B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B.

Akram Idani

▶ **To cite this version:**

Akram Idani. B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B.. domain_stic.inge. Université Joseph-Fourier - Grenoble I, 2006. Français. NNT : . tel-00118718v1

HAL Id: tel-00118718

https://theses.hal.science/tel-00118718v1

Submitted on 6 Dec 2006 (v1), last revised 9 Jan 2015 (v2)

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.

N^o d'ordre: ...

THÈSE

présentée à

L'Université de Grenoble 1

pour obtenir le grade de :

Docteur de l'Université Joseph Fourier
Discipline : Informatique

Titre de la thèse :

B/UML :
Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B

---

par

Akram IDANI

Équipe : **VA**lidation, **S**pécification et **CO**nstruction de logiciels (**VASCO**)
Laboratoire : **L**ogiciels, **S**ystèmes, **R**éseaux (**LSR**) – **IMAG**

Présentée et soutenue publiquement le 29 Novembre 2006

Composition du Jury :

Président du jury	Jean-Pierre Giraudin	Professeur à l'Université de Grenoble 2
Rapporteurs	Régine Laleau	Professeur à l'Université de Paris 12
	Henri Habrias	Professeur à l'Université de Nantes
Examinateur	Michel Léonard	Professeur à l'Université de Genève
Directeurs de thèse	Yves Ledru	Professeur à l'Université de Grenoble 1
	Didier Bert	Chargé de recherche CNRS (LSR/IMAG)

Remerciements

Toutes les personnes m'ayant permis de mener à bien ce travail sont assurées de ma gratitude :
« *Grâce à vous ce qui n'était qu'une question de recherche est devenu une passion* »

À monsieur, Yves Ledru, Professeur à l'Université Joseph Fourier :

« *J'adresse, en signe de reconnaissance, mes plus vifs remerciements pour sa gentillesse et son soutien qu'il a bien voulu manifester à mon égard, me prodiguant les bienveillantes directives dont j'avais besoin pour mener à bien ce travail.* »

À monsieur, Didier Bert, Chargé de recherche CNRS :

« *J'exprime mes sincères remerciements pour ses conseils judicieux, ainsi que ma gratitude pour l'encadrement de qualité dont il m'a fait bénéficier aimablement.* »

Que l'un et l'autre, trouvent ici, l'expression de ma grande estime pour leur personne et de ma profonde considération pour leur compétence professionnelle.

Je remercie vivement M. Jean-Pierre Giraudin qui me fait l'honneur de présider le jury, ainsi que Mme Régine Laleau et M. Henri Habrias pour leurs judicieux commentaires qui ont contribué à l'amélioration de ce document et leurs remarques et critiques très constructives. Je remercie aussi M. Michel Léonard d'avoir accepté d'examiner ce travail.

Je tiens également à remercier M. Farid Ouabdesselam, Directeur du laboratoire Logiciels Systèmes Réseaux, de m'avoir accueilli au sein du LSR où j'ai agréablement effectué mon travail. J'adresse également mes remerciements à Mme Marie-Laure Potet, qui m'a assisté à mes débuts, et qui a contribué à enrichir ma connaissance en B lors de mon projet de DEA.

Merci particulièrement à mes collègues du laboratoire Logiciels Systèmes Réseaux pour leur bonne humeur et leur contribution à l'ambiance chaleureuse. En particulier M. Pierre Berlioux et Nicolas Stouls : partager avec eux le bureau D320 fut un réel plaisir.

Je termine par un grand merci à Messieurs Georges Mariano (INRETS-Lille) et Jean-Louis Boulanger (UT-Compiègne) de m'avoir offert l'opportunité de collaborer avec leurs équipes durant l'année à venir.

Dédicaces

Je dédie ce travail aux deux êtres qui ont donné un sens à ma vie :

- *Ma mère Safia : « Ton amour inconditionnel et ta grande affection forment l'essence de mon existence. Tu es donc avec moi dans tout ce que je fais ».*
- *Mon père Ali qui m'a montré le chemin de l'école : « Tes conseils, ton soutien et ta volonté m'ont toujours inspiré, tu représentes pour moi la clé de ma réussite ».*

Je le dédie aussi à toutes ces personnes qui me sont très chères :

- *Ma soeur Nissaf et mon beau-frère Nizar, ainsi que leur future fille Maysam : « Que dieu vous garde et couronne votre vie de joie et de bonheur ».*
- *Mon frère Nizar pour notre complicité fraternelle et notre estime réciproque : « Tu es loin mais tu es dans mon coeur ».*
- *Mon étoile préférée, ma petite fée, mon ange d'amour Hind pour nos pensées affectueuses : « Main dans la main nous ferons notre destin ; partager ma vie avec toi est mon ultime souhait. Bon courage pour terminer la tienne ».*
- *Mon ami et collègue d'études Mohamed El Benney sans qui ce grand pas n'aurait pas été fait.*

En signe de reconnaissance, je m'adresse à l'éminent professeur M. Braham Abdel Aziz, ainsi qu'aux honorables familles : Campagna, Châabane et El Benney : je n'oublierai jamais qu'à mes débuts, vous m'avez, tous, prêté main forte. Sans votre soutien précieux, ce travail n'aurait pas vu le jour.

Mes pensées vont aussi à M. Tarek Sâadi, Consul Général de Tunisie à Grenoble qui m'a soutenu et encouragé durant les phases cruciales de ce travail de thèse, ainsi qu'à tous ceux qui m'ont aidé de près ou de loin à mener ce travail à bon port.

Je ne saurais oublier mes collègues au LSR ainsi que mes amis en France : Oualid, Mehdi, Khaled, Ahmed, Mohamed Redha, Ben Mostfa, Bachar, Bassem,... : « vous avez tous pris part dans ma vie loin de la famille et loin du pays natal ».

Table des matières

Table des figures **xi**

Liste des tableaux **xvii**

Introduction générale **1**

Partie I B / UML **11**

Chapitre 1 Introduction à la méthode B **13**

1.1 Introduction . . . 14
1.2 Fondements de la méthode B . . . 15
1.2.1 Machine abstraite . . . 15
1.2.2 Typage de données en B . . . 16
1.2.3 Langage des substitutions généralisées . . . 17
1.2.4 Obligations de preuve . . . 20
1.2.5 Exemple . . . 20
1.3 Raffinement . . . 21
1.3.1 Fondements et illustration . . . 21
1.3.2 Obligations de preuve du raffinement . . . 22
1.3.3 Exemple . . . 23
1.4 Modularité en B . . . 24
1.4.1 La clause INCLUDES . . . 24
1.4.2 Les clauses USES et SEES . . . 25
1.4.3 Les autres clauses . . . 26
1.4.4 Aplatissement de spécifications B . . . 26
1.5 Conclusion . . . 28

Chapitre 2 Couplage de spécifications B et UML **29**

2.1 Introduction . . . 29
2.2 Dérivation de UML vers B . . . 31
2.2.1 Aperçu et objectifs . . . 31
2.2.2 Traduction des aspects structurels . . . 32
2.2.3 Traduction des aspects comportementaux . . . 37
2.3 Développement conjoint UML/B . . . 40

2.3.1 Aperçu et objectifs . . . 40
2.3.2 Exemples d'évolutions simultanées de spécifications UML et B . . . 41
2.3.3 Discussion . . . 43
2.4 Dérivation de B vers UML . . . 43
2.4.1 Aperçu et objectifs . . . 44
2.4.2 L'approche uniforme . . . 45
2.4.3 L'approche interactive et incrémentale . . . 49
2.4.4 Notre approche . . . 51
2.5 Conclusion . . . 52

Partie II Correspondances structurelles et sémantiques entre B et UML pour la dérivation de vues structurelles 53

Chapitre 3 Définition d'une syntaxe abstraite UML pour B 55
3.1 Introduction . . . 55
3.2 Méta-modélisation et transformation de modèles : aperçu et usage . . . 56
3.2.1 Transformation de modèles . . . 57
3.2.2 Utilisation des méta-modèles pour la transformation B/UML : notre vision . 57
3.3 Proposition d'un méta-modèle UML pour B . . . 58
3.3.1 Spécification de machines abstraites . . . 59
3.3.2 Spécification de types en B . . . 62
3.3.3 Spécification des structures de raffinement et de composition en B . . . 65
3.4 Conclusion . . . 66

Chapitre 4 Transformation des structures de base 67
4.1 Introduction . . . 68
4.2 Transformation de B vers UML : Illustration par l'exemple . . . 68
4.3 Transformation de machines abstraites . . . 70
4.3.1 Présentation . . . 70
4.3.2 Discussion . . . 72
4.4 Transformation des types de base (*BBasicType*) . . . 73
4.5 Transformation des éléments typés par un *BBasicType* . . . 75
4.5.1 Cas de l'appartenance . . . 75
4.5.2 Cas de l'inclusion . . . 77
4.5.3 Cas de l'égalité . . . 79
4.6 Transformation des éléments typés par un *BPowType* . . . 79
4.7 Transformation des éléments typés par un *BComposedType* . . . 81
4.7.1 Relations et fonctions simples . . . 81
4.7.2 Produit cartésien . . . 84
4.8 Composition de relations et fonctions . . . 84
4.8.1 Composition de relations et fonctions avec un *BPowType* . . . 84
4.8.2 Composition de relations et fonctions avec un produit cartésien . . . 87
4.9 Transformation des opérations d'une machine abstraite . . . 89

4.10 Bilan et discussion 91

Chapitre 5 Transformation des liens entre machines abstraites et application 93

5.1 Introduction 93
5.2 Prise en compte des liens d'inclusion 95
5.3 Prise en compte des liens de raffinement 100
5.4 Application et discussion 101
5.4.1 Application à la spécification *SecureFlight* 101
5.4.2 Comparaison avec d'autres approches 103
5.5 Conclusion 106

Partie III Analyse formelle de concepts pour l'identification de concepts orientés objets à partir de spécifications B 107

Chapitre 6 Analyse formelle de concepts : étude préliminaire 109

6.1 Introduction 109
6.2 Analyse formelle de concepts : aperçu et usage 110
6.3 Prise en compte de la structuration des *BData*s 112
6.3.1 Diagramme de structure préliminaire 112
6.3.2 Discussion 113
6.4 Prise en compte des *BOperation*s 115
6.4.1 Restructuration préliminaire 118
6.4.2 Application 119
6.5 Bilan et discussion 120

Chapitre 7 Analyse formelle de concepts : mise en œuvre 123

7.1 Introduction 123
7.2 Identification des classes candidates 124
7.2.1 Correspondance entre spécifications B et diagrammes préliminaires 124
7.2.2 Relation d'inclusion entre éléments de modélisation préliminaires 126
7.3 Construction de groupements conceptuels de données B 129
7.3.1 Les contextes 129
7.3.2 Construction des contextes 131
7.3.3 Validation et preuve de l'algorithme de construction des contextes 136
7.4 Conclusion 139

Chapitre 8 Transformation des modèles de contextes en diagrammes de classes 141

8.1 Introduction 141
8.2 Règles de transformation 142
8.2.1 Définition 142
8.2.2 Exemple d'application 143
8.3 Des contextes aux classes 144
8.4 Transformation des attributs des contextes 147
8.4.1 Présentation 147

8.4.2 Visibilité des attributs des classes 151
8.5 Transformation des opérations des contextes 151
8.6 Synthèse 152

Chapitre 9 Optimisation et prise en compte des développements structurés 153
9.1 Introduction 153
9.2 Matrices de liens 154
9.2.1 Matrices de liens simples 155
9.2.2 Matrices de liens pondérées 157
9.2.3 Pertinence des modèles de contextes 158
9.2.4 Évaluation 161
9.3 Prise en compte des raffinements 162
9.3.1 Représentation du plus haut niveau d'abstraction 163
9.3.2 Premier niveau de raffinement 164
9.3.3 Derniers niveaux de raffinement 164
9.3.4 Mise en relation des vues produites pour les raffinements 165
9.4 Conclusion 167

Partie IV Construction de vues comportementales à partir de spécifications B 169

Chapitre 10 Vues comportementales issues de spécifications B 171
10.1 Introduction 172
10.2 Notions d'état et de transition 173
10.2.1 Présentation informelle 173
10.2.2 Formalisation 174
10.3 Vues comportementales concrètes 175
10.3.1 Exploration du comportement d'une spécification B 176
10.3.2 Représentation concrète structurée du raffinement 177
10.3.3 Limites des vues concrètes 179
10.4 Vues comportementales abstraites 180
10.4.1 Le gestionnaire de processus 180
10.4.2 Graphes d'états concrets et explosion d'états 181
10.4.3 Focalisation sur une variable particulière 182
10.4.4 Mise en évidence sous forme concurrente de toutes les variables d'état . . . 183
10.4.5 Représentation graphique d'une propriété invariante 185
10.4.6 Représentation du comportement d'un processus 185
10.5 Bilan 187

Chapitre 11 Techniques de dérivation de vues comportementales 189
11.1 Introduction 190
11.2 Outils de construction de vues comportementales 190
11.2.1 Animation basée sur les suites de test 190
11.2.2 Abstraction de graphes d'accessibilité 193

11.2.3 L'approche `GeneSyst` . . . 196
11.2.4 Propriétés des transitions . . . 200
11.3 Identification des états abstraits . . . 200
11.3.1 Recours aux données de la spécification . . . 201
11.3.2 Recours aux besoins informels de l'utilisateur . . . 202
11.3.3 Définition d'un catalogue de patrons d'états abstraits . . . 203
11.4 Dérivation de diagrammes d'états/transitions sur la base des vues structurelles . . . 206
11.4.1 Formalisation . . . 207
11.4.2 Prédicats d'états associés aux classes . . . 208
11.4.3 Application et discussion . . . 211
11.5 Bilan et conclusion . . . 212

Conclusion générale **213**

Bibliographie **223**

Annexe A Nos outils prototypes **233**
A.1 Outil de dérivation de diagrammes de classes . . . 233
A.1.1 Présentation . . . 233
A.1.2 Architecture . . . 234
A.1.3 Quelques résultats . . . 235
A.2 Outil d'abstraction de graphes d'accessibilité . . . 237
A.2.1 Présentation . . . 237
A.2.2 Scénario de fonctionnement . . . 238

Annexe B Méta-modèle d'UML : quelques extraits **241**
B.1 Classifier . . . 241
B.2 Structural Feature . . . 241
B.3 Data Types . . . 243
B.4 Packages . . . 244

Annexe C Exemples d'application des schémas de transformation **245**

Annexe D Questionnaire **253**
D.1 Introduction . . . 253
D.2 Questions générales . . . 253
D.3 Spécifications B considérées . . . 253
D.4 Le gestionnaire de processus (SCHEDULER) . . . 254
D.5 Le contrôleur d'accès . . . 258
D.5.1 Vue Structurelle . . . 259
D.5.2 Vues comportementales . . . 260
D.6 Évaluation . . . 264

Annexe E La spécification SecureFlight **265**

Glossaire **267**

Table des figures

1 Acteurs et documents dans l'approche EDEMOI . . . 4
2 Stratégies d'intégration . . . 5
3 Le processus B/UML . . . 7

1.1 Processus de développement en B . . . 14
1.2 Représentation du raffinement des opérations . . . 22
1.3 Visibilité entre composants B établie par la clause INCLUDES . . . 25
1.4 Visibilité entre composants B établie par la clause USES . . . 25
1.5 Aplatissement du lien de raffinement REFINES . . . 27
1.6 Aplatissement du lien d'inclusion INCLUDES . . . 27

2.1 Traduction du concept d'héritage selon l'approche compilée (Facon *et al.*, 1995) . . . 31
2.2 Traduction du concept d'héritage selon l'approche interprétée (Laleau, 2002) . . . 32
2.3 Traduction des concepts de classe et d'attribut en B. . . . 33
2.4 Traduction du mécanisme d'héritage . . . 34
2.5 Traductions d'associations préconisées par (Laleau, 2002) . . . 35
2.6 Exemple de traduction d'états proposé par (Nguyen, 1998) . . . 38
2.7 Évolution de l'état des spécifications B et UML selon l'approche (Ossami *et al.,* 2004) . . . 40
2.8 Schéma simplifié du processus de développement formel de logiciels extrait de (Gaudel, 1995) . . . 44
2.9 Machine B simple . . . 46
2.10 Application règles proposées par (Voisinet, 2004) à la machine M . . . 47
2.11 Processus de dérivation de B vers UML extrait de (Fekih *et al.,* 2006) . . . 50

3.1 Exemple de transformation de modèles : du modèle Objet au modèle relationnel . . . 58
3.2 Processus de transformation dirigé par les méta-modèles UML et B. . . . 58
3.3 Composition de la méta-classe *BMachine* . . . 59
3.4 Dépendances et composition de la méta-classe *BOperation* . . . 60
3.5 Spécialisations de la méta-classe *BData* . . . 60
3.6 Méta-modèle UML pour la spécification de machines abstraites B . . . 62
3.7 Spécialisations de *BTypedElement* . . . 62
3.8 Spécification du mécanisme de typage . . . 63
3.9 Spécialisations de *BBasicType* . . . 63
3.10 Spécification des types composés . . . 64
3.11 Méta-modèle UML pour la spécification de types en B . . . 65
3.12 Spécification des structures de raffinement et de composition en B . . . 65

4.1 Exemple d'un modèle B composé de deux machines abstraites . . . 69
4.2 Schéma de transformation Machine / Classe . . . 71
4.3 Schéma de transformation Machine / Paquetage . . . 72

4.4 Application des schémas de transformation 1 et 2 73
4.5 Schéma de transformation « types de base » 74
4.6 Illustration du schéma de transformation 3 75
4.7 Schéma de transformation « Éléments typés par un *BBasicType* – Cas de l'opérateur In » (Transformation en attributs de classes) 76
4.8 Illustration du schéma de transformation 4.1 76
4.9 Illustration du schéma de transformation 4.2 77
4.10 Illustration des schémas de transformation 5.2 et 5.3 77
4.11 Illustration du schéma de transformation 5.4 78
4.12 Illustration du schéma de transformation 5.5 79
4.13 Illustration du schéma de transformation 5.6 79
4.14 Illustration du schéma de transformation 8 81
4.15 Schéma de transformation « Éléments typés par un *BComposedType* – Cas 1 » 82
4.16 Illustration du schéma de transformation 9.1 82
4.17 Illustration du schéma de transformation 9.2 83
4.18 Illustration du schéma de transformation 9.3 83
4.19 Illustration du schéma de transformation 9.4 84
4.20 Illustration du schéma de transformation 11.1 85
4.21 Illustration du schéma de transformation 11.2 86
4.22 Illustration du schéma de transformation 11.3 86
4.23 Transformation de $R \in \mathbb{P}(S) \rightarrowtail \mathbb{P}(T)$ 87
4.24 Illustration du schéma de transformation 12 88
4.25 Illustration du schéma de transformation 13 88
4.26 Illustration du schéma de transformation 14 89
4.27 Schéma de transformation 15 90
4.28 Illustration du schéma de transformation 15 90
4.29 Illustration du schéma de transformation 16 91

5.1 Extrait du méta-modèle UML relatif aux dépendances entre paquetages 94
5.2 Schéma de transformation « Inclusion entre machines – Cas 1 » 95
5.3 Illustration du schéma de transformation 17.1 96
5.4 Schéma de transformation « Inclusion entre machines – Cas 2 » 97
5.5 Illustration du schéma de transformation 17.2 97
5.6 Illustration du schéma de transformation 17.3 98
5.7 Structure du patron « *Singleton* » 99
5.8 Application du patron « *Singleton* » 100
5.9 Illustration du schéma de transformation 17.4 100
5.10 Traduction de la machine *BoardingGate* 102
5.11 Traduction de la machine *SecureFlightBoarding* 103
5.12 Application des schémas de transformation à la spécification *SecureFlight* 103
5.13 Application de l'approche de (Voisinet, 2004, Tatibouet et *al.,* 2002) sur BoardingGate 104
5.14 Application de l'approche de (Voisinet, 2004, Tatibouet et al., 2002) sur la machine *SecureFlightBoarding* 105

6.1 Machine *SecureFlightRegistration* 113
6.2 Diagramme de structure préliminaire produit pour la machine *SecureFlightRegistration* 113
6.3 Diagramme construit par (Voisinet, 2004) pour *SecureFlightRegistration* 114
6.4 Extrait du méta-modèle B – dépendance entre opérations et données B 115
6.5 Matrice binaire représentant la relation de dépendance de concepts issue de *SecureFlightRegistration* 116

6.6 Graphe bi-parties représentant le(s) choix de classes pour chaque opération de *SecureFlightRegistration* (relation $\mathcal{I}^{-1}$) 118
6.7 Graphe bi-parties représentant le(s) choix le(s) plus pertinent(s) de classes pour chaque opération de *SecureFlightRegistration* (sous ensemble de la relation $\mathcal{I}^{-1}$) 119
6.8 Groupements conceptuels de données et d'opérations issus de *SecureFlightRegistration* 119
6.9 Restructuration du diagramme de structure préliminaire $-$ 1$^{\text{er}}$ cas 120
6.10 Restructuration du diagramme de structure préliminaire $-$ 2$^{\text{ème}}$ cas 120
6.11 Autres groupements conceptuels de données et d'opérations dérivés de *SecureFlightRegistration* . 121
6.12 Autre diagramme de classes produit pour *SecureFlightRegistration* 121

7.1 Approche proposée pour la construction des diagrammes de classes 124
7.2 Transformation préliminaire de *registered_objects* 125
7.3 Graphe illustrant la relation $\mathcal{I}^{-1}$ issue de SecureFlight (Annexe E) 127
7.4 Graphe illustrant la relation d'inclusion $Incl$ 128
7.5 Relation $\mathcal{J}$ issue de la fusion de *SecureFlightBoarding* et *SecureFlightRegistration* 132
7.6 Sous-graphe du réseau de concepts raffiné $\mathcal{J}_{\mathcal{R}}$ où les racines sont toutes des opérations 133
7.7 Illustration de l'étape (*ii*) de l'algorithme 134
7.8 Modèle de contextes issu de la fusion de *SecureFlightBoarding* et *SecureFlightRegistration* 135
7.9 Exemple de transformation d'un contexte en une classe 136

8.1 Diagramme de classes préliminaire issu de la machine SecureFlight 143
8.2 Diagramme de classes issu du modèle de contextes de la Fig. 7.8 144
8.3 Illustration de la règle $\mathcal{R}_2$ avec $A \subseteq B$ 146
8.4 Illustration de la règle $\mathcal{R}_3$ avec $A \in B \rightarrow C$ 147
8.5 Illustration de la règle $\mathcal{R}_4$ 148
8.6 Illustration de la règle $\mathcal{R}_5$ 149
8.7 Illustration de la règle $\mathcal{R}_6$ $-$ Cas 3. 151
8.8 Exemple de prise en compte des paramètres des méthodes 152

9.1 Modèle de contextes formé uniquement des contextes *Passenger* et *Luggage* 154
9.2 Matrice de liens associée à une *BData* d avec $\mathcal{I}[\{d\}] = \{o_1, o_2, o_3\}$ 156
9.3 Matrice de liens illustrant la prise en compte des opérations privées 156
9.4 Matrice de liens pondérée déterminée pour une combinaison OR des opérations de d 158
9.5 Matrice de liens pondérée prenant en compte les opérations privées 158
9.6 Réseau de concepts $\mathcal{J}$ et réseau de concepts raffiné $\mathcal{J}_{\mathcal{R}}$ issus de *SecureFlightRegistration* 159
9.7 Modèles de contextes dérivés de *SecureFlightRegistration* 159
9.8 Matrices de liens pondérées associées à *registeredP* et *Luggage* 160
9.9 Pertinences $\mathcal{P}_{\mathcal{G}}$ calculées pour les modèles de contextes (a), (b) et (c) 160
9.10 Comparaison des modèles de contextes (a), (b) et (c) 160
9.11 Classement des 65 modèles de contextes issus de la spécification *SecureFlight* 161
9.12 Diagramme de classes issu de l'un des modèles de contextes dont la pertinence moyenne est égale à 95% 161
9.13 Développements en B de SecureFlight 163
9.14 Diagramme de classes issu du plus haut niveau d'abstraction de SecureFlight 163
9.15 Diagramme de classes produit pour le raffinement SecureFlightR1 164
9.16 Diagramme de classes issu du dernier niveau de raffinement 164
9.17 Optimisation du nombre de vues produites pour les différents raffinements 166

10.1 Illustration de l'évolution de l'état de la machine RESERVATION 172
10.2 Diagramme d'états / transitions abstrait dérivé de la machine RESERVATION 173

10.3 Système COMMUNICATION 175
10.4 Diagramme d'états/transition issu du système COMMUNICATION 175
10.5 Raffinement du Système COMMUNICATION 178
10.6 Diagramme d'états/transitions concret issu du raffinement du système COMMUNICATION 178
10.7 Diagramme d'états/transitions hiérarchique concret issu de COMMUNICATION_R$_1$ 178
10.8 Diagramme d'états/transitions abstrait issu de COMMUNICATION_R$_1$ 179
10.9 Spécification B du gestionnaire de processus 180
10.10 Graphe d'états concret produit pour $Process = \{p_1, p_2\}$ 182
10.11 Diagramme d'états/transition abstrait associé à la variable $active$ 182
10.12 Diagramme d'états/transitions concurrent portant sur les variables $active$, $ready$ et $waiting$ . . . 184
10.13 Diagramme d'états/transitions déplié portant sur les variables $active$, $ready$ et $waiting$ 184
10.14 Diagramme d'états/transitions associé à l'invariant : $active = \emptyset \Rightarrow ready = \emptyset$ 185
10.15 Diagramme de classes issu de la machine SCHEDULER 186
10.16 Automate du cycle de vie d'un processus p_i 186

11.1 Représentation graphique de la suite de tests produite par **CASTING** et des états résultants 191
11.2 Représentation graphique abstraite de deux suites de tests 192
11.3 Graphe d'accessibilité produit par ProB étant donné $Process = \{p_1, p_2, p_3\}$ 194
11.4 Preuves générées par l'atelier B pour $active = \emptyset$ et $Process = \{p_1, p_2\}$ 196
11.5 Preuves générées par l'atelier B pour $active = \emptyset$ et $Process = \{p_1, p_2, p_3\}$ 196
11.6 Diagrammes abstraits illustrant le cycle de vie du processus p_1 construits par notre prototype . . . 197
11.7 Diagrammes abstraits illustrant le cycle de vie du processus p_1 construits par l'outil **GeneSyst** . 199
11.8 Ingénierie des besoins vue par A. Davis (Davis, 1993) 202
11.9 Illustration d'une exigence 203
11.10 Application du patron "Énumération" 204
11.11 Application du patron "Inclusion" 204
11.12 Vue hiérarchique produite par application du patron "Inclusion" 205
11.13 Diagramme de classes issu de la machine ACCESSCONTROL 207
11.14 Classes issues de machines B avec leurs diagrammes d'états/transitions 209
11.15 Diagramme d'états/transitions associé à la classe "BATIMENT" 211
11.16 Diagramme d'états/transitions concurrent pour à la classe "BATIMENT" 211

A.1 Interface principale de notre outil de dérivation de diagrammes de classes 233
A.2 Architecture de l'outil de dérivation de diagrammes de classes 234
A.3 Interface de visualisation d'un diagramme de classes correspondant à un choix particulier de cibles 235
A.4 Diagrammes de classes générés par notre outil pour l'exemple *SecureFlightV*1 236
A.5 Interfaces de notre outil d'abstraction de graphes d'accessibilité 237
A.6 Scénario de fonctionnement de l'outil d'abstraction de graphes d'accessibilité 238

B.1 Features diagram of the Kernel package 242
B.2 Classes diagram of the Kernel package 242
B.3 Multiplicities diagram of the Kernel package 243
B.4 Data types diagram of the Kernel package 243
B.5 Namespaces diagram of the Kernel package 244
B.6 The Packages diagram of the Kernel package 244

C.1 Diagramme de classes dérivé de la machine SMA 246
C.2 Diagramme de classes dérivé d'une spécification de gestion de factures 247
C.3 Autre diagramme de classes dérivé de la spécification *SecureFlight* 249

D.1 Spécification B du gestionnaire de processus 254
D.2 Diagrammes d'états/transitions associé à la variable $active$ 255
D.3 Prise en compte d'une propriété invariante 256
D.4 Cycle de vie d'un processus 257
D.5 Spécification B du contrôleur d'accès 258
D.6 Diagramme de classes du contrôleur d'accès 259
D.7 Un diagramme d'états/transitions associé à la classe Batiment 260
D.8 Diagramme d'états/transitions concurrent pour à la classe "BATIMENT" 261
D.9 Diagramme d'états/transitions concurrent pour à la classe "CARTE" 263

Liste des tableaux

1 Méthodes formelles Vs Méthodes semi-formelles 5

1.1 Substitutions primitives 18
1.2 Substitutions dérivées 18
1.3 Exemples d'applications réelles de B extraites de (Behm *et al.,* 1997) 28

2.1 Points communs et divergences entre différentes approches de dérivation de UML vers B pour la traduction des aspects structurels 36
2.2 Points communs et divergences entre différentes approches de dérivation de UML vers B pour la traduction des aspects comportementaux 39

3.1 Table des multiplicités associée aux spécialisations de relations fonctionnelles en B 64

5.1 Succession de schémas de transformation pour transformer la machine *BoardingGate* en un diagramme de classes 102

7.1 Illustration des différents choix de cibles pour chaque opération 135
7.2 Obligations de preuves produites par l'atelier B pour chaque étape de l'algorithme 139

9.1 Nombre de diagrammes de classes construits pour les développements de *SecureFlight* 162

11.1 Suite de tests produite par **CASTING** 191
11.2 Temps de calcul de l'approche par model-checking et de l'approche **GeneSyst** 199
11.3 Utilisation du patron "Intervalle" 205
11.4 Utilisation du patron "Énumération" 206
11.5 Utilisation du patron "Inclusion" 206

Introduction générale

« Le concret c'est l'abstrait rendu familier par l'usage. La notion d'objet, abstraite à l'origine, arbitrairement découpée dans l'univers, nous est devenue familière à tel point que certains d'entre nous pensent que nous ne pouvions pas utiliser autre chose comme base pour construire notre représentation du monde [...] J'ai pour ma part, plus de confiance dans les possibilités de notre évolution mentale. »

P. Langevin
« La pensée et l'action », 1950.

Les exigences qui s'appliquent aux composants logiciels et aux logiciels embarqués justifient l'utilisation des meilleures techniques disponibles pour garantir la qualité des spécifications et conserver cette qualité lors du développement du code. Les travaux de recherche relevant du domaine du génie logiciel se sont largement intéressés à la définition de méthodes de développement et de langages de spécifications ainsi qu'à la mise en place d'outils adéquats assurant le développement de logiciels dans un cadre unifié et ce, dans le but de les intégrer à un contexte industriel. Tel a été l'objectif du standard UML (Booch *et al.,* 2002) − actuellement devenu un standard de facto dans le monde industriel − proposé par l'*Object Management Group*[1] et dont le succès est largement lié à la simplicité de ses notations. En effet, les notations à objets se veulent structurantes, intuitives et faciles à appréhender. En revanche, la sémantique d'UML est souvent qualifiée de floue du fait qu'elle manque de bases mathématiques.

Cependant, la complexité grandissante des logiciels est de plus en plus évidente et la maîtrise des risques inhérents à leur utilisation devient impérative et appelle, de ce fait, une rigueur et une précision accrues lors de leur élaboration. Aussi l'adoption de moyens d'investigation indubitables et de techniques sûres et fiables, reposant sur des fondements mathématiques, s'impose-t-elle, désormais, en tant que garant de la sécurité. C'est ainsi que les récents succès de développement de logiciels dans le cas de METEOR (Behm *et al.,* 1999) et de la méthode B (Abrial, 1996) ou encore l'analyse des codes de la fusée européenne Ariane 501, ont montré que des approches systématiques et rigoureuses constituent des réponses effectives à ces questions de sûreté technique et de garantie.

Ces méthodes, dites formelles, ont été élaborées afin d'assurer un niveau aussi élevé que possible en matière de précision et de cohérence. Leur avantage majeur réside, en somme, dans le fait qu'elles sont basées sur les mathématiques, ce qui permet, d'une part, de neutraliser tous les risques d'ambiguïté et d'incertitude, et d'autre part, de parvenir à un produit fini qui répond aux spécifications requises.

[1] www.omg.org

Cependant, ces méthodes utilisent des notations et des concepts spécifiques qui génèrent souvent une faible lisibilité et une difficulté d'intégration dans les processus de développement et de certification.

Ainsi, il est intéressant d'intégrer ces deux types de méthodes (UML et méthodes formelles) en vue de tirer profit de leurs avantages respectifs. Plusieurs efforts sont alors déployés afin de proposer des environnements de spécification, de développement de programmes et de logiciels, combinant les méthodes formelles et les méthodes graphiques largement utilisées dans les projets industriels, en l'occurrence B et UML. Cet intérêt est justifié par les aspects complémentaires et les apports croisés de ces deux techniques (Dupuy-Chessa, 2000). Notre intérêt portera précisément sur la méthode B qui est une méthode formelle utilisée pour modéliser des systèmes et prouver l'exactitude de leur conception par raffinements successifs. Cependant, les spécifications formelles sont difficiles à lire quand elles ne sont pas accompagnées d'une documentation. Cette lisibilité est essentielle pour une bonne compréhension de la spécification, notamment dans des phases de validation ou de certification. Aujourd'hui, en B, cette documentation est fournie sous forme de texte, avec, quelquefois, des schémas explicitant certaines caractéristiques du système. La documentation et le système sont liés de façon informelle. En vue de pallier cette insuffisance, nous proposons dans le cadre de ce travail de thèse, d'utiliser des diagrammes UML pour produire une documentation standardisée et lisible au sein des projets en B.

Le but de notre investigation est de conférer aux spécifications B une structuration sous forme de vues graphiques telle que leur degré de compréhension et d'accessibilité soit rehaussé : l'ambiguïté des descriptions graphiques sera levée par les développements formels à partir desquels ces descriptions sont issues, la documentation pourra être maintenue durablement et la communication avec des ingénieurs qui ne seraient pas familiarisés avec la méthode B sera plus aisée. Le lien que nous nous proposons d'établir entre diagrammes de UML et modélisation en B a pour objectif d'aboutir à une construction semi-automatique et outillée de vues structurelles et comportementales à partir de spécifications B existantes.

Motivations

Les méthodes formelles commencent à être adoptées dans le contexte industriel comme étant indispensables pour le développement de systèmes dits "critiques". Les principales caractéristiques de ces méthodes sont :

(*i*) Elles permettent de *vérifier la correction d'un logiciel par rapport à sa spécification*.

(*ii*) Elles nécessitent *une forte connaissance de la logique*.

Le premier point, qui représente l'avantage majeur de ces méthodes, relève du fait que les modèles mathématiques permettent un raisonnement rigoureux sur la cohérence du système. Le second point, qui se présente, certes, comme le frein essentiel à l'adoption des méthodes formelles, est fortement lié aux notations, souvent complexes, des langages formels. Ces deux caractéristiques ont fait que les méthodes formelles restent dédiées aux systèmes critiques, pour lesquels la sûreté et la perfection sont indispensables. L'une des solutions proposées dans plusieurs travaux de recherche est de spécifier entièrement le système en se basant sur un langage semi-formel (*e.g.* UML) et de traduire ensuite le modèle ainsi construit en un modèle formel. Les spécifications qui résultent de cette traduction, peuvent alors être exploitées en vue de poursuivre un raisonnement rigoureux du même système. Cependant, ces techniques présentent les inconvénients suivants :

(*i*) Les spécifications formelles générées sont complexes et très éloignées de ce que le développeur aurait pu écrire directement en un langage formel. Ceci est dû au fait que la traduction systématique doit prendre en compte les constructions du langage semi-formel de départ.

(*ii*) Un effort important de compréhension des spécifications formelles est requis en vue de pouvoir les raffiner, les corriger et effectuer les preuves formelles. En cas d'inconsistance, la correction du modèle formel doit induire la correction du modèle semi-formel, ce qui n'est pas trivial.

Bien que cette approche ait été largement exploitée (nous passerons en revue les principaux travaux au niveau du chapitre 2), plusieurs entreprises optent pour l'application directe des méthodes formelles. À titre d'exemples, Siemens Transport (Essamé, 2004), Clearsy (Pouzancre, 2003, Pouzancre *et al.,* 2003), Gemplus (Casset, 2002) et KeesDA (Hallerstede, 2003) ont adopté un processus de développement de logiciels entièrement basé sur la méthode formelle B. Ceci est motivé, principalement, par la qualité des développements formels en B : nous citons, en guise d'exemple, les développements de METEOR[2] qui ont atteint l'objectif du "zéro fautes" (Behm *et al.,* 1999).

Toutefois, la qualité des développements dépend aussi de leur facilité de compréhension, ce qui est capital lors de la validation aussi bien par des clients que par des autorités de certification. Ladite validation, qualifiée de *validation externe* par M.-C. Gaudel (Gaudel, 1995), constitue l'élément moteur de notre recherche. Aussi, notre travail de thèse se propose-t-il de contribuer à une meilleure intégration de B dans un processus industriel de développement en proposant une passerelle entre la méthode B et le langage UML. Finalement, nous soulignons que dans notre démarche, UML ne constitue pas un point de départ, mais un résultat vu qu'il s'agit d'amener UML dans le monde de B et non pas l'inverse. L'objectif principal est un objectif de présentation des spécifications dans des notations graphiques plus faciles à appréhender et utiles lors des phases de validation externe et surtout lorsque ces phases n'impliquent pas nécessairement des personnes familiarisées avec les notations B.

Le projet EDEMOI

Le projet EDEMOI[3] (Élaboration d'une DÉmarche pour la MOdélisation Informatique, la validation et la restructuration de réglementations de «sûreté» (sécurité), et la détection des biais dans les aéroports) représente un domaine d'application intéressant de notre travail de thèse. L'objectif de ce projet porte sur la construction de modèles de la réglementation des aéroports pour la mise en évidence des imprécisions ou biais éventuels. Ces modèles sont de deux natures :

- des modèles graphiques semi-formels exprimés en UML (Laleau *et al.,* 2006), construits à l'aide de techniques d'analyse des exigences (Requirements Engineering). Ces modèles constituent un premier élément de formalisation qui se veut lisible à la fois par les informaticiens et par les autorités de l'aviation civile.
- des modèles formels en B (Bert *et al.,* 2006), Z et Focal (Etienne *et al.,* 2006) qui expriment précisément les propriétés de la réglementation et se prêtent à des traitements automatiques pour vérifier leur cohérence et générer des cas de test.

[2] Métro sans conducteur de Paris.
[3] www-lsr.imag.fr/EDEMOI/

La Fig. 1 extraite de (Ledru *et al.,* 2006) illustre les modèles formels et semi-formels développés dans le cadre du projet EDEMOI ainsi que le rôle des principaux intervenants dans ce projet. Remarquons que les autorités de l'aviation civile n'interviennent qu'au niveau du modèle semi-formel pour valider sa conformité par rapport au standard international de la réglementation. Le modèle formel, produit par des spécialistes indépendamment du modèle semi-formel et dont l'objectif est de permettre une analyse de la cohérence du standard, ne doit pas être très éloigné du modèle semi-formel. Ainsi, assurer la traçabilité entre ces deux modèles (formel et semi-formel) s'avère d'une grande importance dans le cadre du projet. L'application de notre technique permettra de retrouver, à partir des spécifications B du projet, des vues UML proches de ce qui est spécifié dans un cadre semi-formel et présenter ainsi un support de communication compréhensible.

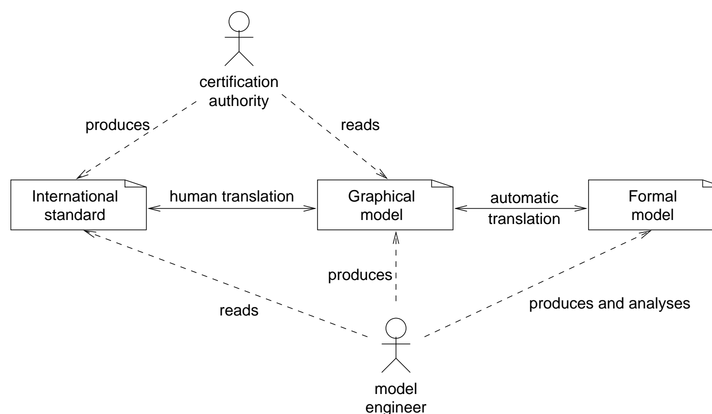


FIG. 1 – Acteurs et documents dans l'approche EDEMOI

Contexte du travail

Dans le domaine du génie logiciel, le terme “méthode” est utilisé pour désigner plusieurs notions : c'est à la fois une technique, une démarche, un formalisme, et des outils d'aide à la résolution de problèmes (Junior, 1997). Notre intérêt portera plus particulièrement sur le formalisme utilisé par chacune des méthodes.

D'une part, les spécifications formelles sont exprimées dans des langages dont la syntaxe et la sémantique sont bien précises. En effet, elles sont construites sur une base théorique solide permettant des validations automatisées. D'autre part, les spécifications semi-formelles sont définies autour de langages textuels ou graphiques, avec une syntaxe précise et une sémantique assez faible (André *et al.,* 1995, Simons *et al.,* 1999, Fecher *et al.,* 2005). Le Tab. 1 synthétise ces notions.

Disposer d'une méthode qui garantisse le développement de systèmes à la fois bien structurés et fiables est l'un des objectifs du génie logiciel (Bruel, 1999). C'est ce qui explique les évolutions des méthodes formelles et semi-formelles. D'une part, l'évolution des méthodes formelles tend vers l'intégration de la notion de modularité pour répondre à des besoins de structuration et de clarté des spécifications. On trouve, par exemple, la notion de machine abstraite en B, de schéma en Z ou encore de module en VDM, etc. D'autre part, les méthodes semi-formelles commencent à intégrer des notations mathématiques pour l'expression de certaines caractéristiques (ou contraintes) des systèmes (OCL (Warmer *et al.,* 1998) par

	Méthodes semi-formelles	**Méthodes formelles**
Formalisme	Textuel ou graphique (Merise, SADT, UML, . . .)	Mathématique (Z, VDM, B, . . .)
Syntaxe du langage	Précise	Précise
Sémantique du langage	Assez faible	Précise
Validation	Syntaxique + Expertise humaine	Preuves Démonstration de théorèmes Model checking Animation et test
Outils	Ateliers de Génie Logiciel (AGL)	Prouveurs + Animateurs
Domaine applicatif	Se veulent généralistes	Systèmes sûrs ou critiques
Objectif	Systèmes bien structurés	Systèmes fiables

TAB. 1 – Méthodes formelles Vs Méthodes semi-formelles

exemple). L'intégration des notations graphiques (plus particulièrement des notations à objets) et mathématiques a été étudiée depuis plusieurs années (Fraser *et al.,* 1991). Nous suggérons ci-dessous (Fig. 2) une classification de ces travaux d'intégration selon la stratégie de développement qu'elle adopte.

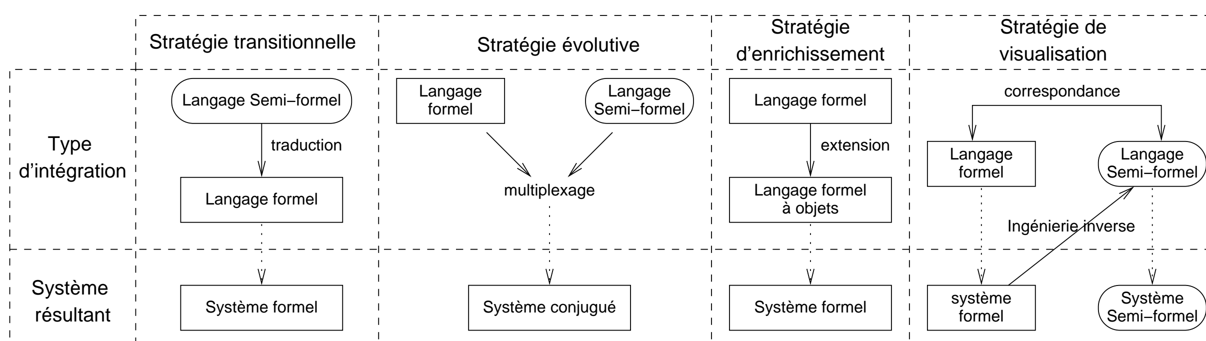


FIG. 2 – Stratégies d'intégration

Dans le cadre d'un développement principalement basé sur un langage semi-formel[4], l'intégration d'un langage formel se fait selon deux stratégies essentielles : la stratégie **transitionnelle**, et la stratégie **évolutive**.

a) **Stratégie transitionnelle** : cette stratégie se base sur une intégration par dérivation (ou traduction) impliquant une phase de transition de spécifications semi-formelles vers des spécifications formelles équivalentes. (Meyer, 2001) distingue différentes approches de l'intégration par dérivation : modélisation, méta-modélisation, méta-méta-modélisation. Dans ce cadre, le modèle formel résultant peut alors être enrichi, raffiné, etc, et l'application des techniques de vérification et de preuves ainsi que l'utilisation d'outils développés autour des méthodes formelles deviennent alors possibles. Le lecteur intéressé peut se référer à (Amálio *et al.,* 2003) où les auteurs présentent une revue de travaux de formalisation de UML en Z et Object-Z. Notons également qu'au niveau du chapitre 2 nous présenterons un aperçu de travaux adoptant une stratégie transitionnelle pour

[4] Par langage semi-formel nous désignons les langages visuels ou graphiques (UML, . . .).

traduire des modèles UML en spécifications B.

b) **Stratégie évolutive** : il s'agit d'étendre le langage semi-formel en y introduisant des notations formelles. Ce mécanisme permet d'exprimer formellement certaines propriétés du modèle semi-formel. Nous citons en guise d'exemple le langage OCL (Warmer *et al.,* 1998) qui a pour but d'exprimer des contraintes sur des diagrammes UML. Dans le même contexte, le formalisme des diagrammes d'états étendus (Attiogbé *et al.,* 2003)[5] permet d'introduire au niveau des diagrammes d'états-transitions d'UML des annotations algébriques en Z ou en B. Le système résultant évolue alors d'une simple description semi-formelle à un système conjugué plus précis.

Notons que plusieurs travaux combinent les deux stratégies. Par exemple, dans (Mammar *et al.,* 2005) les auteurs proposent de traduire un modèle UML enrichi par des spécifications B[6] en un modèle B équivalent ; et ce, en vue de poursuivre un développement formel B aboutissant à la génération d'un code exécutable JAVA/SQL.

Dans le cadre d'un développement fondé sur un langage formel, cette intégration peut s'appliquer également selon deux stratégies : la stratégie **d'enrichissement**, et la stratégie **de visualisation**.

c) **Stratégie d'enrichissement** (appelée aussi *intégration conjointe* dans (André, 1995) et *intégration par extension* dans (Meyer, 2001)) : à ce niveau le paradigme objet est perçu comme un mécanisme de structuration. Il s'agit de définir un langage de spécification formelle "à objets"[7] en étendant un langage de spécification formelle existant. Ce type de stratégie présente deux grands avantages :
   - Disposer d'un langage de spécification formelle permettant une composition et une décomposition modulaire (Cheung *et al.,* 1999).
   - Réutiliser les environnements de spécification et de preuve des méthodes formelles.

   Nous citons, à titre de référence, Z++ (Lano, 1992), Object-Z (Smith, 1995) et VDM++ (Dürr *et al.,* 1992).

d) **Stratégie de visualisation** : l'objectif de cette stratégie est de construire une vue complémentaire graphique, bien que partielle, sur des spécifications formelles. Cette technique nécessite une phase de correspondance établissant les liens structurels et sémantiques entre un sous-ensemble de constructions du langage formel et ceux du langage semi-formel cible. Les modèles obtenus servent, principalement, de documentation graphique du modèle formel. Dans (Dick *et al.,* 1991), Jérémy Dick et Jérôme Loubersac proposent une syntaxe graphique pour VDM en vue d'assurer la traçabilité entre la documentation et les spécifications en VDM. D'autres travaux (Kim *et al.,* 1999, Chen *et al.,* 2004, Kim *et al.,* 2001), utilisent UML pour visualiser des spécifications formelles en Z ou en Object-Z. Finalement, (Leuschel *et al.,* 2003, Voisinet, 2004, Fekih *et al.,* 2004, Bert *et al.,* 2005) se sont attachés à la visualisation des aspects structurels et comportementaux de spécifications B.

Notre travail de thèse s'inscrit dans cette dernière catégorie de travaux. En effet, nous adoptons une stratégie de visualisation pour produire une documentation graphique UML de projets spécifiés initialement en B.

[5] Diagrammes d'états étendus ou Extended State Diagrams (ESD)

[6] Le modèle conjugué est appelé IS-UML (Laleau *et al.,* 2001) où la signature des opérations est exprimée en B.

[7] Connus aussi sous l'appellation "langages formels Orientés Objets" (Lano, 1995).

Contributions & Propositions

Dans l'optique de la validation d'une application, une telle mise en relation apporte certains bénéfices : (*i*) la lisibilité de la spécification initiale, (*ii*) la mise en évidence graphique des propriétés d'un système ; ainsi que (*iii*) l'élaboration de différents points de vue sur le système.

En effet, la présentation de spécifications UML construites à partir de spécifications B, facilitera la compréhension de ces spécifications par les divers acteurs d'un développement de logiciel (client, ingénieur de développement, ingénieur de certification, . . .).

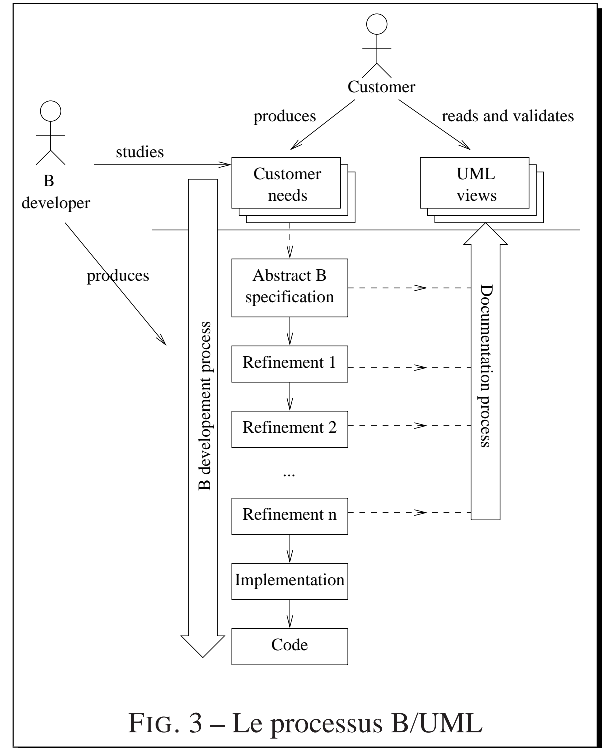


FIG. 3 – Le processus B/UML

La figure 3, ci-contre, présente le processus de documentation que nous proposons en relation avec un processus de développement en B. La méthode B, permet au moyen de raffinements successifs prouvés, de passer progressivement d'une spécification abstraite de haut niveau (éventuellement indéterministe) à une spécification concrète déterministe, automatiquement traduisible en un langage de programmation. Notre processus de documentation B/UML a pour objectif de construire des vues UML à partir des différents documents de spécification (spécification abstraite et raffinements) produits durant le développement en B, et ce, dans le but de disposer de vues graphiques pour une meilleure compréhension du système en cours de développement. En vue de présenter des spécifications B en relation avec des diagrammes UML notre investigation explore trois niveaux de mise en relation :

(*i*) Un niveau de surface dont l'objectif est de donner une syntaxe UML aux spécifications B (Idani, 2006). Pour ce faire, nous proposons une approche *interprétée* où nous identifions des correspondances entre éléments du méta-modèle de UML et un sous-ensemble de constructions en B. Ce premier niveau nous permettra de faire le lien entre les aspects structurels de systèmes spécifiés en B et les diagrammes de classes de UML.

(*ii*) Un niveau plus approfondi dans lequel nous formalisons et nous précisons les fondements théoriques de notre travail. À ce niveau, nous proposons des techniques automatisables pour la production de modèles UML lisibles à partir de spécifications B. Ce niveau prend en compte le premier niveau et est axé autour des points suivants :

- Dérivation de diagrammes de classes (Idani *et al.,* 2005b, Idani *et al.,* 2006b, Idani *et al.,* 2006d) : pour ce faire, nous proposons une démarche d'ingénierie inverse basée sur une technique de formation de concepts. Une analyse formelle de concepts est menée, prenant en compte, outre la structure de la spécification B, les dépendances entre les différents concepts B, ainsi que les dépendances entre données B et opérations B. Un réseau de concepts sera alors défini et exploité en vue de produire des modèles intermédiaires (que nous appelons

modèles de contextes) satisfaisant des critères de pertinence et traduisibles en diagrammes de classes.

- Dérivation de diagrammes d'états/transitions (Idani *et al.,* 2006a) : nous nous inspirons pour cela des travaux de preuve (Bert *et al.,* 2005) et de model-checking (Leuschel *et al.,* 2003) permettant la génération de systèmes de transitions étiquetés à partir de spécifications B. Nous proposons alors de combiner ces techniques en vue de produire des graphes abstraits à partir d'une exploration exhaustive du comportement de la spécification.

(*iii*) Un niveau exploratoire où il s'agit de :

- Mettre en relation les diagrammes structurels et comportementaux (Idani, 2006) produits par nos outils. Pour ce faire nous exploitons le cadre formel, défini par notre analyse formelle de concepts, en vue de pouvoir identifier des états abstraits qui décrivent des états de classes issues de spécifications B.
- Étendre nos techniques pour qu'elles prennent en compte les raffinements en B (Idani *et al.,* 2006d). Nous proposons, pour ce faire, un ensemble de critères pour la préservation de la pertinence des modèles de contextes résultant de chaque niveau de raffinement.
- Évaluer les diagrammes de classes pouvant être dérivés par nos techniques (Idani *et al.,* 2006c). Pour cela, nous proposons un ensemble de matrices, que nous appelons matrices de liens simples et pondérées, et qui permettent de mesurer la pertinence des modèles de contextes. Ces matrices de liens sont utilisées pour exploiter les différentes dépendances entre opérations et données B.
- Développer et expérimenter des outils prototypes permettant la mise en œuvre des différentes techniques que nous proposons dans le cadre de ce travail de thèse.

Plan de la thèse

Ce mémoire de thèse est organisé en 4 parties :

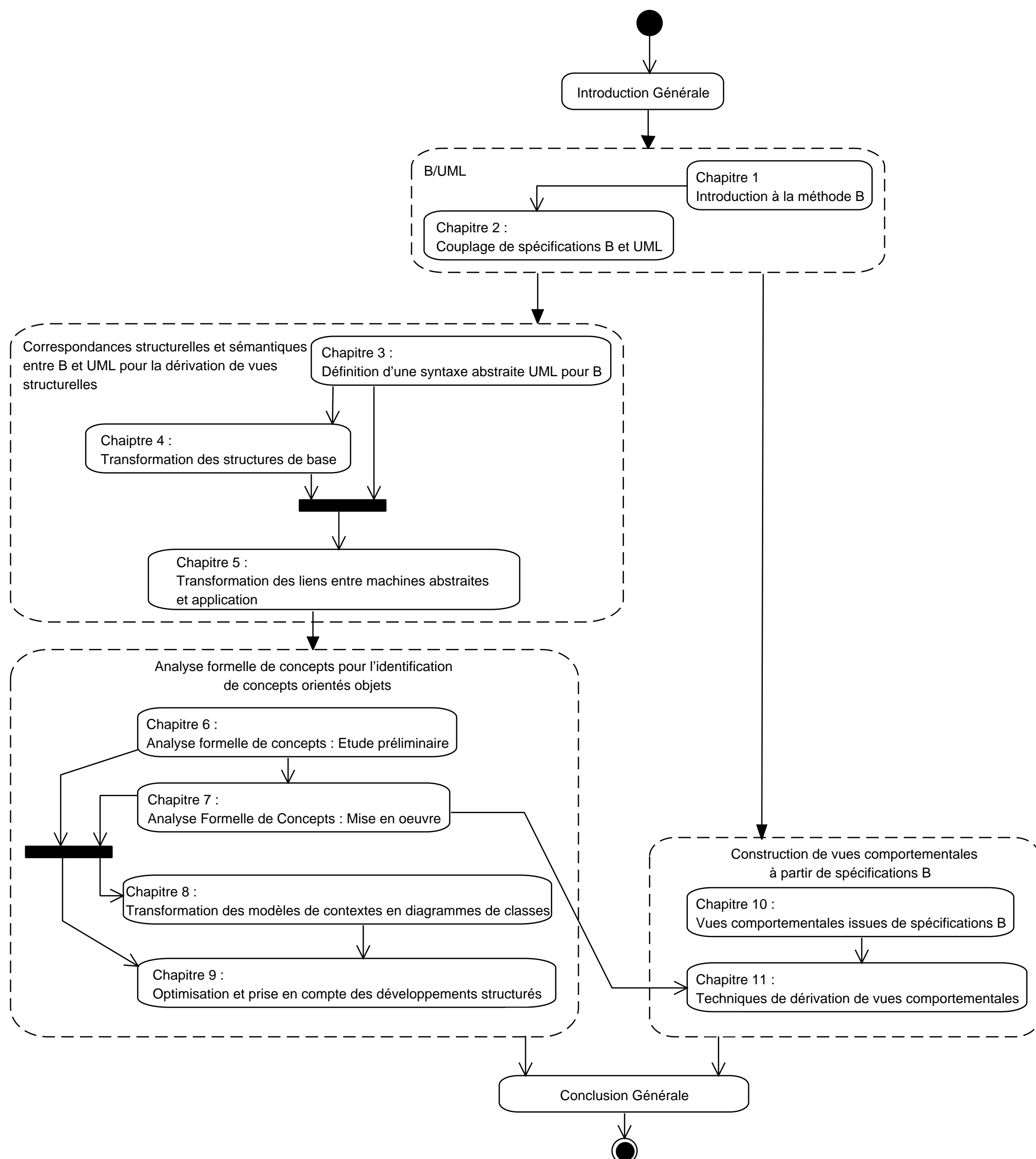
(*i*) La première partie, intitulée **« B/UML »**, est subdivisée en deux chapitres donnant respectivement un état des bases et un état de l'art sur les travaux au sein desquels s'intègre notre recherche. Le premier chapitre de cette partie, présente d'emblée les fondements conceptuels de B en rapport avec la problématique à laquelle nous nous intéressons. Dans le second chapitre de cette première partie, nous menons une étude comparative des approches de couplage de B et UML tout en soulignant, d'une part, leurs aspects complémentaires, et d'autre part, les principales limites des approches de dérivation de B vers UML.

(*ii*) La deuxième partie, intitulée **« Correspondances structurelles et sémantiques entre B et UML pour la dérivation de vues structurelles à partir de spécifications B »**, est consacrée à l'exposition de la partie *interprétée* de notre proposition pour la dérivation de diagrammes de classes à partir de spécifications B. Nous y détaillons le méta-modèle de B (chapitre 3) ainsi que les diverses correspondances avec le méta-modèle de UML (chapitres 4 et 5)

(*iii*) La troisième partie, intitulée **« Analyse formelle de concepts pour l'identification de concepts orientés objets à partir de spécifications B »**, porte sur la partie *calculée* de notre approche. Il

s'agit de présenter l'analyse formelle de concepts que nous avons menée dans le but de guider le choix des transformations à appliquer pour traduire une spécification B en un diagramme de classes (chapitres 6, 7). Cette approche nous a permis d'une part de formaliser les règles de traduction des modèles de contextes en diagrammes de classes (chapitre 8) et d'autre part, de proposer une technique d'optimisation de notre outil (chapitre 9)

(*iv*) La quatrième partie, intitulée **« Construction de vues comportementales à partir de spécifications B »**, présente un constat de vues comportementales pouvant être produites à partir d'une même spécification B (chapitre 10) ainsi que notre approche pour l'automatisation du processus de dérivation de ces vues comportementales (chapitre 11).



Publications issues de ce travail de thèse

⋆ **Revues Internationales**

1. A. Idani and Y. Ledru. « Object Oriented Concepts Identification from Formal B Specifications ». *International Journal of Formal Methods in System Design*. Extended version of FMICS'2004 paper. To appear. Springer.

2. A. Idani and Y. Ledru. « Dynamic Graphical UML Views from Formal B Specifications ». *International Journal of Information and Software Technology*. vol. 48, n° 3, 2006, pp. 154−169, Elsevier.

⋆ **Revues nationales**

3. A. Idani, Y. Ledru and D. Bert. « Analyse formelle de concepts pour la génération de diagrammes de classes UML à partir de spécifications B ». *Technique et Sciences Infomatique*, Article pré-sélectionné pour le numéro spécial *AFADL/TSI 2006*, Version étendue en cours de soumission...

⋆ **Conférences et Workshops Internationaux**

4. A. Idani, Y. Ledru and D. Bert. « A Reverse-Engineering Approach to Understanding B Specifications with UML Diagrams ». In *30th Annual IEEE/NASA Software Engineering Workshop* (SEW-30). USA. April 2006. IEEE Computer Society Press.

5. A. Idani, Y. Ledru and D. Bert. « Derivation of UML Class Diagrams as Static Views of Formal B Developments ». In *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods* (ICFEM 2005). Volume 3785 of LNCS, Springer-Verlag, pages 37-51, Manchester, 2005.

6. A. Idani and Y. Ledru. « Object Oriented Concepts Identification from Formal B Specifications ». In *Proceedings of 9th International Workshop on Formal Methods for Industrial Critical Systems* (FMICS 2004 - In conjunction with 19th IEEE Intl. Conf. on Automated Software Engineering), volume 133 of ENTCS, pages 159-174. Elsevier, 2005.

⋆ **Conférences Nationales**

7. A. Idani. « Couplage de spécifications B et de descriptions UML pour l'aide aux développements formels des systèmes d'informations : Approche par méta-modélisation ». Dans *actes du 24ème congrès INFORSID*. pages 577-594. Tunisie, 2006.

8. A. Idani, Y. Ledru and D. Bert. « Analyse formelle de concepts pour la génération de diagrammes de classes UML à partir de spécifications B ». Dans *actes de la 7ème conférence AFADL - Approches Formelles dans l'Assistance au Développement de Logiciels* (AFADL 2006), Paris, 2006.

Première partie

B / UML

Chapitre 1

Introduction à la méthode B

« *C'est encore en méditant l'objet que le sujet*
a le plus de chance de s'approfondir.
Au lieu de suivre le métaphysicien qui entre dans son poêle,
on peut donc être tenté de suivre
le mathématicien qui entre au laboratoire. »

Bachelard
« Le Nouvel esprit scientifique », 1993.

Sommaire

1.1 Introduction **14**
1.2 Fondements de la méthode B **15**
1.2.1 Machine abstraite 15
1.2.2 Typage de données en B 16
1.2.3 Langage des substitutions généralisées 17
1.2.4 Obligations de preuve 20
1.2.5 Exemple 20
1.3 Raffinement **21**
1.3.1 Fondements et illustration 21
1.3.2 Obligations de preuve du raffinement 22
1.3.3 Exemple 23
1.4 Modularité en B **24**
1.4.1 La clause INCLUDES 24
1.4.2 Les clauses USES et SEES 25
1.4.3 Les autres clauses 26
1.4.4 Aplatissement de spécifications B 26
1.5 Conclusion **28**

1.1 Introduction

Un processus de développement formel est subdivisé selon (Behm *et al.,* 1999), en deux grandes phases : (*i*) la phase de modélisation, et (*ii*) la phase de conception. Dans la première phase, un modèle abstrait est élaboré à partir de besoins informels. Ce modèle reflète une architecture de haut niveau dont la sémantique est bien précise. Les propriétés de sûreté, explicitement exprimées dans la phase d'expression des besoins, y sont formalisées. Quant à la seconde phase, elle transforme, par des étapes successives, le modèle abstrait pour le rendre concret, c'est-à-dire prêt à être traduit en code. La mise en œuvre de ces deux phases est réalisée selon deux activités fondamentales, à savoir, la spécification formelle et la vérification formelle (Mariano, 1997) :

(*i*) La spécification formelle est l'expression, au moyen d'un langage formel, d'un résultat attendu.

(*ii*) La vérification formelle est la production d'une preuve démontrant que le produit respecte effectivement la spécification formelle dont il "prétend" être la réalisation.

Partant d'une spécification abstraite de haut niveau, et par une suite de raffinements prouvés aboutissant à la production de code exécutable, la méthode B (Abrial, 1996), se présente comme une méthode formelle couvrant toutes les phases d'un cycle de développement formel. En effet, les documents formels B (Fig. 1.1) expriment différents niveaux d'abstraction allant de la phase de conception préliminaire jusqu'à la phase de codage. Le passage d'une phase à une autre dans un processus de développement en B correspond à un incrément (dit raffinement) de spécifications et est guidé par un ensemble d'obligations de preuves dont l'objectif est de valider le(s) composant(s) en cours de développement. Cette décomposition par raffinements successifs prouvés de spécifications a pour objectif de permettre un développement modulaire, sûr et fiable de systèmes.

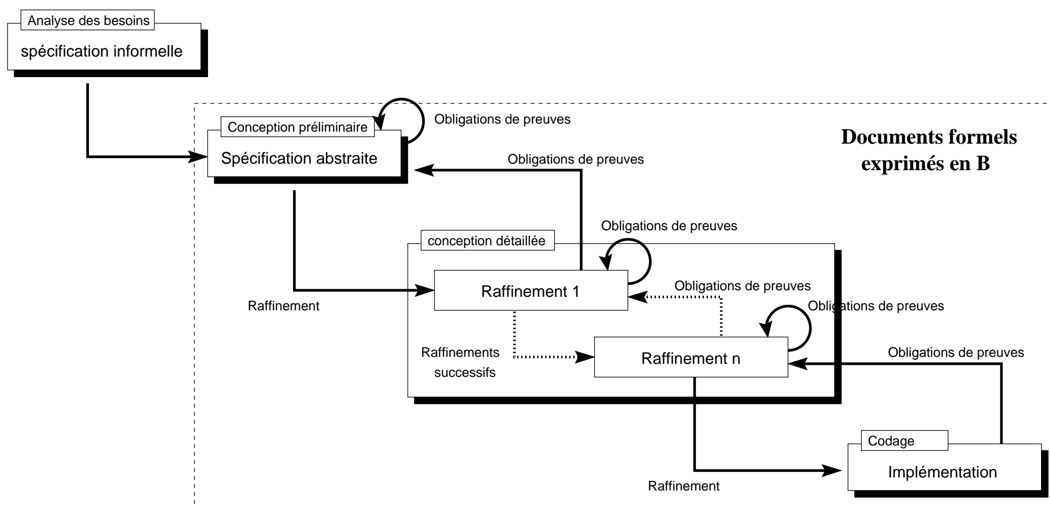


FIG. 1.1 – Processus de développement en B

Depuis quelques années, la méthode B a été utilisée avec succès dans le domaine du ferroviaire, notamment dans le cadre du métro sans conducteur METEOR (Behm *et al.,* 1999) réalisé par *Matra Transport International*. Actuellement ses applications industrielles touchent divers domaines, tels que les cartes à puce (Casset, 2002), le diagnostic automobile (Pouzancre, 2003, Pouzancre *et al.,* 2003) et

les circuits électroniques (Hallerstede, 2003). Dans le cadre du projet EDEMOI[8], qui représente l'une des principales études de cas du présent travail de thèse, l'utilisation de B est consacrée particulièrement aux phases de modélisation pour analyser les propriétés de la réglementation de sûreté dans les aéroports et mettre ainsi en évidence ses imprécisions ou biais éventuels.

Dans ce chapitre nous mettons l'accent sur les fondements conceptuels de la méthode B, afin de montrer, d'emblée, ce qui semble être primordial pour notre travail, et pouvoir entamer, ensuite, la présentation et le développement de notre apport, aussi bien théorique que pratique.

1.2 Fondements de la méthode B

La méthode B fait partie des méthodes de spécification formelle orientées modèle. Elle est fondée sur un langage de spécification relevant de la théorie des ensembles et où l'état du système est modélisé par des types abstraits de données pré-définis (en particulier ensemblistes). Elle permet une modélisation des aspects statiques et dynamiques du logiciel. L'aspect statique concerne les données manipulées et est caractérisé par un ensemble de variables et de constantes (que nous appellerons *données* B) ayant une valeur typée (entier, booléen, . . .). L'état du système spécifié est décrit par l'ensemble des couples (donnée, valeur). Quant à l'aspect dynamique, il est exprimé par un ensemble d'opérations décrivant des changements (ou *transitions*) d'états.

1.2.1 Machine abstraite

J.-R Abrial (Abrial, 1996) compare la notion de machine abstraite en B à une boîte noire ayant une mémoire et un certain nombre de boutons. Les valeurs stockées dans cette mémoire représentent l'état de la machine et les boutons sont les opérations que l'utilisateur de la machine peut activer en vue de modifier les valeurs stockées en mémoire et donc modifier l'état de la machine. Les opérations constituent donc la seule interface de la boîte noire pour l'environnement extérieur, tant en entrée qu'en sortie.

Structure d'une machine abstraite

MACHINE
- PARTIE ENTÊTE
  - Nom de la machine
  - Paramètres de la machine
  - Contraintes sur les paramètres
- PARTIE STATIQUE (DÉCLARATIVE)
  - Déclarations d'ensembles
  - Déclarations de constantes
  - Propriétés des constantes
  - Déclarations des variables (*état*)
  - Invariant (*caractérisation de l'état*)
- PARTIE DYNAMIQUE (OPÉRATIONNELLE)
  - Initialisation des variables
  - Opérations

END

[8] www-lsr.imag.fr/EDEMOI/

Cette notion de machine abstraite est une notion fondamentale dans un développement en B. Elle correspond à l'élément de structuration de base des spécifications et est composée de trois parties : (*i*) la partie entête, (*ii*) la partie statique et (*iii*) la partie dynamique.

La partie entête :
Permet l'identification de la machine abstraite. Elle contient la clause MACHINE, décrivant le nom de la machine suivi éventuellement de paramètres, ainsi que la clause CONSTRAINTS où il s'agit de caractériser ces paramètres.

La partie statique :
Regroupe les déclarations d'ensembles (clause SETS), de constantes (clause CONSTANTS) et de variables (clause VARIABLES). Ces déclarations sont complétées par un ensemble de prédicats décrivant les propriétés des constantes (clause PROPERTIES) ainsi que les invariants (clause INVARIANT) qui explicitent précisément les propriétés qui doivent être toujours satisfaites par l'état de la machine. Les données définies au niveau de ces clauses sont spécifiées en utilisant des formules de la logique du premier ordre et des concepts mathématiques de la théorie des ensembles.

La partie dynamique :
Décrit l'évolution de l'état de la machine. Ceci comprend son initialisation et les opérations offertes par la machine. Les transitions entre états peuvent s'effectuer de manière largement non-déterministe. Pour ce faire, le langage défini est celui des substitutions généralisées qui est une notation spécifique à B. En nous référant à d'autres techniques de spécification formelle orientée modèle telles que Z et VDM, la partie opérationnelle est décrite par des prédicats avant/après spécifiant les états avant et après l'activation d'une opération. En B, nous retrouvons ces notions ; en effet, les substitutions généralisées en B peuvent être réécrites en termes de prédicats avant/après par simple calcul.

1.2.2 Typage de données en B

Le typage en B est un mécanisme de vérification statique des données (ClearSy, 2002b). Le type d'une entité e est le plus grand ensemble (parmi les ensembles définis dans le langage B) auquel appartient l'entité e. En B, ce mécanisme s'effectue en utilisant des prédicats ou des substitutions particulières désignés par prédicats de typage ou substitutions de typage. Dans notre travail nous allons nous intéresser plutôt aux prédicats de typage[9] (l'appartenance ($\in$), l'inclusion ($\subset, \subseteq$) et l'égalité ($=$)). Ces prédicats permettent d'associer explicitement un type à une donnée B, par exemple, le prédicat $d \in NAT$ spécifie que la donnée d est de type entier.

Il existe deux catégories de types en B (ClearSy, 2002b) : les types de base et les types construits à l'aide d'un constructeur de type :

(i) Les types de base :
- L'ensemble des entiers ($\mathbb{Z}$) à partir duquel d'autres types sont définis : $\mathbb{N}$ (désignant les entiers positifs), $\mathbb{N}_1$ (désignant les entiers strictement positifs), NAT (l'ensemble des entiers positifs représentables (ou concrets))...,
- L'ensemble des booléens (BOOL = {TRUE, FALSE}, avec TRUE $\neq$ FALSE),
- L'ensemble des chaînes de caractères (STRING), et

[9] La notion de substitution de typage est précisée dans le manuel de référence de B (ClearSy, 2002b)

– Les ensembles abstraits et énumérés définis au niveau de la clause SETS.

(*ii*) Les constructeurs de type :

– L'ensemble des parties finies d'un ensemble ($\mathbb{P}$). Par exemple, $\mathbb{P}(E)$ définit l'ensemble dont les éléments sont des ensembles d'éléments de E.

$$T \in \mathbb{P}(E) \Leftrightarrow \forall t \cdot (t \in T \Rightarrow t \in E)$$

– Le produit cartésien entre deux ensembles ($\times$). Par exemple, $E_1 \times E_2$ définit l'ensemble des paires ordonnées dont le premier élément est de type E_1 et le second est de type E_2.

$$(x \mapsto y) \in E_1 \times E_2 \Leftrightarrow x \in E_1 \wedge y \in E_2$$

Cette notion de type est une notion fondamentale dans notre travail, en effet, la traduction d'une donnée B en un élément d'un diagramme de classes ou d'un diagramme d'états/transitions sera guidée, entre autres, par le type associé à cette donnée.

1.2.3 Langage des substitutions généralisées

Les opérations d'une machine abstraite représentent un ensemble de services permettant d'initialiser puis de faire évoluer les données de la machine (ou état de la machine). Le langage des substitutions généralisées est défini pour exprimer ce mécanisme d'évolution de données. À ce niveau, nous n'allons pas citer les différents concepts relevant de la logique des prédicats mais plutôt introduire la notion de substitution[10] généralisée utilisée en B et qui est, en fait, un transformateur de prédicats[11]. La sémantique des substitutions généralisées en B est précisée par le calcul de la plus faible pré-condition (ou $\mathcal{W}p$) défini par Dijkstra en 1975 (Dijkstra, 1975) et inspiré de la logique de Hoare (Hoare, 1969). Dans ce qui suit, la formule $[S]R$ est équivalente au calcul de la plus faible pré-condition qui garantit que l'instruction S se termine et que l'assertion R est vraie après la terminaison de S. Par exemple, dans le cadre de l'instruction particulière d'affectation, la formule $[x := e]R$ dénote la plus faible pré-condition qui garantit que le prédicat R est vrai après la substitution uniforme de la variable x par l'expression e. En d'autres termes, $[x := e]R$ dénote la formule obtenue après remplacement de toutes les occurrences libres de x dans R par e. Ceci sera noté : $R[e/x]$ (voir le B-Book (Abrial, 1996), § 1.3.3).

Substitutions primitives. Les substitutions de base en B sont dites les substitutions primitives. Ce sont des substitutions généralisées à partir desquelles les autres sont définies (voir Tab. 1.1) :

– La substitution pré-conditionnée : fixe les pré-conditions sous lesquelles une opération est appelée. En effet, si la pré-condition P est fausse, alors le prédicat $P \wedge [S]R$ est faux et la substitution S peut ne jamais se terminer ;

– La substitution choix borné : permet de définir un nombre fini (ou borné) de comportements possibles sans préciser lequel sera effectivement implanté ;

– La substitution gardée : définit différents comportements possibles en fonction de la validité de gardes. Une substitution gardée spécifie une opération qui est activée sous une hypothèse correspondant à la garde ;

– La substitution choix non-borné : permet d'utiliser dans la substitution S la donnée abstraite z. Cette substitution définit un comportement non-déterministe dépendant du choix de z. On peut alors dire que la substitution choix non-borné est une généralisation de la substitution choix borné.

[10] Une substitution est tout simplement le remplacement d'une variable par une valeur qu'elle peut prendre.

[11] Cette notion de transformation de prédicats a été introduite par Dijkstra dans (Dijkstra, 1976).

Nom de la substitution	Notation mathématique	Notation verbeuse	Sémantique (calcul du $\mathcal{W}p$)
simple	$x := e$	$x := e$	$[x := e]R \Leftrightarrow R[e/x]$
multiple simple	$x, y := e, f$	$x, y := e, f$	$[x, y := e, f]R \Leftrightarrow [z := f][x := e][y := z]R$
séquencée	$S;T$	$S;T$	$[S;T]R \Leftrightarrow [S][T]R$
sans effet	$skip$	$skip$	$[skip]R \Leftrightarrow R$
pré-conditionnée	$P \vert S$	PRE P THEN S END	$[P\vert S]R \Leftrightarrow P \wedge [S]R$
choix borné	$T[]S$	CHOICE T OR S END	$[T[]S]R \Leftrightarrow [T]R \wedge [S]R$
gardée	$P \Longrightarrow S$	SELECT P THEN S END	$[P \Longrightarrow S]R \Leftrightarrow (P \Rightarrow [S]R)$
choix non borné	$@z \cdot S$	VAR z IN S END	$[@z \cdot S]R \Leftrightarrow \forall z \cdot [S]R$
Avec : x et y des variables ; e et f des expressions ; S et T des substitutions ; P et R des prédicats. et où z est non-libre dans R.			

TAB. 1.1 – Substitutions primitives

Substitutions dérivées. Les substitutions dérivées sont construites à partir des substitutions primitives précédentes. Le tableau 1.2 présente celles qui, à notre sens, sont les plus utilisées.

Notation verbeuse	Notation mathématique
IF P THEN S ELSE T END	$P \Longrightarrow S[]\neg P \Longrightarrow T$
CHOICE S OR T OR ... OR U END	$S \;[]\; T \;[]\; \ldots \;[]\; U$
ANY z WHERE P THEN S END	$@z \cdot (P \Longrightarrow S)$
$x :\in E$	$@x' \cdot (x' \in E \Longrightarrow x := x')$
$x :\vert\ P$	$@x' \cdot ([x\$0, x := x, x']P \Longrightarrow x := x')$
ASSERT P THEN S END	$P \vert (P \Longrightarrow S)$
Avec : x une variable ; E un ensemble ; S, T et U des substitutions ; P un prédicat.	

TAB. 1.2 – Substitutions dérivées

La notation $x\$0$ figurant au niveau de la substitution “devient tel que” ($x :\vert\ P$) permet de caractériser la nouvelle valeur de x par rapport à sa valeur avant la substitution. Par exemple, la substitution suivante : $x :\vert\ x > x\$0$, signifie que la nouvelle valeur de x doit être strictement supérieure à son ancienne valeur.

Il existe en B une notation pour réaliser simultanément deux (ou plusieurs) substitutions : c’est la substitution multiple généralisée, notée $S \;\vert\vert\; T$ et n’est définie que si les variables modifiées par S et celles modifiées par T sont disjointes. Il existe également des variations sur la substitution de choix borné, comme par exemple les deux suivantes :

Notation	Définition

SELECT P THEN
    S
WHEN Q THEN
    T
...
WHEN R THEN
    U
END
	CHOICE
	    $P \Longrightarrow S$
	OR
	    $Q \Longrightarrow T$
	...
	OR
	    $R \Longrightarrow U$
	END

Notation	Définition

CASE E OF
  EITHER l THEN
    S
  OR p THEN
    T
  ...
  OR q THEN
    U
  END
END
	SELECT $E \in \{l\}$ THEN
	    S
	WHEN $E \in \{p\}$ THEN
	    T
	...
	WHEN $E \in \{q\}$ THEN
	    U
	END

Manipulation des substitutions généralisées. Une substitution peut être caractérisée par sa terminaison, sa faisabilité et l'état avant et après son exécution. Pour ce faire, le langage B définit le prédicat de terminaison, le prédicat de faisabilité et le prédicat avant/après pour une substitution $\mathcal{S}$ comme suit :

– **Prédicat de terminaison** $-\ \mathbf{trm}(\mathcal{S})$
Ce prédicat permet d'indiquer si l'exécution d'une substitution permet toujours d'obtenir un résultat. Le prédicat $trm(\mathcal{S})$ est défini par : $trm(\mathcal{S}) \mathrel{\widehat{=}} [\mathcal{S}]btrue$ ou encore $trm(\mathcal{S}) \mathrel{\widehat{=}} [\mathcal{S}](x = x)$. La terminaison des substitutions généralisées énoncées précédemment se déduit de cette définition. Par exemple, la terminaison de la substitution choix borné est calculée comme suit :

$$\begin{array}{lcl} trm(\mathcal{S} \, [] \, \mathcal{U}) & \Leftrightarrow & [\mathcal{S} \, [] \, \mathcal{U}](btrue) \\ & \Leftrightarrow & [\mathcal{S}](btrue) \wedge [\mathcal{U}](btrue) \\ & \Leftrightarrow & trm(\mathcal{S}) \wedge trm(\mathcal{U}) \end{array}$$

– **Prédicat de faisabilité** $-\mathbf{fis}(\mathcal{S})$
Ce prédicat caractérise l'espace d'états pour lequel il existe un résultat. Il est défini comme suit : $fis(\mathcal{S}) \mathrel{\widehat{=}} \neg[\mathcal{S}](bfalse)$. Par exemple, la faisabilité de la substitution choix borné revient à :

$$\begin{array}{lcl} fis(\mathcal{S} \, [] \, \mathcal{U}) & \Leftrightarrow & \neg[\mathcal{S} \, [] \, \mathcal{U}](bfalse) \\ & \Leftrightarrow & \neg([\mathcal{S}](bfalse) \wedge [\mathcal{U}](bfalse)) \\ & \Leftrightarrow & fis(\mathcal{S}) \vee fis(\mathcal{U}) \end{array}$$

Ce qui correspond à l'union des espaces d'états assurant la faisabilité des substitutions $\mathcal{S}$ et $\mathcal{U}$.

– **Prédicat avant/après** $-\mathbf{prd_x}(\mathcal{S})$
Ce prédicat caractérise toutes les valeurs possibles des variables d'état après l'exécution d'une substitution. La valeur avant d'une variable est notée x et la valeur après est notée x'. Le prédicat $prd_x(\mathcal{S})$ est défini par : $prd_x(\mathcal{S}) \mathrel{\widehat{=}} \neg[\mathcal{S}](x' \neq x)$. Par exemple, la valeur après l'exécution d'une substitution simple $x := E$ est calculée comme suit :

$$\begin{array}{lcl} prd_x(x := E) & \Leftrightarrow & \neg[x := E](x' \neq x) \\ & \Leftrightarrow & \neg(x' \neq E) \\ & \Leftrightarrow & x' = E \end{array}$$

Ces prédicats caractéristiques des substitutions généralisées permettent d'établir le lien avec d'autres langages de spécification formelle basés sur la logique (Z, VDM, TLA, etc). En effet, deux propriétés sont déduites de ces prédicats et sont prouvées dans (Abrial, 1996) :

1. Une substitution $\mathcal{S}$ portant sur une variable x, est faisable si et seulement si une valeur après x' peut être calculée à partir de la valeur avant x. Ce qui revient à définir la faisabilité en termes de prédicats avant/après : $fis(\mathcal{S}) \Leftrightarrow \exists x' \cdot (prd_x(\mathcal{S}))$
2. Pour toute substitution généralisée $\mathcal{S}$, une forme normale existe :

$$\mathcal{S} = trm(\mathcal{S}) \mid @x' \cdot (prd_x(\mathcal{S}) \Longrightarrow x := x')$$

Où x est l'ensemble des variables de la machine et x' représente l'ensemble des valeurs de x après l'exécution de $\mathcal{S}$.

1.2.4 Obligations de preuve

Définition. « Une obligation de preuve est une formule mathématique à démontrer afin d'assurer qu'un composant B[12] est correct. La théorie *B* indique quelles sont les obligations de preuve à démontrer pour assurer la correction d'un composant *B* donné. Dans cette optique, les obligations de preuve sont une aide au processus de vérification ».

Les obligations de preuve sont générées à partir du calcul de la plus faible pré-condition. Elles représentent des conditions de validité permettant de vérifier la correction d'un développement B (à ce niveau nous parlons de correction d'une machine abstraite). Il s'agit de vérifier que l'invariant est respecté aussi bien par l'initialisation que par les opérations de la machine.

Obligations de preuves de l'initialisation. Soit $\mathcal{I}$ l'invariant (clause INVARIANT), $\mathcal{R}$ les propriétés (clause PROPERTIES) et $\mathcal{C}$ les contraintes (clause CONSTRAINTS) de la spécification, et soit $\mathcal{S}$ la substitution définissant l'initialisation de la machine alors l'initialisation est dite correcte si et seulement si :

$$\boxed{\mathcal{R} \wedge \mathcal{C} \Rightarrow [\mathcal{S}]\mathcal{I}}$$

En d'autres termes, il s'agit de démontrer que pour n'importe quel état s satisfaisant les propriétés et des contraintes de la machine, l'exécution de la substitution d'initialisation $\mathcal{S}$ à partir de s aboutit à un état qui satisfait l'invariant $\mathcal{I}$.

Obligations de preuve des opérations. Les obligations de preuve des opérations ont pour objectif de vérifier que l'invariant de la machine est vrai après application de la substitution de l'opération sachant que l'invariant était vrai avant l'appel de l'opération. Étant donné que la forme générale d'une opération en B est : $r \longleftarrow nom_op(p) =$ PRE $\mathcal{P}$ THEN $\mathcal{K}$ END ; alors l'obligation de preuve associée revient à vérifier que le prédicat suivant est satisfait :

$$\boxed{\mathcal{I} \wedge \mathcal{R} \wedge \mathcal{C} \wedge \mathcal{P} \Rightarrow [\mathcal{K}]\mathcal{I}}$$

1.2.5 Exemple

La spécification B ci-dessus gère une réservation de places[13]. Le nombre de places maximal est défini par la constante Max (entier naturel positif strictement supérieur à 10). L'état est constitué d'une variable $places$ qui indique le nombre de places disponibles. Cette variable est typée par l'ensemble $0..Max$.

L'opération $reserver$ permet de réserver une place en décrémentant le nombre de places libres, et ce, contrairement à l'opération $annuler$ qui permet de libérer une place et donc d'incrémenter le nombre de places libres. Ces deux opérations correspondent à des substitutions pré-conditionnées. En effet, l'opération de réservation n'est possible que s'il existe encore des places disponibles ($places > 0$). Aussi, l'opération d'annulation n'est possible que si au moins une place est occupée ($places < Max$).

[12] Nous évoquerons plus tard la notion de composants en B. À ce niveau, une machine abstraite est un composant B.

[13] Cette spécification est inspirée de (Bert *et al.,* 2004)

Exemple 1.1

```
MACHINE
   RESERVATION
CONSTANTS
   Max
PROPERTIES
   Max ∈ NAT ∧ Max > 10
VARIABLES
   places
INVARIANT
   places ∈ 0 .. Max
INITIALISATION
   places := Max
OPERATIONS
   reserver =
         PRE places > 0 THEN
            places := places - 1
         END;
   annuler =
         PRE places < Max THEN
            places := places + 1
         END
END
```

1.3 Raffinement

Le raffinement en B est une technique de développement incrémental permettant de préciser progressivement les données et les opérations de la spécification abstraite de départ. L'objectif de cette technique est de passer progressivement d'une spécification non-déterministe de haut niveau à une spécification concrète déterministe, automatiquement traduisible dans un langage de programmation.

1.3.1 Fondements et illustration

Le raffinement d'une machine abstraite n'est autre qu'un composant qui conserve la même interface et le même comportement que la machine abstraite mais qui permet, d'une part de reformuler la machine en une expression de plus en plus concrète, et d'autre part, de l'enrichir en y introduisant de nouvelles données et invariants. Deux types de raffinements sont établis en B :

– le raffinement de données : où les données ou variables abstraites définies dans la clause VARIABLES peuvent être conservées, disparaître ou changer de forme. Aussi, le raffinement de données peut consister en l'introduction de nouvelles variables. Dans ce cas, un invariant, dit de liaison (ou de collage), est défini en vue de spécifier la relation entre données abstraites et données concrètes (celles du raffinement).

– le raffinement des opérations : chaque opération raffinée doit réaliser ce qui est spécifié dans l'abstraction, à l'aide des données du raffinement et de substitutions plus concrètes et plus déterministes.

La Fig. 1.2, inspirée de (Lano, 1996), illustre ces propos. Il s'agit d'une opération $O_{\mathcal{N}}$ qui raffine une opération possiblement non-déterministe $O_{\mathcal{M}}$ avec une relation de raffinement R définie par l'invariant de liaison. u représente l'état abstrait avant l'exécution de l'opération abstraite $O_{\mathcal{M}}$, alors que u'_1 et u'_2 représentent les états abstraits après l'exécution de cette même opération. v et v' représentent

respectivement les états concrets avant et après l'exécution de l'opération concrète $O_{\mathcal{N}}$. Le raffinement des opérations garantit que toute exécution de l'opération concrète $O_{\mathcal{N}}$ à partir d'un état v lié à un état abstrait u par une relation de raffinement (il s'agit ici de R) se termine dans un état v' lié par la même relation de raffinement à un état u' (il s'agit ici de u'_1) qui peut être atteint par une exécution de l'opération abstraite $O_{\mathcal{M}}$ à partir de u.

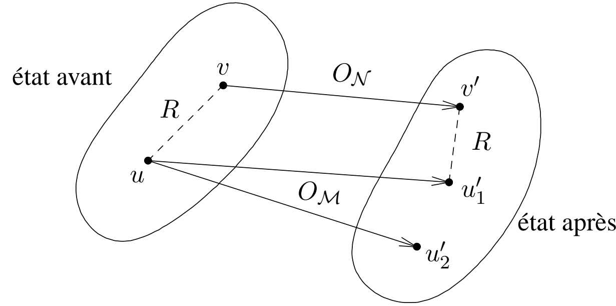


FIG. 1.2 – Représentation du raffinement des opérations

Plusieurs niveaux de raffinement peuvent ainsi être utilisés afin de reformuler progressivement, par étapes successives, la machine abstraite. Dans le cadre du projet EDEMOI, le raffinement est vu comme un précepte de construction de spécifications par ajout de détails, sans nécessairement se préoccuper de la production de code.

1.3.2 Obligations de preuve du raffinement

Comme pour les obligations de preuve d'une machine abstraite (§ 1.2.4, page 20) nous allons évoquer les obligations de preuve de l'initialisation et celles des opérations du raffinement. Soit $\mathcal{I}$ l'invariant du composant raffiné (ou abstrait), $\mathcal{R}$ ses propriétés et $\mathcal{C}$ ses contraintes, et soit $\mathcal{S}$ la substitution définissant son initialisation.

Obligations de preuve de l'initialisation. Nous désignons par $\mathcal{J}$ l'invariant du raffinement, et par $\mathcal{R}_r$ les propriétés (clause PROPERTIES) du raffinement. Soit $\mathcal{S}_r$ la substitution définissant l'initialisation du raffinement alors l'obligation de preuve de l'initialisation du raffinement revient à vérifier que le prédicat suivant[14] est satisfait :

$$\boxed{\mathcal{R} \wedge \mathcal{C} \wedge \mathcal{R}_r \Rightarrow [\mathcal{S}_r]\neg[\mathcal{S}]\neg\mathcal{J}}$$

Il s'agit, de vérifier que l'initialisation du raffinement $\mathcal{S}_r$ établit l'invariant du raffinement $\mathcal{J}$ sans contredire l'initialisation $\mathcal{S}$ du composant raffiné, et ce, sous l'hypothèse formée par la conjonction des prédicats $\mathcal{R}$, $\mathcal{C}$ et $\mathcal{R}_r$.

Obligations de preuve des opérations.

– Preuve de la pré-condition : soit $\mathcal{P}$ la pré-condition de l'opération abstraite et $\mathcal{Q}$ la pré-condition de l'opération concrète alors la preuve de la pré-condition de l'opération concrète se ramène à :

[14] $\neg[\mathcal{S}]\neg\mathcal{J}$ signifie qu'il existe une exécution de la substitution $\mathcal{S}$ qui établit le prédicat $\mathcal{J}$ (Lano, 1996).

$$\mathcal{R} \wedge \mathcal{C} \wedge \mathcal{R}_r \wedge \mathcal{I} \wedge \mathcal{J} \wedge \mathcal{P} \Rightarrow \mathcal{Q}$$

– Preuve de l'opération : soit $\mathcal{L}$ la substitution de l'opération concrète et $\mathcal{K}$ celle de l'opération abstraite alors l'obligation de preuve de l'opération est la suivante :

$$\mathcal{R} \wedge \mathcal{C} \wedge \mathcal{R}_r \wedge \mathcal{I} \wedge \mathcal{J} \wedge \mathcal{P} \Rightarrow [\mathcal{L}]\neg[\mathcal{K}]\neg\mathcal{J}$$

1.3.3 Exemple

Soit, par exemple, le raffinement RESERVATION_REF ci-dessous, de la machine abstraite RESERVATION (exemple 1.1, page 21). Dans ce raffinement nous introduisons l'ensemble $Sieges$ désignant l'ensemble des numéros de sièges, ainsi que la variable *occupes* représentant l'ensemble des numéros des sièges occupés. Le nombre de places libres défini par la variable $places$ de la machine abstraite RESERVATION est égal à $Max - \mathbf{card}(occupes)$. Cet invariant représente l'invariant de liaison[15] entre la machine RESERVATION et son raffinement RESERVATION_REF.

L'opération $reserver$, chargée de décrémenter le nombre de places libres, permet, d'une manière non-déterministe, d'introduire un numéro de siège libre (appartenant à $Sieges - occupes$) dans l'ensemble des numéros de sièges occupés. L'opération $annuler$, initialement chargée d'incrémenter le nombre de places libres, permet de soustraire un numéro de siège de l'ensemble des numéros de sièges occupés.

Exemple 1.2

```
REFINEMENT
    RESERVATION_REF
REFINES
    RESERVATION
CONSTANTS
    Sieges
PROPERTIES
    Sieges = 1 .. Max
VARIABLES
    occupes
INVARIANT
    occupes ⊆ Sieges ∧
    places = Max - card(occupes)
INITIALISATION
    occupes := ∅
OPERATIONS
    reserver =
        ANY pp WHERE pp ∈ Sieges - occupes THEN
            occupes := occupes ∪ {pp}
        END;
    annuler =
        ANY pp WHERE pp ∈ occupes THEN
            occupes := occupes - {pp}
        END
END
```

[15] Dit aussi invariant de collage

L'obligation de preuve à ce niveau, garantit que les opérations concrètes $reserver$ et $annuler$ préservent le comportement de la spécification abstraite, et ce, sous l'invariant de liaison $places = Max - \mathbf{card}(occupes)$ qui permet de lier l'état concret à l'état abstrait.

1.4 Modularité en B

En génie logiciel, la modularité est perçue comme étant une solution efficace pour la maîtrise de la complexité d'un système (Baldwin *et al.,* 2000). En effet, la décomposition en modules (ou composants) permet de disposer de divers sous-systèmes plus faciles à appréhender. En revanche, des questions d'intégration, de cohérence et de dépendance entre composants deviennent inévitables. Pour ce faire, la méthode B fournit un mécanisme d'assemblage (dit aussi de liaison) qui permet de structurer les spécifications en plusieurs modules interdépendants qu'on appellera composants B. Dans ce contexte, M.-L. Potet (Potet, 2002) distingue deux sortes de composition en B :

(i) La composition verticale : c'est le mécanisme qui permet de considérer différents niveaux d'abstraction d'un composant. Il s'agit là précisément du raffinement présenté précédemment et qui est défini par la clause REFINES et l'invariant de liaison. Notons que dans le cadre de ce mécanisme d'assemblage particulier, nous parlerons de composant concret (ou raffinant) et de composant abstrait (ou raffiné).

(ii) La composition horizontale : c'est le mécanisme qui permet de construire un composant en le réduisant à des composants plus petits. En B, ce mécanisme est réalisé via des règles de visibilité entre machines B et est défini par les clauses : INCLUDES, SEES, USES, EXTENDS, PROMOTES et IMPORTS. Nous distinguons deux modes de visibilité selon l'accès aux variables de l'extérieur :
- Mode *ouvert* : les variables sont visibles de l'extérieur sans contrainte, (*i.e.* clauses INCLUDES, IMPORTS). Ces variables sont visibles au niveau des prédicats et accessibles à travers les opérations de la machine incluse au niveau des substitutions.
- Mode *semi-ouvert* : les variables ne sont visibles de l'extérieur qu'en lecture (*i.e.* clauses USES et SEES),

Les liens PROMOTES et EXTENDS servent à intégrer une ou plusieurs opérations d'un composant M dans un autre composant M'. Ceci permet de considérer les opérations intégrées comme faisant partie de la spécification M'. Quant au lien IMPORTS, il est utilisé uniquement pour indiquer le lien entre une implantation et une (ou plusieurs) machine(s) abstraite(s).

Dans notre démarche nous allons, dans un premier lieu, traiter des composants B séparément, ensuite, nous nous baserons sur les différentes clauses d'assemblage du langage B pour mettre en relation les vues UML que nous produisons pour chaque composant B. Dans notre travail nous n'allons pas nous intéresser aux mécanismes d'assemblage définis par les clauses : EXTENDS, PROMOTES et IMPORTS. Nous pensons que leur intégration représente une continuation intéressante du présent travail de thèse. La démarche de construction de vues UML que nous proposons au niveau du présent document porte donc principalement sur les clauses REFINES, INCLUDES, SEES et USES.

1.4.1 La clause INCLUDES

Le mécanisme d'inclusion de machines abstraites établi par la clause INCLUDES permet de rendre visibles, dans la machine incluante, les données et les opérations du composant inclus. Soient, par exemple,

deux machines abstraites M et N telles que la machine N inclut la machine M (figure 1.3) alors les règles de visibilité de l'inclusion sont les suivantes :

– les paramètres de M doivent être instanciés et donc ne sont plus visibles dans N,
– les variables de M peuvent être utilisées au niveau des prédicats et ne sont accessibles au niveau des substitutions qu'à travers les opérations de M,
– les propriétés de N peuvent faire référence à n'importe quel ensemble ou constante de M,
– les invariants de N peuvent être exprimés en fonction des ensembles, constantes et variables définies dans M.

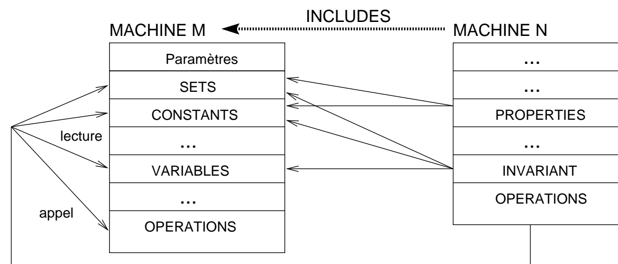


FIG. 1.3 – Visibilité entre composants B établie par la clause INCLUDES

Il est à noter que l'inclusion en B est transitive. En effet, l'inclusion de plusieurs composants dans un composant M est automatiquement reproduite dans un autre composant N si N inclut M.

1.4.2 Les clauses USES et SEES

La clause USES est utilisée pour consulter les informations d'autres composants (figure 1.4). En effet, si un composant N utilise un autre composant M alors il peut utiliser toutes ses données en lecture uniquement (ou consultation). Les opérations de N n'utilisent donc aucune donnée du composant M et ne peuvent faire appel à aucune de ses opérations. Il existe en B une autre clause de consultation : la clause SEES qui, contrairement à la clauses USES, ne permet de consulter ni les paramètres, ni les variables du composant référencé. Par ailleurs, au niveau de la clause SEES l'appel d'opérations ne peut s'effectuer que dans le cas où les opérations appelées ne modifient pas les variables de leur machine abstraite. Il est à noter que ces 2 mécanismes ne sont pas transitifs.

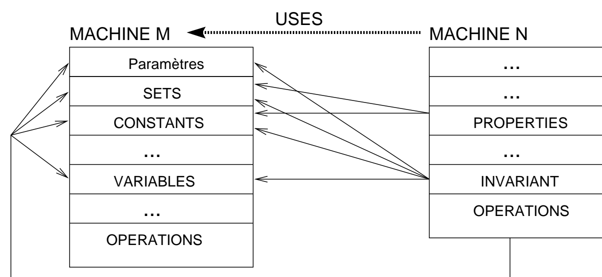


FIG. 1.4 – Visibilité entre composants B établie par la clause USES

1.4.3 Les autres clauses

Clause PROMOTES. Indique la liste des opérations *promues* des instances de machines incluses. Promouvoir une opération o d'une instance de machine M dans un composant M' équivaut à définir dans M' une opération dont le nom, le corps et la signature sont ceux de o (éventuellement précédé du préfixe de renommage de M si M est renommée).

Clause EXTENDS. Indique la liste des instances de machines *étendues*. Le principe de ce mécanisme de composition est similaire au principe de la clause PROMOTES sauf qu'il s'agit de promouvoir toutes les opérations de toutes les machines incluses.

Clause IMPORTS. Il s'agit d'un lien entre une implantation est une ou plusieurs instances de machines abstraites. Ce lien permet de réaliser l'implantation du composant concerné sur la base des données et opérations d'instances de machines importées.

1.4.4 Aplatissement de spécifications B

Lors de la construction des diagrammes de classes à partir de spécifications B, la prise en compte des compositions et raffinements est basée sur une étape préliminaire d'aplatissement ou de dépliage (ces deux termes seront utilisés indifféremment) de spécifications. Il s'agit, de dériver, à partir d'un ensemble de composants B, un composant unique équivalent à l'ensemble des composants aplatis. Cette technique a été présentée implicitement dans le B-Book (Abrial, 1996), et a été développée au niveau des thèses de S. Behnia (Behnia, 2000) et D. Petit (Petit, 2003). Nous en présentons un bref aperçu à ce niveau.

Aplatissement du lien de raffinement REFINES. En B, un raffinement hérite de certaines déclarations du niveau abstrait précédent telles que, par exemple, la définition des ensembles, des constantes concrètes[16] et des variables. L'aplatissement du raffinement collecte les informations (déclarations, type, ...) concernant les ensembles, les constantes concrètes et les variables et les insère au niveau concret pour être analysées. En vue de présenter l'aplatissement du lien de raffinement (Fig. 1.5) nous nous basons principalement sur la présentation faite dans le B-Book (Abrial, 1996) (Chapitre 11, page 524).

Soient $\mathcal{M}$ et $\mathcal{N}$ un composant abstrait et son raffinement. La phase d'aplatissement produit une machine $\mathcal{N}'$ en effectuant une combinaison syntaxique de $\mathcal{M}$ et $\mathcal{N}$. Il s'agit, dans un premier temps, d'insérer, au niveau du composant raffinant, les ensembles ainsi que les constantes concrètes du composant abstrait. Les propriétés sur les constantes ainsi formées sont définies par conjonction des propriétés concrètes suivies des propriétés abstraites ($\mathcal{P}_\mathcal{N} \wedge \mathcal{P}_\mathcal{M}$). Les variables de $\mathcal{N}'$, désignées par $\mathcal{V}'_\mathcal{N}$, correspondent aux variables du raffinement $\mathcal{V}_\mathcal{N}$. Ces dernières sont renommées dans le cas où elles sont homonymes aux variables du composant raffiné $\mathcal{V}_\mathcal{M}$. Ce renommage est également pris en compte au niveau de l'invariant $\mathcal{I}'_\mathcal{N}$ et l'initialisation $Init'_\mathcal{N}$. Étant donné que les variables $\mathcal{V}_\mathcal{M}$ ne sont pas nécessairement redéfinies au niveau de $\mathcal{N}'$ alors qu'elles sont susceptibles d'être utilisées au niveau de $\mathcal{N}$, l'invariant de $\mathcal{N}'$ les intègre tout en conservant les obligations de preuves associées au raffinement : $\exists \mathcal{V}_\mathcal{M} \cdot (\mathcal{I}_\mathcal{M} \wedge \mathcal{I}'_\mathcal{N})$. Ce même principe est appliqué à l'opération $\mathcal{O}$ lors de son aplatissement.

[16] Une constante concrète est une donnée implémentable dans un langage informatique dont la valeur reste constante et qui est implicitement conservée au cours du raffinement jusqu'à l'implantation.

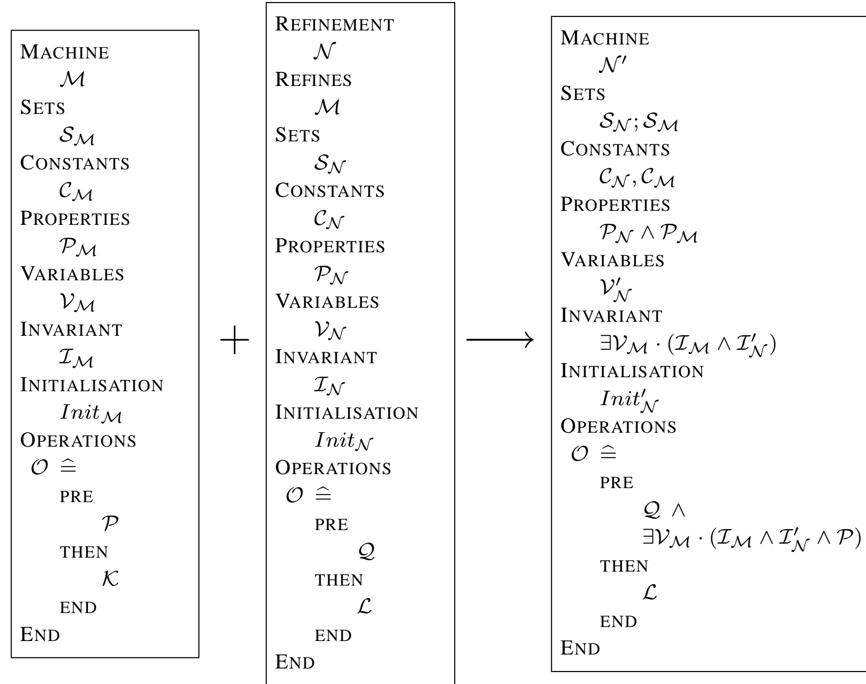


FIG. 1.5 – Aplatissement du lien de raffinement REFINES

Aplatissement du lien d'inclusion INCLUDES. Les règles de visibilité définies par la relation d'inclusion entre composants B (Sect. 1.4.1, page 24) permettent d'avoir accès à des données d'instances de machines incluses, ainsi que leurs propriétés.

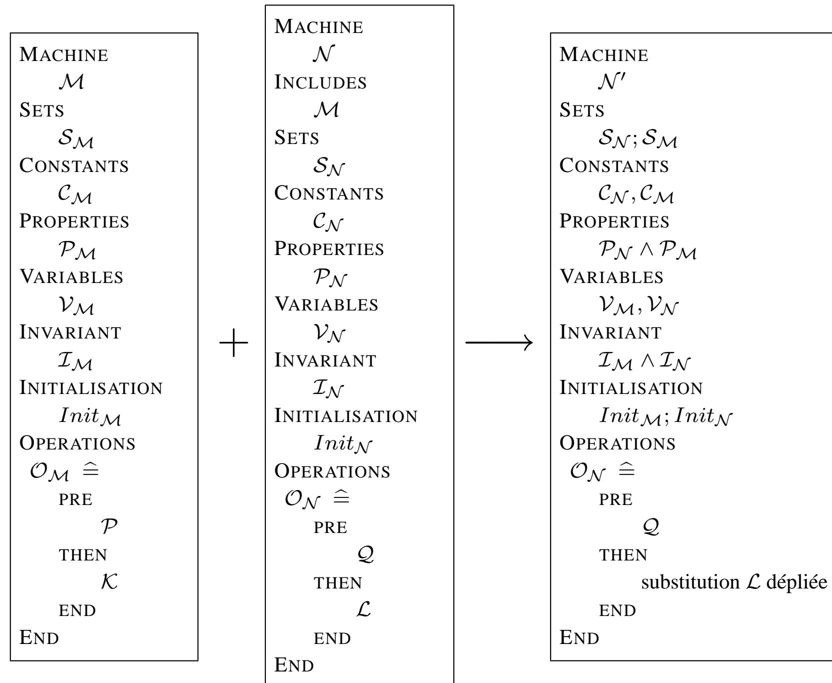


FIG. 1.6 – Aplatissement du lien d'inclusion INCLUDES

Ce mécanisme aboutit à un composant enrichi par les données incluses. L'aplatissement de l'inclusion entre machines permet ainsi de récupérer au niveau de la machine incluante toutes les données de

la machine incluse. Les opérations du composant incluant sont conservées ; cependant, les appels aux opérations du composant inclus sont remplacés par les corps des opérations correspondant aux appels.

Notons que l'aplatissement de machines composées par les liens SEES et USES est réalisé de manière similaire mis à part le dépliage des opérations. En effet, dans le cadre d'un lien USES aucun appel d'opérations ne peut être effectué (il s'agit du mode semi-ouvert).

1.5 Conclusion

La méthode B se présente donc comme une méthode de développement formelle mettant en valeur aussi bien les aspects statiques que dynamiques d'un système. D'un côté, les aspects statiques sont définis au moyen de concepts mathématiques de la théorie des ensembles et de la logique des prédicats, et d'un autre côté, les aspects dynamiques sont modélisés en utilisant le langage des substitutions généralisées. Dans notre travail nous allons exploiter ces deux caractéristiques d'une spécification B en vue d'en dériver des diagrammes aussi bien structurels que comportementaux. Il n'est certes pas possible de décrire d'une manière exhaustive, tous les fondements théoriques de B, nous nous en sommes tenu, dans ce chapitre, aux notions essentielles qui sont en rapport avec la problématique à laquelle nous nous intéressons. En somme, les points forts de la méthode B sont :

- la modularité du développement ;
- la preuve des propriétés des systèmes ;
- la génération automatique du code ;
- la disponibilité d'outils adéquats.

Le tableau suivant présente quelques exemples d'application de B tirés de (Behm *et al.,* 1997) et témoignant de la maturité de B :

Projet	Nbre de lignes de code en B	Nbre de composants
KVB	60000	250
CSL	65000	230
METEOR	107000	1075

– KVB (Contrôle de Vitesse par Balises)
– CSL (Calculateur de Sécurité Locale)
– METEOR (Métro sans conducteur de Paris)

TAB. 1.3 – Exemples d'applications réelles de B extraites de (Behm *et al.,* 1997)

Finalement, nous soulignons le fait que contrairement aux langages de spécification formelle "à objets" (*e.g.* Object-Z), B n'a pas été conçu pour intégrer des concepts du paradigme objet, ce qui rend l'élaboration de liens entre B et UML moins évidente.

Chapitre 2

Couplage de spécifications B et UML

« *Teaching specifications is to be situated between Russel: Matehmatics is the only science where one never knows what one is talking about nor whether what is said is true; and Emile Borel: Mathematics is the only science in which one knows exactly what one is talking about and one is certain that what on saying is true.* »

H. Habrias & S. Faucou
« Linking paradigms: semi-formal and formal notations », TFM *2004.*

Sommaire

2.1 Introduction **29**
2.2 Dérivation de UML vers B **31**
2.2.1 Aperçu et objectifs 31
2.2.2 Traduction des aspects structurels 32
2.2.3 Traduction des aspects comportementaux 37
2.3 Développement conjoint UML/B **40**
2.3.1 Aperçu et objectifs 40
2.3.2 Exemples d'évolutions simultanées de spécifications UML et B 41
2.3.3 Discussion 43
2.4 Dérivation de B vers UML **43**
2.4.1 Aperçu et objectifs 44
2.4.2 L'approche uniforme 45
2.4.3 L'approche interactive et incrémentale 49
2.4.4 Notre approche 51
2.5 Conclusion **52**

2.1 Introduction

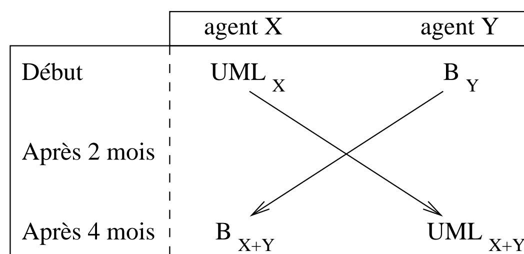
L'intégration de notations formelles et semi-formelles a été étudiée depuis plusieurs années (Fraser *et al.,* 1991) et a fait l'objet de nombreuses recherches. Son principal objectif est de surmonter les limites des unes en tirant profit des apports des autres. Dans ce chapitre, nous nous intéressons particulièrement aux travaux portant sur les langages B et UML, et nous distinguons les trois catégories d'intégration suivantes :

1. dérivation de UML vers B,
2. développement conjoint UML/B, et
3. dérivation de B vers UML.

Plusieurs équipes ont travaillé sur le lien entre UML et la méthode B. Nous pouvons citer notamment en France les travaux du CEDRIC/CNAM à Evry (Nguyen, 1998, Mammar, 2002, Laleau, 2002), du LORIA à Nancy (Meyer, 2001, Ledang, 2002), ou de l'Université de Versailles (Marcano, 2002), et à l'étranger des travaux menés au Royaume Uni à l'Université de Southampton (Snook *et al.,* 2006) ou à l'Imperial College de Londres (Lano *et al.,* 2004). Ces équipes ont privilégié la traduction d'UML en B, et son extension par des annotations écrites en B. L'objectif de tels travaux est de préciser la sémantique d'UML, de compléter ce langage pour augmenter son pouvoir d'expression, et de traduire les spécifications semi-formelles en B pour profiter des outils de la méthode B.

Ainsi, le sens de dérivation de UML vers B est largement étudié aujourd'hui, se concrétisant par l'existence de plusieurs outils de traduction : *e.g.* **UML2B** (Hazem *et al.,* 2004), **UML2SQL** (Laleau *et al.,* 2000, Mammar *et al.,* 2004), **U2B** (Snook *et al.,* 2004), **ArgoUML+B** (Ledang *et al.,* 2003). En revanche, peu de travaux portent sur les deux autres catégories d'intégration. En effet, dans ces catégories nous distinguons, d'une part, les travaux de D. D. Okalas Ossami (Ossami *et al.,* 2004, Ossami *et al.,* 2005) pour un développement conjoint en UML et B, et d'autre part, les travaux de H. Fekih (Fekih *et al.,* 2004, Fekih *et al.,* 2006) et de J.-C. Voisinet (Voisinet, 2004) pour une dérivation de B vers UML.

Ces travaux sont certes complémentaires car ils traitent tous les sens du couplage de ces deux formalismes. Une étude comparative menée par M. Satpathy et ses co-auteurs (Satpathy *et al.,* 2001) montre l'importance d'un tel couplage sur la base de développements B et UML d'un même système et atteste que : « *a combination of both is needed in order to achieve good visibility as well as precision* ». Dans cette étude, deux agents X et Y commencent par réaliser un premier jet de spécifications respectivement en UML (appelé UML_X) et en B (appelé B_Y) puis interchangent, au bout de quelques mois de développement leurs spécifications pour que X continue le développement en B (B_{X+Y}) et Y continue le développement en UML (UML_{X+Y}).



Nous présentons, ci-dessous, les principales "leçons" consignées par cette étude (Satpathy *et al.,* 2001) :

- La spécification de la structure générale du modèle est plus intuitive en UML grâce à l'utilisation de diagrammes,
- Pour conduire des spécifications précises et détaillées, UML est insuffisant et doit être augmenté par des notations formelles,
- Les ambiguïtés du cahier des charges sont mieux identifiées lors du développement en B que lors du développement en UML,

– X trouve que B_Y manque de documentation alors que Y trouve que UML_X est très abstrait et manque de détails.

Ces constatations qui découlent d'expérimentations confirment la complémentarité des techniques d'intégration de UML et B. Dans ce chapitre, nous présentons les fondements conceptuels de ces techniques. Finalement, nous notons que dans le cadre de notre travail de thèse nous ciblons la troisième catégorie d'intégration et plus particulièrement la documentation graphique en UML de spécifications B.

2.2 Dérivation de UML vers B

2.2.1 Aperçu et objectifs

P. Facon et R. Laleau, dans (Facon *et al.,* 1995) distinguent deux approches de dérivation de spécifications formelles à partir de spécifications semi-formelles : l'approche *interprétée* et l'approche *compilée*. Notons qu'une illustration détaillée et complète de l'application de ces deux approches à une dérivation de UML vers B est présentée dans (Laleau, 2002).

L'approche compilée (ou par traduction). Cette approche consiste à donner des règles permettant de traduire un modèle semi-formel directement dans un langage formel. Nous citons, par exemple, la règle de traduction du concept d'héritage extraite de (Facon *et al.,* 1995) où les entités "Secrétaire" et "Ingénieur" sont des spécialisations de l'entité "Personne" :

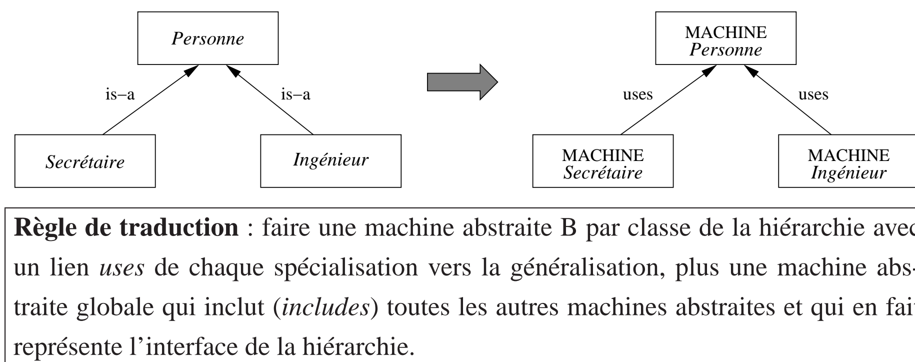


FIG. 2.1 – Traduction du concept d'héritage selon l'approche compilée (Facon *et al.*, 1995)

Notons que la plupart des travaux de dérivation de UML vers B adoptent une approche compilée (Mammar, 2002, Laleau, 2002, Meyer, 2001, Snook *et al.,* 2006) fondée sur un ensemble de règles de traduction. L'intérêt majeur d'une telle technique est qu'elle est réalisée par une traduction directe d'un modèle semi-formel à une spécification formelle. Cependant, son principal inconvénient est que la sémantique des correspondances entre B et UML n'est pas explicitement formalisée étant donné qu'elle est cachée au niveau des règles de traduction.

L'approche interprétée (ou par méta-modélisation). Cette approche est fondée sur un mélange des spécifications formelles issues aussi bien des concepts du méta-modèle que des éléments du modèle semi-formel sujet de la traduction. Il s'agit précisément de proposer, une fois pour toutes, une formalisation du méta-modèle du modèle semi-formel ; et d'effectuer, ensuite, la traduction de la partie propre à chaque

application et l'injecter au niveau de la formalisation du méta-modèle. Par exemple, pour traiter l'héritage, une classe n'est plus comme précédemment immédiatement traduite par une machine abstraite. Nous avons à la place une seule machine abstraite générique qui comprend :

- L'ensemble de tous les objets du système,
- Une fonction associant à chaque nom de classe, les ensembles d'objets instances de la classe,
- Une fonction associant à chaque nom d'association la relation correspondante entre objets, et
- Une fonction associant à chaque objet et attribut la valeur correspondante.

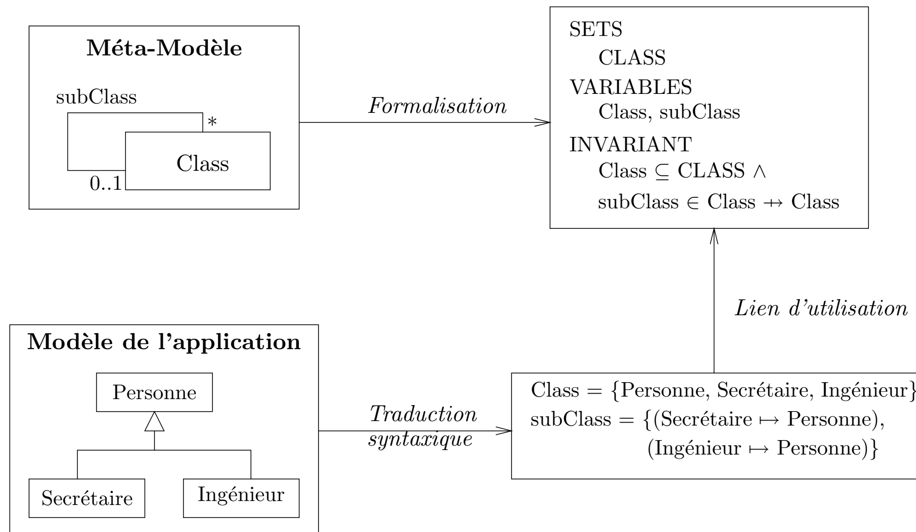


FIG. 2.2 – Traduction du concept d'héritage selon l'approche interprétée (Laleau, 2002)

Peu de travaux se sont attachés à ce type d'approches pour la dérivation de spécifications B à partir de modèles UML. Nous distinguons principalement dans ce contexte le travail de R. Laleau (Laleau, 2002).

Discussion. Les termes "*shallow embedding*" et "*deep embedding*" (Boulton *et al.,* 1993, Wildmoser *et al.,* 2004) sont souvent utilisés pour désigner un changement ou une intégration de formalismes. Le premier terme désigne une traduction directe d'un modèle source vers un modèle cible, alors que le second terme désigne une traduction de formalismes aboutissant à des constructions qui représentent des types de données. L'approche interprétée et l'approche compilée suivent une logique semblable. En effet, contrairement à l'approche *interprétée*, l'approche *compilée* se veut plus directe car les spécifications formelles qui en découlent reflètent de manière naturelle et explicite les éléments du modèle (ou de l'application) en question. Cependant, pour des besoins propres à des systèmes particuliers, l'approche interprétée permet de préciser et de clarifier les éléments des méta-modèles qui leurs sont dédiés. Notre travail n'étant pas spécifique à des systèmes bien déterminés et vu que nous ciblons le niveau "modèles", nous allons nous concentrer uniquement sur les techniques de dérivation par traduction directe de UML vers B.

2.2.2 Traduction des aspects structurels

Les principaux travaux sur lesquels nous allons nous baser à ce niveau sont (Mammar, 2002, Laleau, 2002, Meyer, 2001, Snook *et al.,* 2006) avec une mise en évidence des traductions des éléments de modé-

lisation UML suivants : classes, attributs de classes, relation de généralisation/spécialisation, associations et classes associatives.

Traduction de classes. La transformation d'une classe C en des données B passe par la formalisation de l'ensemble des instances possibles (que nous notons *Instances_Possibles*) de la classe. Ce consensus des différentes approches de dérivation de UML vers B se traduit par un ensemble abstrait au niveau de la spécification formelle (voir Fig. 2.3) à partir duquel est défini l'ensemble des instances existantes (que nous notons *Instances_Existantes*) de C. En d'autres termes, il s'agit d'exprimer l'invariant *Instances_Existantes* $\subseteq$ *Instances_Possibles* ou alors *Instances_Existantes* $\in \mathbb{P}($*Instances_Possibles*$)$. Dans ce contexte, (Meyer, 2001) rajoute une couche pour distinguer l'ensemble de tous les objets possibles *OBJECTS*[17] et définit *Instances_Possibles* par une constante telle que *Instances_Possibles* $\subseteq$ *OBJECTS*. Quand plusieurs classes sont traduites, deux points de vue peuvent être distingués pour formaliser l'ensemble de toutes les classes d'un diagramme de classes : le point de vue machine unique et le point de vue ensemble de machines liées par les clauses *USES*, *SEES* ou *INCLUDES*. Néanmoins, le second point de vue est le plus privilégié par ces techniques de dérivation de UML vers B car il permet la modularité et la structuration des spécifications B finales. De plus, le point de vue machine unique peut être obtenu par application d'une technique d'aplatissement des machines B issues des diverses classes.

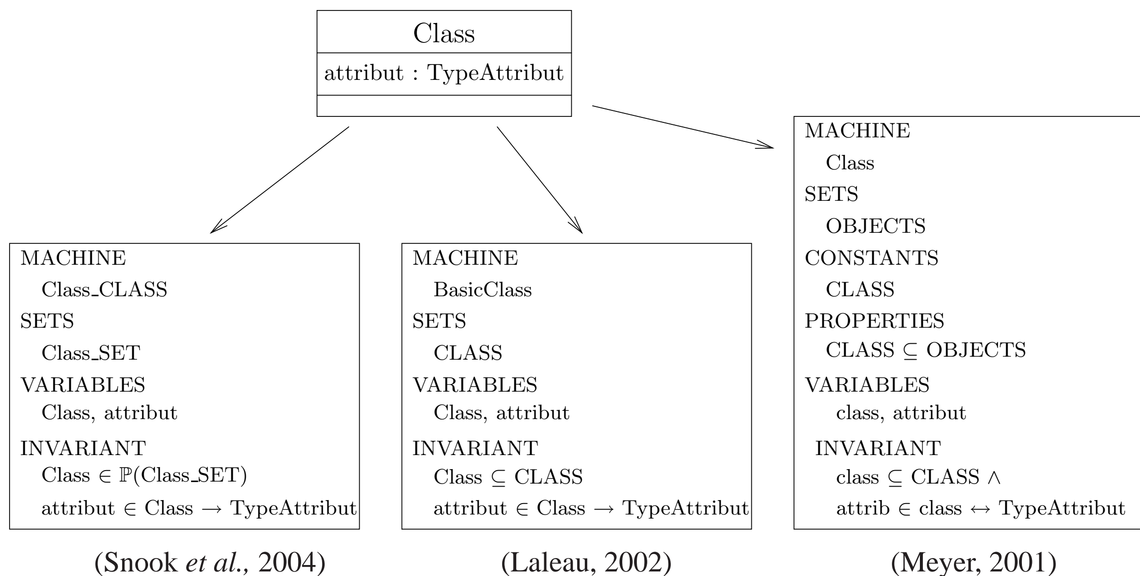


FIG. 2.3 – Traduction des concepts de classe et d'attribut en B.

Traduction d'attributs. La traduction d'un attribut t d'une classe C est réalisée sur la base de l'ensemble des instances existantes de C et du type *TypeAttribut* de t. Cet attribut est formalisé par une relation distinguant les valeurs de t pour chaque instance de C : $t \in$ *Instances_Existantes* $\leftrightarrow$ *TypeAttribut*. Par ailleurs, dans (Nguyen, 1998, Mammar, 2002, Laleau, 2002) si l'attribut est mono-valué et selon que t soit obligatoire ou optionnel, la formalisation de cet attribut se traduit par la spécialisation de la relation

[17] L'ensemble *OBJECTS* est représenté au sein de la même machine que la formalisation de la classe dans la Fig. 2.3 uniquement pour des raisons de clarté. Notons que (Meyer, 2001) déclare cet ensemble dans une machine abstraite spéciale appelée *Types* dans laquelle sont également modélisés tous les types qui apparaissent dans le diagramme de classes.

qui lui est asscoiée en une fonction totale ou partielle. Cette distinction entre attributs mono/mutli-valués n'est pas considérée par les autres approches. Par exemple, (Meyer, 2001) traduit un attribut par une simple relation "$\leftrightarrow$" et (Snook *et al.,* 2006) ne considère que des fonctions totales "$\rightarrow$".

Traduction du mécanisme d'héritage. La traduction d'une sous-classe est réalisée de la même manière que la traduction d'une classe avec le rajout d'un invariant particulier indiquant la relation sous-classe/super-classe. Soit *Class* une super-classe et *SubClass* sa sous-classe alors la traduction de ces deux classes en B définit deux variables *Class* et *SubClass* avec l'invariant suivant : $SubClass \subseteq Class$ ou $SubClass \in \mathbb{P}(Class)$. Le deuxième invariant est extrait de l'approche (Snook *et al.,* 2006) où plusieurs autres cas sont identifiés selon que le nombre d'instances soit fixé ou non. Lorsqu'une machine B est dérivée pour chaque classe du diagramme UML, un lien *USES* est défini de la machine issue de la sous-classe vers la machine issue de la super-classe. Par ailleurs, (Snook *et al.,* 2006) préconise également de mettre toutes les données B issues d'une même hiérarchie d'héritage au sein d'une même machine B. Contrairement à cette deuxième traduction, la première est généralement destinée à conserver la structuration des classes au niveau des machines B et garder ainsi l'intuition du développement incrémental assurée par le mécanisme d'héritage. La Fig. 2.4 illustre ces deux traductions sur la base de (Snook *et al.,* 2006) et où *SubClass* est une sous-classe de la classe *Class* (Fig. 2.3).

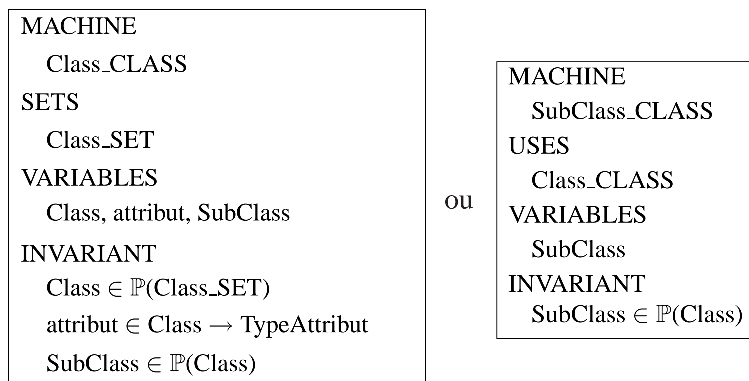


FIG. 2.4 – Traduction du mécanisme d'héritage

Traduction des associations. Intuitivement la traduction d'une association doit prendre en compte : (*i*) le nom de l'association, (*ii*) les multiplicités des deux côtés de l'association, (*iii*) les données B dérivées des deux classes extrémités de l'association et (*iv*) le sens de navigation. Soient $Class_1$ et $Class_2$ deux variables issues de deux classes UML avec une association *Assoc* de $Class_1$ vers $Class_2$, alors une traduction naturelle de *Assoc* en B est de définir la relation *Assoc* entre les ensembles d'objets existants comme suit : $Assoc \in Class_1 \leftrightarrow Class_2$. Des spécialisations de la relation B ainsi créée peuvent être définies selon les multiplicités de l'association, par exemple, si *Assoc* est qualifiée par une multiplicité "$*$" du côté de $Class_1$ et une multiplicité "1" du côté de $Class_2$ alors la relation créée de $Class_1$ vers $Class_2$ est une fonction totale. Pour un objectif de structuration des spécifications B, (Meyer, 2001) propose de produire pour chaque association une machine abstraite distincte qui, d'une part, définit la variable *Assoc*, et d'autre part, utilise (clause *USES*) les deux machines produites à partir de $Class_1$ et $Class_2$. Dans ce même contexte, (Snook *et al.,* 2006) ne définit aucune machine supplémentaire pour traduire

l'association mais impose l'utilisation de noms de rôles pour qualifier les extrémités de l'association et en dériver des variables au niveau des machines générées pour $Class_1$ et $Class_2$.

Les principales limites de ces deux traductions portent sur la complexité des spécifications qui en résultent. En effet, la première transformation produit trop de machines abstraites, alors que la seconde inclut trop d'informations au niveau d'une même machine. En vue de remédier à ces limites, (Nguyen, 1998, Mammar, 2002, Laleau, 2002) abordent ces deux cas de transformation en distinguant deux types d'associations : fixes et non-fixes. L'association *Assoc* est dite association fixe pour $Class_1$ (respectivement $Class_2$) si l'ensemble des liens concernant chaque objet de $Class_1$ (respectivement $Class_2$) est créé en même temps que l'objet et ne peut être modifié ensuite. Dans le cas contraire *Assoc* est dite non-fixe pour $Class_1$ (respectivement $Class_2$). À partir de là, la première transformation est privilégiée par (Nguyen, 1998, Mammar, 2002, Laleau, 2002) uniquement pour traduire les associations non-fixes (ni pour $Class_1$ ni pour $Class_2$). Par contre, dans le cas où *Assoc* est fixe aussi bien pour $Class_1$ que pour $Class_2$ alors le choix d'une transformation particulière est laissée au spécifieur. Finalement, les autres cas sont traités en appliquant la seconde transformation.

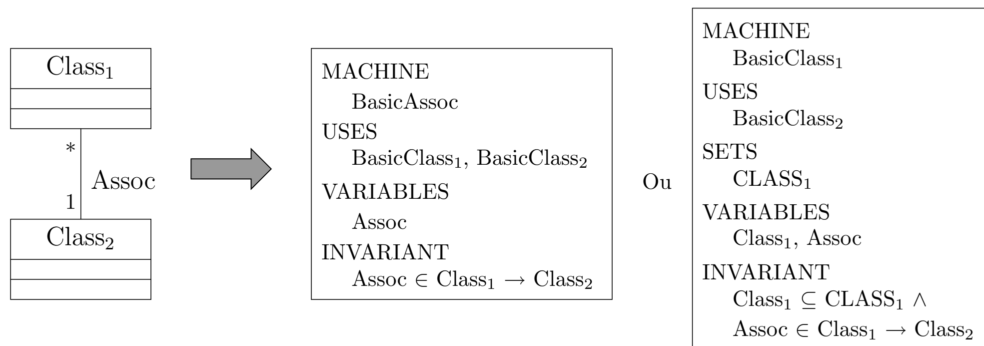


FIG. 2.5 – Traductions d'associations préconisées par (Laleau, 2002)

(Nguyen, 1998, Meyer, 2001) traitent l'existence de contraintes entre associations (*e.g.* contrainte Xor, totalité, ...) : "soient $Assoc_1$ et $Assoc_2$ deux associations ayant une classe *Class* en commun. Une contrainte Xor indique qu'une instance de *Class* ne peut être reliée à une autre instance de classe que par une seule instance d'association à la fois (ou exclusif)". La traduction de cette contrainte est réalisée en rajoutant l'invariant : $\mathrm{dom}(Assoc_1) \cap \mathrm{dom}(Assoc_2) = \emptyset$

Traduction des classes associatives. La transformation des classes associatives est généralement réalisées par une combinaison des règles de traduction des classes et des associations simples en des machines abstraites distinctes. Notons que (Snook *et al.,* 2006) ne prend pas en compte la traduction des classes associatives. Par ailleurs, (Nguyen, 1998, Mammar, 2002, Laleau, 2002) distinguent les classes associatives qui participent à d'autres associations ou à des liens de généralisation/spécialisation. Dans ce cas, un choix de modélisation est déterminé considérant la classe associative en tant que classe liée par deux associations simples (r_1 et r_2) aux deux classes extrémités de la classe associative. Soit *Assoc* une classe associative reliant les deux classes $Class_1$ et $Class_2$ et correspondant à ce cas particulier, alors *Assoc* est transformée en un ensemble abstrait *ASSOC* avec l'invariant : $Assoc \subseteq ASSOC$. Les associations r_1 et r_2 donnent lieu à deux variables r_1 et r_2 et l'invariant de typage suivant : $r_1 \in Assoc \rightarrow Classe_1 \wedge r_2 \in Assoc \rightarrow Classe_2$. L'invariant supplémentaire suivant : $r_1 \otimes r_2 \in Assoc \rightarrowtail Classe_1 \times Classe_2$ est 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.

Discussion. Dans cette sous-section, nous avons mis l'accent sur les transformations des structures statiques auxquelles nous nous intéressons dans le cadre de notre travail. En effet, en vue de représenter la structure de spécifications B nous cherchons à dériver des classes, des attributs de classes, des associations, des classes associatives et de l'héritage. Le Tab. 2.1 présente un récapitulatif des différents points communs et spécificités des principales approches de dérivation de UML vers B.

	(Nguyen, 1998) (Mammar, 2002) (Laleau, 2002)	(Meyer, 2001) (Ledang, 2002)	(Snook et *al.,* 2006)
Classes (instances indéterminées)	+	+	+
Classes (instances fixes)	-	-	+
Attributs de classes	+	+	+
Distinction entre attributs multi/mono-valués	+	-	-
Mécanisme d'héritage	+	+	+
Multiplicités des associations	+	+	+
Sens de navigation des associations	-	+	+
Noms de rôles	+	-	+
Contraintes entre associations	+	+	-
Distinction entre associations fixes/non-fixes	+	-	-
Classes associatives	+	+	-
Classes associatives participant à des associations ou à des liens de spécialisation/généralisation	+	-	-
Paramétrisation des classes	-	-	+

Légende : "+" (critère pris en compte dans la transformation) ; "-" (critère non considéré)

TAB. 2.1 – Points communs et divergences entre différentes approches de dérivation de UML vers B pour la traduction des aspects structurels

Une idée naturelle et simpliste pour l'étude du sens inverse de dérivation (*i.e* de B vers UML), serait de formaliser les traductions précédentes des aspects statiques de UML en B par une fonction**[18]** φ et d'appliquer la fonction φ^{-1} pour dériver de l'UML à partir de B. Nous pensons qu'une telle manière de faire est assez restrictive car nous ne nous basons pas sur des spécifications B dérivées à partir de modèles UML. En effet, notre objectif n'est pas de garantir le retour vers une représentation UML établie au préalable. Notre point de départ est donc une spécification B développée indépendamment de tout modèle UML.

Pour assurer cette double correspondance UML $\xrightarrow{\varphi}$ B $\xrightarrow{\varphi^{-1}}$ UML, R. Laleau et F. Polack dans (Laleau *et al.,* 2002) proposent deux méta-modèles spécifiques : l'un précisant les modèles UML pris en compte dans leur approche (modèles nommés IS-UML) et l'autre délimitant les spécifications B produites à partir de ces modèles UML. Sur cette base conceptuelle homogène, des correspondances "un à un" sont établies entre méta-modèles (fonction φ) et par conséquent l'application de la fonction φ^{-1} devient restreinte à un sous-ensemble de constructions en B.

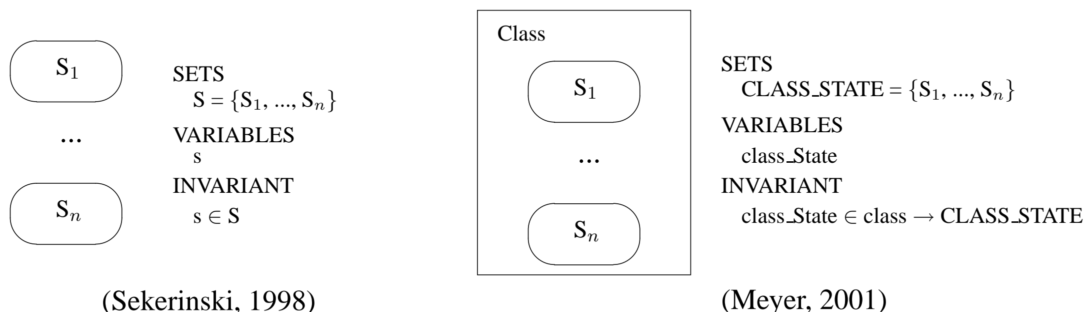
[18] Nous supposons que φ est une fonction injective garantissant que si une traduction existe pour une même construction en UML alors cette traduction est unique.

Dans le cadre de notre travail, nous allons présenter un méta-modèle B qui se veut générique et nous allons dans un premier temps délimiter les liens possibles entre B et UML, sur la base de leurs méta-modèles respectifs. Vu que nos spécifications B de départ sont indépendantes de tout modèle UML alors les liens entre B et UML que nous définissons sont établis par une correspondance “un à plusieurs” entre méta-modèles. Aussi, une part importante d'interaction est-elle requise pour sélectionner la transformation la plus adéquate. Nous sommes donc amené à proposer dans un second temps d'autres techniques pour réduire au mieux cette interaction et guider le choix d'une représentation parmi d'autres. Nous parlerons alors d'approche *interprétée* et d'approche *calculée* pour qualifier ces deux phases de notre travail de thèse.

2.2.3 Traduction des aspects comportementaux

À ce niveau, nous allons nous focaliser en particulier sur les travaux de traduction des diagrammes d'états/transitions d'UML car dans notre travail nous nous limitons à ces diagrammes pour représenter les aspects comportementaux de spécifications B. L'un des premiers travaux pour la combinaison des diagrammes d'états et B avait pour finalité de concevoir, de manière graphique, des systèmes réactifs (Sekerinski, 1998). Dans cette approche, la conception des systèmes réactifs est initialement réalisée au moyen de statecharts (Harel, 1987) qui sont traduits, par la suite, en une machine abstraite B. Cette traduction permet, dans une certaine mesure, d'attribuer une sémantique formelle B aux diagrammes d'états de D. Harel, généralement utilisés pour leur forme graphique expressive. Différents aspects de ce travail nous paraissent intéressants à mettre en évidence tels que la prise en compte des notions d'états hiérarchiques et concurrents, ainsi que la notion d'états communicants. Bien que les règles proposées par (Sekerinski, 1998) soient assez précises et couvrent la quasi-totalité des spécificités des diagrammes d'états, elles ne prennent pas en compte l'aspect statique, ou l'existence d'un lien avec un diagramme de classes. Les travaux de H. P. Nguyen (Nguyen, 1998) et E. Meyer (Meyer, 2001) sont, entre autres, des tentatives de formalisation des diagrammes d'états/transitions d'UML ciblant la prise en compte des aspects statiques. En effet, au niveau des diagrammes d'états/transitions traités par ces travaux, les états décrivent des situations possibles dans la vie d'un objet.

Traduction des états. (Sekerinski, 1998) représente les états d'un diagramme d'états/transitions par un ensemble énuméré $States = \{State_1, \ldots, State_n\}$ avec la définition d'une variable abstraite *state* telle que $state \in States$. Les transitions entre états sont alors assurés par un changement de valeurs de cette variable. Pour traduire des états d'une classe *Class*, (Meyer, 2001) reprend cette formalisation par un ensemble énuméré et définit la variable *state* par la fonction totale : $state \in Class \rightarrow States$ dans la machine B issue de la classe *Class* :



(Sekerinski, 1998) (Meyer, 2001)

Dans ce même contexte, (Nguyen, 1998) formalise la variable *state* directement par la fonction $state \in Class \rightarrow \{State_1, \ldots, State_n\}$ et définit au niveau de la clause *DEFINITION* les états distinctement par des prédicats comme suit : $State_i(o) \mathrel{\widehat{=}} o \in state^{-1}[\{State_i\}]$ (où $i \in 1..n$). Par conséquent o désigne une instance de la classe *Class*. La Fig. 2.6 illustre cette dernière transformation. Remarquons que cette représentation par des prédicats permet également d'indiquer une abstraction particulière des attributs de l'objet o (voire aussi de l'existence ou non de liens avec des instances d'autres classes). En effet, la fonction totale $Etat_Voiture \in Voiture \rightarrow \{Marche, Arrêt\}$ peut provenir de la traduction d'un attribut énuméré *Etat_Voiture* de la classe *Voiture*. Dans ce cas précis, les états *Marche* et *Arrêt* associés à la classe *Voiture* sont exprimés tout simplement à partir des valeurs possibles de l'attribut *Etat_Voiture*.

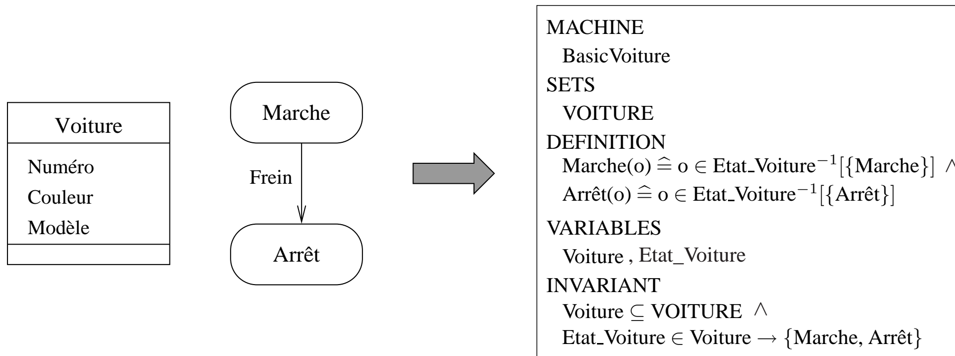


FIG. 2.6 – Exemple de traduction d'états proposé par (Nguyen, 1998)

Traduction des transitions. Les traductions des transitions considèrent plusieurs cas de figure : événements simples, événements conditionnés, événements conditionnés avec action, événements pouvant déclencher plusieurs transitions, ... Nous présentons ci-dessous les propositions faites par (Sekerinski, 1998) pour traiter ces différents cas, puis nous parlerons de leurs adaptations au niveau de (Nguyen, 1998) et (Meyer, 2001) pour la prise en compte de l'aspect statique.

	Diagramme	Traduction
(a)	S_1 —Ev→ S_2	Ev ≙ IF s = S_1 THEN s := S_2
(b)	S_1 —Ev→ S'_1 ... ... S_i —Ev→ S'_i	Ev ≙ CASE s OF EITHER S_1 THEN s := S'_1 ... OR S_i THEN s := S'_i END
(c)	S_1 —Ev[cond]/act→ S_2	Ev ≙ IF s = S_1 ∧ cond THEN s := S_2 || act END
(d)	S_1 —Ev[cond$_1$]/act$_1$→ S'_1 ... S_1 —Ev[cond$_i$]/act$_i$→ S'_i	Ev ≙ SELECT s = S_1 ∧ cond$_1$ THEN s := S'_1 || act$_1$ ... WHEN s = S_1 ∧ cond$_i$ THEN s := S'_i || act$_i$ ELSE *skip* END

H. P. Nguyen dans (Nguyen, 1998) distingue ces différents cas de figure en termes de scénarios mono-entité[19]. L'auteur considère que l'utilisation de structures telles que **IF** s = S_1 ... **THEN** ... **END** (cas (a) et (c)) est une approche *défensive* car la vérification de la condition de déclenchement s'effectue dans le corps de l'opération. L'utilisation de la structure **PRE** ... **THEN** ... **END** est donc privilégiée. Vu que dans cette approche les états sont exprimés par des prédicats et non par des éléments d'ensembles, alors la condition s = S_1 est remplacée par le prédicat décrivant l'état S_1 (*i.e.* $State_1(o)$). Par exemple, la transition *Frein* au niveau de la Fig. 2.6 est traduite par $Frein(o) \widehat{=}$ **PRE** $Marche(o)$ **THEN** *action* **END** et où *action* est une substitution correspondant au changement de situation en *Arrêt* de *Voiture*. (Meyer, 2001) également préconise l'utilisation d'une structure similaire et conduit à transformer la transition *Frein* comme suit :

```
Frein(o) ≙  PRE
                 o ∈ voiture ∧
                 voitureState(o) = Marche
            THEN
                 voitureState(o) := Arrêt
            END
```

avec

– *voiture* est une variable abstraite décrivant l'ensemble des instances existantes de la classe *Voiture*,
– *voitureState* est une fonction totale de l'ensemble *voiture* vers l'ensemble énuméré délimitant les états *Marche* et *Arrêt*.

Discussion. Nous avons présenté à ce niveau un aperçu de traductions typiques des aspects comportementaux (états et transitions simples) en B. Nous constatons, d'un côté des similarités, et d'un autre côté une complémentarité des techniques de dérivation de spécifications B sur la base d'un diagramme de comportement. Notons que les différentes transformations ont été initialement développées par (Sekerinski, 1998) et adaptées par la suite au niveau des autres approches pour la formalisation des états d'instances de classes. Bien que (Nguyen, 1998) se veuille complet, de par la couverture de la quasi-totalité des concepts de modèles dynamiques et statiques, son inconvénient majeur est qu'elle n'aborde pas les notions de hiérarchie et de concurrence contrairement à (Sekerinski, 1998, Meyer, 2001). Le tableau 2.2 présente les points communs ainsi que les divergences entre ces différentes approches de dérivation de UML vers B pour la traduction des aspects comportementaux.

	(Sekerinski, 1998)	(Nguyen, 1998)	(Meyer, 2001)
États simples	+	+	+
États hiérarchiques	+	-	+
États concurrents	+	-	+
Transitions mono-entité	-	+	+
Transitions multi-entités	-	+	-
Transitions conditionnées	+	+	+
Transitions paramétrées	+	+	-
Transitions mono-sources et pluri-cibles (cas (d))	+	+	-
Transitions pluri-sources et pluri-cibles (cas (b))	+	+	-

Légende : "+" (critère pris en compte dans la transformation) ; "-" (critère non considéré)

TAB. 2.2 – Points communs et divergences entre différentes approches de dérivation de UML vers B pour la traduction des aspects comportementaux

[19] C'est-à-dire scénarios rattachés à une seule entité ou classe.

Le travail de H. Ledang (Ledang, 2002) est une extension du travail de E. Meyer (Meyer, 2001) où ces notions de concurrence et de communication sont développées. L'originalité de cette approche porte principalement sur la formalisation d'événements selon qu'ils seraient différés ou non et de raffiner l'opération abstraite correspondante. Dans le cadre de notre travail, nous nous intéressons en particulier aux concepts de base des diagrammes d'états/transitions et plus particulièrement les 4 cas de transitions présentés précédemment. Comme dans le cadre des travaux de formalisation des diagrammes de classes nous pensons que si φ est la fonction de traduction de UML vers B pour la formalisation de diagrammes d'états/transitions, alors φ^{-1} se ramène à une étude syntaxique (en effectuant par exemple un "pattern matching"[20]) d'une spécification B quelconque pour en dériver un(des) diagramme(s) d'états/transitions. Une telle technique a été étudiée par (Hammad *et al.,* 2002, Fekih *et al.,* 2004) et présente de nombreux inconvénients que nous allons détailler au niveau de la section 2.4.

2.3 Développement conjoint UML/B

Cette section est basée sur les travaux de D.-D. O. Ossami (Ossami *et al.,* 2004, Ossami *et al.,* 2005) qui s'attache à une approche de multi-modélisation utilisant simultanément B et UML. L'intérêt de ce travail est qu'il se concentre sur un développement conjoint de deux vues d'un même système, l'une formelle et l'autre semi-formelle, tout en conservant la cohérence et la traçabilité entre ces deux vues. Avant de présenter les fondements théoriques de cette approche, nous notons que ce travail est largement influencé par l'approche unidirectionnelle présentée au niveau de la section précédente et plus précisément par les travaux de E. Meyer (Meyer, 2001) et H. Ledang (Ledang, 2002).

2.3.1 Aperçu et objectifs

L'idée de base de cette approche est de définir des "*opérations de construction*" permettant de faire évoluer simultanément les deux spécifications UML et B. Sur le plan méthodologique, il s'agit de distinguer un ensemble de *tactiques* offrant chacune un catalogue d'*opérations de construction* et dépendant principalement de l'état courant du développement.

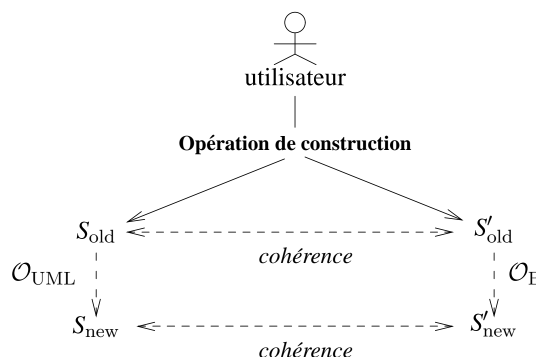


FIG. 2.7 – Évolution de l'état des spécifications B et UML selon l'approche (Ossami *et al.,* 2004)

La Fig. 2.7, extraite de (Ossami *et al.,* 2004) illustre ce procédé de développement en mettant l'accent sur les états des spécifications UML et B avant (*i.e.* S_{old} et S'_{old}) et après (*i.e.* S_{new} et S'_{new}) l'application

[20] Pattern matching : mise en correspondance de formes selon un ensemble prédéfini de règles ou de critères.

d'opérations de construction de $\mathcal{O}_{\mathrm{UML}}$ et de $\mathcal{O}_{\mathrm{B}}$. Cette évolution parallèle des développements UML et B veut garantir la cohérence des deux représentations. Elle est basée sur une hypothèse fondamentale indiquant que les spécifications B et UML ne contredisent pas la fonction φ de dérivation de UML vers B. Par conséquent, si $S'_{\mathrm{old}} = \varphi(S_{\mathrm{old}})$ alors une application simultanée de $\mathcal{O}_{\mathrm{UML}}$ et de $\mathcal{O}_{\mathrm{B}}$ produit les états S_{new} et S'_{new} avec $S'_{\mathrm{new}} = \varphi(S_{\mathrm{new}})$.

L'intérêt de l'application parallèle de $\mathcal{O}_{\mathrm{UML}}$ et de $\mathcal{O}_{\mathrm{B}}$ est de pouvoir aborder un niveau de granularité plus fin que les approches de dérivation de UML vers B. Elle permet de se focaliser, en même temps, sur les deux vues formelle et semi-formelle lors de chaque incrément de spécification.

2.3.2 Exemples d'évolutions simultanées de spécifications UML et B

En vue d'illustrer les notions de tactique et d'opérations de construction qui fondent cette approche de développement conjoint en UML et B, nous allons nous baser sur l'exemple de la classe *Class* présentée au niveau de la Fig. 2.3 avec la spécification B produite par (Meyer, 2001). Notons que la construction de la spécification UML ou de la spécification B se base sur deux activités principales : l'introduction de composants et l'introduction de données. Au niveau de la vue UML les notions de composants et de données portent respectivement sur la classe "*Class*" et l'attribut "*attribut*", alors qu'au niveau de la vue B, ces notions indiquent respectivement la machine "*Class*" et la variable abstraite "*attribut*".

Introduction d'un composant B. L'opération de construction associée à l'introduction de la machine abstraite *Class* au niveau de la spécification B est nommée *newComponent* et induit les modifications suivantes :

1. **Sur la vue B**, l'application de *newComponent* donne lieu aux deux machines B suivantes :
   (a) **Types** dans laquelle sont définis :
      (*i*) l'ensemble abstrait *OBJECTS* décrivant tous les objets possibles du modèle,
      (*ii*) la constante *CLASS* correspondant à l'ensemble de tous les objets possibles de la classe UML représentant la machine *Class*,
      (*iii*) l'invariant $CLASS \subseteq OBJECTS$.
   (b) **Class** représentant le composant B à introduire et dans laquelle sont définis :
      (*i*) un lien *SEES* indiquant l'utilisation de la machine *Class* des données de la machine *Types*,
      (*ii*) la variable *class* indiquant l'ensemble des instances existantes du modèle,
      (*iii*) l'invariant $class \subseteq Class$.
2. **Sur la vue UML**, deux éléments de modélisation sont introduits :
   (a) une classe abstraite *OBJECTS* qui représente l'ensemble de tous les objets possibles,
   (b) la classe *Class* correspondant à la machine B introduite dans le modèle.

Ci-dessous, nous présentons la formalisation proposée par (Ossami *et al.,* 2004) pour cette opération de construction. Notons que *newComponent* est aussi appelée tactique et se décompose en deux ensembles d'opérations de construction : l'un s'attachant à l'évolution de la vue B et l'autre porte sur l'évolution de la vue UML.

Opération de construction	$\mathcal{O}_{\mathrm{B}}$	$\mathcal{O}_{\mathrm{UML}}$
newComponent	*newMachine* *addName* *addCompositionLink*[a] *newDataType*[b] *newVariable*[c]	*newClass* *addName*

[a] utilisé lors de la création de toute machine hormis la machine *Types*.
[b] permet d'introduire les ensembles *OBJECTS* et *CLASS* dans la machine *Types*.
[c] introduit la variable *class* dans la machine *Class*.

Nous pouvons remarquer à partir de cette opération d'introduction d'un composant B que dans cette approche le concept "Machine abstraite B" apparaît comme étroitement lié au concept "classe" en UML. Nous pensons qu'une telle vision est assez contraignante car elle représente une restriction assez forte au niveau des développements en B. En effet, bien que des opérateurs de construction autres que *newComponent* peuvent donner lieu à la création d'une machine B, sans que cette machine ne représente une classe UML, les développements B restent aussi complexes et artificiels que ceux obtenus par traduction de UML vers B.

Introduction de données. Nous supposons à ce niveau que la machine et la classe désignées par "*Class*" sont déjà créées via l'application de l'opération précédente et que l'analyste cherche à introduire l'attribut "*attribut*" au niveau de la classe "*Class*". Pour ce faire, il s'agit d'appliquer une opération de construction nommée *newData* et qui induit les modifications suivantes :

1. **Sur la vue B**, introduction d'une variable caractérisée par :
   (*i*) son nom : le nom de l'attribut (*i.e.* "*attribut*"),
   (*ii*) son invariant de typage : $attribut \in class \leftrightarrow TypeAttribut$,
   (*iii*) sa valeur initiale : si celle-ci est définie.
2. **Sur la vue UML**, introduction de l'attribut "*attribut*" typé par *TypeAttribut* et ayant éventuellement une valeur initiale.

La formalisation de cette opération de construction est la suivante :

Opération de construction	$\mathcal{O}_{\mathrm{B}}$	$\mathcal{O}_{\mathrm{UML}}$
newData	*newVariable* *addName* *addInvariant*[a] |*addProperty*[b] *addInitialisation*[c]	*newAttribute* *addName* *addType* *addInitialValue* *addMultiplicity* *addProperty*

[a] valable pour les variables et les constantes.
[b] valable pour les constantes.
[c] valable pour les variables.

L'opération de construction *newData* assimile une variable abstraite ou une constante par un attribut de classe vu l'analogie entre les notions de machine abstraite en B et de classe en UML. Cette analogie sur laquelle se base ce travail provient particulièrement des travaux de dérivation de UML vers B.

2.3.3 Discussion

Les règles de traduction de UML vers B proposées par différentes approches, notamment celles sur lesquelles se base cette technique de développement conjoint en UML et B, ne peuvent être appliquées pour un aller-retour entre constructions UML et B que lorsque le point de départ est une modélisation en UML (*i.e.* traduction UML → B → UML). Vu cette dérivation unidirectionnelle, la technique d'évolution simultanée des deux spécifications doit rester compatible avec cette traduction, et par conséquent des restrictions sur les constructions en B ont été envisagées.

Il est, certes, évident que cette limitation affecte considérablement le raisonnement en B car une évolution directe de la spécification formelle B doit satisfaire des schémas particuliers pour pouvoir garantir l'existence d'une évolution analogue au niveau de la spécification UML. Ces schémas particuliers sont intuitivement précisés par les règles de dérivation de UML vers B. Nous pensons qu'une telle restriction n'est pas aisée à mettre en œuvre lorsqu'il s'agit de faire évoluer directement le modèle B. En effet, les spécifications formelles produites par les règles de dérivation de UML vers B sont parfois très éloignées de ce que le développeur aurait pu écrire directement en B. Cette distance provient du fait qu'un développeur B n'adopte pas en général une approche Orientée Objets et ne raisonne donc pas en termes de classes et d'attributs de classes, mais plutôt en termes de machines abstraites, de données B et du traitement de ces données dans une approche formelle guidée par les preuves. Par exemple, une machine B peut intégrer plusieurs ensembles abstraits sans pour autant que ces ensembles abstraits ne soient des attributs de la classe produite par l'opération de construction *newComponent*. Étant donné qu'il existe une analogie entre les notions d'ensembles abstraits et d'instances de classes alors, un ensemble abstrait E défini dans une machine M peut également être traduit par une classe distincte de la classe issue de M. Par conséquent, l'appel de l'opération de construction *newData* ne doit pas nécessairement faire évoluer la vue UML en y introduisant un attribut de classe mais plutôt une classe ou d'autres constructions UML. Cette limite est justifiée par le fait que dans cette approche de construction simultanée, l'opération *newData* est d'abord destinée à faire évoluer la vue UML, puis à répercuter cette évolution sur la spécification B. Les travaux actuels de D.-D. O. Ossami (Ossami *et al.,* 2004, Ossami *et al.,* 2005) prennent donc implicitement la vue UML comme point de départ.

Cette constatation nous amène à conclure que l'étude des transformations possibles d'une spécification B en UML sans que cette spécification B ne soit développée « à l'UML » permet d'améliorer et d'enrichir cette approche de développement conjoint. En effet, nous pensons que l'élaboration de correspondances de B vers UML permet de distinguer d'autres opérations de construction sans affecter le raisonnement en B et donne ainsi une plus grande couverture des évolutions simultanées des constructions des modèles formel et semi-formel.

2.4 Dérivation de B vers UML

Les approches de dérivation de B vers UML peuvent être classées en deux familles selon les diagrammes qu'elles construisent : dérivation de diagrammes de classes, et/ou dérivation des diagrammes d'états. Dans cette section, nous allons nous intéresser à ces travaux de dérivation de B vers UML en présentant un aperçu de règles de transformation qu'ils proposent. Nous allons nous intéresser en particulier à l'étude des travaux de J.-C. Voisinet *et al.,* (Hammad *et al.,* 2002, Tatibouet *et al.,* 2002, Voisinet, 2004) ainsi qu'aux travaux de H. Fekih *et al.,* (Fekih *et al.,* 2004, Fekih *et al.,* 2006). Par ailleurs, au niveau de

la deuxième famille d'approches de dérivation de B vers UML nous retrouvons également ceux de M.-L Potet, D. Bert et N. Stouls (Potet *et al.,* 2004, Bert *et al.,* 2005) qui adoptent une technique complètement différente où il s'agit de définir des obligations de preuve pour l'étude des états et des transitions entre ces états. Vu la complémentarité entre leur approche et notre proposition pour la construction de diagrammes d'états à partir de spécifications B (chapitres 10 et 11) nous n'allons pas présenter les fondements de ce travail à ce niveau. Néanmoins, nous allons présenter une comparaison détaillée entre notre approche et l'approche par preuve au niveau du chapitre 11.

2.4.1 Aperçu et objectifs

Les travaux de dérivation de B vers UML s'inscrivent dans le cadre des travaux dont la vocation est de surmonter les limites des méthodes formelles. Plusieurs références discutent ces limites (Hall, 1990, Gaudel, 1991, Bowen *et al.,* 1995) et mettent en avant la difficulté de compréhension des notations sur lesquelles ces méthodes s'appuient. C'est principalement cette difficulté que les travaux de dérivation de B vers UML cherchent à combler. En génie logiciel, la compréhension des spécifications est indispensable, entre autres, lors des phases de validation et de certification. Par ailleurs, ces phases, qualifiées de "*validation externe*" par M.-C. Gaudel (Gaudel, 1991, Gaudel, 1995), ne sont pas aisées dans le contexte des méthodes formelles car elles sont généralement réalisées par des intervenants qui ne sont pas forcément familiarisés avec les notations mathématiques.

Validation interne vs validation externe. (Gaudel, 1995) distingue deux activités de validation de spécifications lors d'un processus de développement entièrement basé sur les méthodes formelles : la validation **interne** et la validation **externe**. Au niveau de la Fig. 2.8, extraite de (Gaudel, 1995), nous distinguons deux types de documents : (*i*) les besoins de l'utilisateur qui sont de nature informelle, car ils reflètent des aspects du monde réel, et (*ii*) les documents formels qui composent la spécification et le produit final.

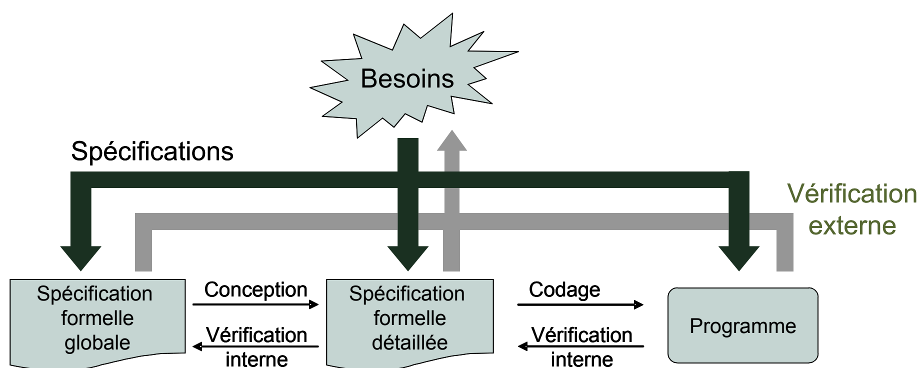


FIG. 2.8 – Schéma simplifié du processus de développement formel de logiciels extrait de (Gaudel, 1995)

L'activité de spécification, qualifiée par "*Top-level Specification*" au niveau de (Gaudel, 1991) est étroitement liée aux documents informels fournis par l'utilisateur. Tandis que l'activité de conception (correspondant à une activité de raffinement dans un processus de développement en B) ne porte que sur les différents documents formels. Par conséquent l'activité de *vérification interne* peut être complètement assistée par la preuve, ce qui permet de garantir que le programme final est conforme à sa spécification

(globale et détaillée). En revanche, une spécification formelle peut être incorrecte pour deux raisons fondamentales : l'une provenant d'une interprétation erronée des besoins de l'utilisateur et l'autre résultant de besoins mal exprimés. Ainsi, la vérification de la conformité des spécifications par rapport aux besoins de l'utilisateur est essentielle pour une validation externe de ces spécifications. Toutefois, les besoins de l'utilisateur étant informels, cette activité de *vérification externe* ne peut être réalisée par la preuve.

Validation externe de spécifications B. Dans le cadre d'un processus de développement en B, la vérification de la spécification par rapport aux besoins s'effectue en général manuellement, par relecture des sources des composants B et analyse de traçabilité par rapport au cahier des charges. Dans ce contexte, des techniques de test, notamment celle de S. Behnia dans (Behnia, 2000), ont été envisagées et visent, entre autres, une validation externe des spécifications formelles B. Les travaux de dérivation de B vers UML ciblent des objectifs similaires où cette validation externe est réalisée de manière informelle sur la base d'une documentation construite selon le standard UML. Ainsi, cette activité implique divers acteurs, principalement les utilisateurs finaux, et est assistée par des diagrammes visant l'aide à la compréhension de spécifications formelles.

Nous avons montré au moyen des deux autres catégories de dérivation que des similarités structurelles peuvent être identifiées entre UML et un sous-ensemble de constructions en B. Les travaux dont l'intention est de dériver des diagrammes UML à partir de spécifications par application de règles de traduction exploitent implicitement ces similarités. Dans ce contexte, J.-C Voisinet (Voisinet, 2004) propose une approche uniforme et systématique, mais qui manque de flexibilité et de souplesse. Contrairement à cette approche uniforme, H. Fekih *et al.,* (Fekih *et al.,* 2004, Fekih *et al.,* 2006) adhèrent à une approche interactive et incrémentale garantissant une grande souplesse lors de la construction de diagrammes mais qui sollicite fortement l'analyste. Dans notre travail, nous cherchons à définir une technique semi-automatique se situant entre l'approche uniforme et l'approche interactive. Dans ce qui suit, nous donnons un aperçu de ces deux techniques avant de présenter notre apport théorique et pratique pour la documentation graphique UML de projets en B.

2.4.2 L'approche uniforme

L'approche de J.-C Voisinet (Voisinet, 2004) est fondée sur un ensemble de règles permettant une traduction partielle de spécifications B en diagrammes de classes et d'états/transition de manière automatisée et systématique. Pour ce faire, la traduction est uniforme et sous-entend que pour chaque donnée B couverte par cette technique une seule transformation en UML existe. Dans cette perspective, l'intérêt porté à l'extraction de vues UML à partir de spécifications B est de permettre, à la personne chargée d'évaluer le modèle formel sur la base des représentations graphiques, de ne pas trop s'attacher à la recherche de la bonne visualisation. Une correspondance "un à un" est donc établie entre un sous-ensemble de constructions en B et celles de UML. De ce fait, le(s) diagramme(s) UML résultant des règles de traduction ne sont pas nécessairement les plus pertinents des points de vue lisibilité et compréhension.

2.4.2.1 Dérivation de diagrammes de classes

La construction de diagrammes de classes à partir de spécifications B selon l'approche de J.-C. Voisinet (Voisinet, 2004) permet de donner une vue d'ensemble estimée synthétique sans que cette vue

ne soit nécessairement proche de la structure des données spécifiées au niveau du modèle B de départ. Les règles proposées dans ce contexte sont, principalement, les suivantes :

Règle 1. Une machine abstraite M est transformée en une classe S.

Règle 2. Toutes les opérations deviennent des méthodes de la classe S.

Règle 3. Les variables entières et booléennes deviennent des attributs de la classe S.

Règle 4. Un ensemble abstrait A est traduit, d'une part par une classe E_A, et d'autre part, par une association entre E_A et S. Le nom de rôle du côté de E_A est A avec une multiplicité égale à $*$. Finalement, la multiplicité du côté de S est égale à 1.

Règle 5. Un ensemble énuméré D est représenté de façon similaire à un ensemble abstrait (*i.e.* par une classe E_D et une association entre E_D et S). Vu que le cardinal de D est connu, alors un attribut $values$ est rajouté dans E_D. Le type de cet attribut est D_VALUES avec D_VALUES est une classe ≪*enumeration*≫ indiquant les éléments de l'ensemble D.

Règle 6. Une variable a appartenant à un ensemble abstrait A est transformée en une agrégation entre S et E_A. Le nom de rôle du côté de E_A est a avec une multiplicité égale à 1.

Règle 7. Un ensemble B inclus dans un autre ensemble abstrait A est transformé par une association entre E_A et S. Le nom de rôle du côté de E_A est B avec une multiplicité égale à $*$. La multiplicité du côté de S est égale à 1.

Règle 8. Une relation R entre deux ensembles A et B est représentée par :
- Une classe E_A_B,
- Une association entre E_A_B et S. Le nom de rôle du côté de E_A_B est A_B avec une multiplicité égale à $*$ du côté de E_A_B et une multiplicité égale à 1 du côté de S,
- Une agrégation entre E_A_B et E_A, avec une multiplicité égale à 1 du côté de E_A.
- Une agrégation entre E_A_B et E_B, avec une multiplicité égale à 1 du côté de E_B.

MACHINE
  M
SETS
  A ; B ;
  $D = \{elem1, elem2, elem3\}$
VARIABLES
  a, d, R, C
INVARIANT
  $a \in A \wedge C \subseteq A \wedge$
  $d \in D \wedge R \in A \leftrightarrow B$
OPERATIONS
  Op1 $\widehat{=}$ ...
  Op2 $\widehat{=}$ ...
END

FIG. 2.9 – Machine B simple

Dans le but d'illustrer ces règles de traduction nous prenons l'exemple de la machine abstraite M ci-dessus où la structure est simplement définie par deux ensembles abstraits (A et B), un ensemble

énuméré (D) et quatre variables : éléments d'ensembles a et d, sous-ensemble C et relation R. La partie dynamique est composée des opérations Op1 et Op2. L'application de l'ensemble des règles précédentes produit le diagramme de classes de la Fig. 2.10 où 5 classes liées par de nombreuses associations sont construites.

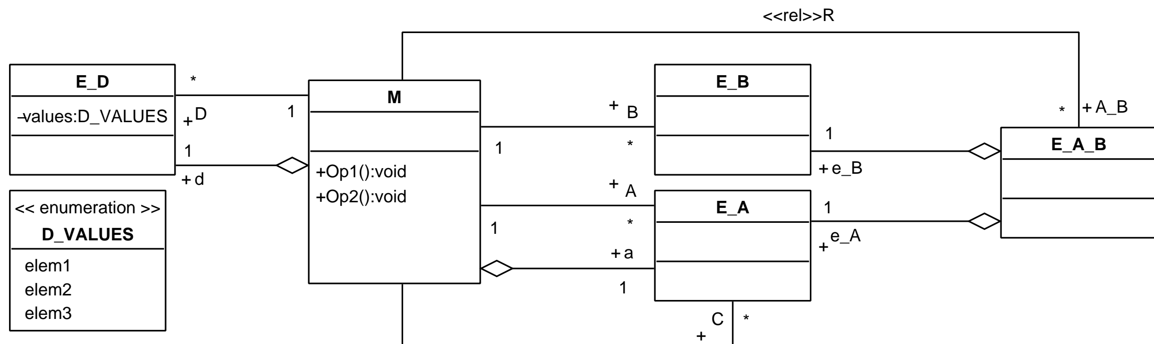


FIG. 2.10 – Application règles proposées par (Voisinet, 2004) à la machine M

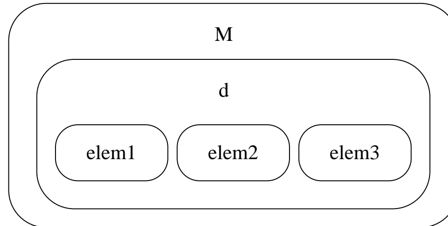
Discussion. La création d'une classe pour chaque machine abstraite B a l'avantage de pouvoir traiter la composition au niveau de la spécification de manière intuitive et simple. En effet, ceci est réalisé par des relations entre les classes issues directement des machines B. L'auteur propose de transposer les clauses d'assemblage de machines abstraites au niveau du modèle UML par des associations ou des agrégations entre ces classes. La visibilité des données B est tout simplement traduite par une visibilité *public* des attributs de classes. Bien que cette approche présente un moyen automatique pour la production de diagrammes de classes à partir de spécifications B, son inconvénient majeur est, nettement, la complexité des diagrammes générés. La traduction systématique des machines, des ensembles, des relations et fonctions en classes ne semble pas être le moyen le plus efficace pour produire une vue UML compréhensible. Cette technique conduit à une explosion du nombre de classes et d'associations au niveau du diagramme lorsque les spécifications sont de taille importante. En effet, nous remarquons que, d'un côté, la traduction d'un ensemble abstrait produit deux éléments de modélisation (*i.e.* une classe et une association), et d'un autre côté, la traduction d'une relation produit quatre éléments de modélisation (*i.e.* une classe, une association et deux agrégations). Cependant, ces transformations ne sont pas à exclure étant donné que d'un point de vue conceptuel les diagrammes construits par cette technique restent cohérents.

2.4.2.2 Dérivation de diagrammes d'états/transitions

Les règles de traduction de spécifications B en diagrammes de classes se basent uniquement sur les structures des données B : ensembles abstraits, ensembles énumérés, éléments d'ensembles . . . Ceci est justifié par le fait que le diagramme de classes illustre une vue statique de la spécification. Cependant, vu que la transformation des opérations est réalisée de manière uniforme en les mettant dans une seule classe, alors la génération de diagrammes d'états/transitions ne peut se rattacher à la description du comportement des différentes classes de la vue statique. En effet, les transitions entre états correspondent naturellement aux méthodes des classes. Par conséquent, les vues comportementales, produites par l'approche uniforme, sont soit rattachées à la classe encapsulant toutes les opérations, soit dérivées indépendamment du diagramme de classes. Dans les deux cas, ces diagrammes illustrent le comporte-

ment de la spécification B et non le comportement des classes de la vue structurelle.

Dans ce cadre, (Voisinet, 2004) traduit systématiquement une machine B par un super-état ayant le même nom que cette machine et se limite aux spécifications abstraites possédant des variables scalaires typées à l'aide d'ensembles énumérés. Chacune de ces variables, est par la suite traduite en un sous-état du super-état issu de la machine B et qui est lui-même un super-état dont les sous-états sont les valuations possibles de la variable. L'application de cette technique à la machine M (Fig. 2.9) produit la hiérarchie d'états suivante :



Ceci étant, la construction des transitions entre états se base sur une analyse syntaxique de l'initialisation et des opérations (ou événements) tout en distinguant différentes transformations selon les substitutions rencontrées :

1. La substitution $skip$ est traduite par une transition réflexive sur l'état courant ;
2. Les substitutions *devient tel que* " :()" et *devient élément de* ":$\in$" sont traduites par des transitions mono-source et multi-cibles.
3. Les substitutions devient égal ":=" sont traduites par des transitions mono-source et mono-cible.

Ci-dessous, nous présentons quelques exemples de transformations proposées dans (Voisinet, 2004). Notons que pour réduire la complexité des transitons conditionnées (conditions [P] ou [¬P]), une étape de simplification déterminant si les états sources de chaque transition satisfont les prédicats P ou $\neg P$ est suggérée.

Op $\widehat{=}$ BEGIN $v := val_i$ END	val_1 ... val_i ... val_n ; Op, Op, Op, Op, Op
Op $\widehat{=}$ SELECT P THEN $v := val_i$ END	val_1 ... val_i ... val_n ; Op[P], Op[P], Op[P], Op[P], Op[P]
Op $\widehat{=}$ IF P THEN $v := val_i$ END	Op[¬P], Op[¬P], Op[¬P] ; val_1 ... val_i ... val_n ; Op[P], Op[P], Op[P], Op[P], Op[P]

Discussion. Nous distinguons deux limites principales de cette approche : la première porte sur l'identifications des états, et la seconde concerne l'utilisation d'une démarche purement syntaxique pour l'étude des transitions :

(*i*) La restriction sur des variables v_i délimitées par des ensembles énumérés $v_i \in \{e_1, e_2, \ldots, e_n\}$, sous-entend que les systèmes étudiés sont des systèmes finis et que les états de la spécification B sont entièrement définis par des éléments d'ensembles. Cependant, l'intérêt d'une méthode formelle telle que B réside aussi dans sa capacité d'abstraction où les données ne se limitent pas nécessairement à des éléments d'ensembles. Rappelons que dans le cas d'une machine B, la notion d'état définie par J.-R. Abrial dans (Abrial, 1996) est représentée par les variables d'états et que celles-ci peuvent être indéterminées ou représentées par des sous-ensembles, des relations... C'est pourquoi nous aborderons un autre niveau de granularité dans notre technique où nous mettons en relief les notions d'états **concrets** et d'états **abstraits**.

(*ii*) L'utilisation d'une grammaire particulière pour définir un noyau de spécifications B acceptées par cette démarche, représente une restriction assez forte pour un passage à l'échelle. D'un autre côté, vu que les états traités sont complètement valués, alors les transitions entre ces états peuvent être obtenues grâce à un outil d'animation (*e.g.* l'animateur de l'atelier B) ou à un outil de model-checking (*e.g.* ProB (Leuschel *et al.,* 2003)) sans passer par une étude syntaxique. Par ailleurs, la technique de simplification des pré-conditions peut compliquer davantage le traitement des transitions, surtout quand les prédicats associés sont complexes.

2.4.3 L'approche interactive et incrémentale

Cette approche se base sur un ensemble de règles heuristiques ayant pour objectif de construire d'une manière interactive, et passant par plusieurs itérations, des diagrammes de classes et d'états/transitions à partir d'une spécification B. La Fig. 2.11, extraite de (Fekih *et al.,* 2006), illustre ce processus de transformation. Trois étapes sont définies et sont supportées par plusieurs incréments dépendant de l'appréciation accordée par l'analyste aux différents diagrammes. Ci-dessous nous citons les règles de transformations proposées par cette technique. Notons que ces règles sont extraites de (Fekih *et al.,* 2004) :

Règle 1. Si un ensemble abstrait apparaît dans le domaine de relations fonctionnelles avec d'autres ensembles alors il pourra être transformé en une classe.

Règle 2. Si un ensemble abstrait ou énuméré apparaît dans le co-domaine de certaines relations sans apparaître dans le domaine d'autres relations fonctionnelles, alors il pourra être transformé en un type d'attribut pour les classes représentant le domaine de ces relations.

Règle 3. La relation d'inclusion entre deux ensembles S et E peut représenter :
- le concept de généralisation entre une sous-classe S et une super-classe E,
- un état possible S d'une instance de la classe E,
- l'ensemble des instances effectives S d'une classe E.

Règle 4. Une relation $R \in A \leftrightarrow B$ entre deux ensembles abstraits A et B traduits en deux classes A et B peut être transformée en une association R entre ces classes ou en un attribut R de type B dans la classe A.

Règle 5. Les événements (ou opérations) représentent des méthodes dans les classes et/ou des transitions dans un diagramme d'états/transition.

Remarquons à partir des règles qui produisent des éléments de modélisation d'un diagramme de classes que l'intention de cette approche lors de la construction de vues statiques est d'illustrer les données spécifiées en explicitant les différentes relations (ou liens) entre ces données. Les vues structu-

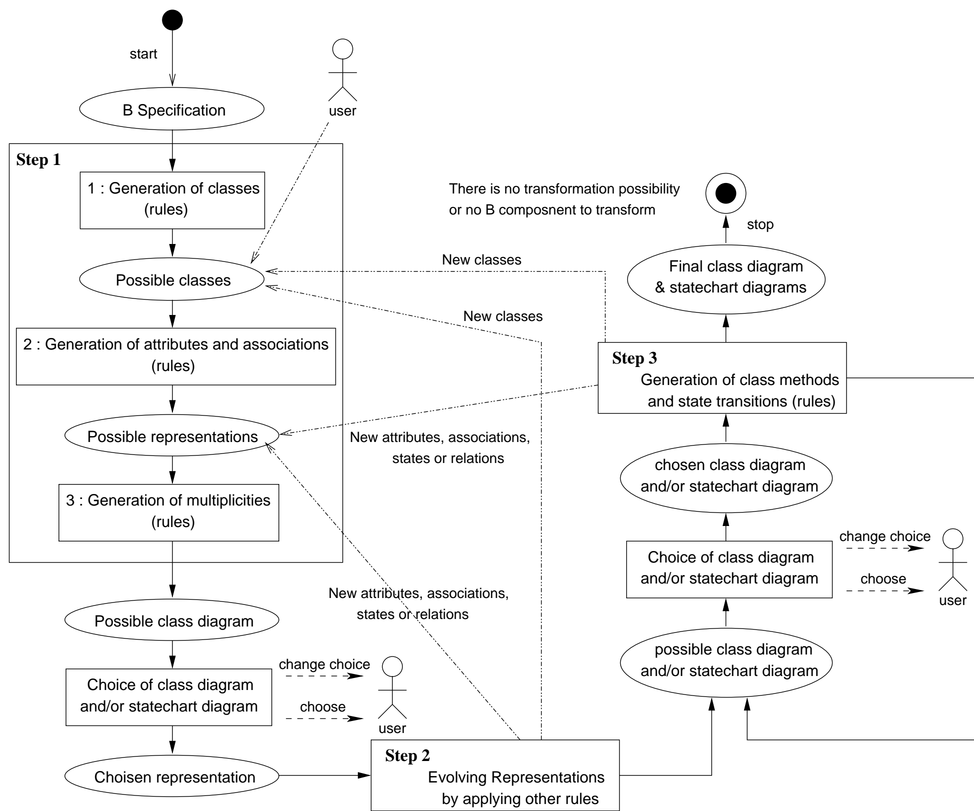


FIG. 2.11 – Processus de dérivation de B vers UML extrait de (Fekih *et al.,* 2006)

relles ainsi construites sont estimées proches de la structure des données B issues du modèle formel. Par exemple, l'invariant $R \in A \leftrightarrow B$ peut être traduit par deux classes A et B liées par l'association R. Le résultat ainsi obtenu est grandement plus intuitif que celui proposé au niveau de l'approche uniforme. En revanche, cette transformation particulière nécessite que B figure lui aussi en tant que domaine d'une autre relation fonctionnelle R' (ce qui n'est pas le cas pour la machine de la Fig. 2.9). Or, indépendamment du fait qu'un ensemble apparaisse dans le domaine ou le co-domaine d'une relation, un ensemble abstrait peut être vu comme une abstraction d'un ensemble d'objets et transformé par conséquent en une classe. La règle 2 apparaît donc comme trop restrictive. Finalement, nous notons que ces règles heuristiques, à vocation empirique, ont été adaptées au niveau de (Fekih *et al.,* 2006) pour permettre d'autres transformations. Par exemple la règle de transformation **Règle 1** est adaptée comme suit :

Règle 1′. Un ensemble abstrait T peut être transformé en une classe si :
(i) T apparaît dans le domaine d'une relation,
(ii) T apparaît comme un super-ensemble d'une inclusion,
(iii) l'utilisateur juge opportun de traduire T en une classe.

Discussion. Contrairement à (Voisinet, 2004) qui présente une technique uniforme et systématique de B vers UML, la technique de H. Fekih associe plusieurs transformations possibles à chaque construction en B. Par exemple, la relation d'inclusion est transformée de trois manières (**Règle 3**). Ainsi pour chaque construction B, l'analyste doit faire un choix parmi une liste de transformations possibles. La spécifica-

tion est donc ré-exprimée, d'une manière interactive, élément par élément. Cette conclusion est confirmée par la **Règle 1′** – étape (*iii*). L'avantage d'une telle démarche est qu'elle permet une grande souplesse pour obtenir des diagrammes lisibles et satisfaisants mais se prête difficilement à l'automatisation et au passage en vraie grandeur ; et ce, pour deux raisons principales :

(*i*) Les règles proposées restent informelles et ne traitent que partiellement les constructions en B. Ceci est dû au fait qu'aucune traçabilité sémantique entre B et UML n'est explicitement définie et que les transformations sont énoncées à partir d'expérimentations ;

(*ii*) Aucun guide méthodologique permettant d'assister l'analyste dans le choix des meilleures règles de traduction n'est présenté. L'analyste, étant sollicité à tous les niveaux et dans toutes les étapes du processus de transformation, ne pourra gérer des spécifications de tailles importantes.

2.4.4 Notre approche

Dans les propositions actuelles deux paradigmes opposés ont été mis en œuvre : soit assurer un fort niveau d'automatisation mais produire des diagrammes complexes, soit produire des diagrammes lisibles mais avec un fort niveau d'interaction. Dans notre travail, nous visons un compromis entre ces deux paradigmes. En effet, la solution que nous proposons réduit amplement l'interaction avec l'analyste tout en produisant des diagrammes lisibles.

Dans le but de dériver des diagrammes de classes UML à partir de spécifications B, nous proposons un processus guidé par un ensemble de correspondances structurelles et sémantiques de B vers UML ; et ce, dans une perspective de transformation dirigée par les méta-modèles. Nous proposons alors un méta-modèle du langage B permettant de définir un cadre conceptuel homogène pour la formalisation de ces correspondances. Dans notre démarche, nous nous focalisons, dans un premier niveau, sur la possibilité de représenter de manière explicite, différentes transformations jugées intéressantes pour un objectif de documentation. Enuite, et en vue de sélectionner la(les) transformation(s) adéquate(s), nous essayerons d'identifier des critères (dits de pertinence) permettant de réduire de manière significative l'interaction avec l'analyste. Ces critères sont formalisés à partir des facteurs suivants :

(*i*) La distribution des opérations sur les différentes classes,

(*ii*) Les dépendances entre opérations et données B,

(*iii*) Les liens entre données B elles-mêmes.

Sur cette base, nous proposons un algorithme de formation de concepts permettant de produire, de manière semi-automatique, des vues statiques satisfaisant ces critères de pertinence. Le cadre formel qui fonde notre approche nous permet une étude précise et rigoureuse des aspects structurels de la spécification B.

Quant à la construction de vues comportementales, nous exploitons deux techniques complémentaires : l'animation et la preuve. Notre proposition à ce niveau, se base dans un premier temps sur une exploration effective du comportement de la spécification B, puis dans un second, nous nous servons d'une technique d'abstraction de graphes pour construire des diagrammes d'états/transitions moins complexes et de manière automatique. Nous montrons également que le cadre formel que nous avons défini pour la construction de diagrammes de classes, présente un moyen efficace pour étudier le lien entre les vues comportementales et structurelles pouvant être dérivées de spécifications B.

2.5 Conclusion

UML et B sont deux techniques de spécifications reconnues en génie logiciel ; leur couplage est motivé par le souhait de pouvoir les utiliser ensemble dans un processus de développement de logiciels intégrant à la fois structuration et précision. Dans ce chapitre, nous avons présenté un aperçu de travaux de couplage de ces deux formalismes tout en mettant l'accent sur l'importance de combiner la lisibilité d'une méthode graphique telle que UML et la rigueur d'une méthode formelle telle que B pour un objectif de documentation de spécifications.

De UML vers B. Les travaux de dérivation de UML vers B visent, d'une part, l'utilisation d'UML comme point de départ du processus de développement de spécifications B, et d'autre part, l'utilisation des outils de B pour analyser et vérifier la correction des spécifications formelles qui en découlent. Ces spécifications peuvent également être exploitées pour vérifier des contraintes d'intégrité exprimées initialement en OCL (Ledang, 2002) ou encore pour produire du code source (*e.g.* SQL/JAVA) après une suite de raffinements prouvés (Mammar *et al.,* 2005) . . .

De B vers UML. Les travaux de B vers UML adoptent l'approche inverse : le point de départ est B et non une spécification semi-formelle. En effet, ces travaux visent la production de vues graphiques à partir de spécifications existantes entièrement développées en B. L'objectif principal est de rendre accessibles les spécifications B à des personnes qui ne seraient pas familiarisées avec les notations mathématiques, et permettre par conséquent une meilleure conduite de la phase de validation externe de spécifications.

Deuxième partie

Correspondances structurelles et sémantiques entre B et UML pour la dérivation de vues structurelles

Chapitre 3

Définition d'une syntaxe abstraite UML pour B

> « *C'est probablement une exigence de l'esprit humain d'avoir une représentation du monde qui soit unifiée et cohérente* [...]. *La science ne vise pas d'emblée à une explication complète et définitive de l'univers. Elle opère sur des phénomènes qu'elle parvient à circonscrire et définir.* »
>
> *François Jacob*
> « *Le jeu des possibles 1981* »

Sommaire

3.1 Introduction **55**
3.2 Méta-modélisation et transformation de modèles : aperçu et usage **56**
3.2.1 Transformation de modèles 57
3.2.2 Utilisation des méta-modèles pour la transformation B/UML : notre vision 57
3.3 Proposition d'un méta-modèle UML pour B **58**
3.3.1 Spécification de machines abstraites 59
3.3.2 Spécification de types en B 62
3.3.3 Spécification des structures de raffinement et de composition en B 65
3.4 Conclusion **66**

3.1 Introduction

Nous distinguons deux perspectives essentielles dans l'usage d'un langage de modélisation graphique comme UML : la **perspective logicielle** et la **perspective conceptuelle**. Dans la première perspective, les différents modèles et concepts UML correspondent à des éléments d'un système logiciel, et donc UML est perçu comme un langage de programmation bien qu'il soit incomplet pour cette finalité. Dans l'autre, l'objectif assigné à l'usage d'UML est d'aboutir à une description des concepts d'un domaine d'étude : les modèles et concepts UML ne correspondent pas alors à des éléments logiciels mais construisent un "*vocabulaire*" pour discuter d'un domaine particulier. C'est dans cette deuxième perspective que s'inscrivent nos travaux de thèse. En effet, notre usage d'UML ne s'inscrit pas dans une perspective logicielle pour plusieurs raisons :

◇ La phase de codage est actuellement assez couverte par un développement en B (Badeau *et al.,* 2004). Nous excluons, donc, l'usage d'UML comme langage de programmation.

◇ Les fondements conceptuels de B n'ont pas été élaborés dans une optique orientée objets. Ainsi, les intersections et incompatibilités entre B et UML font que nous ne pouvons pas traduire toutes les données B en des éléments de modélisation UML, ce qui nous contraint à identifier un sous-ensemble de constructions en B pour lesquelles une ou plusieurs contreparties en UML existent.

◇ Les vues UML que nous produisons ont pour principale finalité d'aider à la compréhension de spécifications. Nous ciblons ainsi les phases conceptuelles d'un cycle de développement classique et non les phases de codage. En effet, nous ne couvrons pas les composants IMPLEMENTATION (voir Fig. 3 page 7).

Par ailleurs, dans le cadre d'une perspective conceptuelle d'UML, (Fowler, 2003) établit une distinction entre UML comme **croquis** (*sketch*) et UML comme **modèle** (*blueprint*). Le premier usage d'UML a pour objectif de communiquer, d'une manière plutôt informelle et dynamique certains aspects du système. Quant au second usage, l'idée centrale en est que les modèles UML soient assez exhaustifs pour former une spécification suffisamment détaillée et complète.

Aussi, notre usage d'UML vise-t-il un objectif de communication qui se situe entre "croquis" et "modèle". UML est, par conséquent, perçu comme un ensemble d'illustrations pour lesquelles la communication prime sur la complétude. Nous n'allons pas produire des vues UML à partir de spécifications B pour construire une spécification équivalente du même système ; nos diagrammes sont plutôt exploratoires et reflètent les caractéristiques structurelles et comportementales des spécifications B.

Dans le but d'expliciter les correspondances entre les constructions de B prises en compte dans notre travail et UML nous commençons tout d'abord par définir un cadre conceptuel précis et homogène dans lequel les constructions issues de ces deux paradigmes sont définies par des méta-concepts. Nous proposons alors dans ce chapitre un méta-modèle UML pour la syntaxe B que nous découpons en trois parties principales : (*i*) spécifications de machines abstraites, (*ii*) spécifications de types en B et (*iii*) spécification des structures de raffinement et de composition.

3.2 Méta-modélisation et transformation de modèles : aperçu et usage

En vue de préciser les notions fondamentales de cette deuxième partie de notre travail de thèse nous commençons tout d'abord par définir les notions de **modèle** et de **méta-modèle**.

Nous retrouvons, dans la "littérature", plusieurs définitions du mot **modèle**, d'ailleurs l'une des plus récentes est celle de J. Bézivin et ses co-auteurs (Bézivin *et al.,* 2001) : "*un modèle est une abstraction d'un système conçue pour être plus facile à utiliser que le système lui-même et pour permettre de répondre à un certain nombre de questions sur celui-ci*".

La notion de modèle a, néanmoins, été introduite depuis de nombreuses années, et utilisée dans plusieurs domaines (les mathématiques, la physique, . . .), par exemple en 1968, Minsky (Minsky, 1968) atteste que "*Un objet O est un modèle d'une réalité R si O permet de répondre aux questions que l'on se pose sur R*". Ces différentes définitions conduisent, au final, à appréhender un modèle selon son pouvoir d'expression, et par conséquent, à considérer l'**interprétation** du modèle comme une activité fondamentale. En effet, H. Habrias distingue deux sens du mot modèle :

– **1er sens.** Attribution d'un sens à des énoncés formels de sorte qu'ils soient vérifiés.
– **2ème sens.** Association d'un énoncé formel à une "réalité empirique".

Dans ce contexte, H. Sinaceur dans (Lecourt, 1999) indique que : "*Les deux sens du concept de modèle ne sont que les deux faces complémentaires d'une même activité :* ***interpréter****. Interpréter est inéluctable qu'il s'agisse d'interpréter un formalisme, ou inversement d'interpréter mathématiquement un ensemble de données. D'une part, parcequ'un langage qui n'aurait pas de modèle n'a aucun intérêt, d'autre part et réciproquement, parce que l'expression n'est pas le miroir de l'expérience.*"[21]

Quant à la notion de **méta-modèle**, elle est définie par (Bézivin *et al.,* 2001) comme suit : "*Un méta-modèle est une spécification explicite d'une abstraction. Cette spécification permet d'identifier l'ensemble des concepts utilisés pour exprimer des modèles ainsi que les différentes relations entre ces concepts. En d'autres termes, le méta-modèle définit la terminologie à adopter pour définir des modèles.*"

3.2.1 Transformation de modèles

La transformation de modèles est une technique qui s'inscrit dans le domaine de l'ingénierie dirigée par les modèles et est souvent associée à l'approche MDA[22] (OMG, 2003, Blanc, 2005). En effet, la mise en œuvre d'une MDA est entièrement basée sur la définition de modèles et de leurs transformations. Pour ce faire, l'OMG a proposé une formalisation et une standardisation des techniques de transformation de modèles pour garantir la compatibilité entre les outils MDA. Dans cette optique, l'OMG fournit le langage MOF[23] (*Meta-Object Facility* ou langage de méta-modélisation) comme langage d'échange de constructions utilisées par les modèles. L'intérêt de ce langage est de faire interopérer des méta-modèles différents ce qui permet de manipuler les modèles à l'aide d'opérations génériques sans connaissance du domaine. Dans ce contexte l'OMG propose de définir la transformation de modèles par des projections entre méta-modèles représentant des concepts de domaines variés. Cependant, l'OMG ne fournit pas avec le MOF de mode d'emploi. Il revient alors à chaque utilisateur de définir sa propre organisation de mise en œuvre du MOF en fonction de ses besoins (Marvie, 2002).

La Fig. 3.1, ci-dessous, présente un exemple simple de transformation basée sur la projection de méta-modèles représentant le modèle objet et le modèle relationnel. Cette transformation est basée sur deux règles de correspondances structurelles entre les méta-modèles objet et relationnel :

Règle 1. À chaque classe UML correspond une table dans le modèle relationnel.

Règle 2. À chaque attribut atomique (mono-valué et non composé d'autres attributs) d'une classe UML correspond une colonne dans une table dans le modèle relationnel.

3.2.2 Utilisation des méta-modèles pour la transformation B/UML : notre vision

Nous n'allons pas détailler les notions relevant de l'approche MDA, ni les techniques de transformation de modèles, nous allons plutôt nous inspirer de ces approches pour présenter un cadre conceptuel cohérent permettant, par la suite, d'expliciter différents schémas de transformation d'un modèle B en un modèle UML. Dans (Blanc, 2005) une transformation de modèles est perçue comme un ensemble de règles dont l'exécution se fait au niveau des modèles, alors que la spécification se fait au niveau des

[21] Citation reprise de (Habrias, 2006).
[22] *Model Driven Architecture*.
[23] MOF est un standard de méta-modélisation proposé par l'OMG (www.omg.org/mof/).

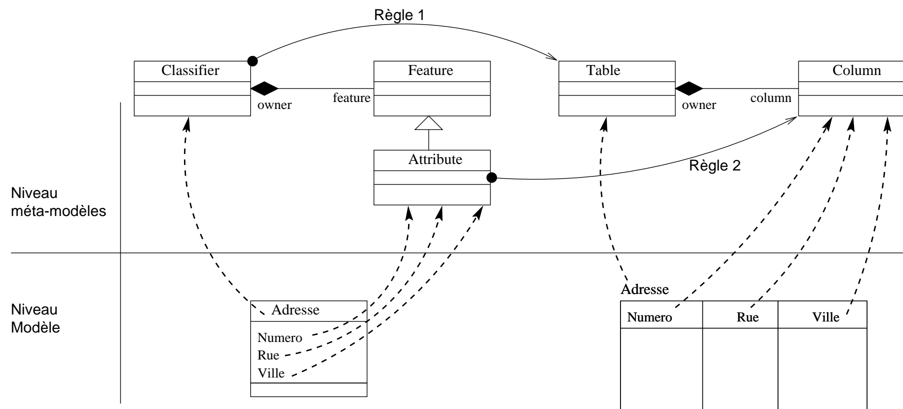


FIG. 3.1 – Exemple de transformation de modèles : du modèle Objet au modèle relationnel

méta-modèles. C'est dans cette optique que nous allons orienter notre travail. En effet, nous allons proposer un méta-modèle pour B en vue de pouvoir délimiter l'ensemble des correspondances possibles entre constructions B et UML. Nous adoptons alors une approche *interprétée* dont l'objectif est, d'une part, de cerner les constructions en B pouvant être traduites en UML, et d'autre part, de donner une spécification complète de la transformation que nous proposons. Ainsi, le présent chapitre est une introduction à la partie supérieure de la Fig. 3.2 (la définition de la transformation par établissement de liens entre méta-modèles sera présentée au niveau des chapitres 4 et 5). Nous laissons la partie *calculée* de notre travail aux chapitres 6 et 7.

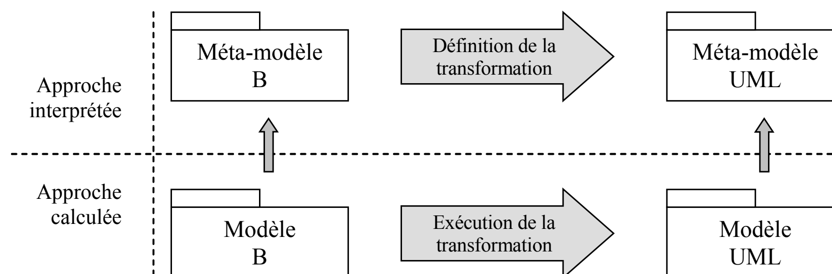


FIG. 3.2 – Processus de transformation dirigé par les méta-modèles UML et B.

3.3 Proposition d'un méta-modèle UML pour B

Le méta-modèle que nous proposons pour B donne une vue structurelle de haut niveau reprenant les notions définies au niveau de la syntaxe B. Nous nous en servons ici pour donner une vision globale des dépendances entre les différentes constructions du langage B.

Dans un premier temps, nous procéderons à la spécification du méta-modèle correspondant à une machine abstraite B (Fig. 3.6), ensuite nous proposerons un méta-modèle exprimant le typage en B (Fig. 3.11), et finalement, nous terminons par le méta-modèle relatif aux compositions et aux raffinements (Fig. 3.12). Ces méta-modèles seront étayés par une description des principales méta-classes ainsi que des principales contraintes d'utilisation (ou règles de bonne formation). Pour des raisons de clarté, nous décrirons ces contraintes d'utilisation d'une manière informelle.

3.3.1 Spécification de machines abstraites

Comme indiqué au niveau du chapitre 1 (page 13), une machine abstraite, représentée par la méta-classe *BMachine*, est constituée de plusieurs clauses permettant de spécifier les parties statiques et dynamiques du système. La partie statique correspond aux déclarations d'ensembles, de constantes et de variables et une caractérisation de ces données en termes de propriétés des constantes (clause PROPERTIES) et d'invariants. Nous ne faisons pas cette distinction par clauses au niveau du méta-modèle relatif aux machines abstraites car nous ne trouvons pas de contrepartie en UML. Il est, certes, possible de proposer une extension d'UML pour la prise en compte d'une telle structuration. Cependant, nous limitons notre étude, à ce niveau, aux constructions de base d'UML en vue de produire des diagrammes simples et compréhensibles. C'est pourquoi nous n'évoquerons au niveau de notre méta-modèle que les constructions en B susceptibles d'être transformées en UML. Nous définissons la méta-classe abstraite *BData* pour spécifier toutes les données déclarées au niveau d'une machine B (ensembles abstraits, constantes et variables). Quant à la partie dynamique, elle est spécifiée par les deux méta-classes *BOperation* et *BInitialisation* qui permettent de représenter respectivement les opérations et l'initialisation d'une spécification B. Notons que le méta-modèle complet relatif à la spécification des machines abstraites et présenté au niveau de la Fig. 3.6. Toutefois, nous allons en présenter certains morceaux ci-dessous pour illustrer les différentes définitions.

BMachine (Fig. 3.3). Représente une machine abstraite B. Elle est composée d'un ensemble de données B (méta-classe *BData*), d'un ensemble d'opérations (méta-classe *BOperation*), d'un ensemble d'invariants[24] (méta-classe *BInvariant*), d'un ensemble de paramètres (méta-classe *BMachineParam*), et finalement de l'initialisation (méta-classe *BInitialisation*). Nous la définissons pour constituer la spécification d'un module B.

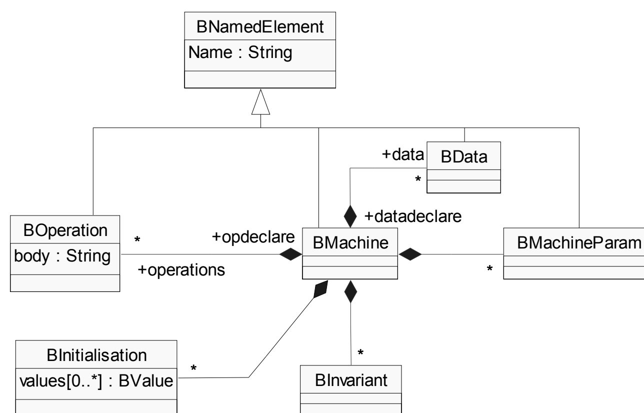


FIG. 3.3 – Composition de la méta-classe *BMachine*

BOperation (Fig. 3.4). Spécifie les opérations d'une machine abstraite B. Chaque opération est composée de paramètres (*BOpParameter*) et de substitutions (*BSubstitution*). Chaque accès d'une *BOperation* à une *BData* est qualifié par un type d'accès (attribut *Kind*). Cette notion d'accès sera traitée plus finement dans le chapitre 9. Néanmoins, nous notons que l'accès d'une *BOperation* $\mathcal{O}$ à une *BData* $\mathcal{D}$ en *précondition* correspond au fait que $\mathcal{D}$ apparait dans le corps d'une instance de *BPrecondition* (voir Fig. 3.4)

[24] Par invariant nous indiquons aussi bien la clause INVARIANT que la clause PROPERTIES.

liée à une substitution (Méta-classe *BSubstitution*) de $\mathcal{O}$. Dans le cadre d'un développement B construit par composition, il peut y avoir des appels entre opérations (cas d'une inclusion entre machines (Fig. 3.12)). Ces appels ainsi que les règles de visibilité associées seront spécifiés dans la section 3.3.3. Au niveau de la syntaxe B nous distinguons plusieurs contraintes d'utilisation des opérations telles que :

(*i*) Les paramètres formels d'une opération doivent être deux à deux distincts ;

(*ii*) Les paramètres d'entrée d'une opération doivent être typés dans le prédicat pré-condition. Ce qui signifie que si une opération est paramétrée alors une pré-condition lui est associée ;

(*iii*) Les paramètres sont distincts des variables et des constantes de la *BMachine*.

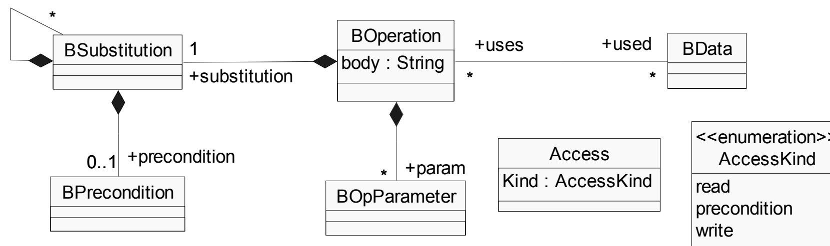


FIG. 3.4 – Dépendances et composition de la méta-classe *BOperation*

BPrecondition. Cette méta-classe ne spécifie pas uniquement la précondition associée à une substitution préconditionnée. En effet, elle prend en compte des prédicats associées à d'autres substitutions. Nous disons que P est une instance de la méta-classe *BPrecondition* dans les cas suivants :

IF P THEN S ELSE T END
ASSERT P THEN S END
PRE P THEN S END
SELECT P THEN S END
ANY z WHERE P THEN S END

BData (Fig. 3.5). Définit les variables, constantes et ensembles abstraits déclarés et spécifiés au niveau des clauses SETS, CONSTANTS et VARIABLES.

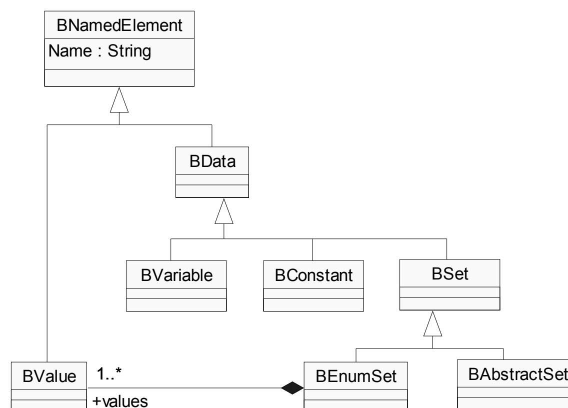


FIG. 3.5 – Spécialisations de la méta-classe *BData*

Les contraintes d'utilisation de la méta-classe *BData* sont principalement associées aux spécialisations *BConstant* et *BSet*. Il s'agit précisément de restreindre l'accès des opérations à un accès en *précondition* ou en *lecture* quand il s'agit d'un accès à un *BSet* ou à une *BConstant*.

(*iv*) Si une instance de la classe associative *Access* est créée entre une opération quelconque (instance de *BOperation*) et une instance de *BSet* ou de *BConstant*, alors son attribut *Kind* vaut "*read*" ou "*precondition*".

BNamedElement. Représente tous les éléments du modèle qui doivent être nommés au niveau de la spécification : machines, opérations, données B et paramètres.

BInitialisation. Définit l'initialisation des variables de la machine abstraite. Une instance de *BInitialisation* correspond à une substitution généralisée qui permet d'associer une (ou plusieurs) valeur(s) (*BValue*) de départ pour une variable (*BVariable*) de la machine. Bien qu'il soit possible d'utiliser dans l'initialisation toutes les sortes d'instructions, la manière usuelle d'initialiser les variables consiste à utiliser des substitutions devient égal ":=". Nous parlons alors d'initialisation déterministe. Dans ce cas, la valeur initiale de la variable est identifiée par l'attribut `values[0..*] : BValue` de la méta-classe *BInitialisation*. Le caractère multi-valué de l'attribut *values* permet de considérer le cas d'une initialisation non-déterministe (*e.g.* substitution devient appartient ":∈" ou devient tel que ":()") et telle que les valeurs initiales possibles d'une variable sont explicitement définies (*e.g.* une variable v initialisée par $v :\in E$ où E est un ensemble énuméré). Lorsque ces valeurs ne sont par définies explicitement au niveau de la spécification, l'attribut *values* n'est pas instancié vu que c'est un attribut optionnel.

BSet. Spécifie les ensembles déclarés au niveau de la clause SETS d'une machine abstraite. Nous distinguons deux spécialisations de la méta-classe *BSet* : la méta-classe *BEnumSet* et la méta-classe *BAbstractSet* (Fig. 3.5). Ces sous-classes de *BSet* caractérisent respectivement les ensembles énumérés (*BEnumSet*) et les ensembles abstraits (*BAbstractSet*).

BValue. Représente l'ensemble des valeurs pré-définies par le langage B (par exemple les valeur TRUE et FALSE de l'ensemble BOOL des booléens) ou définies au niveau d'une machine abstraite par énumération.

BPredicate. Classe abstraite représentant les prédicats associés aux invariants et aux préconditions des substitutions. Étant donné que nous nous intéressons à la structure des machines B nous mettons en évidence les données et valeurs qui composent un prédicat.

La Fig. 3.6 illustre le méta-modèle que nous proposons en vue de représenter la spécification des machines abstraites en B. Notons que nous ne détaillons pas les notions de *BSubstitution* et de *BExpression* étant donné que nous n'allons pas nous en servir au niveau de notre travail. Nous mettons en évidence à ce niveau les constituants d'une machine abstraite et les dépendances entre ces constituants sans nous attacher aux détails ni à la manière dont on peut former les substitutions généralisées. L'accès d'une *BOperation* o à une *BData* d peut être affiné au moyen d'un lien entre la(les) *BSubstitution*(s) qui composent l'opération o et la donnée d si ces *BSubstitution*s utilisent d. Pour faciliter l'expression des schémas de transformation de B vers UML que nous présenterons dans le chapitre suivant nous nous contentons du lien *Access* entre *BOperation*s et *BData*s.

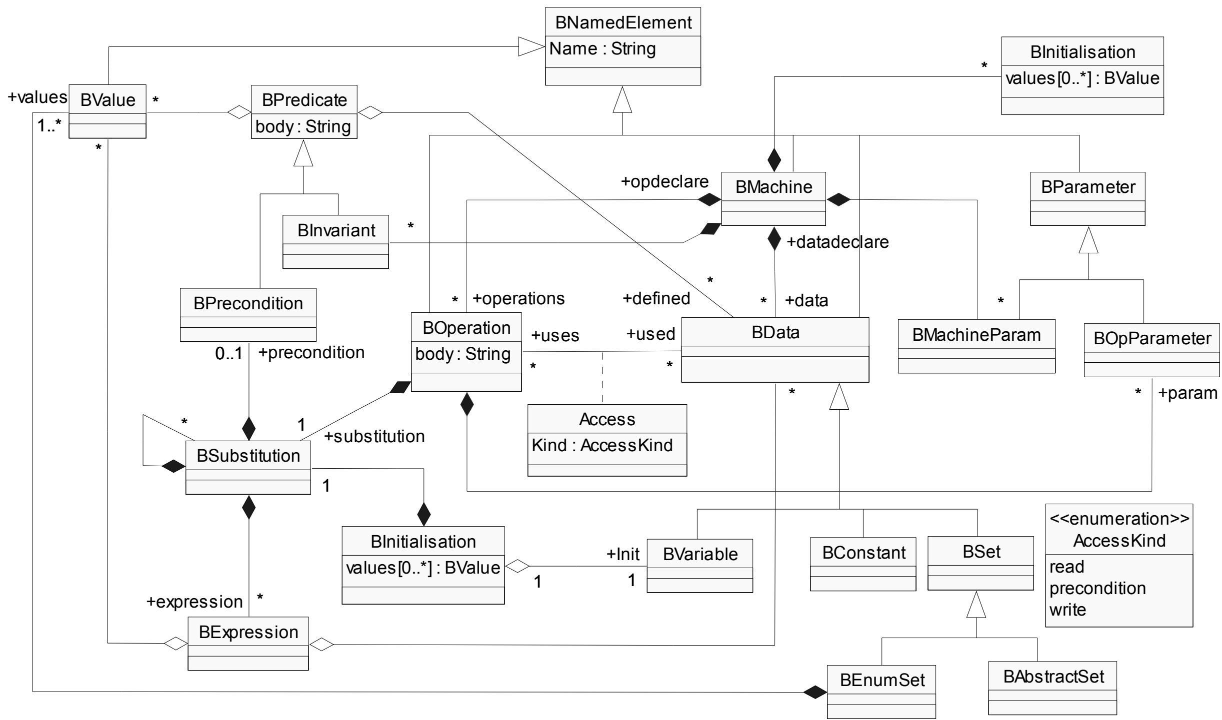


FIG. 3.6 – Méta-modèle UML pour la spécification de machines abstraites B

3.3.2 Spécification de types en B

Cette partie se base sur les définitions de types présentées au niveau de la section 1.2.2 (page 16). Nous faisons le choix ici de représenter une syntaxe abstraite issue des constructions les plus couramment utilisées au sein des machines B. En effet, nous n'illustrons pas à un méta-niveau les fondements (ou axiomes) de B. Par exemple, une variable R peut être définie au niveau de l'invariant par :

$$R = \{r | r \in \mathbb{P}(E_1 \times E_2) \wedge \forall x, y, z \cdot (x, y \in r \wedge x, z \in r \Rightarrow y = z)\}$$

avec E_1 et E_2 deux ensembles abstraits (instances de *BAbstractSet*). Cependant, une telle variable est couramment définie par : $R \in E_1 \nrightarrow E_2$ et correspond à une fonction partielle. Notre méta-modèle convient donc à cette deuxième construction de R. La Fig. 3.11 présente le méta-modèle complet correspondant aux spécifications de types en B.

BTypedElement. Regroupe les éléments typés d'une machine abstraite : les paramètres, les constantes, les variables ainsi que les valeurs pré-définies par le langage ou définies par énumération.

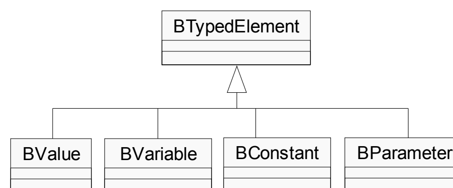


FIG. 3.7 – Spécialisations de *BTypedElement*

TypingPredicate (Fig. 3.8). Définit les prédicats de typage en B à partir des opérateurs particuliers d'égalité, d'appartenance et d'inclusion. Rappelons que dans la théorie B le typage est simplement réalisé par l'appartenance à un ensemble. Cet ensemble peut être un ensemble de base, ou défini par un constructeur de type ("$\times$","$\mathbb{P}$"). Toutefois, étant donné que nous ne représentons pas la théorie B, mais plutôt les notions directement utilisées au sein des machines B, nous faisons la distinction entre les divers opérateurs de typage (la méta-classe *TypingOperator*). Dans le cadre d'une *BValue* un prédicat de typage implicite est défini par une appartenance. Un prédicat de typage *TypingPredicate* est un prédicat B défini par la méta-classe *BPredicate*. Dans le cadre d'une opération paramétrée le *TypingPredicate* représente une partie de la *BPrecondition* associée à la substitution de l'opération en question (voir Fig. 3.4) et permet de typer les paramètres de l'opération.

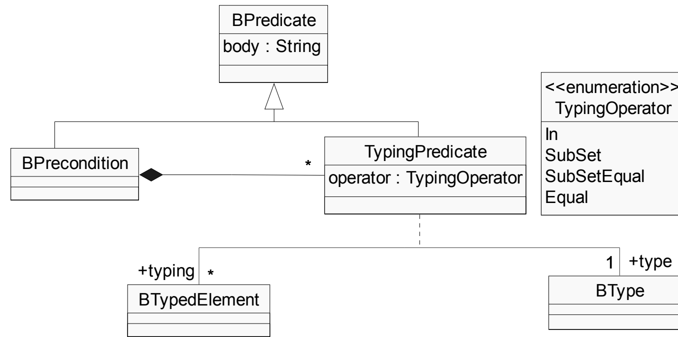


FIG. 3.8 – Spécification du mécanisme de typage

BBasicType. Représente les types de base du langage B. Ces derniers sont constitués des types prédéfinis tels que les types BOOL, Z, ... mais également des ensembles abstraits et des ensembles énumérés (méta-classe *BSet*). Les ensembles abstraits sont utilisés pour désigner des objets dont on ne veut pas définir la structure au niveau d'une abstraction. Ils sont alors définis uniquement par leurs noms. Quant aux ensembles énumérés ils sont définis aussi bien par leurs noms que par la liste ordonnée et non vide de leurs éléments énumérés. Les énumérés littéraux d'un ensemble S défini par énumération sont représentés par l'ensemble des instances de la méta-classe *BValue* liées à l'instance S de la méta-classe *BEnumSet*. La contrainte de bonne formation de cette classe définit le fait que les éléments d'un *BEnumSet* S sont typés uniquement par l'ensemble énuméré S.

(v) Lorsque le *BTypedElement* d est une *BValue*, et telle que d est liée par un lien "*values*" à un *BEnumSet* S, alors le *BType* associé à d via un lien *TypingPredicate* est S.

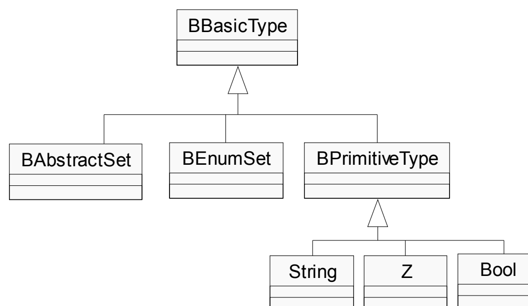


FIG. 3.9 – Spécialisations de *BBasicType*

BComposedType. Nous désignons par « type composé » tout type défini à partir d'au moins deux autres types. Il s'agit là précisément de types construits par des expressions de relations fonctionnelles ("↔","→", etc) ou par un produit cartésien. Chaque *BComposedType* est donc défini par les *BType* qui composent son domaine (nom de rôle **`+dom`**) et son co-domaine (nom de rôle **`+ran`**). Des imbrications de types composés peuvent ainsi être représentées. La méta-classe *Multiplicity*, est définie particulièrement pour désigner les relations fonctionnelles. Bien que ces multiplicités ne soient pas explicitement définies en B, nous les introduisons au niveau du méta-modèle pour leur adéquation à la transformation de relations B en associations UML. Ces multiplicités sont présentées au niveau du tableau 3.1 et présentent différentes contraintes d'utilisation établissant le lien entre l'expression utilisée pour définir le type composé et les bornes supérieures et inférieures des multiplicités du domaine et du co-domaine.

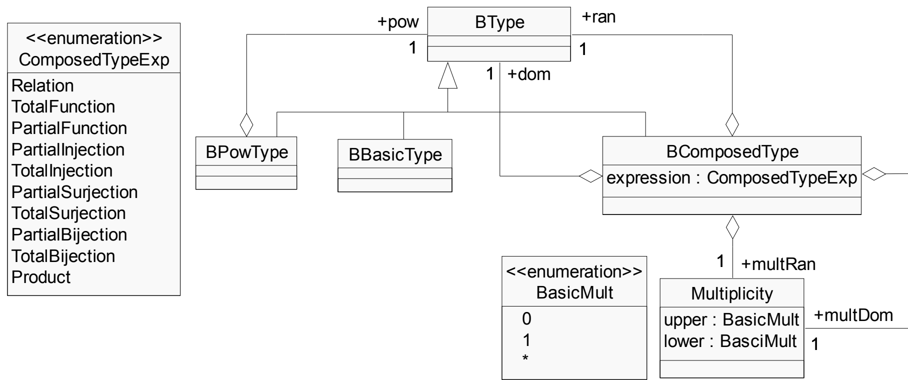


FIG. 3.10 – Spécification des types composés

		`multDom`		**`multRan`**	
`ComposedType`	**`expression`**	**`lower`**	**`upper`**	**`lower`**	**`upper`**
Relation	↔	0	*	0	*
Fonction partielle	⇸	0	*	0	1
Fonction totale	→	0	*	1	1
Injection partielle	⤔	0	1	0	1
Injection totale	↣	0	1	1	1
Surjection partielle	⤀	1	*	0	1
Surjection totale	↠	1	*	1	1
Bijection partielle	⤗	1	1	0	1
Bijection totale	⤖	1	1	1	1

TAB. 3.1 – Table des multiplicités associée aux spécialisations de relations fonctionnelles en B

BPowType. Correspond à un Type construit par le constructeur de type $\mathbb{P}$ (ensemble des sous-ensembles) appliqué à un *BType*. Des spécialisations de cette méta-classe peuvent être définies : $\mathbb{P}_1$ (ensemble des sous-ensembles non vides), $\mathbb{F}$ (ensemble des sous-ensembles finis) et $\mathbb{F}_1$ (ensemble des sous-ensembles finis non vides).

La Fig. 3.11 présente le méta-modèle proposé pour la spécification des types en B. La distinction entre types de base et types composés nous permettra par la suite de distinguer entre deux familles de schémas de transformation : les transformations de base et les transformation composées. Les transformations de base présentent le niveau de granularité le plus fin car elles sont établies à partir de la spécification des *BBasicType*, alors que les transformations composées sont établies sur la base de *BCom-*

posedType et peuvent correspondre à une composition de transformations de base. Notre objectif par là est de distinguer à partir de cette spécification abstraite un ensemble de traductions élémentaires, de telle sorte que toute composition de types peut être traitée. Ainsi, des constructions B complexes peuvent être transformées en UML en les ramenant à des constructions traduisibles par ces traductions élémentaires.

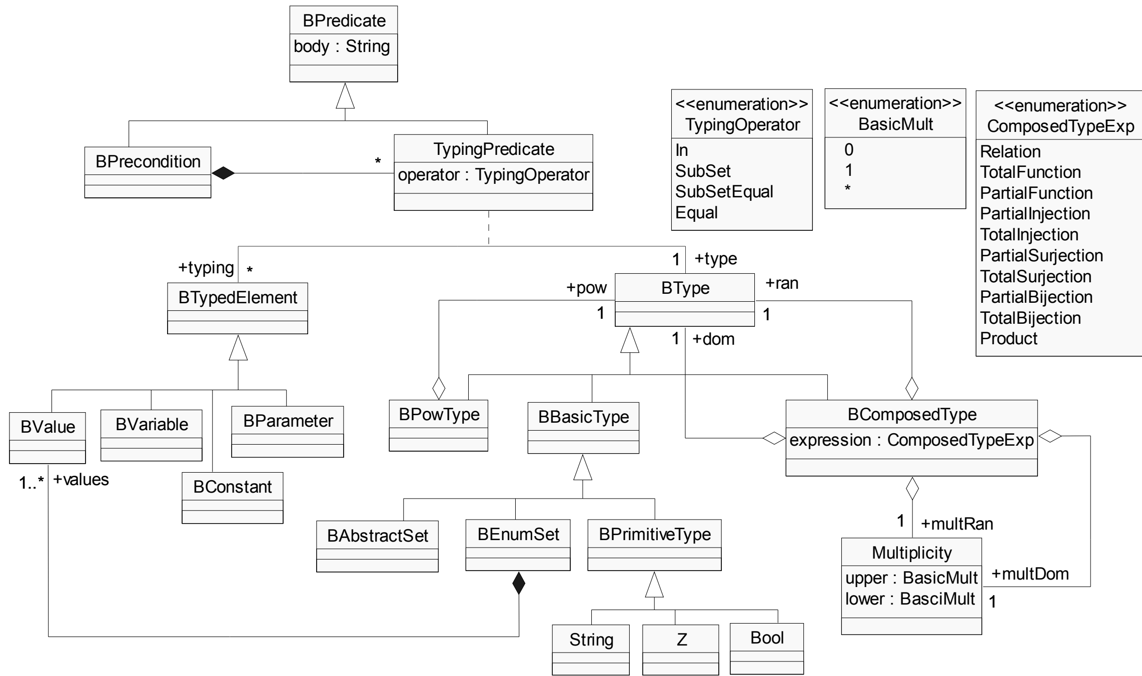


FIG. 3.11 – Méta-modèle UML pour la spécification de types en B

3.3.3 Spécification des structures de raffinement et de composition en B

À ce niveau, nous présentons, uniquement les liens INCLUDES et REFINES entre machines abstraites. La spécification des compositions et des raffinements au niveau du méta-modèle correspondant est plutôt centrée sur les contraintes d'utilisation des méta-classes *BMachine*, *BOperation* et *BData*. Ces contraintes sont des contraintes structurelles définies à partir de la syntaxe abstraite que nous proposons pour B.

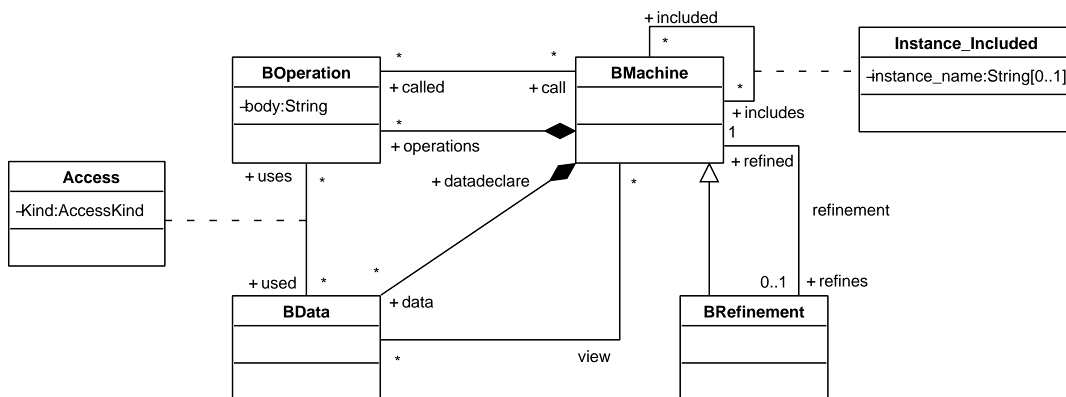


FIG. 3.12 – Spécification des structures de raffinement et de composition en B

Les notions d'invariant de collage, de préservation de cohérence entre raffinements, etc, ne sont pas spécifiées au niveau du méta-modèle. En effet, le méta-modèle que nous proposons présente une vue structurelle des dépendances entre constructions B.

Instance_Included. Représente l'instance d'une machine abstraite incluse. L'attribut *instance_name* représente le nom de cette instance. Les noms de rôle `+includes` et `+included` sont associés respectivement à la machine incluante et à la machine incluse. Un lien entre une *BOperation* et une *BMachine* définissant les rôles `+call` et `+called` n'est possible que dans le cas ou un lien *Instance* entre la *BMachine* appelant l'opération et la *BMachine* qui la définit.

(*vi*) Une *BOperation* $\mathcal{O}$ peut être appelée (`+called`) par une *BMachine* $\mathcal{M}$, si $\mathcal{O}$ est déclarée dans une machine $\mathcal{M}'$ et telle que $\mathcal{M}'$ joue le rôle de machine incluse (`+included`) dans $\mathcal{M}$.

BRefinement. Illustre un raffinement B sous forme d'une spécialisation de *BMachine*. En effet, un raffinement peut être vu comme une machine abstraite particulière liée par la clause REFINES à une autre machine abstraite. Les noms de rôle `+refines` et `+refined` sont associés respectivement au composant raffinant et au composant raffiné.

(*vii*) Une machine abstraite ou un raffinement ne peuvent inclure que des machines abstraites. En effet, un *BRefinement* ne peut être inclus dans une *BMachine*.

(*viii*) Dans le cas d'un *BRefinement*, les opérations du raffinement doivent avoir les mêmes noms que celles de la machine raffinée.

Relation view. Définit un référencement de données d'une *BMachine* incluse par une *BMachine* incluante. L'utilisation des données (association "*view*") est spécifiée par les règles de visibilité. Par exemple, une machine incluante ne peut accéder aux données de la machine incluse qu'en consultation ou via les opérations de la machine incluse. Ces règles sont présentées au niveau de la section 1.4 page 24.

(*ix*) Une machine B ne peut référencer des données venant d'autres machines que si un lien d'inclusion ou de raffinement est établi.

Les liens SEES et USES sont similaires au lien d'inclusion et sont naturellement définis par une association réflexive analogue à l'association *Instance_Included*. Pour ne pas encombrer le diagramme, nous nous sommes limité à la représentation de la clause INCLUDES.

3.4 Conclusion

Ce chapitre a présenté une introduction à notre approche de transformation que nous allons spécifier à un méta-niveau et a été consacré à la présentation d'un méta-modèle UML pour B. L'objectif d'une telle représentation est de disposer d'un cadre conceptuel dans lequel nous traitons les correspondances entre UML et B de manière homogène. Nous n'avons pas détaillé toutes les règles de bonne formation, nous nous sommes contenté de celles qui serviront pour la suite de notre approche.

Chapitre 4

Transformation des structures de base

« *L'image, en somme, est vue, non pas comme le lieu de la signification, mais comme un instrument de figuration de la signification. L'imagerie, lorsqu'elle accompagne les processus de compréhension, élabore des produits cognitifs optionnels, dont la nature et la structure restent foncièrement distinctes de celles des représentations qui codent la signification de l'énoncé.* »

Michel Denis

« Presses universitaires de france », 2ème édition, 1994.

Sommaire

4.1 Introduction **68**

4.2 Transformation de B vers UML : Illustration par l'exemple **68**

4.3 Transformation de machines abstraites **70**

4.3.1 Présentation 70

4.3.2 Discussion 72

4.4 Transformation des types de base (*BBasicType*) **73**

4.5 Transformation des éléments typés par un *BBasicType* **75**

4.5.1 Cas de l'appartenance 75

4.5.2 Cas de l'inclusion 77

4.5.3 Cas de l'égalité 79

4.6 Transformation des éléments typés par un *BPowType* **79**

4.7 Transformation des éléments typés par un *BComposedType* **81**

4.7.1 Relations et fonctions simples 81

4.7.2 Produit cartésien 84

4.8 Composition de relations et fonctions **84**

4.8.1 Composition de relations et fonctions avec un *BPowType* 84

4.8.2 Composition de relations et fonctions avec un produit cartésien 87

4.9 Transformation des opérations d'une machine abstraite **89**

4.10 Bilan et discussion **91**

4.1 Introduction

Dans le but de pouvoir pallier les difficultés rencontrées par les démarches existantes (Voisinet, 2004, Fekih *et al.,* 2004, Fekih *et al.,* 2006), plus particulièrement, l'absence d'une base conceptuelle clairement définie, nous adoptons une démarche par méta-modélisation et transformation de modèles. Nous allons alors nous appuyer sur les méta-modèles de B (Chapitre 3) et de UML[25] (OMG, 2005) en vue de spécifier la traçabilité entre B et UML et présenter ainsi un catalogue de schémas de transformation. Nous définissons la structure de ces derniers comme suit :

Définition 4.1 *Un schéma de transformation représente un ensemble de correspondances cohérentes dont les entrées sont spécifiées par des méta-concepts B et les sorties par des méta-concepts UML. La structure d'un schéma de transformation est la suivante :*

Schéma *i.*
Paramètres
Pré-condition
Transformation

La liste des paramètres d'un schéma de transformation sont des instances de méta-concepts B répondant à des conditions d'entrées (ou pré-conditions). La partie "Transformation" *décrit la traduction des constructions B passées en paramètre lorsque le schéma de transformation est exécuté.*

Dans ce chapitre, nous présentons un ensemble de transformations portant sur les structures que nous qualifions de base. Nous allons discuter la pertinence de chaque transformation tout en définissant le schéma correspondant. Notons que les liens entre méta-modèles seront illustrés pour les cas les plus importants à notre sens, et pour lesquels nous jugeons opportun de mettre en évidence ces correspondances.

4.2 Transformation de B vers UML : Illustration par l'exemple

Nous présentons dans cette section un exemple de spécification en B permettant à la fois d'illustrer différentes transformations possibles d'un modèle B en un modèle UML et aussi d'expliciter un ensemble de schémas de transformation basés sur les méta-modèles B et UML. Nous ne mettons pas en avant la notion de règles de transformation comme souvent évoqué par l'approche MDA, étant donné que la transformation que nous proposons dépend d'un ensemble de choix de modélisation et ne se base pas sur une correspondance unique entre concepts B et concepts UML. Plusieurs transformations sont donc évoquées pour certaines constructions en B.

Spécification informelle du système. L'exemple traité ici correspond à un modèle B composé de deux machines abstraites : la machine *SecureFlightBoarding* et la machine *BoardingGate* (Fig. 4.1). Notons que cet exemple est inspiré du projet EDEMOI[26] où une modélisation plus fine et détaillée de la sécurité des aéroports a été menée. Dans ce modèle, nous ne considérons que les passagers et les objets qu'ils transportent et nous nous intéressons principalement à : (*i*) l'ouverture et la fermeture de la salle d'embarquement, (*ii*) l'entrée des passagers dans la salle d'embarquement, et (*iii*) l'embarquement.

[25] Les principales parties du méta-modèle de UML dont nous faisons référence ici sont présentées au niveau de l'annexe B.
[26] www-lsr.imag.fr/EDEMOI/

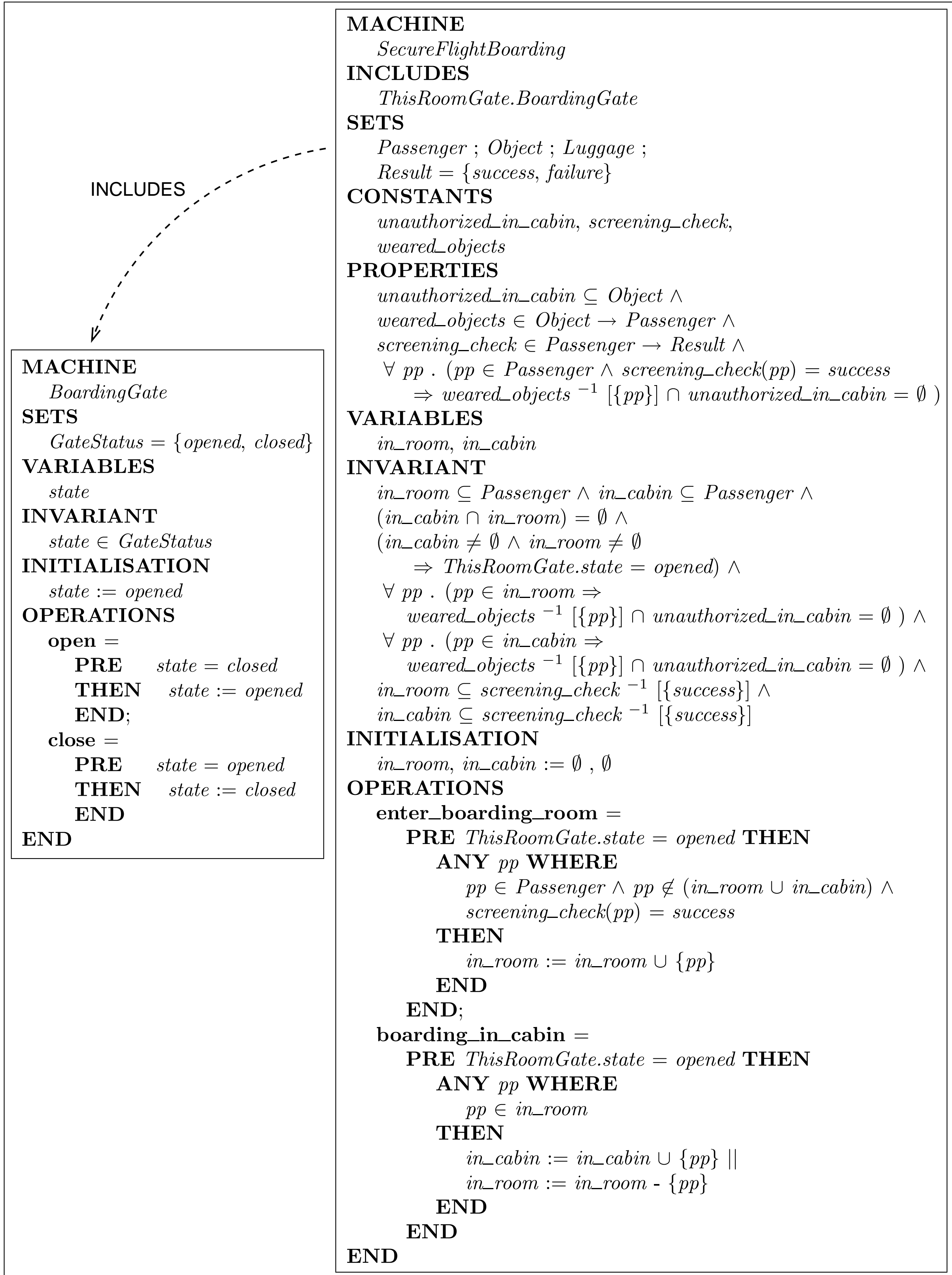


FIG. 4.1 – Exemple d'un modèle B composé de deux machines abstraites

Machine BoardingGate. Il s'agit de la spécification d'une porte d'embarquement permettant aux voyageurs d'accéder à l'avion. Cette machine représente les états de la porte d'embarquement par l'ensemble énuméré *GateStatus*. L'ouverture et la fermeture de la porte sont réalisées respectivement par les opérations *open* et *close*. La variable *state* représente l'état actuel de la porte et prend ses valeurs de l'ensemble *GateStatus*.

Machine SecureFlightBoarding. C'est la spécification d'une salle d'embarquement dont la porte d'accès à l'avion est désignée par l'instance *ThisRoomGate* de la machine *BoardingGate*. Les ensembles abstraits *Passenger* et *Object* représentent respectivement l'ensemble des passagers et l'ensemble des objets que transportent ces passagers. La constante *weared_objects* donne l'ensemble des objets détenus par chaque passager. La constante *unauthorized_in_cabin* désigne l'ensemble des objets non autorisés dans la cabine. Quant à *screening_check*, elle représente le résultat du contrôle de chaque passager. Finalement, l'invariant de cette machine *SecureFlightBoarding* désigne ce qui suit :

- Un passager ne peut être à la fois dans l'avion et dans la salle d'embarquement.
- Les objets transportés par les passagers dans l'avion ou lors du passage par la salle d'embarquement ne sont pas reconnus comme non autorisés dans la cabine.
- Les passagers qui sont dans la salle d'embarquement ou dans l'avion ont été contrôlés avec succès.
- La porte d'embarquement reste ouverte dès qu'un passager a embarqué et tant qu'il existe des passagers contrôlés et qui n'ont pas encore accédé à l'avion.

4.3 Transformation de machines abstraites

4.3.1 Présentation

Le concept de machine abstraite représente l'élément de base pour la structuration des spécifications B. La décomposition de spécifications en plusieurs machines abstraites se base sur des critères purement logiques dont le but est de regrouper au sein de chaque machine un ensemble de données et d'opérations formant un tout cohérent.

Au sein d'une vue statique UML, cette structuration est réalisée par deux mécanismes : les classes et les paquetages.

- Les **classes** en UML constituent les éléments de structuration de base au niveau d'un diagramme de classes. Chaque classe représente une description abstraite d'un ensemble d'objets et regroupe des attributs et des opérations décrivant la structure et le comportement de ces objets.
- Les **paquetages** en UML offrent un mécanisme général pour la partition des modèles et le regroupement d'éléments de modélisation (classes, associations, . . .) (Muller *et al.,* 2001). Leur objectif est d'assurer une meilleure organisation du système en le structurant en *catégories* ou en *sous-systèmes*.

Les notions fondamentales d'**encapsulation** et de **structuration** qui surgissent de ces définitions, conduisent à distinguer entre deux points de vue UML différents pour une machine B : le point de vue classe et le point de vue paquetage. Le premier point de vue permet de voir une machine B comme une classe encapsulant des attributs (données B) et des méthodes (opérations B) ; alors que dans le second point de vue, une machine B est vue comme une entité assez complexe, dont les constituants (données

et opérations) peuvent être traduits par des éléments de modélisation bien structurés (classes, associations, etc). Rappelons qu'une classe en UML introduit la notion d'instance, alors qu'en B, une machine n'a généralement qu'une seule instance. Cette notion d'instance unique provient plus particulièrement des machines de haut niveau décrivant une structure de programmes. En revanche, dans le cadre d'une inclusion de machines, il peut exister des instances variées (explicitement identifiées par renommage) d'une même machine B. Dans ces deux cas, la traduction d'une machine en une classe n'est pas erronée. En effet, lorsqu'il s'agit d'une machine de haut niveau qui n'est pas utilisée dans des clauses d'inclusion, cette traduction en classe sous-entend qu'il s'agit d'une classe disposant d'une instance unique. Par ailleurs, lorsque la machine B est incluse dans d'autres machines donnant lieu à plusieurs instances nous identifions chaque instance par un lien entre les classes issues de la machine incluse et des machines incluantes. Cette vision particulière sera détaillée au niveau du chapitre suivant lors de la prise en compte de la notion d'instance de machine (méta-classe associative *Instance_Included* de la Fig. 3.12 page 65). Ce faisant, nous nous limitons à ce niveau à cette traduction de base d'une machine en une classe.

Schéma 1 *Schéma de transformation Machine / Classe* (Fig. 4.2)

Paramètres	M : *BMachine*
Transformation	La machine M est transformée en une classe UML. Les caractéristiques structurelles et comportementales de la classe correspondent respectivement aux données (*BData*) et aux opérations (*BOperation*) définies au niveau de la machine abstraite. Si la machine est paramétrée alors la classe résultante est également une classe paramétrée (ou *template*).

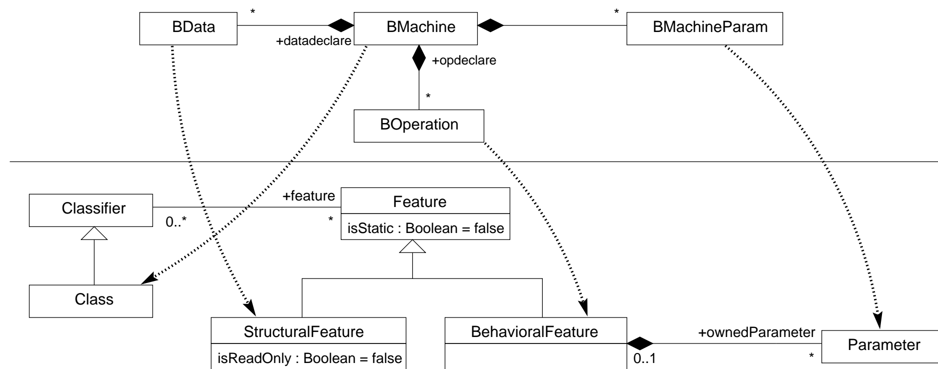


FIG. 4.2 – Schéma de transformation Machine / Classe

Nous ne montrons pas comment une *BData* (respectivement une *BOperation*) est transformée en un attribut (respectivement en une méthode) de la classe associée à la machine abstraite. Ces transformations seront détaillées dans les schémas « BData / Attribut » et « BOperation / Méthode ». Néanmoins, nous mettons l'accent sur le fait que la partie statique (déclarative) et la partie dynamique (opérationnelle) d'une machine B correspondent respectivement aux caractéristiques structurelles (méta-classe *StructuralFeature*) et comportementales (méta-classe *BehavioralFeature*) de la classe résultant de ce schéma de transformation. Quant à la traduction des paramètres de la machine en paramètres de classes, elle a pour principal objectif d'indiquer au niveau de la classe produite par ce schéma de transformation que la machine a été initialement conçue avec des paramètres formels et que son utilisation nécessite des paramètres effectifs.

Notons que l'initialisation peut être associée directement à la classe en tant que caractéristique comportementale ayant pour objectif d'initialiser les attributs de la classe ainsi construite. Cependant, nous n'évoquerons sa transformation qu'au niveau de la dérivation de diagrammes d'états/transitions.

Schéma 2 *Schéma de transformation Machine / Paquetage* (Fig. 4.3)

Paramètres	M : *BMachine*
Transformation	La machine abstraite M est traduite en un paquetage regroupant un ensemble d'éléments de modélisation. Ces derniers correspondent aux données (*BData*) et aux opérations (*BOperation*) définies au niveau de la machine abstraite.

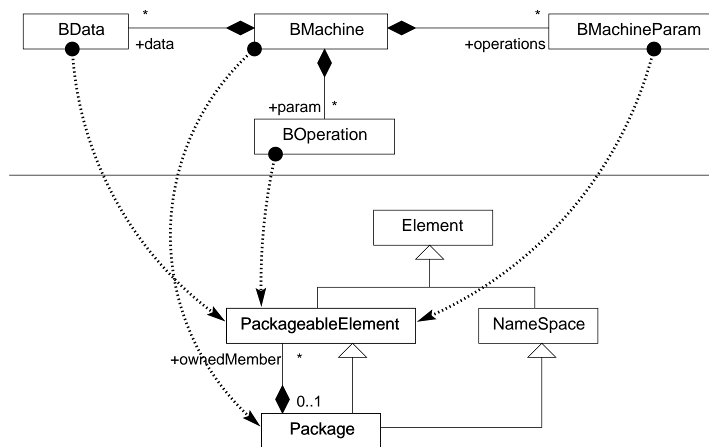


FIG. 4.3 – Schéma de transformation Machine / Paquetage

Dans ce schéma de transformation, la machine B est vue comme un espace de noms (méta-classe *NameSpace*) formé, d'une part, par les variables, constantes et ensembles et, d'autre part, par les opérations de la spécification. Ces données et opérations B sont alors transformées en des éléments de modélisation empaquetés (méta-classe *PackageableElement*). Notons que le document de référence d'UML 2.0 (OMG, 2005) spécifie *PackageableElement* comme étant tout élément de modélisation pouvant appartenir directement à un paquetage sans cataloguer explicitement ces éléments. Dans la figure ci-dessus nous considérons que les *BData*s, les *BOperation*s et les *BMachineParam*s sont transformées en des éléments empaquetés sans donner aucune information sur leur distribution à l'intérieur du paquetage. En effet, une *BOperation* est susceptible de devenir une méthode d'une classe issue d'une *BData* ou encore une transition dans un diagramme d'états/transitions. Nous considérons donc que toutes ces données et opérations sont traitées comme étant des éléments du paquetage issu de la machine à laquelle le schéma 2 est appliqué.

4.3.2 Discussion

Le choix d'un schéma de transformation parmi les deux précédents se base sur le niveau de détail et la complexité des dépendances entre données B. En effet, une spécification B constituée de plusieurs ensembles abstraits avec plusieurs relations entre ces ensembles peut être vue comme un paquetage regroupant un ensemble de classes reliées par des associations. Cependant, il est possible de voir cette même spécification B comme une seule classe dont la structure est détaillée par plusieurs attributs. L'identifi-

cation du niveau de granularité dépend de la vision du développeur et de sa manière de concevoir des systèmes en B. Toutefois, nous proposons de guider ce choix par la règle heuristique suivante :

> L'application du schéma de transformation 2 est conditionnée par l'existence d'au moins une donnée B au niveau de la spécification susceptible d'être transformée en classe, sinon le schéma de transformation 1 est appliqué.

L'identification des classes candidates à partir d'une spécification B sera présentée au niveau du chapitre 7 où nous proposons des critères de pertinence permettant d'assurer la cohérence entre un concept B et les choix d'une contrepartie UML. Au niveau du présent chapitre nous présentons principalement les différentes transformations possibles d'une spécification B en nous basant sur les méta-modèles UML et B. Les correspondances ainsi définies ne forment pas une correspondance déterministe entre concepts B et UML ce qui rend le processus de transformation non trivial. Proposer des techniques qui permettent de guider le choix d'un schéma parmi d'autres devient alors inéluctable.

La spécification présentée au niveau de la Fig. 4.1 permet d'illustrer ces deux points de vue. En effet, d'une part, la machine *BoardingGate* peut être vue comme une classe représentant les portes d'embarquement et encapsulant l'attribut *state* et les opérations *open* et *close* ; et d'autre part, la machine *SecureFlightBoarding* peut être vue comme un paquetage regroupant un ensemble de classes (e.g. *Passenger*, *Object*, ...) avec différents liens entre ces classes.

FIG. 4.4 – Application des schémas de transformation 1 et 2

Dans ce qui suit, la notation :

$$\text{element B : meta-concept B} \xrightarrow{\text{Schema}_i} \text{element UML : meta-classe UML}$$

désigne la *traduction* d'un élément B (instance d'un certain méta-concept B défini au niveau du méta-modèle B) en un élément UML (instance d'une certaine méta-classe UML définie au niveau du méta-modèle UML) par application du schéma de transformation i. Par exemple, la Fig. 4.4 résulte de l'application des traductions suivantes :

$\mathcal{T}_1 \mathrel{\widehat{=}} BoardingGate$: *BMachine* $\xrightarrow{Schema_1}$ $BoardingGate$: *Class*

$\mathcal{T}_2 \mathrel{\widehat{=}} SecureFlightBoarding$: *BMachine* $\xrightarrow{Schema_2}$ $SecureFlightBoarding$: *Package*

Une *traduction* peut donc être vue comme une instanciation d'un schéma de transformation. Par conséquent, la traduction d'un modèle B en un modèle UML peut effectivement être réalisée en exécutant une suite d'instances de schémas de transformation.

4.4 Transformation des types de base (*BBasicType*)

La définition des types en UML est similaire à la définition de types en B. Il s'agit d'un ensemble de valeurs permettant de contraindre les éléments typés (par exemple un attribut ...). En UML, nous pou-

vons distinguer entre types de données concrets (méta-classe *DataType*) et types abstraits tels que la définition de classes (méta-classe *Class*). Les types concrets correspondent aux types primitifs (**`Boolean`**, **`integer`**, ...) et aux types énumérés. La notion de type primitif en UML est généralement définie à partir des types de données prédéfinis au niveau des langages de programmation. En B, nous distinguons : $String$, Z et $Bool$. Quant aux types abstraits, ils correspondent précisément aux classes UML.

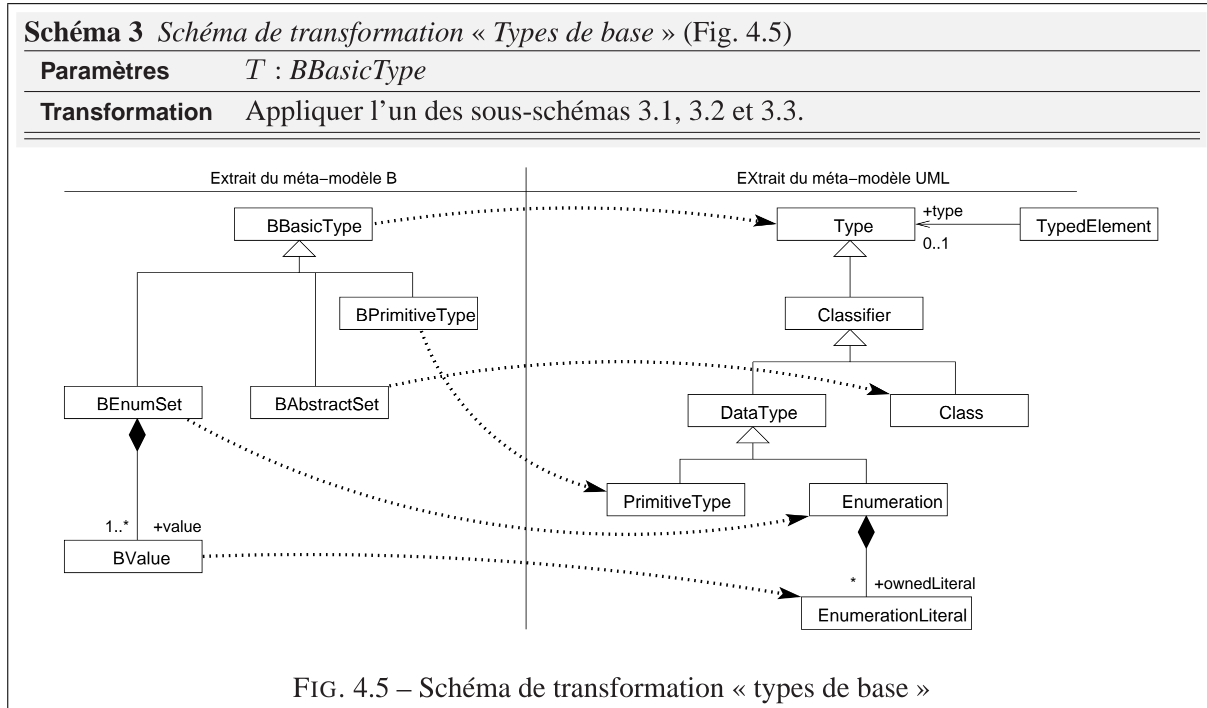


Schéma 3 *Schéma de transformation « Types de base »* (Fig. 4.5)

Paramètres	T : *BBasicType*
Transformation	Appliquer l'un des sous-schémas 3.1, 3.2 et 3.3.

FIG. 4.5 – Schéma de transformation « types de base »

Les différents sous-schémas du schéma de transformation 3 correspondent aux trois spécialisations de la méta-classe *BBasicType* (*i. e. BAbsractSet*, *BEnumSet* et *BPrimitiveType*) que nous avons présentées au niveau du méta-modèle B.

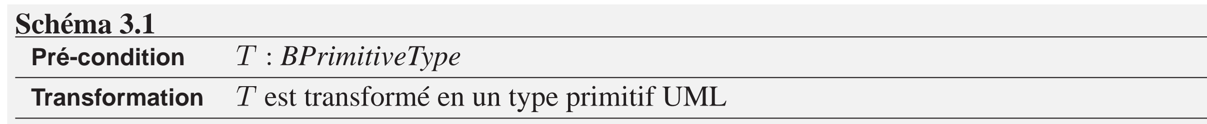


Schéma 3.1

Pré-condition	T : *BPrimitiveType*
Transformation	T est transformé en un type primitif UML

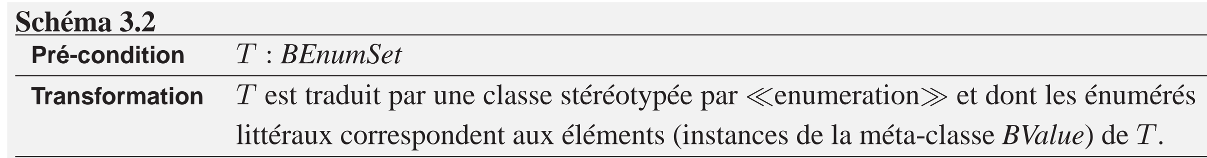


Schéma 3.2

Pré-condition	T : *BEnumSet*
Transformation	T est traduit par une classe stéréotypée par ≪enumeration≫ et dont les énumérés littéraux correspondent aux éléments (instances de la méta-classe *BValue*) de T.

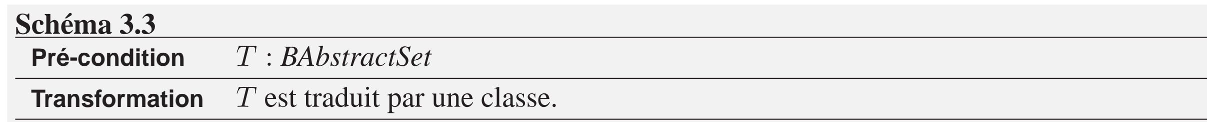


Schéma 3.3

Pré-condition	T : *BAbstractSet*
Transformation	T est traduit par une classe.

Dans le paradigme orienté objet une classe est une abstraction d'entités avec des caractéristiques communes. Elle représente un ensemble d'objets concrets appelés instances. La traduction d'un ensemble abstrait (*BAbsractSet*) en une classe dans le diagramme UML est justifiée par le fait qu'un ensemble

abstrait en B est une représentation de haut niveau d'un ensemble d'éléments concrets. D'un autre côté, la définition d'un ensemble abstrait S dans une machine B est généralement lié à l'existence d'opérations et/ou de données B en relation avec l'ensemble S. Ces opérations et données peuvent être considérées comme des caractéristiques structurelles et comportementales de la classe produite à partir de S.

Les travaux de formalisation de UML en B (Mammar, 2002, Laleau, 2002, Ledang, 2002, Snook *et al.,* 2004) traduisent une classe UML en un ensemble abstrait B vu les similarités sémantiques entre ces deux concepts. Ces travaux se basent principalement sur le fait qu'un ensemble abstrait permet de formaliser l'existence d'un ensemble d'objets indépendamment de leur structure. C'est cette même caractéristique qui motive le schéma de transformation « *Types de base* » proposé dans cette section. En effet, l'application de ce schéma aux ensembles abstraits (*BAbstractSet*) produit des classes sans aucune structure dont l'objectif est de définir un ensemble de types abstraits au niveau du diagramme de classes. Par exemple, l'application de ce schéma de transformation à la spécification B de la Fig. 4.1 (page 69) permet de construire les classes *GateStatus*, *Result*, *Passenger*, *Object*, *in_cabin*, *in_room* et *unauthorized_in_cabin*. Les ensembles énumérés *GateStatus* et *Result* sont transformés en classes ≪enumeration≫ (*i.e.* instances de la méta-classe *enumeration*).

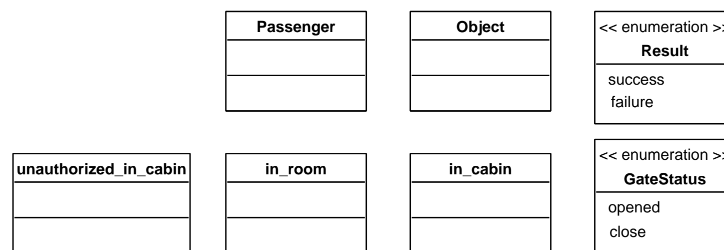


FIG. 4.6 – Illustration du schéma de transformation 3

4.5 Transformation des éléments typés par un *BBasicType*

Différents schémas de transformation peuvent être définis en vue de transformer un élément B, typé (*i.e.* instance de *BTypedElement*) par une instance de la méta-classe *BBasicType*, en un élément UML. Ces schémas de transformation dépendent de deux facteurs : (*i*) de la contrepartie UML produite par application du schéma de transformation 3 en vue de traduire le *BBasicType*, et (*ii*) de l'opérateur de typage (*i.e.* attribut `operator : TypingOperator` de la méta-classe associative *TypingPredicate*).

4.5.1 Cas de l'appartenance

Étant donné qu'un type de base T est traduit soit en un type primitif, soit en une classe stéréotypée par ≪*enumeration*≫, soit en une classe normale, alors l'appartenance d'un élément t à T peut traduire un élément UML (un attribut par exemple) typé par T ou encore une instance particulière de la classe associée à T.

Schéma 4 *Schéma de transformation « Éléments typés par un BBasicType – Opérateur In »*

Paramètres	T : *BBasicType* ; d : BTypedElement
Pré-condition	$d \in T$: *TypingPredicate*
Transformation	Exécuter l'un des sous-schémas 4.1 et 4.2.

Nous distinguons donc deux sous-schémas de transformation distincts :
– Transformation en un attribut de classe (schéma 4.1) ;
– Transformation en instance de classe (schéma 4.2).

Schéma 4.1 *Transformation en attributs de classes* (Fig. 4.7)

Transformation	d est transformé en un attribut de classe mono-valué typé par le type UML associé à T après l'application du schéma de transformation 3.

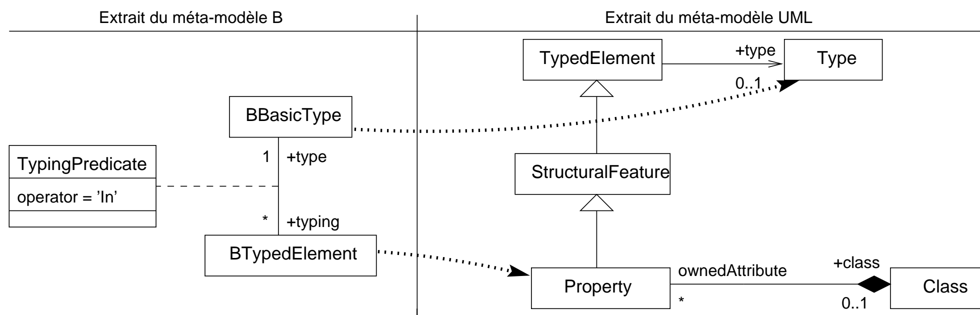


FIG. 4.7 – Schéma de transformation « Éléments typés par un *BBasicType* − Cas de l'opérateur In » (Transformation en attributs de classes)

L'identification de la classe à laquelle associer l'attribut créé par application de ce schéma de transformation dépend des relations pouvant exister entre l'élément B à l'origine de l'attribut et les autres éléments B traduits en classes. Cette notion de dépendance sera détaillée au niveau des chapitres suivants où il s'agit de distinguer la classe la plus appropriée pour encapsuler l'attribut en question.

En guise d'exemple, nous appliquons le schéma de transformation 1 (page 71) pour traduire la machine *BoardingGate* en une classe. Aussi, la variable *state* typée par l'ensemble énuméré *GateStatus* peut-elle être transformée via le schéma de transformation 4.1 en un attribut de la classe *BoardingGate*.

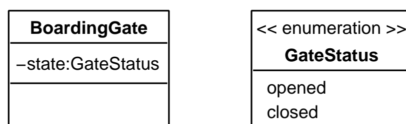


FIG. 4.8 – Illustration du schéma de transformation 4.1

Schéma 4.2 *Transformation en instances de classes*

Pré-condition	T : *BAbstractSet*
Transformation	d est transformé en un objet instance de la classe associée à T. Une dépendance stéréotypée par ≪instantiate≫ est rajoutée de l'instance d vers la classe T.

Au niveau du schéma de transformation en instances de classes, nous considérons une représentation alternative destinée uniquement au cas où le *BBasicType* est un ensemble abstrait (*BAbstractSet*) : l'élément B en question, n'est pas traduit par un attribut, mais plutôt par une instance de la classe associée à l'ensemble abstrait. Par exemple, l'invariant *person* $\in$ *Passenger* peut être traduit par une classe *Passenger* à laquelle est associée l'instance *person* (Fig. 4.9).

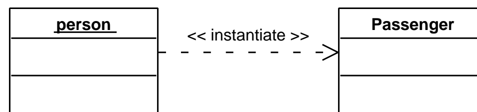


FIG. 4.9 – Illustration du schéma de transformation 4.2

4.5.2 Cas de l'inclusion

À la différence de l'appartenance, l'inclusion ne définit pas un élément particulier mais plutôt un ensemble d'éléments. C'est pourquoi nous considérons les transformations en attributs multi-valués ou en sous-classes.

Schéma 5 *Schéma de transformation « Éléments typés par un BBasicType – Opérateur Subset »*

Paramètres	T : *BBasicType* ; d : BTypedElement
Transformation	Exécuter l'un des sous-schémas suivants.

Ci-dessous, nous distinguons différents cas de transformation dépendant de la spécialisation du *BBasicType* T pris en compte au niveau des paramètres de ce schéma. Notons que dans le cas d'un *BEnumSet* $\text{card}(T)$ est connu et correspond au nombre d'instances de *BValue* liées à T.

Schéma 5.1

Pré-condition	T : *BEnumSet* ; $d \subset T$: *TypingPredicate*
Transformation	d est transformé en une classe disposant d'un attribut "*values*" multi-valué de type T et de multiplicité égale à $0..\text{card}(T) - 1$ ($values : T[0..\text{card}(T) - 1]$).

Schéma 5.2

Pré-condition	T : *BEnumSet* ; $d \subseteq T$: *TypingPredicate*
Transformation	d est transformé en une classe disposant d'un attribut "*values*" multi-valué de type T et de multiplicité égale à $0..\text{card}(T)$ ($values : T[0..\text{card}(T)]$).

Schéma 5.3

Pré-condition	T : *BPrimitiveType* ; $d \subset T$: *TypingPredicate* ou $d \subseteq T$: *TypingPredicate*
Transformation	d est transformé en une classe disposant d'un attribut "*values*" multi-valué de type T et dont la multiplicité est égale à $*$ ($values : T[*]$).

Soit par exemple l'ensemble énuméré $Couleur = \{bleu, blanc, rouge, noir, vert\}$ et les variables $CouleurDrapeau$ et $stats$ telles que $CouleurDrapeau \subseteq Couleur$ et $stats \subseteq NAT$, alors l'application des schémas de transformation 5.2 et 5.3 produit les classes suivantes :

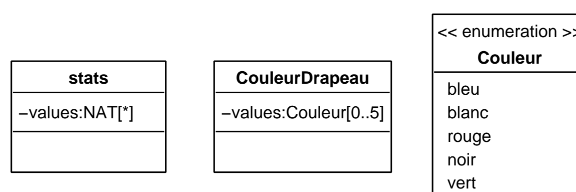


FIG. 4.10 – Illustration des schémas de transformation 5.2 et 5.3

La transformation en attributs multi-valués ne porte que sur les éléments typés par un *BEnumeSet* ou par un *BPrimitiveType*. Dans le cadre d'ensembles abstraits sans aucune structure deux transformations distinctes sont envisagées : en une sous-classe et en un attribut booléen.

Schéma 5.4 *Transformation en un lien d'héritage*	
Pré-condition	T : *BAbstractSet* ; $d \subset T$: *TypingPredicate* ou $d \subseteq T$: *TypingPredicate*
Transformation	d est transformé en une sous-classe de la classe produite pour T par application du schéma de transformation 3.

Ce schéma de transformation est justifié par le fait qu'en UML l'héritage exprime une relation "*est un*" ou "*est une sorte de*" entre une sous-classe et une super-classe et permet d'encapsuler des informations et des traitements supplémentaires au niveau de la sous-classe. Ceci correspond parfaitement à l'inclusion entre ensembles abstraits en B ; en effet, soient les ensembles S et S' tels que $S \subseteq S'$ alors un élément de S est aussi un élément de S'. De plus la définition d'un sous-ensemble en B peut désigner l'existence de traitements particuliers et d'informations supplémentaires se rattachant au sous-ensemble.

L'application de ce schéma de transformation sur la spécification B de la Fig. 4.1 permet d'identifier (*i*) les deux sous-classes *in_cabin* et *in_room* de la classe *Passenger* et (*ii*) la sous-classe *unauthorized_in_cabin* de la classe *Object*.

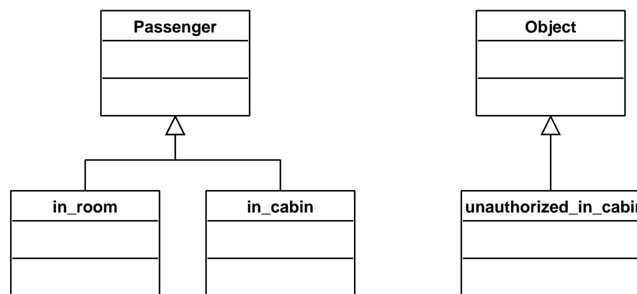


FIG. 4.11 – Illustration du schéma de transformation 5.4

Notons que la transformation (*i*) n'est pas interdite par nos transformations bien qu'elle ne soit pas cohérente en termes d'évolution d'états. En effet, une telle représentation en UML sous-entend qu'une instance p de *Passenger* peut se trouver dans deux états distincts : *in_room* ou *in_cabin*. Cependant, bien que la spécification B permette l'évolution de p d'un état à l'autre via l'opération *boarding_in_cabin*, la représentation hiérarchique par des sous-classes ne garantit pas cette évolution dynamique du contenu des sous-classes. Une transformation alternative est alors proposée ci-dessous où le sous-ensemble est traité comme un attribut booléen. Les états *in_cabin* et *in_room* sont alors associés aux valeurs "*true*" ou "*false*" des attributs *in_cabin* et *in_room*.

Schéma 5.5 *Transformation en un attribut booléen*	
Pré-condition	T : *BAbstractSet* ; $d \subset T$: *TypingPredicate* ou $d \subseteq T$: *TypingPredicate*
Transformation	d est transformé en un attribut booléen de la classe produite pour T par application du schéma de transformation 3.

Le choix de ce schéma de transformation est également justifié par le fait que initialement en B la définition d'un sous-ensemble S d'un ensemble S' a pour objectif de conditionner la réalisation de

certaines opérations. Ainsi, un élément de S' peut se traduire par un objet instance de la classe S et dont l'attribut S' est vrai.

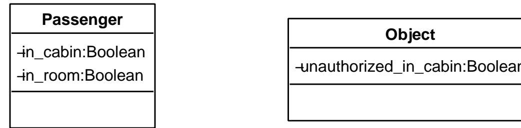


FIG. 4.12 – Illustration du schéma de transformation 5.5

Remarquons que les cinq cas de transformation précédents caractérisant le schéma 5 conduisent à traduire la machine déclarant le *BBasicType* T et le *BTypedElement* d en un paquetage. Cependant, lorsqu'une machine B est transformée en une classe, il devient opportun de traduire d par un attribut de cette classe.

Schéma 5.6 *Transformation en un attribut booléen*

Pré-condition	$d \subset T$: *TypingPredicate* ou $d \subseteq T$: *TypingPredicate*
Transformation	d est transformé en un attribut de classe multi-valué et typé par T.

Supposons par exemple que la variable *state* de la machine *BoardingGate* est définie telle que *state* $\subseteq$ *GateStatus* alors une transformation possible est :

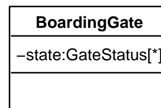


FIG. 4.13 – Illustration du schéma de transformation 5.6

4.5.3 Cas de l'égalité

L'égalité entre éléments B sous-entend qu'il s'agit d'effectuer une copie du même élément B. Nous gardons alors uniforme la transformation pour les deux éléments B de l'égalité. Par ailleurs, nous notons que l'application du schéma de transformation 6 doit être menée avec précaution car cette transformation produit deux éléments de modélisation de noms différents mais indiquant un même concept. Nous faisons le choix ici de traduire les constructions en B faisant partie de l'égalité de manière uniforme en vue de garantir une transformation cohérente.

Schéma 6 *Schéma de transformation « Éléments typés par un BBasicType − Opérateur Equal »*

Paramètres	T : *BBasicType* ; d : BTypedElement
Pré-condition	$d = T$: *TypingPredicate*
Transformation	d est transformé par application du même schéma de transformation que celui utilisé pour T.

4.6 Transformation des éléments typés par un *BPowType*

En vue de transformer les éléments B dont le type est construit par le constructeur de type $\mathbb{P}$ nous allons distinguer également entre plusieurs cas de figure dépendant d'une part de l'opérateur de typage,

et d'autre part, des spécialisations du *BBasicType*.

Schéma 7 *Schéma de transformation « Éléments typés par un BPowType − Opérateur In »*

Paramètres	t : *BTypedElement* ; T : *BBasicType*
Pré-condition	$t \in \mathbb{P}(T)$: *TypingPredicate*
Transformation	Appliquer schéma 7.1 ou 7.2 ou 7.3 ou 7.4.

Dans le cadre d'un opérateur d'appartenance, le résultat du schéma de transformation 7 est identique à celui du schéma 5. Ceci est justifié par l'équivalence suivante :

$$t \in \mathbb{P}(T) \Leftrightarrow t \subseteq T$$

Schéma 7.1

Pré-condition	T : *BAbstractSet*
Transformation	t est transformé en un attribut booléen de la classe produite pour T par application du schéma de transformation 3.

Schéma 7.2

Pré-condition	T : *BAbstractSet*
Transformation	t devient une sous-classe de la classe produite pour T par application du schéma de transformation 3.

Schéma 7.3

Pré-condition	T : *BEnumSet*
Transformation	t est transformé en un attribut multi-valué *values* de la classe produite pour T par application du schéma de transformation 3 et de multiplicité égale à $0..\operatorname{card}(T)$ ($values$: $T[0..\operatorname{card}(T)]$).

Schéma 7.4

Pré-condition	T : *BEnumSet*
Transformation	t est transformé en un attribut multi-valué *values* de la classe produite pour T par application du schéma de transformation 3 et de multiplicité égale à $*$ ($values$: $T[*]$).

Dans le cadre des opérateurs d'inclusion ou d'égalité nous allons considérer uniquement des ensembles abstraits (*BAbstractSet*) avec le schéma de transformation suivant :

Schéma 8 *Schéma de transformation « Éléments typés par un BPowType − SubSet / Equal »*

Paramètres	t : *BTypedElement* ; T : *BAbstractSet*
Pré-condition	$t \subseteq \mathbb{P}(T)$: *TypingPredicate* **ou** $t \subset \mathbb{P}(T)$: *TypingPredicate* **ou** $t = \mathbb{P}(T)$: *TypingPredicate*
Transformation	Une relation d'agrégation entre les classes t et T est créée avec des multiplicités égales à $*$ des deux côtés. La classe t joue le rôle de l'agrégat tandis que la classe T correspond à l'élément agrégé.

Pour illustrer les schémas de transformation 7 et 8 nous prenons en guise d'exemple les prédicats de typage suivants : $Group \subseteq \mathbb{P}(Employee)$ et $qualified \in \mathbb{P}(Employee)$. En d'autres termes, $qualified$ est un sous-ensemble d'$Employee$ et $Group$ est un ensemble d'ensembles contenant des groupes d'employés. Les deux ensembles $Employee$ et $Group$ sont transformés via le schéma de transformation 3 en classes. L'élément $qualified$ est transformé soit en un attribut booléen soit en une sous-classe de la classe *Employee* selon le choix du schéma 7.1 ou 7.2. Ici, nous faisons le choix du schéma de transformation 7.1.

Ceci étant, l'application du schéma de transformation 8 produit le diagramme de la Fig. 4.14. Dans ce diagramme de classes, les classes *Group* et *Employee* sont liées par une relation d'agrégation où un groupe est vu comme un agrégat d'employés. Les éléments de l'ensemble $Employee$ faisant partie ou non de l'ensemble $qualified$ sont représentés par l'attribut booléen *qualified*.

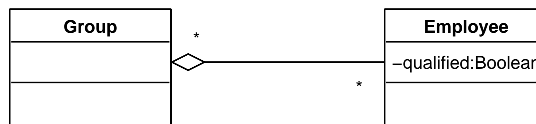


FIG. 4.14 – Illustration du schéma de transformation 8

Ce schéma de transformation est motivé par le fait que l'agrégation permet d'exprimer une relation de type "ensemble / élément" montrant une subordination entre deux classes. À ce niveau, cette subordination correspond à la définition d'un ensemble à partir des éléments d'un autre ensemble ; en effet, un élément de l'ensemble $Group$ est un ensemble d'éléments de l'ensemble $Employee$. Ceci est documenté en UML par le fait qu'un groupe (instance de la classe *Group*) est un agrégat d'employés (instances de la classe *Employee*).

4.7 Transformation des éléments typés par un *BComposedType*

Rappelons que les éléments typés par un *BComposedType* sont des éléments dont le type est construit par les constructeurs de type $\mathbb{P}$ et/ou $\times$. Au niveau du méta-modèle nous avons mis l'accent distinctement sur les différentes relations fonctionnelles définies à partir de ces constructeurs de types. Ainsi, nous considérons dans nos transformations les compositions de types les plus couramment utilisées. Les autres peuvent être réécrites en constructions plus simples en vue d'être prises en compte par nos schémas de transformation. Nous allons traiter dans un premier temps les relations et fonctions simples ensuite nous aborderons le produit cartésien.

4.7.1 Relations et fonctions simples

Dans le cadre de relations ou fonctions simples (*e.g.* $R \in T \leftrightarrow P$) plusieurs transformations sont envisagées : association, classe associative ou attribut de classes.

Schéma 9 *Schéma de transformation « Éléments typés par un BComposedType »*

Paramètres	R : *BTypedElement* ; T, P : *BType* ; w : *ComposedTypeExp*
Pré-condition	$R \in T\ w\ P$: *TypingPredicate* ; $w \neq$ *Product*
Transformation	Appliquer schéma 9.1, 9.2, 9.3 ou 9.4.

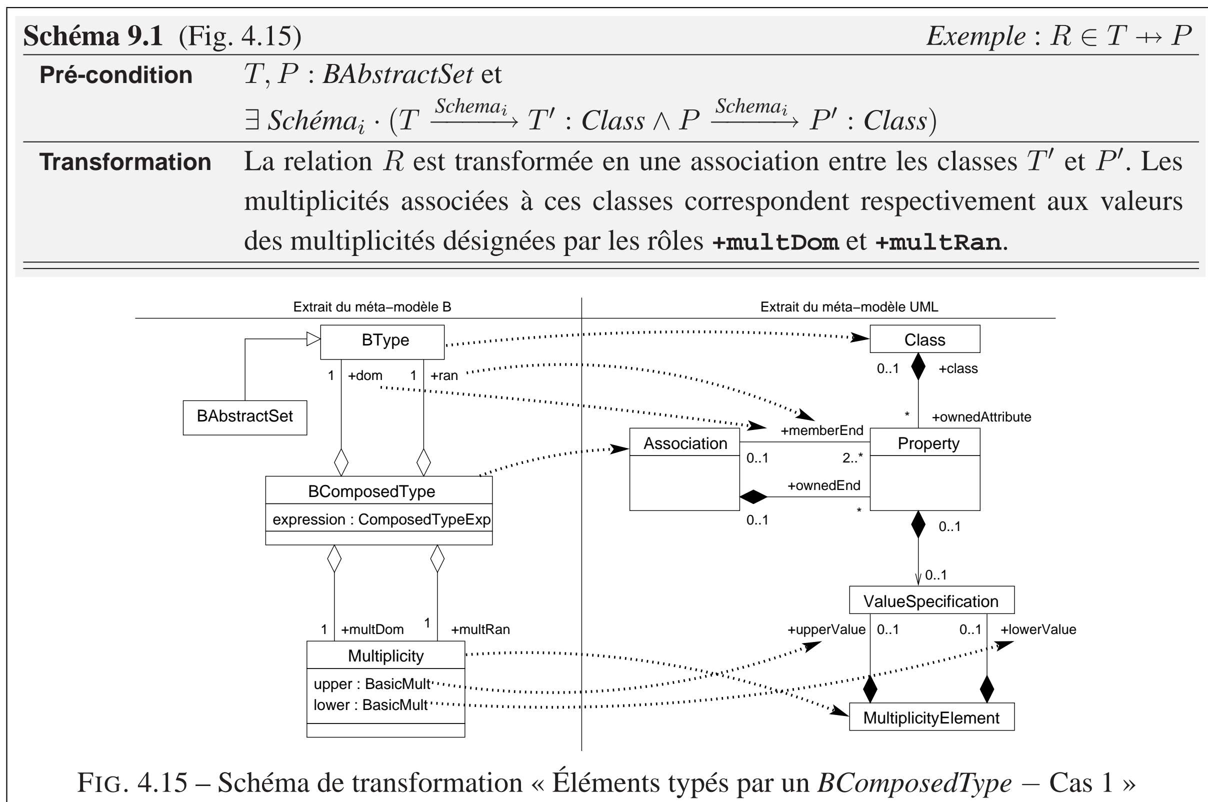


Schéma 9.1 (Fig. 4.15) — *Exemple* : $R \in T \twoheadrightarrow P$

Pré-condition	T, P : *BAbstractSet* et $\exists$ *Schéma*$_i \cdot (T \xrightarrow{Schema_i} T'$: *Class* $\wedge$ $P \xrightarrow{Schema_i} P'$: *Class*$)$
Transformation	La relation R est transformée en une association entre les classes T' et P'. Les multiplicités associées à ces classes correspondent respectivement aux valeurs des multiplicités désignées par les rôles **`+multDom`** et **`+multRan`**.

FIG. 4.15 – Schéma de transformation « Éléments typés par un *BComposedType* $-$ Cas 1 »

L'application du schéma de transformation 9.1 au prédicat de typage de la constante $weared_objects$ au niveau de la machine *SecureFlightBoarding* (Fig. 4.1, page 69) :

$$weared_objects \in Object \rightarrow Passenger$$

produit le diagramme suivant :

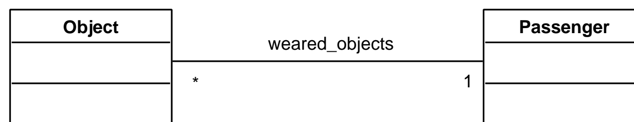


FIG. 4.16 – Illustration du schéma de transformation 9.1

Au niveau de ce diagramme, la fonction totale définie de l'ensemble abstrait $Object$ vers l'ensemble abstrait $Passenger$ et désignée par $weared_objects$ est transformée en une association $(*, 1)$ entre les classes produites pour ces deux ensembles abstraits. Un passager peut, en effet, être propriétaire de plusieurs objets, et un objet ne peut appartenir qu'à un et un seul passager.

Des représentations équivalentes à la notation par association peuvent être mises en évidence telles que les représentations par attribut. Celles-ci sont définies au niveau des schémas 9.2 et 9.3 suivants :

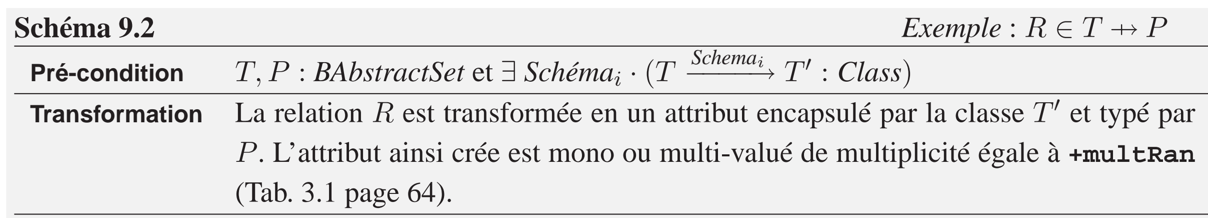


Schéma 9.2 — *Exemple* : $R \in T \twoheadrightarrow P$

Pré-condition	T, P : *BAbstractSet* et $\exists$ *Schéma*$_i \cdot (T \xrightarrow{Schema_i} T'$: *Class*$)$
Transformation	La relation R est transformée en un attribut encapsulé par la classe T' et typé par P. L'attribut ainsi crée est mono ou multi-valué de multiplicité égale à **`+multRan`** (Tab. 3.1 page 64).

Étant donné que *Passenger* est un ensemble abstrait au niveau de la spécification *SecureFlightBoarding*, alors il est traduit par un type abstrait en appliquant les schémas de transformation des types de base. L'application du schéma de transformation 9.2 au prédicat de typage de la constante *weared_objects* permet de traduire cette constante en un attribut de type *Passenger* et de multiplicité égale à 1 au niveau de la classe *Object* (Fig. 4.17). La constante *weared_objects* est donc transformée en un attribut mono-valué de cardinalité $[1..1]$.

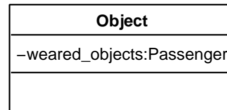


FIG. 4.17 – Illustration du schéma de transformation 9.2

Notons que dans cette transformation on perd l'information sur **`+multRan`**, en effet, ce diagramme ne représente que le fait qu'un objet est transporté par un et un seul passager.

Schéma 9.3	*Exemple* : $R \in T \leftrightarrow P$
Pré-condition	T, P : *BAbstractSet* $\exists$ *Schéma*$_i \cdot (P \xrightarrow{Schema_i} P'$: *Class*$)$
Transformation	La relation R est transformée en un attribut encapsulé par la classe P' et typé par T. L'attribut ainsi créée est mono ou multi-valué de multiplicité égale à **`+multDom`** (Tab. 3.1 page 64).

Le résultat de ce schéma de transformation lorsque celui-ci est appliqué à la constante *weared_objects* est un attribut multi-valué (multiplicité égale à $*$) non-obligatoire encapsulé par la classe *Passenger* et typé par *Object* (Fig.4.18).

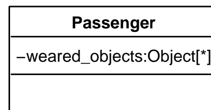


FIG. 4.18 – Illustration du schéma de transformation 9.3

Comme pour le cas du schéma de transformation 9.2, dans ce schéma on perd l'information sur **`+multDom`**, en effet, cette représentation n'illustre que le fait qu'un passager peut transporter plusieurs objets sans se préoccuper du fait qu'un objet est transporté par un et un seul passager.

Schéma 9.4	*Exemple* : $R \in T \leftrightarrow P$
Pré-condition	T, P : *BAbstractSet*
Transformation	La relation R est transformée en une classe associative entre les classes T et P. Les multiplicités associées aux extrémités de l'association correspondent respectivement aux valeurs des multiplicités désignées par les rôles **`+multDom`** et **`+multRan`** (Tab. 3.1 page 64).

Le schéma de transformation 9.4 est justifié par le fait que la spécification peut définir des données B et/ou des opérations B liées uniquement avec la relation R. L'existence d'une classe R permet d'identifier une structure pouvant encapsuler ces données et opérations. Par exemple, *weared_objects* peut être considérée comme une classe associative entre *Passenger* et *Object* comme présenté au niveau de la Fig. 4.19.

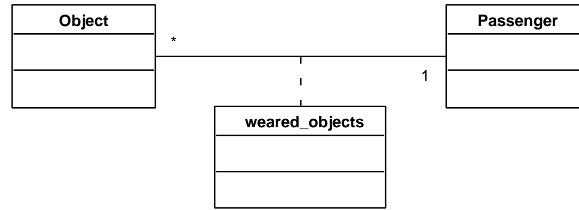


FIG. 4.19 – Illustration du schéma de transformation 9.4

4.7.2 Produit cartésien

Schéma 10 *Schéma de transformation « Éléments typés par un BComposedType (Produit cartésien) »*

Paramètres	R : *BTypedElement* ; T, P : *BType* ; w : *ComposedTypeExp*
Pré-condition	$R \subset T\ w\ P$: *TypingPredicate* **ou** $R \subseteq T\ w\ P$: *TypingPredicate* **ou** $R = T\ w\ P$: *TypingPredicate* ; w = *Product*
Transformation	– Définir le prédicat $R \in T \leftrightarrow P$; – Appliquer Schéma$_9(R, T, P, \leftrightarrow)$.

Le schéma de transformation 10 produit soit une relation, soit un attribut soit alors une classe associative *R* si T et P sont des ensembles abstraits. Le remplacement du produit cartésien par une relation permet la ré-utilisation du schéma de transformation 9. Ceci revient à la définition des relations en B ($T \leftrightarrow P \mathrel{\widehat{=}} \mathbb{P}(T \times P)$).

$$\begin{aligned} R \subseteq T \times P \quad &\Rightarrow \quad R \in \mathbb{P}(T \times P) \\ &\Rightarrow \quad R \in T \leftrightarrow P \end{aligned}$$

Nous n'allons pas considérer à ce niveau le cas de l'appartenance car elle traduit un couple particulier du produit cartésien. Ceci est susceptible d'être transformé en des instances de classes. Cependant, nous ne considérons pas dans notre démarche les diagrammes d'objets. Nous nous limitons alors aux données B pouvant donner lieu à des éléments d'un diagramme de classes.

4.8 Composition de relations et fonctions

Dans cette section, nous nous intéresserons à la transformation de constructions B établies par composition de certaines structures de base. Plus particulièrement la composition de relations et fonctions avec un *BPowType* ou avec un produit cartésien. Nous essayerons de ramener les transformations de compositions de relations et fonctions à une application des schémas de base décrits dans les sections précédentes.

4.8.1 Composition de relations et fonctions avec un *BPowType*

Schéma 11 *Schéma de transformation « Éléments typés par un BComposedType »*

Paramètres	R : *BTypedElement* ; T, P : *BType* ; w : *ComposedTypeExp*
Pré-condition	$R \in T\ w\ P$: *TypingPredicate* ; $w \neq$ *Product*
Transformation	Appliquer schéma 11.1, 11.2, 11.3 ou 11.4.

Schéma 11.1	*Exemple* : $R \in T \leftrightarrow \mathbb{P}(S)$
Pré-condition	T : *BAbstractSet* ; P : *BPowType* ; $P \widehat{=} \mathbb{P}(S)$; S : *BAbstractSet* ; w : *ComposedTypeExp*
Transformation	Définir l'ensemble abstrait *GroupOf_S* telle que *GroupOf_S* = $\mathbb{P}(S)$, ainsi que le prédicat : $R \in$ T w *GroupOf_S*, puis appliquer successivement les schémas suivants : – Schéma$_3$(*GroupOf_S*) ; – Schéma$_{9.1}$(R, *GroupOf_S*, T, w) ou Schéma$_{9.4}$(R, *GroupOf_S*, T, w) ; – Schéma$_8$(*GroupOf_S*, S)

Ce schéma introduit une nouvelle classe appelée *GroupOf_S* représentant un agrégat d'éléments de la classe S. Ainsi, un lien R entre une instance t de la classe T et une instance g de la classe *GroupOf_S* permet de déterminer un ensemble d'éléments de S associés à t par la relation R. Étant donné, par exemple, le prédicat $R \in T \leftrightarrow \mathbb{P}(S)$, alors le schéma de transformation 11.1 n'est autre que l'application d'une suite de schémas de base après la réécriture de ce prédicat de la manière suivante :

$$R \in T \leftrightarrow \mathit{GroupOf_S} \quad \wedge \quad \mathit{GroupOf_S} = \mathbb{P}(S)$$

Le choix entre Schéma$_{9.1}$ et Schéma$_{9.4}$ lors de la transformation introduit du non-déterminisme au schéma 11.1. Les deux transformations sélectionnées à ce niveau sont la traduction de R en une association simple ou en une classe associative entre les classes T et *GroupOf_S*. Les deux autres spécialisations du schéma 9 traduisent la relation R en attribut, ce qui ne peut être considéré car la classe *GroupOf_S* a été rajoutée pour mettre en évidence la relation "element/ensemble", désignée par R, entre T et S. Le diagramme suivant illustre le cas où Schéma$_{9.1}$ a été choisi.

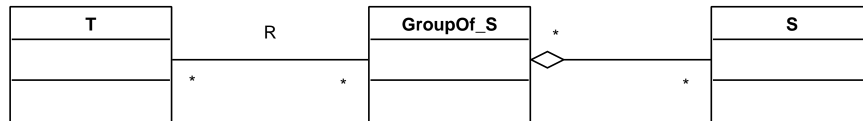


FIG. 4.20 – Illustration du schéma de transformation 11.1

Nous proposons un cas particulier de transformation (Schema 11.2) qui n'est applicable que dans le cadre d'éléments typés par composition d'une fonction et d'un *BPowType* et où la multiplicité maximale définie par le rôle **`+multRan`** ne dépasse pas 1 (Tab. 3.1 page 64). Nous excluons donc de ce schéma particulier le cas des relations où **`+multRan`** indique une multiplicité égale à $*$. En effet, l'invariant $R \in T \leftrightarrow \mathbb{P}(S)$ définit le fait qu'un élément de T peut être lié par R à plusieurs ensembles d'éléments de S, alors qu'au niveau de ce schéma nous considérons qu'un seul ensemble d'éléments de S au plus est associé à un élément de T. Dans ce cas, toutes les spécialisations du schéma 9 peuvent être envisagées.

Schéma 11.2	*Exemple* : $R \in T \rightarrow \mathbb{P}(S)$
Pré-condition	T : *BAbstractSet* ; P : *BPowType* ; $P \widehat{=} \mathbb{P}(S)$; S : *BAbstractSet* ; $w \neq$ *Relation*
Transformation	Appliquer Schéma$_9$(R, T, S, $\leftrightarrow$).

Soit par exemple, $weared_objects$ telle que $weared_objects \in Passenger \rightarrow \mathbb{P}(Object)$. Ce prédicat peut être interprété comme suit : d'une part, chaque passager détient un ensemble d'objets ; et d'autre

part, un ensemble d'objets peut appartenir à plusieurs passagers. Ceci conduit à considérer qu'un passager détient zéro ou plusieurs objets et qu'un objet peut, d'une part, appartenir à plusieurs passagers, et d'autre part, ne pas avoir de propriétaire. Ainsi, $weared_objects$ peut être vu comme une association de multiplicités $(0..*, 0..*)$, ce qui correspond exactement à l'application de Schéma$_9$(*weared_objects*, *Passenger*, *Object*, $\leftrightarrow$).

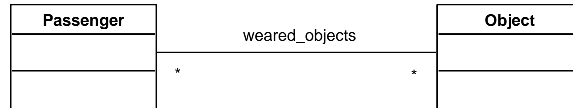


FIG. 4.21 – Illustration du schéma de transformation 11.2

Ce diagramme illustre le fait que l'existence d'un lien *weared_objects* entre une instance p de la classe *Passenger* et une instance o de la classe *Object* représente le fait que o appartient à l'ensemble des objets portés par le passager p.

Une transformation analogue aux schémas 11.1 et 11.2 peut être envisagée pour des cas tels que $R \in \mathbb{P}(S) \leftrightarrow T$ où il s'agit d'appliquer les mêmes schémas sur R^{-1}. Par exemple, comme pour le schéma de transformation 11.1, le schéma ci-dessous produit une nouvelle classe nommée *GroupOf_S* (par application du schéma 3) et représentant un agrégat d'éléments de la classe S. L'application du schéma 9.1 construit une relation R entre *GroupOf_S* et P avec les multiplicités adéquates. Finalement, l'application du schéma de transformation 8 définit le lien d'agrégation entre les classes *GroupOf_S* et S.

Schéma 11.3	*Exemple* : $R \in \mathbb{P}(S) \nrightarrow P$
Pré-condition	T : *BPowType* ; P : *BAbstractSet* ; $T \mathrel{\widehat{=}} \mathbb{P}(S)$; S : *BAbstractSet* ; w : *ComposedTypeExp*
Transformation	Définir l'ensemble abstrait *GroupOf_S* telle que *GroupOf_S* = $\mathbb{P}(S)$, ainsi que le prédicat : $R \in$ *GroupOf_S* w P, puis appliquer successivement les schémas suivants : – Schéma 3(*GroupOf_S*) ; – Schéma 9.1(R, *GroupOf_S*, P, w) ; ou Schéma 9.4(R, *GroupOf_S*, P, w) – Schéma 8(*GroupOf_S*, S)

Supposons, par exemple, que la constante $weared_objects$ est définie comme suit : $weared_objects \in \mathbb{P}(Object) \nrightarrow Passenger$, alors l'application du schéma de transformation 11.3 produit le diagramme de la Fig. 4.22.

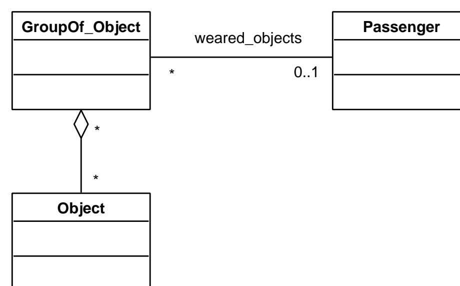


FIG. 4.22 – Illustration du schéma de transformation 11.3

Ce diagramme de classes illustre le fait que chaque élément instance de la classe *GroupOf_Object* est une collection d'instances de la classe *Object*. Un passager peut porter plusieurs groupes d'objets, et chaque ensemble d'objets ne peut être porté que par au plus un passager.

De même, nous proposons un cas particulier de transformation (Schema 11.4) qui n'est applicable que dans le cadre d'éléments typés par composition d'une fonction et d'un *BPowType* et où la multiplicité maximale définie par le rôle **`+multDom`** ne dépasse pas 1.

Schéma 11.4	*Exemple* : $R \in \mathbb{P}(S) \rightarrowtail\!\!\!\!\!\!\!\shortmid\;\; P$
Pré-condition	T : *BPowType* ; P : *BAbstractSet* ; $T \mathrel{\widehat{=}} \mathbb{P}(S)$; S : *BAbstractSet* ; $w \in \{$*PartialInjection, TotalInjection, PartialBijection, TotalBijection*$\}$
Transformation	Appliquer Schéma$_9$(R, S, P, $\leftrightarrow$).

Discussion. Les quatre cas de transformation précédents sont définis comme étant des cas de transformation élémentaires car la composition de types peut être plus complexe. Nous nous limitons à ces cas particuliers et nous soulignons le fait que les compositions de relations et fonctions avec un *BPowType* qui ne font pas partie de cette famille de structures composées, doivent être réécrites en des structures élémentaires. Par exemple, l'invariant $R \in \mathbb{P}(S) \mathrel{+\!\!\!\rightarrow} \mathbb{P}(T)$ peut être traité en passant par la réécriture suivante : *GroupOf_S* $= \mathbb{P}(S) \land R \in$ *GroupOf_S* $\mathrel{+\!\!\!\rightarrow} \mathbb{P}(T)$. Il suffit d'appliquer par la suite les schémas suivants : Schéma$_8$(*GroupOf_S*, S) et Schéma$_{11.1}$(R,*GroupOf_S*, T, $\mathrel{+\!\!\!\rightarrow}$) en vue d'aboutir à une transformation possible :

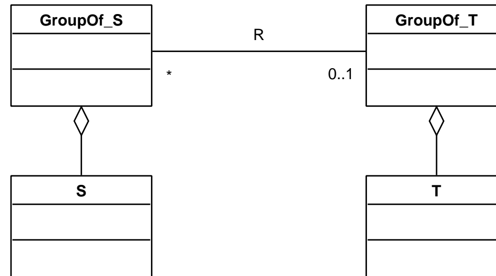


FIG. 4.23 – Transformation de $R \in \mathbb{P}(S) \mathrel{+\!\!\!\rightarrow} \mathbb{P}(T)$

4.8.2 Composition de relations et fonctions avec un produit cartésien

Schéma 12	*Exemple* : $R \in T \leftrightarrow (Q \times S)$
Paramètres	R : *BTypedElement* ; T, P : *BType* ; w : *ComposedTypeExp*
Pré-condition	T : *BAbstractSet* ; P : *BComposedType* ; $P \mathrel{\widehat{=}} Q\ w'\ S$; Q, S : *BAbstractSet* ; w : *ComposedTypeExp* ; $w' =$ *Product*
Transformation	a. Définir la relation Q_S telle que $Q_S = Q\ w'\ S$; b. Remplacer $Q\ w'\ S$ par Q_S ; c. Exécuter Schéma$_{9.4}$(Q_S, Q, S, $\leftrightarrow$) ; d. Exécuter Schéma$_{9.1}$(R, T, Q_S, w) ou Schéma$_{9.3}$(R, T, Q_S, w) ou Schéma$_{9.4}$(R, T, Q_S, w) ;

Les deux transformations exécutées par le schéma 12 permettent successivement de :

(i) Créer la classe associative *Q_S* entre les deux classes *Q* et *S*

(ii) Créer, selon le cas,

- une association *R* entre les classes *T* et *Q_S*, ou
- un attribut *R* (mono ou multi-valué) de la classe *Q_S* et typé par *T*, ou
- une classe associative *R* entre les classes *T* et *Q_S*.

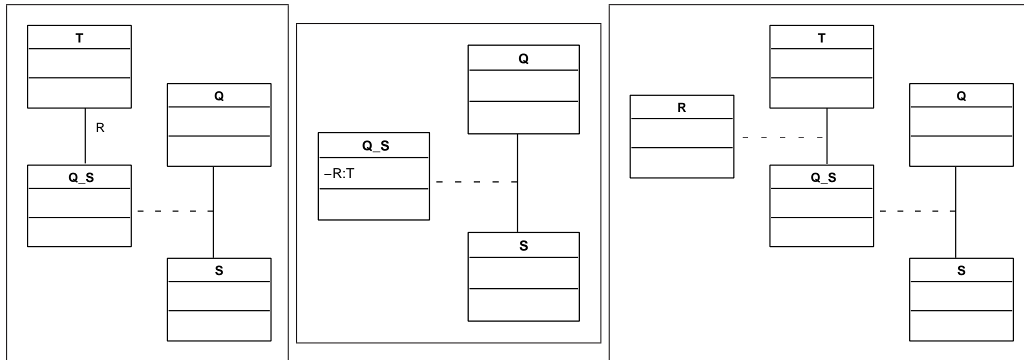


FIG. 4.24 – Illustration du schéma de transformation 12

L'application de ce schéma de transformation passe nécessairement par la création d'une classe associative (*i.e.* *Q_S*). Ceci permet d'identifier les couples de $Q \times S$ liés aux éléments de T par la relation R. Par analogie à ce schéma de transformation, nous identifions également le cas où le type composé par un *Product* porte sur le domaine de la relation.

Schéma 13	*Exemple* : $R \in (Q \times S) \leftrightarrow P$
Paramètres	R : *BTypedElement* ; T, P : *BType* ; w : *ComposedTypeExp*
Pré-condition	T : *BComposedType* ; P : *BAbstractSet* ; $T \mathrel{\widehat{=}} Q\ w'\ S$; Q, S : *BAbstractSet* ; w : *ComposedTypeExp* ; $w' =$ *Product*
Transformation	a. Définir la relation Q_S telle que $Q_S = Q\ w'\ S$; b. Remplacer $Q\ w'\ S$ par Q_S ; c. Exécuter Schéma$_{9.4}$(Q_S, Q, S, $\leftrightarrow$) ; d. Exécuter Schéma$_{9.1}$(R, Q_S, P, w) ou Schéma$_{9.3}$(R, Q_S, P, w) ou Schéma$_{9.4}$(R, Q_S, P, w) ;

Soit par exemple, le prédicat : $registered_objects \in (Object \times Luggage) \rightarrow Passenger$, alors le schéma de transformation 13 construit, entre autres, le diagramme de classes de la Fig. 4.25.

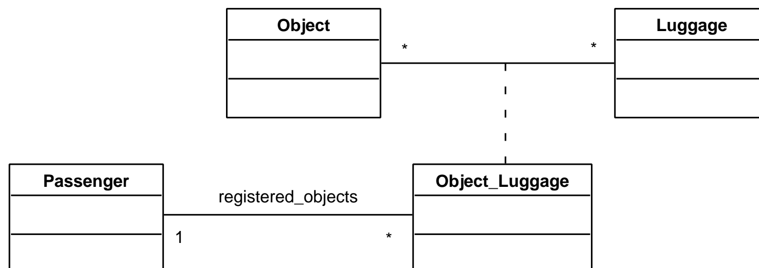


FIG. 4.25 – Illustration du schéma de transformation 13

La fonction totale $registered_objects$ peut également être transformée en un attribut mono-valué obligatoire de la classe associative $Object_Luggage$, ou encore en une classe associative entre $Passenger$ et $Object_Luggage$.

Schéma 14	*Exemple* : $R \in (Q \times S) \leftrightarrow (M \times N)$
Paramètres	R : *BTypedElement* ; T, P : *BType* ; w : *ComposedTypeExp*
Pré-condition	T, P : *BComposedType* ; $T \widehat{=} Q\ w'\ S$; $P \widehat{=} M\ w'\ N$ Q, S, M, N : *BAbstractSet* ; w : *ComposedTypeExp* ; $w' = Product$
Transformation	Exécuter les schémas 12 et 13 en alternant leurs étapes.

Étant donné, par exemple, l'invariant suivant : $R \in (Q \times S) \rightarrow (M \times N)$; le schéma de transformation 14 passe par les six étapes suivantes :

1	`définir` : $M_N = M \times N$	**`schéma 12.(a)`**
2	`définir` : $Q_S = Q \times S$	**`schéma 13.(b)`**
3	`Réécrire` $R : R \in (Q \times S) \rightarrow M_N$	**`schéma 12.(a)`**
4	`Réécrire` $R : R \in Q_S \rightarrow M_N$	**`schéma 13.(b)`**
5	`Exécuter` : `Schéma`$_{9.4}(M_N, M, N, \leftrightarrow)$	**`schéma 12.(a)`**
6	`Exécuter` : `Schéma`$_{9.4}(Q_S, Q, S, \leftrightarrow)$	**`schéma 13.(b)`**

La dernière étape de ce schéma de transformation permet de construire une association ou une classe associative R entre les deux classes associatives M_N et Q_S.

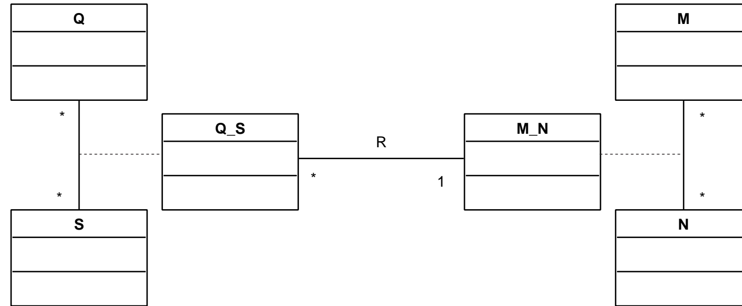


FIG. 4.26 – Illustration du schéma de transformation 14

La Fig. 4.26 illustre le cas où R est transformée en association, et ce, par application du schéma 9.1 avec les paramètres ordonnés suivants : "R", "Q_S", "M_N" et "$\rightarrow$".

4.9 Transformation des opérations d'une machine abstraite

Dans le cadre d'une représentation de spécifications B sous-forme de vues structurelles, une manière naturelle de traduire les opérations est de les considérer comme des méthodes de classes. Cependant, les schémas que nous présentons produisent plusieurs classes selon différents cas de figure. Par exemple, une classe peut provenir :

(i) d'une machine abstraite (*i. e.* instance de la méta-classe *BMachine*) par application du schéma de transformation 1 ; ou,

(*ii*) d'une donnée B (*i. e.* instance de la méta-classe *BData*) par application des schémas de transformation 3, 5.2 ou 9.4. En d'autres termes, nous considérons les schémas qui produisent une classe, une sous-classe ou encore une classe associative.

Dans le premier cas de figure, chaque opération d'une machine abstraite M est systématiquement traduite en une méthode de la classe produite pour M via le schéma 1. Ceci se traduit par le schéma de transformation suivant :

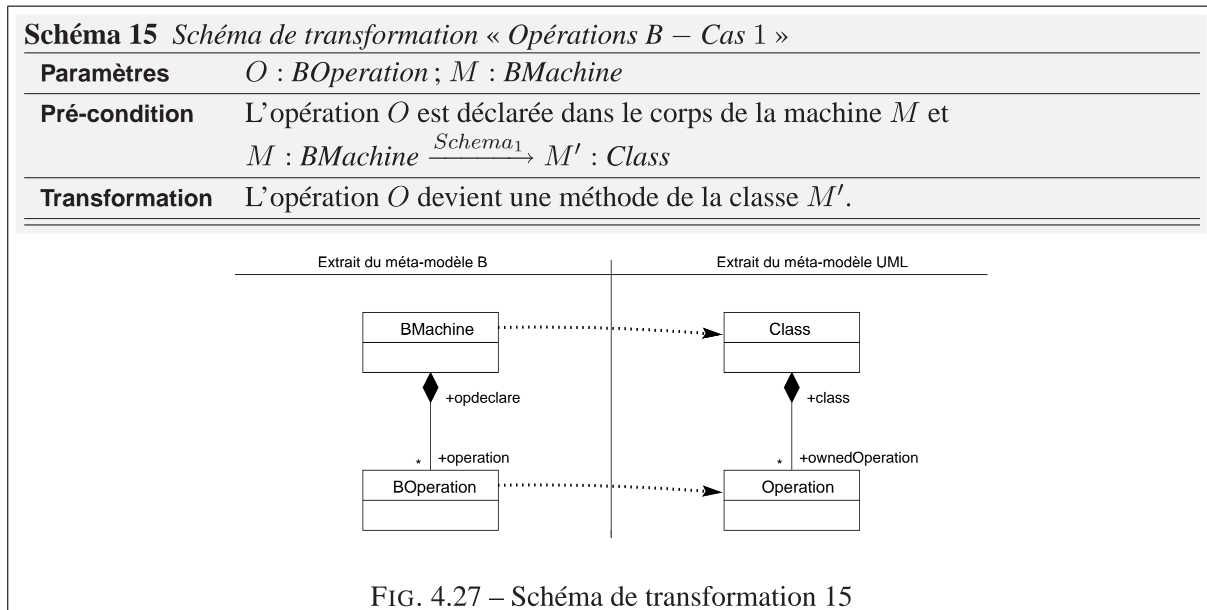


Schéma 15 *Schéma de transformation « Opérations B – Cas* 1 »

Paramètres	O : *BOperation* ; M : *BMachine*
Pré-condition	L'opération O est déclarée dans le corps de la machine M et M : *BMachine* $\xrightarrow{Schema_1}$ M' : *Class*
Transformation	L'opération O devient une méthode de la classe M'.

FIG. 4.27 – Schéma de transformation 15

L'application de cette transformation à la machine *BoardingGate* produit une classe *BoardingGate* encapsulant les méthodes : *open* et *close*.

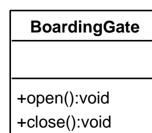


FIG. 4.28 – Illustration du schéma de transformation 15

Dans le cas de figure où une *BMachine* est transformée en un paquetage (*Package*) via le schéma 2, les opérations deviennent des éléments empaquetés et seront encapsulées par les classes du paquetage. Cela suppose l'existence d'une *BData* de la machine B transformée en une classe.

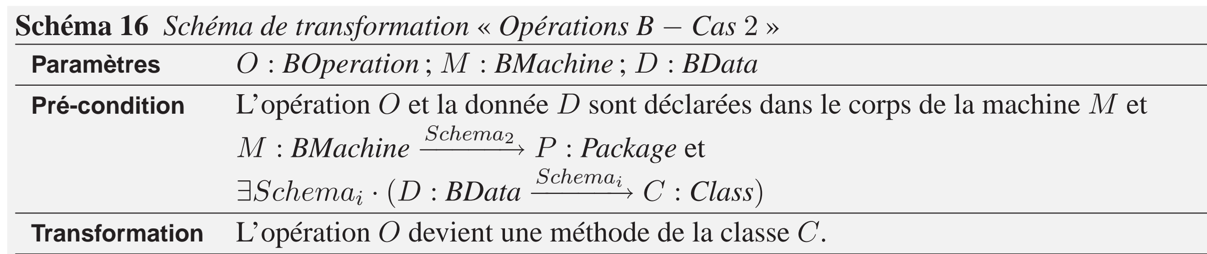


Schéma 16 *Schéma de transformation « Opérations B – Cas* 2 »

Paramètres	O : *BOperation* ; M : *BMachine* ; D : *BData*
Pré-condition	L'opération O et la donnée D sont déclarées dans le corps de la machine M et M : *BMachine* $\xrightarrow{Schema_2}$ P : *Package* et $\exists Schema_i \cdot (D$: *BData* $\xrightarrow{Schema_i}$ C : *Class*$)$
Transformation	L'opération O devient une méthode de la classe C.

L'exécution de ce schéma de transformation autant de fois que d'opérations permet de distribuer les opérations de la *BMachine* M sur les différentes classes du paquetage P. Supposons que la machine

SecureFlightBoarding est transformée en un paquetage contenant la classe *Passenger* ; alors l'exécution de ce schéma avec les paramètres successifs suivants : "*enter_boarding_room*", "*SecureFlightBoarding*" et "*Passenger*" permet de construire le diagramme de la Fig. 4.29.

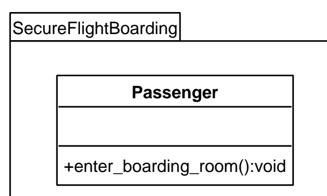


FIG. 4.29 – Illustration du schéma de transformation 16

4.10 Bilan et discussion

Dans ce chapitre nous avons présenté un ensemble de dérivations possibles de B vers UML portant sur des structures de base : machines abstraites, types de base, éléments typés par des types de bases, éléments typés par $\mathbb{P}$, relations et fonctions simples, produit cartésien et finalement les opérations d'une machine abstraite. Nous avons montré également qu'une composition de ces schémas de base peut être envisagée pour traiter la transformation de structures composées. Cependant, d'autres constructions plus complexes peuvent exister telles que :

$$P \subseteq S \times \mathbb{P}(T)$$

En vue de prendre en compte de tels prédicats de typage, il est indispensable de passer par une phase de décomposition (ou réécriture) en prédicats plus simples adaptés aux schémas de base auxquels nous nous sommes limité. Par exemple, la décomposition du prédicat précédent en :

$$\begin{gathered} \mathit{GroupOf_T} = \mathbb{P}(T) \\ P \subseteq S \times \mathit{GroupOf_T} \end{gathered}$$

permet d'appliquer, entre autres schémas de transformation, les schémas 8 et 10. Ce faisant, nous présentons à l'annexe C (page 245) quelques exemples de réécriture ainsi que les transformations associées.

Remarquons qu'une même construction B peut être transformée de plusieurs manières selon le(s) schéma(s) choisi(s) et que l'application de deux schémas différents sur une même construction B peut la traduire de deux manières différentes au sein d'un même diagramme. À ce niveau de l'investigation nous ne marquons aucune préférence entre ces schémas, nous mentionnons simplement que la mise en œuvre de ces transformations est fortement interactive et le choix des schémas à appliquer dépend principalement de l'appréciation de l'analyste.

Chapitre 5

Transformation des liens entre machines abstraites et application

« *Ainsi pensée, représentation et logique sont-elles intimement liées. C'est par la pensée, qui s'exprime par le langage, que l'on peut appréhender la forme logique du monde, c'est-à-dire considérer les rapports nécessaires entre les faits.* »

Ludwig Wittgenstein
« Investigations philosophiques », 1953.

Sommaire

5.1 Introduction **93**
5.2 Prise en compte des liens d'inclusion **95**
5.3 Prise en compte des liens de raffinement **100**
5.4 Application et discussion **101**
5.4.1 Application à la spécification *SecureFlight* 101
5.4.2 Comparaison avec d'autres approches 103
5.5 Conclusion **106**

5.1 Introduction

Etant donné qu'une machine B peut être transformée en une classe ou en un paquetage selon le schéma de transformation sélectionné, alors les liens de dépendance (inclusion, raffinement, . . .) entre machines seront traduits en fonction des entités (classes ou paquetages) résultantes de chaque schéma de transformation.

Dépendances entre paquetages. Les dépendances entre paquetages en UML sont définies comme des liens de référencement permettant au paquetage source d'accéder à certains éléments (dits publics ou visibles de l'extérieur) du paquetage cible. Explicitement ces liens sont spécifiés par les méta-classes *ElementImport* et *PackageImport* et indiquent une visibilité privée ou publique des éléments référencés de l'extérieur (Fig. 5.1). La méta-classe *ElementImport* précise les éléments du paquetage cible référencés par le paquetage source alors que *PackageImport* porte sur tous les éléments du paquetage cible.

Dans les deux cas, seuls les éléments du paquetage cible disposant d'une visibilité publique peuvent être référencés de l'extérieur. Quant à la portée des éléments référencés, elle est définie en UML par les stéréotypes ≪import≫ et ≪access≫.

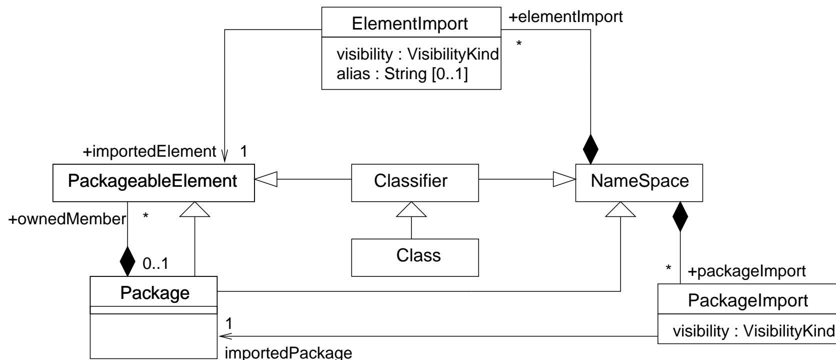


FIG. 5.1 – Extrait du méta-modèle UML relatif aux dépendances entre paquetages

Dans le cas d'un lien ≪import≫, la dépendance entre paquetages est transitive ; en effet, le référencement de plusieurs éléments d'un paquetage $\mathcal{P}_1$ par un paquetage $\mathcal{P}_2$ est reproduit pour un paquetage $\mathcal{P}_3$ qui importe $\mathcal{P}_2$. Cependant, une dépendance ≪access≫ n'est pas transitive, et la portée des éléments référencés, dans ce cas, se limite au paquetage source.

L'extrait du méta-modèle de UML présenté ci-dessus permet aussi de distinguer des dépendances entre classes et paquetages. En effet :

– Un paquetage est un espace de nommage (*NameSpace*) pouvant importer (**`+elementImport`**) un *PackageableElement*. Ce dernier peut être une classe (**`+importedElement`**) ou un paquetage .
– Une classe est un espace de nommage (*NameSpace*) pouvant importer (**`+packageImport`**) un paquetage (**`+importedPackage`**) ou une classe (**`+importedElement`**).

Dépendances entre classes. Les dépendances entre classes peuvent exister pour plusieurs raisons (Fowler, 2004) : une classe envoie un message à une autre classe ; une partie des données d'une classe est stockée dans une autre classe ; une classe en utilise une autre comme paramètre de l'une de ses méthodes ; etc. Nous n'allons considérer que trois dépendances principales :

– les associations : permettent de mettre en relation les classes ;
– l'appel d'opérations : désigné par le stéréotype ≪call≫ ;
– l'utilisation des attributs : spécifiée par le stéréotype ≪use≫.

Notons que les deux dernières dépendances sont définies dans le cas où aucune association n'est identifiée entre les classes source et cible de la dépendance. Notre objectif étant la visualisation de certains traits liés à la spécification, nous nous en servons dans le cadre où la classe cible de la dépendance dispose d'une instance unique ou est vue comme une classe utilitaire...

Vu que pour chaque machine abstraite les schémas de transformation 1 et 2 produisent soit une classe soit un paquetage, alors différents cas de figure sont à prendre en compte pour chaque type de dépendance entre ces machines :

$\mathcal{M}_1 : BMachine \xrightarrow{Schema_2} \mathcal{P}_1 : Package$ $\mathcal{M}_2 : BMachine \xrightarrow{Schema_2} \mathcal{P}_2 : Package$	Cas 1
$\mathcal{M}_1 : BMachine \xrightarrow{Schema_1} \mathcal{C}_1 : Class$ $\mathcal{M}_2 : BMachine \xrightarrow{Schema_1} \mathcal{C}_2 : Class$	Cas 2
$\mathcal{M}_1 : BMachine \xrightarrow{Schema_2} \mathcal{P} : Package$ $\mathcal{M}_2 : BMachine \xrightarrow{Schema_1} \mathcal{C} : Class$	Cas 3
$\mathcal{M}_1 : BMachine \xrightarrow{Schema_1} \mathcal{C} : Class$ $\mathcal{M}_2 : BMachine \xrightarrow{Schema_2} \mathcal{P} : Package$	Cas 4

Dans ce qui suit, nous allons considérer une dépendance de $\mathcal{M}_1$ vers $\mathcal{M}_2$ (par exemple, dans le cadre d'une inclusion nous considérons que $\mathcal{M}_1$ inclut $\mathcal{M}_2 \ldots$).

5.2 Prise en compte des liens d'inclusion

Schéma 17 *Schéma de transformation « Inclusion entre machines »*

Paramètres	$\mathcal{M}_1, \mathcal{M}_2 : BMachine$
Pré-condition	$\mathcal{M}_1$ inclut $\mathcal{M}_2$
Transformation	Exécuter l'un des sous-schémas suivants.

Cas 1. Les données de $\mathcal{M}_1$ et de $\mathcal{M}_2$ sont transformées en des éléments de modélisation empaquetés respectivement dans $\mathcal{P}_1$ et $\mathcal{P}_2$ et telles que les éléments de $\mathcal{P}_2$ sont utilisés par ceux de $\mathcal{P}_1$.

Schéma 17.1 *Schéma de transformation « Inclusion entre machines − Cas* 1 » (Fig. 5.2)

Pré-condition	$\mathcal{M}_1 : BMachine \xrightarrow{Schema_2} \mathcal{P}_1 : Package$ et $\mathcal{M}_2 : BMachine \xrightarrow{Schema_2} \mathcal{P}_2 : Package$
Transformation	L'inclusion entre $\mathcal{M}_1$ et $\mathcal{M}_2$ est traduite par une dépendance de $\mathcal{P}_1$ vers $\mathcal{P}_2$ et est stéréotypée par ≪import≫ vu le caractère transitif de l'inclusion. Le nom de l'alias correspondant est celui associé à l'instance de $\mathcal{M}_2$ dans $\mathcal{M}_1$.

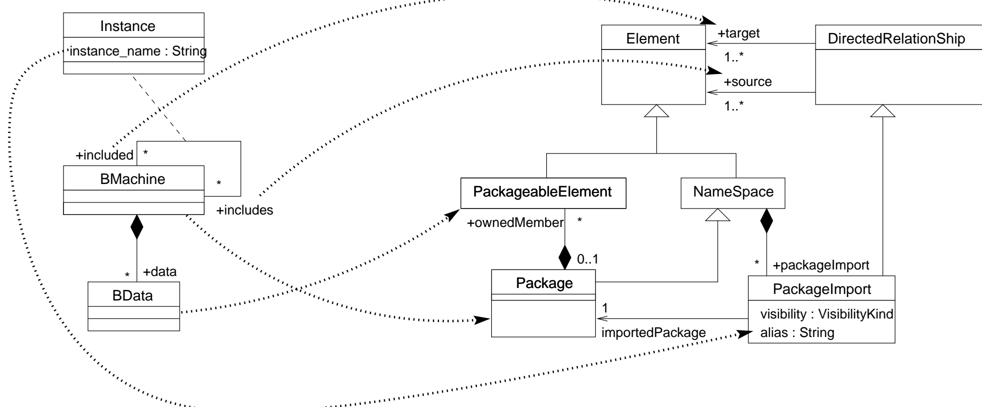


FIG. 5.2 – Schéma de transformation « Inclusion entre machines − Cas 1 »

L'ensemble des correspondances mises en évidence au niveau de la Fig. 5.2 illustrent le fait que les machines abstraites incluses (nom de rôle **`+included`**) et incluantes (nom de rôle **`+includes`**) sont transformées en paquetages et que leurs données sont traduites par des éléments empaquetés. Ceci correspond à l'application du schéma de transformation 2 pour les machines jouant un rôle dans l'inclusion.

Par ailleurs, une instance de la méta-classe *PackageImport* est créée pour désigner l'inclusion et telle que la source (nom de rôle **`+source`**) et la cible (nom de rôle **`+target`**) de cette instance sont respectivement le paquetage correspondant à la machine incluante et le paquetage correspondant à la machine incluse. Quant au nom de l'instance associée à la machine incluse (attribut **`instance_name`**), il est traduit par un alias au niveau de l'instance de *PackageImport*. Finalement, la valeur de l'attribut **`visibility`** est **`public`**, ce qui correspond au stéréotype ≪import≫.

Notons aussi, que la dépendance ≪import≫ ainsi identifiée peut porter uniquement sur certains éléments de $\mathcal{P}_2$, particulièrement ceux utilisés dans le corps de $\mathcal{M}_1$. Ceci est recommandé lorsque le paquetage $\mathcal{P}_2$ est assez complexe et que la dépendance ≪import≫ construite pour traduire l'inclusion entre machines ne couvre que quelques éléments de $\mathcal{P}_2$. Dans ce cas, la dépendance est transformée non pas par une instance de la méta-classe *PackageImport*, mais plutôt par une instance de la méta-classe *ElementImport*. L'élement source de cette instance est le paquetage correspondant à la machine incluante alors que la cible est un élément empaqueté contenu dans le paquetage correspondant à la machine incluse. Supposons, par exemple, que la machine *BoardingGate* est transformée en un paquetage nommé *PBoardingGate* et contenant la classe *BoardingGate* alors le diagramme résultant peut être illustré comme suit :

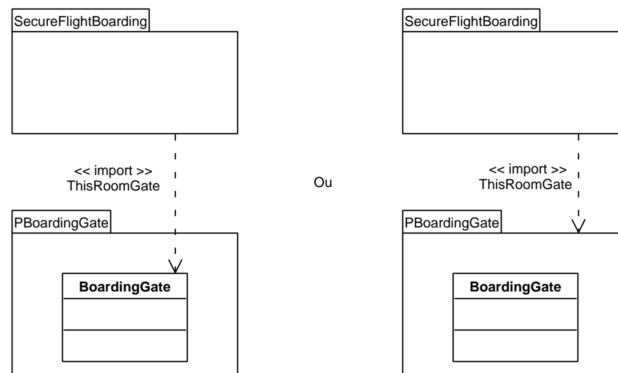


FIG. 5.3 – Illustration du schéma de transformation 17.1

Vu que la traduction du lien INCLUDES en une dépendance ≪import≫ s'attache en particulier à la visibilité des éléments de modélisation du paquetage cible et au caractère transitif de ce lien, alors nous pouvons envisager des traductions semblables dans le cadre d'autres clauses de composition de machines. Par exemple, la non-transitivité des clauses USES et SEES peut se réduite à une dépendance ≪access≫ entre paquetages.

Des mécanismes d'extension d'UML, connus sous le nom de *profile* UML, peuvent être élaborés pour projeter les règles de visibilité induites par les clauses d'assemblage en B au niveau des dépendances entre paquetage en UML. Cependant, notre intérêt au niveau du présent travail de thèse ne porte pas sur l'extension d'UML, ni la proposition de nouveau stéréotypes. Nous nous servons donc des constructions de base offertes par le langage UML pour étudier nos traductions de spécifications B en diagrammes UML.

Cas 2. Les données de $\mathcal{M}_1$ et de $\mathcal{M}_2$ sont transformées en des attributs des classes respectives $\mathcal{C}_1$ et $\mathcal{C}_2$, et les opérations de chaque machine correspondent aux méthodes de ces classes.

Schéma 17.2 *Schéma de transformation « Inclusion entre machines – Cas* 2 » (Fig. 5.4)

Pré-condition	$\mathcal{M}_1 : BMachine \xrightarrow{Schema_1} \mathcal{C}_1 : Class$ et $\mathcal{M}_2 : BMachine \xrightarrow{Schema_1} \mathcal{C}_2 : Class$
Transformation	Les attributs de $\mathcal{C}_2$ correspondant à des *BData*s utilisées dans les clause PROPERTIES ou INVARIANT de $\mathcal{M}_1$, ou encore référencées dans le corps des opérations de $\mathcal{M}_1$ deviennent des attributs publics (donc visibles pour la classe $\mathcal{C}_1$). Le lien d'inclusion est représenté par une association unidirectionnelle de $\mathcal{C}_1$ vers $\mathcal{C}_2$ où les multiplicités associées à $\mathcal{C}_1$ et $\mathcal{C}_2$ sont respectivement $0..1$ et 1. Si un nom d'instance de $\mathcal{M}_2$ est défini explicitement au niveau de $\mathcal{M}_1$ alors ce nom devient un nom de rôle du côté de $\mathcal{C}_2$. Sinon le nom de rôle utilisé est le nom de la classe $\mathcal{C}_2$.

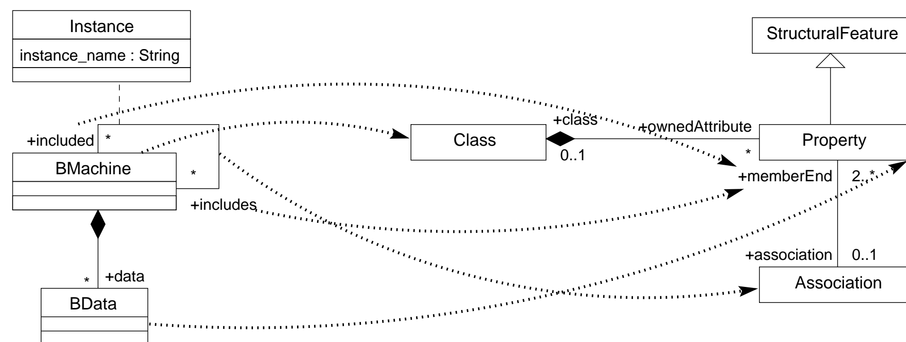


FIG. 5.4 – Schéma de transformation « Inclusion entre machines – Cas 2 »

Supposons par exemple que les machines *SecureFlightBoarding* et *BoardingGate* sont transformées en classes, alors l'application du schéma de transformation ci-dessus permet de construire le diagramme de classes suivant :

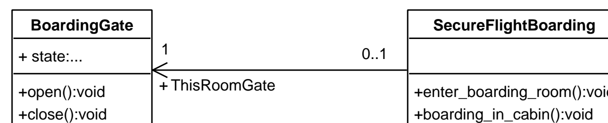
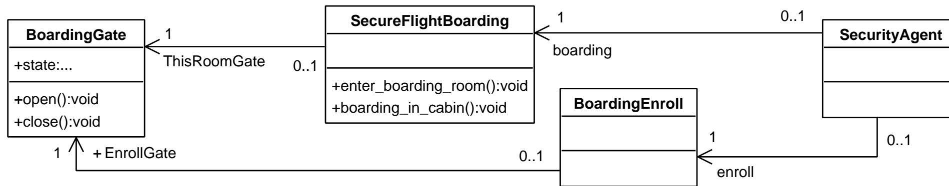


FIG. 5.5 – Illustration du schéma de transformation 17.2

L'attribut *state* de la classe *BoardingGate* est public car, d'une part, il est référencé au niveau de l'invariant de *SecureFlightBoarding*, et d'autre part, il est utilisé par les opérations *boarding_in_cabin* et *enter_boarding_room*. D'un point de vue documentation, le diagramme de la Fig. 5.5 peut être interprété comme suit : les méthodes de la classe *SecureFlightBoarding* peuvent accéder aux attributs et méthodes de la classe *BoardingGate* alors que l'inverse est impossible. Une instance de *SecureFlightBoarding* est nécessairement liée à une instance unique de *BoardingGate* qualifiée par le nom de rôle *ThisRoomGate*. D'un autre côté, ce diagramme indique qu'une instance *BoardingGate* ne joue le rôle de *ThisRoomGate* que pour une instance, au plus, de *SecureFlightBoarding*. Notons qu'en B lorsqu'une machine $\mathcal{M}$ est incluse dans deux machines différentes, alors deux instances distinctes de $\mathcal{M}$ sont créées. Cependant,

la transformation que nous avons proposée à ce niveau peut aboutir à un diagramme de classes où cette notion d'instances distinctes n'est pas respectée. Par exemple, dans le diagramme ci-dessous où toutes les classes sont issues de machines abstraites, une instance de la classe *BoardingGate* peut être à la fois liée à une instance de *SecureFlightBoarding* et à une instance de *BoardingEnroll*. En vue d'indiquer qu'une telle situation n'est réellement pas envisageable trois solutions existent : (*i*) indiquer une contrainte "Xor" entre *ThisRoomGate* et *EnrollGate*, ou (*ii*) considérer une contrainte OCL, ou alors (*iii*) rajouter un stéréotype spécifique à la visualisation de spécifications B (*e.g.* ≪*MachineInclusion*≫) indiquant une sémantique particulière d'une association provenant d'une relation d'inclusion entre machines.



Cas 3. D'une part, les données de $\mathcal{M}_1$ sont transformées en des éléments de modélisation empaquetés dans le paquetage $\mathcal{P}$; et d'autre part, les données de $\mathcal{M}_2$ sont transformées en des attributs de la classe désignée par $\mathcal{C}$.

Schéma 17.3 *Schéma de transformation « Inclusion entre machines* − *Cas* 3a »

Pré-condition	$\mathcal{M}_1 : BMachine \xrightarrow{Schema_2} \mathcal{P} : Package$ et $\mathcal{M}_2 : BMachine \xrightarrow{Schema_1} \mathcal{C} : Class$
Transformation	Les attributs de $\mathcal{C}$ correspondant à des concepts utilisés dans les clause PROPERTIES ou INVARIANT de $\mathcal{M}_1$, ou encore référencés dans le corps des opérations de $\mathcal{M}_1$ deviennent des attributs publics (donc visibles pour les classes de $\mathcal{P}$). L'inclusion est traduite par une dépendance ≪import≫ de $\mathcal{P}$ vers $\mathcal{C}$ et dont le nom d'alias correspond au nom de l'instance de $\mathcal{M}_2$ dans $\mathcal{M}_1$. Les classes de $\mathcal{P}$ sont liées à la classe $\mathcal{C}$ par des dépendances ≪use≫ ou ≪call≫ selon qu'elles manipulent les attributs ou les opérations de $\mathcal{C}$.

La classe $\mathcal{C}$ peut être considérée comme une classe utilitaire donnant des services externes. Par exemple, la Fig. 5.6 illustre l'application de ce schéma de transformation à l'exemple de la Fig. 4.1. Dans ce diagramme de classes nous considérons que l'ensemble abstrait *Passenger* est transformé en une classe dans le paquetage *SecureFlightBoarding*. Nous considérons également que l'opération *boarding_in_cabin* est une méthode de la classe *Passenger*.

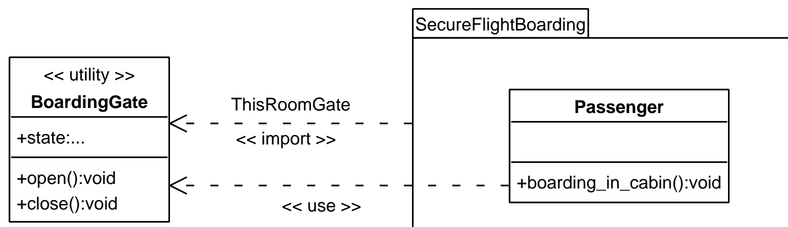


FIG. 5.6 – Illustration du schéma de transformation 17.3

L'inclusion entre les deux machines *SecureFlightBoarding* et *BoardingGate* est représentée par la dépendance ≪import≫ avec le nom d'alias *ThisRoomGate*. La classe *BoardingGate* agit alors comme

un module avec des variables globales et des procédures accessibles par les éléments de modélisation empaquetés au niveau du paquetage *SecureFlightBoarding*. En UML, ceci peut être représenté par le stéréotype ≪utility≫. À ce niveau, l'attribut public *state* est accessible en pré-condition par la méthode *boarding_in_cabin* de la classe *Passenger*. Le rajout du lien de dépendance ≪use≫ entre les classes *Passenger* et *BoardingGate* permet d'illustrer cet accès.

Nous utilisons la représentation en classe utilitaire dans l'intention de garantir l'existence d'un seul ensemble de services et données accessible par toutes les classes du paquetage. Cette unicité est définie d'une manière plus fine et précise au niveau du patron de conception « *Singleton* » (Gamma *et al.,* 1999) dont la structure est la suivante[27] :

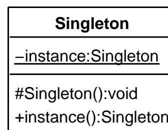


FIG. 5.7 – Structure du patron « *Singleton* »

Les recommandations proposées par E. Gamma *et al.,* (Gamma *et al.,* 1999) pour l'utilisation de ce patron de conception indiquent qu'il s'agit de garantir qu'une classe n'a qu'une seule instance et fournit un point d'accès de type global. La responsabilité d'assurer l'unicité de l'instance est confiée à la classe elle-même. La Fig. 5.7 illustre la structure de ce patron : l'attribut statique privé "*instance*" permet à la classe de contrôler comment et quand les clients y accèdent. Ces derniers n'accèdent à l'instance en question que par le seul intermédiaire de l'opération *Instance*() de la classe *Singleton*.

Schéma 17.4 *Schéma de transformation* « *Inclusion entre machines − Cas* 3b »

Pré-condition	$\mathcal{M}_1 : BMachine \xrightarrow{Schema_2} \mathcal{P} : Package$ et $\mathcal{M}_2 : BMachine \xrightarrow{Schema_1} \mathcal{C} : Class$
Transformation	Les attributs de $\mathcal{C}$ correspondant à des concepts utilisés dans les clauses PROPERTIES ou INVARIANT de $\mathcal{M}_1$, ou encore référencés dans le corps des opérations de $\mathcal{M}_1$ deviennent des attributs publics (donc visibles pour les classes de $\mathcal{P}$). L'inclusion est traduite par une classe « *Singleton* » héritant de la classe $\mathcal{C}$ et liée par une dépendance ≪use≫ ou ≪call≫ aux classes de $\mathcal{P}$ dont les opérations utilisent les attributs ou les opérations de $\mathcal{C}$.

L'intérêt que nous portons à ce patron de conception est justifié par le fait que les opérations au sein d'une machine incluante accèdent à des instances bien déterminées de la machine incluse. Par exemple, les opérations de *SecureFlightBoarding* ont accès à l'instance *ThisRoomGate* de la machine *BoardingGate*.

Pour illustrer l'application du patron « *Singleton* » sur notre cas d'étude nous gardons la transformation de *SecureFlightBoarding* en un paquetage contenant la classe *Passenger*. Ce qui produit le diagramme de la Fig. 5.8. Dans ce diagramme de classes, la classe *ThisRoomGate* est une spécialisation de la classe *BoardingGate* héritant, de ce fait, l'attribut *state* et les méthodes *open* et *close*. L'application du patron « *Singleton* » à ce niveau permet de garantir que *ThisRoomGate* dispose d'une instance unique. Aussi, les éléments du paquetage *SecureFlightBoarding* n'ont-ils accès qu'à cette instance.

[27] Un attribut souligné est un attribut statique.

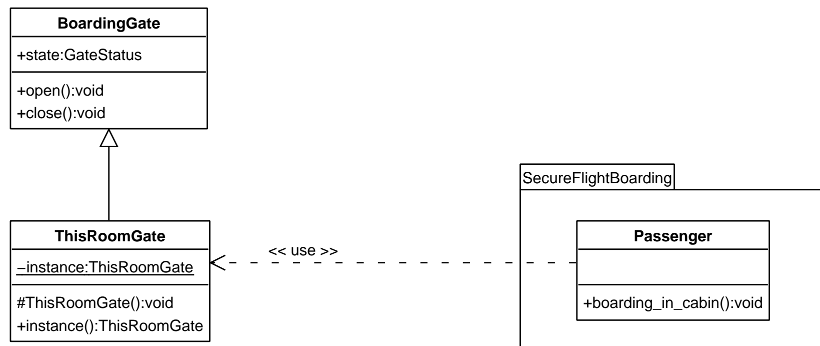


FIG. 5.8 – Application du patron « *Singleton* »

Cas 4. Les données de $\mathcal{M}_1$ sont transformées en des attributs de la classe $\mathcal{C}$; et les données de $\mathcal{M}_2$ sont transformées en des éléments de modélisation empaquetés dans le paquetage $\mathcal{P}$.

Schéma 17.5 *Schéma de transformation « Inclusion entre machines – Cas* 4 »

Pré-condition	$\mathcal{M}_1 : BMachine \xrightarrow{Schema_1} \mathcal{C} : Class$ et $\mathcal{M}_2 : BMachine \xrightarrow{Schema_2} \mathcal{P} : Package$
Transformation	Les attributs de $\mathcal{C}$ correspondant à des concepts utilisés dans les clauses PROPERTIES ou INVARIANT de $\mathcal{M}_1$, ou encore référencés dans le corps des opérations de $\mathcal{M}_1$ deviennent des attributs publics (donc visibles pour les classes de $\mathcal{P}$). L'inclusion est traduite par une dépendance ≪import≫ de $\mathcal{C}$ vers $\mathcal{P}$ et dont le nom d'alias correspond au nom de l'instance de $\mathcal{M}_2$ dans $\mathcal{M}_1$.

Ce schéma de transformation est similaire au schéma de transformation 17.1. La dépendance ≪import≫ créée permet à la classe résultant de $\mathcal{M}_1$ d'accéder aux données du paquetage $\mathcal{P}$ produit pour $\mathcal{M}_2$. La Fig. 5.9 présente une application de ce schéma de transformation considérant que la machine *SecureFlightBoarding* est transformée en une classe et que *BoardingGate* est transformée en un paquetage.

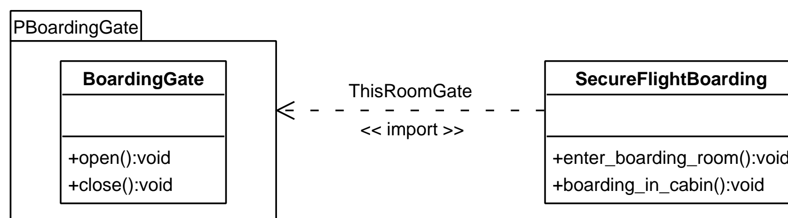


FIG. 5.9 – Illustration du schéma de transformation 17.4

5.3 Prise en compte des liens de raffinement

En UML, aucun mécanisme de raffinement semblable au raffinement en B n'est explicitement défini. Cependant, dans (OMG, 2005), plusieurs stéréotypes standards sont présentés (Annexe C page 669) en vue d'enrichir la sémantique d'UML. Nous considérons ici le stéréotype ≪*refine*≫. Ce dernier est introduit au niveau d'UML2.0 pour indiquer une relation de dépendance entre éléments de modélisation issus de différents niveaux sémantiques, tels que l'analyse et la conception. Dans ce contexte, cette relation

est interprétée de différentes manières sans précision de ses fondements. Plusieurs travaux de recherches se sont alors intéressés à ce mécanisme particulier d'extension en vue de lui associer une sémantique assez précise (Pons, 2006, Lano *et al.,* 2005). Bien que ce mécanisme d'extension d'UML paraisse assez éloigné de la notion de raffinement en B, il peut servir pour indiquer une relation particulière entre traductions UML issues de deux niveaux de raffinement successifs.

Le principe d'aplatissement de spécifications que nous avons présenté au niveau du chapitre 1 peut être utilisé lors de la prise en compte des raffinements, et une sélection de schémas de transformations adéquats permet de produire des vues structurelles UML pour chaque aplatissement. L'avantage de cette technique est qu'elle permet la réutilisation de nos schémas de transformation lors de la transformation des raffinements. Cependant, elle ne garantit pas une homogénéité des diagrammes produits pour deux aplatissements de deux raffinement successifs. Cette homogénéité peut être obtenue par application des mêmes transformations sur les constructions B partagées par les deux niveaux de raffinements. Par exemple, si la relation $weared_objects$ est transformée en une classe associative au niveau abstrait par application du schéma de transformation 9.4, alors il devient plus adéquat de la transformer de manière similaire au niveau concret. Dans le chapitre 9 nous aborderons des critères formels pour préserver l'homogénéité des vues produites pour chaque raffinement.

5.4 Application et discussion

L'élaboration d'un méta-modèle pour B nous a permis, d'une part, de cerner l'ensemble des constructions en B prises en compte par notre démarche de construction de vues structurelles UML, et d'autre part, de spécifier en termes de méta-concepts B et UML les schémas de transformation. À ce niveau, nous ne proposons pas encore un processus automatique de transformation ; nous montrons simplement que le choix du meilleur schéma à appliquer permet de construire des diagrammes structurels lisibles pouvant servir pour un objectif de documentation. Dans cette section, nous présentons l'application des schémas de transformation des structures de base et des structures composées étayée d'une comparaison avec les approches existantes.

5.4.1 Application à la spécification *SecureFlight*

La spécification *SecureFlight* (Fig. 4.1 page 69) est composée de deux machines abstraites pouvant chacune être traduite en classe ou en paquetage. Il est noter qu'à ce niveau le choix du schéma de transformation est laissé à la simple appréciation de l'analyste. Cependant, nous avons présenté une règle heuristique permettant de guider ce choix selon l'existence d'une *BData* susceptible d'être transformée en une classe. Nous estimons alors qu'une manière adéquate de transformer ces deux machines abstraites : *SecureFlightBoarding* et *BoardingGate*, est d'appliquer le schéma de transformation 2 (page 72). En effet, les *BAbstractSet* ainsi que les *BEnumSet* peuvent être traduits en classes via les schémas de transformation 3.2 et 3.3 (page 74).

Transformation de la machine *BoardingGate*. L'ensemble énuméré *GateStatus* de la machine *BoardingGate*, est traduit par une classe ≪*enumeration*≫ grâce au schéma 3.2. Cependant la variable *state*, typée par un *BBasicType* via l'opérateur d'appartenance, ne peut être traduite qu'en un attribut d'une classe, et ce, en appliquant le schéma de transformation 4.1 (page 76). Pour ce faire, Nous appliquons

également le schéma 1 (page 71) en vue de traduire la machine *BoardingGate* en une classe et pouvoir ainsi prendre en compte l'attribut *state*. Aussi, la *BVariable state* devient-elle un attribut de type *BoardingStatus* de la classe *BoardingGate*. Il en est de même pour les opérations de la machine *BoardingGate*. Le tableau 5.1 illustre une succession de schémas de transformation produisant un diagramme de classes UML à partir de la machine *BoardingGate*.

	Schéma	Entrées	Sorties	Commentaires
1	2	*BoardingGate* : *BMachine*	*BoardingGate* : *Package*	
2	1	*BoardingGate* : *BMachine*	*BoardingGate* : *Class*	Classe empaquetée dans le package *BoardingGate*
3	3.2	*GateStatus* : *BEnumSet* *opened* : *BValue* *closed* : *BValue*	*GateStatus* : *Enumeration* *opened* : *EnumerationLiteral* *closed* : *EnumerationLiteral*	Classe ≪*enumeration*≫
4	4.1	*GateStatus* : *BEnumSet* *state* : *BVariable*	*state* : *Property*	Attribut de la classe *BoardingGate* de type *GateStatus*
5	15	*open* : *BOperation* *BoardingGate* : *BMachine*	*open* : *Operation*	Méthode de la classe *BoardingGate*
6	15	*close* : *BOperation* *BoardingGate* : *BMachine*	*close* : *Operation*	Méthode de la classe *BoardingGate*

TAB. 5.1 – Succession de schémas de transformation pour transformer la machine *BoardingGate* en un diagramme de classes

L'application de cette suite de schémas de transformation permet de construire le diagramme de la Fig. 5.10. Il s'agit d'un paquetage contenant la classe *BoardingGate* et la classe ≪enumeration≫ *GateStatus*. La classe *BoardingGate* représente les portes d'embarquement et est caractérisée par l'attribut privé *state*. Ce dernier identifie les états *opened* et *closed* de chaque porte d'embarquement.

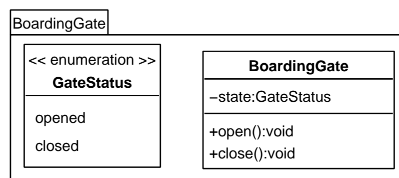


FIG. 5.10 – Traduction de la machine *BoardingGate*

Transformation de la machine *SecureFlightBoarding*. À ce niveau, nous n'allons pas présenter une suite de schémas de traduction aboutissant à la production d'un diagramme de classes à partir de la machine *SecureFlightBoarding* étant donné que cela a été présenté progressivement tout au long du chapitre 4. Ce faisant, nous présentons une intégration des figures : 4.4 (page 73), 4.6 (page 75), 4.11 (page 78), 4.12 (page 79) et 4.16 (page 82) en vue de construire le diagramme de la Fig. 5.11. Dans ce diagramme, le paquetage *SecureFlightBoarding* dispose d'une structure plus complexe que le paquetage *BoardingGate*. En effet, plusieurs classes avec plusieurs dépendances (associations, héritage, agrégation) ont été identifiées. Quant aux opérations *enter_boarding_room* et *boarding_in_cabin*, nous avons choisi de les associer respectivement aux classes *Passenger* et *in_room* pour les raisons suivantes :

– *enter_boarding_room* agit sur les passagers en vue d'assurer leur accès à la salle d'embarquement. De plus elle utilise l'attribut *screening_check* en vue de vérifier si un passager donné peut ou non

accéder à une salle d'embarquement ;

– *boarding_in_cabin* ne concerne que les passagers déjà présents dans la salle d'embarquement et permet de réaliser leur embarquement dans l'avion.

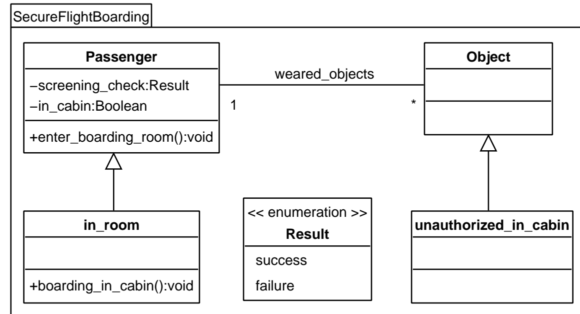


FIG. 5.11 – Traduction de la machine *SecureFlightBoarding*

Transformation de l'inclusion entre *BoardingGate* et *SecureFlightBoarding*. En vue de traduire l'inclusion entre ces deux machines nous allons considérer les schémas de transformation suivants :

– Schéma 17.1 (page 95) car les deux machines considérées sont transformées en paquetages. Ceci permet de créer une dépendance ≪*import*≫ entre ces paquetages avec le nom d'alias *ThisRoomGate*.

– Schéma 17.3 (page 98) ou 17.4 (page 99) car *BoardingGate* est également transformée en une classe. À ce niveau, il nous paraît plus opportun de choisir le schéma 17.4 où il s'agit d'utiliser le patron « *Singleton* ». En effet, l'inclusion entre ces machines définit une instance unique de *BoardingGate* appelée *ThisRoomGate*.

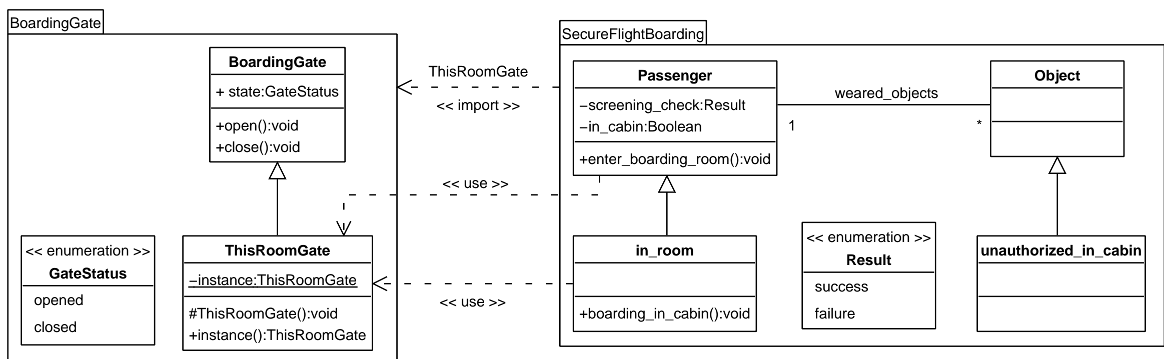


FIG. 5.12 – Application des schémas de transformation à la spécification *SecureFlight*

5.4.2 Comparaison avec d'autres approches

Nous avons présenté un catalogue de traductions de structures de bases et composées étayées par un ensemble de relations sémantiques entre B et UML. Nous n'avons pas abordé un processus d'automatisation orienté méta-modèles étant donné que les transformations que nous proposons ne sont pas

déterministes. Nous ne proposons donc pas la mise en œuvre d'un environnement interactif où l'analyste interagit fortement pour sélectionner le bon schéma de transformation. Cependant, le cadre conceptuel que nous avons proposé ainsi que les différentes transformations présentent ensemble un progrès important par rapport aux travaux existants, et ce, sur deux axes essentiels :

- Une plus grande couverture des constructions en B permettant d'aboutir à des diagrammes plus complets ;
- La définition d'un cadre de travail (*framework*) homogène qui se prête facilement à la mise en œuvre et à l'évolution.

Application à la machine *BoardingGate*. L'approche de (Fekih *et al.,* 2004, Fekih *et al.,* 2006) ne présente aucune règle pour la traduction des machines abstraites, ni pour la traduction des variables définies telle que : *state* ∈ *BoardingStatus*. Nous ne trouvons donc aucune transformation possible pour la machine *BoardingGate*. Par ailleurs, l'application de l'approche de (Voisinet, 2004, Tatibouet *et al.,* 2002) à cette machine aboutit au diagramme de la Fig. 5.13.

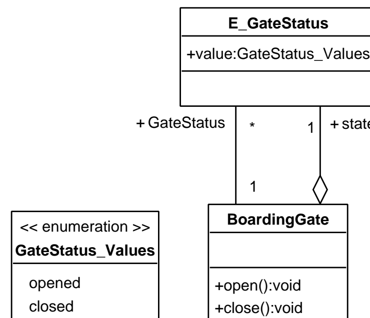


FIG. 5.13 – Application de l'approche de (Voisinet, 2004, Tatibouet et *al.,* 2002) sur BoardingGate

Bien que ce diagramme soit correct d'un point de vue conceptuel, son interprétation en tant que support d'aide à la compréhension de la spécification *BoardingGate* ne semble pas aisée. Remarquons, par exemple, qu'une instance de la classe *BoardingGate* peut être liée à plusieurs objets dont le rôle est *GateStatus* et que chacun de ces objets dispose d'un attribut de type énuméré dont les valeurs possibles sont *opened* et *closed*.

Cette interprétation présente une variété d'instances qui ne sont pas envisagées au niveau de la structure même de la machine *BoardingGate*. En effet, au niveau de la spécification B, deux éléments uniquement jouent le rôle de *GateStatus* pour une *BoardingGate*, à savoir l'élément *closed* et l'élément *opened*. Au niveau du diagramme de classes de la Fig. 5.13, cette ambiguïté est compensée par l'agrégation rajoutée pour chaque élément appartenant à l'ensemble énuméré *GateStatus* (*e.g.* variable *state*). En revanche, le diagramme n'indique pas le fait que l'instance unique de *E_GateStatus* liée à une *BoardingGate* et qualifiée par le nom de rôle *state* est l'une des instances possibles de *E_GateStatus* qualifiées par le nom de rôle *GateStatus* et liées à la même *BoardingGate*.

Application à la machine *SecureFlightBoarding*. À l'exception de $screening_check$ le choix de certaines règles heuristiques parmi celles proposées par (Fekih *et al.,* 2004, Fekih *et al.,* 2006) peut conduire à un diagramme de classes similaire à celui que nous produisons via nos schémas de transformation pour la machine *SecureFlightBoarding*. La relation $screening_check$ ne peut être traduite par les règles de

(Fekih *et al.,* 2004, Fekih *et al.,* 2006) pour la raison suivante : les ensembles énumérés ne sont pris en compte que pour identifier des états dans un diagramme d'états/transitions. La règle proposée dit que : « *si* $r \in A \rightarrow B$ *est une variable où* A *est une classe et* B *un ensemble énuméré alors* B *peut être interprété comme un ensemble d'états possibles de la classe* A ». De ce fait, $success$ et $failure$ peuvent être considérés comme des états de la classe *Passenger*. Nous pensons qu'il est plus adéquat dans une vue structurelle de les considérer comme des valeurs possibles de l'attribut *screening_check* de la classe *Passenger*. L'application de l'approche de (Voisinet, 2004, Tatibouet *et al.,* 2002) sur la machine *SecureFlightBoarding* produit 7 classes liées par plusieurs associations (Fig. 5.14). Ceci est dû au fait qu'une classe est produite pour chaque ensemble et pour chaque relation de la spécification. D'un autre côté les opérations *enter_boarding_room* et *boarding_in_cabin* sont regroupées au sein d'une classe centrale. Ce diagramme nous semble donc plus complexe et moins lisible que la Fig. 5.11.

FIG. 5.14 – Application de l'approche de (Voisinet, 2004, Tatibouet et al., 2002) sur la machine *SecureFlightBoarding*

Prise en compte de l'inclusion entre *BoardingGate* et *SecureFlightBoarding*. Contrairement à notre travail et à celui de (Voisinet, 2004, Tatibouet *et al.,* 2002), l'approche de (Fekih *et al.,* 2004, Fekih *et al.,* 2006) ne présente aucun moyen de prise en compte des dépendances entre machines abstraites. Notre intérêt porte principalement à ce niveau sur l'inclusion en B. Celle-ci est systématiquement traduite par (Voisinet, 2004, Tatibouet *et al.,* 2002) en des associations entre les classes produites pour chaque machine abstraite. Ce qui résulte en un grand nombre d'associations et un diagramme surchargé. Ce faisant, les attributs de la classe générée pour la machine incluse sont transformés en attributs publics. Outre cette transformation qui est appliquée uniquement dans le cas où chaque machine est traduite par une classe, nous avons présenté d'autres transformations possibles du lien d'inclusion. Là encore, exploiter plusieurs traductions s'avère opportun pour aboutir à une plus grande flexibilité et une couverture plus importante des constructions en B.

Comparaison avec des approches de transformation par méta-modélisation. D'autres approches ont adopté une technique basée sur les méta-modèles pour l'intégration de spécifications formelles et semi-formelles. Nous citons, par exemple, (Kim *et al.,* 2005) qui proposent une technique dirigée par

les modèles pour transformer des spécifications Object-Z en des modèles UML. Ce travail identifie les liens entre méta-modèles UML et Object-Z par une correspondance "un à un" vu que les spécifications en Object-Z sont fondées sur le paradigme objet. Dans notre technique ce lien est établi par une correspondance "un à plusieurs". Ce qui rend complexe le processus de transformation. En effet, la partie interprétée de notre approche est insuffisante à elle seule pour un processus de transformation systématique. D'autres critères sont donc requis en vue de pouvoir distinguer la meilleure traduction du modèle B de départ.

Dans le cadre des transformations entre UML et B, R. Laleau et F. Polack (Laleau *et al.,* 2002) ont également adopté une approche par méta-modélisation. Les auteurs proposent un méta-modèle partiel pour B en vue d'assurer la traçabilité entre les modèles UML et la spécification B correspondante. Le méta-modèle de B proposé dans cette approche porte sur des spécifications produites à partir de diagrammes UML, ce qui rend la traçabilité UML/B plus intuitive et plus aisée. Notre technique, exploite ce type de correspondances entre méta-modèles pour la dérivation de vues statiques produites à partir d'une spécification B développée indépendamment d'un modèle UML. Nous pensons que la mise en relation de notre travail avec celui de R. Laleau et F. Polack (Laleau *et al.,* 2002) permettra une meilleure prise en compte de systèmes décrits en UML et dans lesquels certaines parties de sûreté sont spécifiées formellement en B. Nous pensons que cette mise en relation permettra d'étudier la concordance entre les différentes parties formelles et semi-formelles. En effet, leur travail s'avère adéquat pour les parties initialement décrites en UML, alors que le notre s'adresse aux parties initialement développées en B. Une telle étude ouvre la voie vers des perspectives intéressantes et peut se ramener à une mise en relation adéquate des correspondances au niveau des méta-modèles.

5.5 Conclusion

À ce stade de notre démarche, nous nous sommes intéressé à la phase conceptuelle de notre approche où nous nous sommes situé à un niveau de surface pour la mise en relation de spécifications B et de descriptions UML. En effet, notre réflexion a principalement porté sur une approche *interprétée* où il s'agit de définir une syntaxe UML pour B. Notre principal objectif était de spécifier un catalogue de correspondances entre constructions B et UML dans un cadre conceptuel homogène

Une idée importante des techniques de transformation dirigées par les méta-modèles est que les transformations entre deux modèles peuvent être décrites en des termes définis au niveau de chacun de leurs méta-modèles. Partant du fait qu'un méta-modèle définit une syntaxe abstraite à partir de laquelle on peut décrire la sémantique d'un modèle, les règles de transformation qui surgissent de ces techniques sont souvent qualifiées de précises. Cependant, ces règles sont restrictives car elles passent par certains choix de transformation. Par exemple, nous avons présenté au niveau de la Fig. 3.1 une règle de transformation (extraite de (Lemesle, 2000)) du modèle objet au modèle relationnel où il s'agit de traduire chaque classe par une table. Or, certains désirent obtenir autant de tables relationnelles que de classes dans le modèle à objets tandis que d'autres ne voudront des tables relationnelles que pour les classes concrètes du modèle.

Au niveau de la phase *interprétée* de notre démarche nous ne faisons aucune restriction ni aucun choix particulier de transformation. Le choix de la(les) transformation(s) adéquate(s) sera présenté dans les chapitres suivants où nous abordons un niveau plus approfondi avec une approche *calculée* permettant d'atteindre un niveau d'automatisation considérable.

Troisième partie

Analyse formelle de concepts pour l'identification de concepts orientés objets à partir de spécifications B

Chapitre 6

Analyse formelle de concepts : étude préliminaire

« *Très peu de mots suffisent pour montrer la puissance de la pensée abstraite. Via les processus autoréférents, nous pouvons même définir les objets indéfinissables et décrire ceux qui sont indescriptibles. Le paradoxe syntaxique apparent n'est qu'un moyen dynamique d'engendrer de la sémantique.* »

N. Lygeros

« De la pensée de l'abstraction à l'abstraction de la pensée », vol 4, 2003.

Sommaire

6.1 Introduction **109**

6.2 Analyse formelle de concepts : aperçu et usage **110**

6.3 Prise en compte de la structuration des *BDatas* **112**

6.3.1 Diagramme de structure préliminaire 112

6.3.2 Discussion 113

6.4 Prise en compte des *BOperations* **115**

6.4.1 Restructuration préliminaire 118

6.4.2 Application 119

6.5 Bilan et discussion **120**

6.1 Introduction

Dans les chapitres précédents nous avons présenté un cadre conceptuel homogène dans lequel nous avons explicité les correspondances entre B et UML. Bien qu'un tel contexte semi-formel permette de faciliter la mise en œuvre d'un environnement de multi-modélisation prenant en compte et les constructions B et les constructions UML, le processus de dérivation de diagrammes UML reste sujet à plusieurs interventions de la part de l'utilisateur. Cet aspect interactif est dû aux facteurs suivants :

- L'existence de plusieurs transformations possibles pour une même construction en B. Par exemple un sous-ensemble peut être traduit en une sous-classe ou en un attribut booléen.

◇ Les schémas de transformation sont interdépendants. Par exemple, l'application d'un schéma produisant un attribut de classes sous-entend l'exécution au préalable d'un autre schéma produisant une classe.

◇ Les constructions B elles-mêmes sont liées par des dépendances sémantiques. Par exemple, l'association d'un attribut à une classe suppose qu'il existe un lien sémantique (*e.g.* inclusion, appartenance, ...) entre les éléments B à partir desquels l'attribut et la classe ont été dérivés.

Il n'est certes pas aisé de prendre en compte de tels facteurs dans un contexte semi-formel dirigé par les méta-modèles, car cela nous forcerait à proposer une liste exhaustive de cas particuliers − en général énoncés à partir d'expérimentations. Nous adoptons alors une démarche plus précise et rigoureuse pour combler ces limites. Nous montrons dans ce chapitre préliminaire, que l'analyse de ces facteurs dans un cadre formel assure une meilleure conduite des schémas de transformation et permet de franchir un pas important pour l'automatisation du processus de dérivation des vues UML. Nos articles (Idani *et al.,* 2005a) et (Idani *et al.,* 2006b) ont porté sur cette étude préliminaire et montrent que l'application de règles de traduction directe de B vers UML reste superficielle et que l'analyse formelle de concepts présente une contribution importante pour approfondir l'étude de ces transformations.

6.2 Analyse formelle de concepts : aperçu et usage

L'analyse formelle de concepts ou AFC (Wille, 1980, Bernhard *et al.,* 1999) est une technique d'analyse de données qui a été largement utilisée dans plusieurs domaines tels que la fouille de données (Groh *et al.,* 1999), les sciences sociales (Ganter *et al.,* 1989), la psychologie ... En génie logiciel, l'AFC a été typiquement utilisée lors des phases de maintenance et plus particulièrement sur deux axes[28] : (*i*) la restructuration (ou *refactoring*) du code existant (Snelting *et al.,* 1998), et (*ii*) l'identifications de concepts orientés objets (Sahraoui *et al.,* 1999).

L'idée primordiale de l'AFC est de décrire, de manière mathématique et rigoureuse, le monde réel en termes d'objets et d'attributs et d'étudier leurs dépendances grâce à des treillis de concepts (Godin *et al.,* 1995c, Valtchev *et al.,* 2002). Nous n'allons pas présenter les fondations mathématiques de l'AFC, ceci est largement détaillé dans (Bernhard *et al.,* 1999) ; nous allons cependant nous inspirer des notions basiques de l'AFC en vue de proposer une approche de formation de concepts aboutissant à l'identification de concepts orientés objets à partir de spécifications B.

La notion de "**contexte formel**" (appelé aussi "**concept formel**") est le cœur de l'AFC, il s'agit d'une représentation formelle d'une relation binaire entre un ensemble d'éléments du monde réel E et un ensemble de caractéristiques A qui est définie comme suit :

Définition 6.1
Soit R une relation binaire telle que $R \subseteq E \times A$. Les ensembles non-vides $X \in \mathbb{P}(E)$ et $Y \in \mathbb{P}(A)$ qui définissent un contexte formel $C = (X, Y)$ sur R sont tels que :

(*i*) $Y = \{y \in A | \forall x \cdot (x \in X \Rightarrow y \in R[\{x\}])\}$

(*ii*) $X = \{x \in E | \forall y \cdot (y \in Y \Rightarrow x \in R^{-1}[\{y\}])\}$

[28] Le lecteur intéressé peut se référer à (Tilley *et al.,* 2005) où une classification des domaines d'application de l'Analyse Formelle de Concepts en génie logiciel est présentée.

Les approches fondées sur des techniques de formation de concepts pour la ré-ingénierie du code existant (Godin *et al.,* 1995b, Sahraoui *et al.,* 1999), associent habituellement la relation R à un lien de référencement où les éléments de E correspondent à des procédures et fonctions et les éléments de A correspondent à des variables. Soit, par exemple, un code procédural avec trois procédures O_1, O_2, O_3 et trois variables globales A_1, A_2, A_3 et telle que la relation binaire de référencement R est représentée par la matrice booléenne suivante :

$R \widehat{=}$

	A_1	A_2	A_3
O_1	×		
O_2	×		×
O_3		×	

alors les contextes formels construits à partir de R par la définition 6.1 sont :

$$\begin{cases} C_1 = \{\{O_1, O_2\}, \{A_1\}\} \\ C_2 = \{\{O_3\}, \{A_2\}\} \\ C_3 = \{\{O_2\}, \{A_1, A_3\}\} \end{cases}$$

Les couples qui constituent chaque contexte formel de R sont des couples *complets* vu le caractère symétrique de la définition. La complétude à ce niveau découle du fait que pour un couple particulier (X, Y), s'il existe des éléments de O qui n'appartiennent pas à X mais qui sont reliés à tous les éléments de Y par R, alors ces éléments doivent être ajoutés à X (pour que le couple soit complet). Vu la symétrie entre X et Y dans la définition, le même raisonnement s'applique à Y.

Cette notion de complétude associée aux contextes formels est primordiale car elle permet une structuration conceptuelle des données (Godin *et al.,* 1995a). Les techniques de formation de concepts existantes (Godin *et al.,* 1995b, Godin *et al.,* 1995c, Sahraoui *et al.,* 1999, Snelting *et al.,* 1998) se basent alors sur ces regroupements de données et proposent des algorithmes pour mettre en évidence de façon exhaustive les regroupements estimés potentiellement intéressants. Par exemple, le contexte formel C_2 peut être pris à part entière et traduit en orienté objet par une classe encapsulant l'attribut A_2 et la méthode O_3.

Dans le cadre de notre travail, nous nous plaçons à un plus haut niveau d'abstraction où nous partons de spécifications formelles abstraites en B pour produire une structuration conceptuelle de ces spécifications sous forme de diagrammes de classes UML. Pour ce faire, nous nous inspirons de la notion de *contexte formel*, présentée ci-dessus, en vue de proposer un algorithme de formation de concepts basé sur les facteurs suivants :

(*i*) Les dépendances entre opérations et données B,

(*ii*) Les liens entre données B elles-mêmes,

(*iii*) La distribution des opérations sur les différentes classes.

La formalisation et la mise en œuvre de notre algorithme seront présentées dans les chapitres suivants. Cependant, nous montrons dans ce chapitre qu'une prise en compte adéquate de ces facteurs nous permet de sélectionner le(s) schéma(s) de transformation le(s) mieux approprié(s) et d'automatiser ainsi le processus de dérivation des vues UML.

6.3 Prise en compte de la structuration des *BData*s

Nous commençons par distinguer une vue structurelle préliminaire en proposant des règles de transformation systématiques et aboutissant au diagramme le plus détaillé possible. Ensuite, nous définissons des critères de restructuration de ces diagrammes préliminaires en vue de ne garder que les éléments de modélisation les plus pertinents. Ainsi, par un mécanisme de réorganisation, guidé par le cadre conceptuel présenté dans les chapitres 3, 4 et 5 nous arrivons à construire une vue structurelle moins détaillée et plus naturelle et intuitive.

6.3.1 Diagramme de structure préliminaire

Cette première étape de notre démarche se base uniquement sur la partie statique de la spécification B. Nous proposons à ce niveau de construire un premier diagramme d'une manière automatique et simple. Notons que (Voisinet, 2004, Tatibouet *et al.,* 2002) ont suivi une telle démarche d'automatisation via des règles de traduction ; cependant, comme nous avons pu observer, cette automatisation peut conduire à des diagrammes peu lisibles et moins adaptés d'un point de vue documentation. Nous nous servons des diagrammes préliminaires, en grande partie, pour illustrer notre démarche de formation de concepts (celle-ci sera détaillée dans le chapitre 7). Ainsi, nous notons que ces diagrammes ne sont pas présentés au niveau du présent chapitre comme un support d'aide à la compréhension de spécifications B (c'est d'ailleurs pourquoi nous les qualifions de préliminaires). La construction de ces diagrammes suit quatre règles principales :

Règle 6.1 *Chaque instance de la méta-classe BBasicType est traduite par une classe dans le diagramme préliminaire.*

⟶ Ceci est une forme plus faible du schéma de transformation 3 (page 74) où il s'agit de traduire chaque ensemble (abstrait, énuméré ou représentant un type primitif) par une classe préliminaire.

Règle 6.2 *Chaque sous-ensemble est traduit par un héritage.*

⟶ Nous excluons ainsi les transformations sous forme d'attributs booléens. Vu que chaque ensemble est traduit par une classe, alors cette règle permet de distinguer une relation de spécialisation entre les classes produites pour chaque ensemble. Plus précisément il s'agit d'une application directe du schéma de transformation 5.4 page 78.

Règle 6.3 *Chaque relation est traduite par une classe associative.*

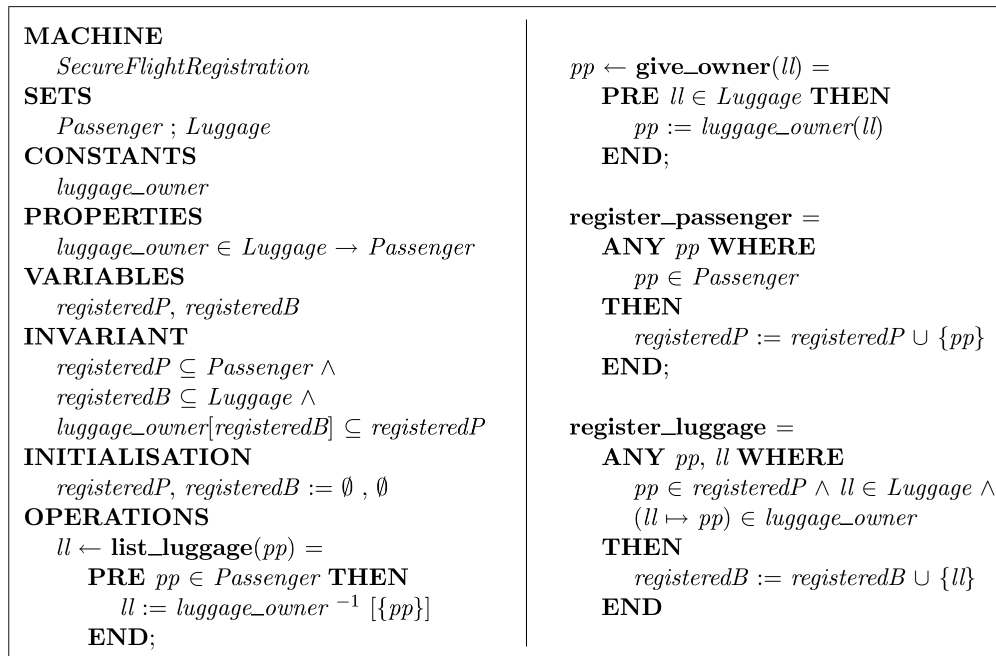
⟶ La transformation d'un ensemble en une classe et d'un sous-ensemble en une sous-classe nous amène à traduire chaque relation entre ensembles par une association entre classes. Vu qu'en B les relations peuvent être manipulées par les opérations de la spécification, nous faisons le choix de les traduire en classes associatives susceptibles d'encapsuler ces opérations. Il s'agit donc d'appliquer le schéma de transformation 9.4 page 83.

Règle 6.4 *Une relation ou fonction R définie par composition avec un BPowType ou un produit cartésien est transformée par application des schémas correspondants* (schémas 13 et 14) *de telle sorte que R soit transformée en une classe associative.*

En vue d'illustrer ces règles de construction d'un diagramme préliminaire, nous allons les appliquer sur une version simple de la spécification *SecureFlight* où nous considérons uniquement des passagers et leurs bagages. Au niveau de la spécification *SecureFlightRegistration*, ci-dessous, les ensembles *registredP* et *registeredB* représentent respectivement les passagers et les bagages enregistrés. La fonction totale *luggage_owner* associe à chaque bagage son propriétaire. Finalement, l'invariant :

$$luggage_owner[registeredB] \subseteq registeredP$$

signifie que les propriétaires des bagages enregistrés sont des passagers déjà enregistrés.



```
MACHINE
    SecureFlightRegistration
SETS
    Passenger ; Luggage
CONSTANTS
    luggage_owner
PROPERTIES
    luggage_owner ∈ Luggage → Passenger
VARIABLES
    registeredP, registeredB
INVARIANT
    registeredP ⊆ Passenger ∧
    registeredB ⊆ Luggage ∧
    luggage_owner[registeredB] ⊆ registeredP
INITIALISATION
    registeredP, registeredB := ∅ , ∅
OPERATIONS
    ll ← list_luggage(pp) =
        PRE pp ∈ Passenger THEN
            ll := luggage_owner ⁻¹ [{pp}]
        END;

    pp ← give_owner(ll) =
        PRE ll ∈ Luggage THEN
            pp := luggage_owner(ll)
        END;

    register_passenger =
        ANY pp WHERE
            pp ∈ Passenger
        THEN
            registeredP := registeredP ∪ {pp}
        END;

    register_luggage =
        ANY pp, ll WHERE
            pp ∈ registeredP ∧ ll ∈ Luggage ∧
            (ll ↦ pp) ∈ luggage_owner
        THEN
            registeredB := registeredB ∪ {ll}
        END
```

FIG. 6.1 – Machine *SecureFlightRegistration*

6.3.2 Discussion

L'application des règles de dérivation du diagramme de structure préliminaire produit le diagramme de la Fig. 6.2 ci-dessous :

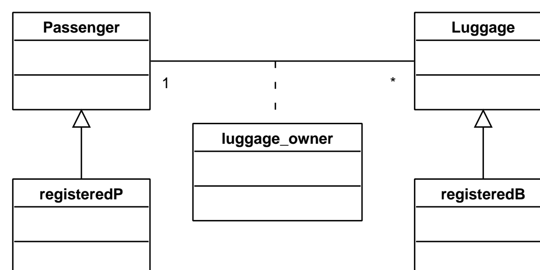


FIG. 6.2 – Diagramme de structure préliminaire produit pour la machine *SecureFlightRegistration*

Bien que ce diagramme permette de donner une vue compréhensible de la structure de la spécification *SecureFlightRegistration*, le résultat devient assez complexe lorsqu'il s'agit de spécifications de

tailles plus importantes. Toutefois, cette partie préliminaire de notre travail fournit de meilleurs résultats, ou du moins des résultats similaires, en comparaison avec les approches existantes. En effet, d'un côté la complexité intentionnelle des diagrammes de structure préliminaires reste moins importante que la complexité des diagrammes produits par l'approche uniforme (Voisinet, 2004, Tatibouet *et al.,* 2002). Celle-ci produit le diagramme de la Fig. 6.3 pour le cas de *SecureFlightRegistration*. Bien que ce diagramme soit construit autour de 4 classes, cette complexité provient du grand nombre d'associations mises en jeu dans le diagramme de classes (7 associations pour le cas de *SecureFlightRegistration*). De plus notre diagramme préliminaire permet une meilleure distribution des opérations. Cette distribution sera étudiée au niveau de la section suivante.

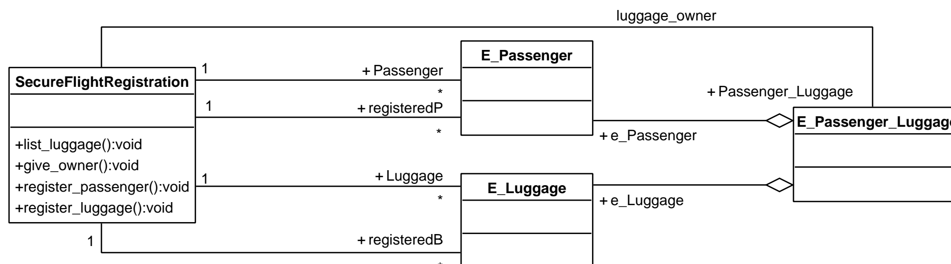


FIG. 6.3 – Diagramme construit par (Voisinet, 2004) pour *SecureFlightRegistration*

D'un autre côté, les règles proposées par (Fekih *et al.,* 2004) produisent des classes uniquement pour les ensembles abstraits qui apparaissent dans le domaine de relations fonctionnelles, ce qui conduit à un diagramme de classes formé par la classe unique *Luggage* et ne permettant pas de modéliser les passagers sans bagages. Une telle transformation peut s'avérer incomplète, étant donné que les opérations *register_passenger* et *list_luggage* ne peuvent être encapsulées que par la classe *Passenger*. En effet, ces opérations sont définies au niveau de la spécification pour réaliser des traitements liés aux passagers et non aux bagages : pour un passager donné, l'une permet de l'enregistrer et l'autre permet de lister ses bagages.

Notons qu'une amélioration de ce travail a été proposée dans (Fekih *et al.,* 2006) où deux conditions ont été rajoutées pour permettre la traduction d'un ensemble abstrait T par une classe :

(i) Si T apparaît comme un super-ensemble d'une inclusion, et/ou

(ii) Si l'utilisateur juge opportun de traduire T en une classe.

L'ensemble abstrait *Passenger* est alors transformé en une classe vu l'existence du sous-ensemble *registeredP*. Nous pouvons donc constater, que si *registeredP* n'est pas défini au niveau de la spécification alors la transformation de *Passenger* en classe dépend uniquement de l'appréciation de l'utilisateur.

Cette première phase de notre démarche permet de construire un diagramme structurel d'une manière automatique dans lequel certaines constructions peuvent s'avérer incorrectes d'un point de vue orienté objet. Par exemple, un ensemble énuméré (*e.g. Result* dans Fig. 4.1, page 69) peut être transformé en une classe alors qu'il devrait être modélisé par un type ou un stéréotype[29] (d'après les schémas de transformation présentés au chapitre 4). De plus, dans ce diagramme, nous ne prenons pas en compte les opérations de la spécification. Ceci fera l'objet de la section suivante. Nous montrons par la suite que la prise en

[29] La classe énumération ne représente pas un ensemble d'instances, mais plutôt tu type défini par énumération.

compte des dépendances entre opérations et données B (*BData*) permet de pallier cette insuffisance des diagrammes préliminaires et d'en dériver des vues UML moins complexes et satisfaisant les schémas de transformation.

6.4 Prise en compte des *BOperation*s

Notre étude ici est axée autour de la dépendance entre les opérations de la spécification et les données B prises en compte dans le diagramme de structure préliminaire. Rappelons que cette dépendance correspond à l'existence d'une instance du lien *Access* du méta-modèle B entre une *BOperation* et une *BData* (Fig. 6.4). Notons également qu'à ce niveau nous allons traiter d'une manière uniforme les différents types d'accès (lecture, écriture et pré-condition). La distinction entre ces différents types d'accès sera détaillée dans le chapitre 9 et aura pour finalité d'optimiser notre technique de formation de concepts.

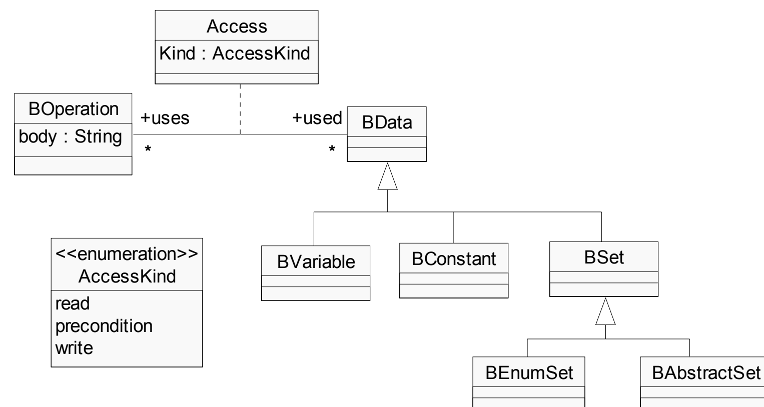


FIG. 6.4 – Extrait du méta-modèle B – dépendance entre opérations et données B

La prise en compte des opérations a pour objectif d'identifier, d'une part, une distribution optimale des opérations sur les classes du diagramme de classes et, d'autre part, de définir un premier critère de pertinence des éléments de modélisation construits au niveau du diagramme préliminaire. Cette notion de pertinence peut être énoncée, d'une manière informelle, comme suit :

La **pertinence** d'une *BOperation* par rapport à une classe $\mathcal{C}$ donnée est définie à partir de l'existence d'un lien *Access* :
- entre la *BOperation* et la *BData* à l'origine de $\mathcal{C}$, ou
- entre la *BOperation* et au moins une *BData* transformée en un attribut de $\mathcal{C}$.

Par exemple, l'opération *register_passenger* ne peut être une opération de la classe *Luggage* car elle n'agit que sur la *BData Passenger*, en outre, il est plus judicieux que cette opération soit encapsulée par la classe *Passenger*. Nous disons alors qu'une classe du diagramme préliminaire est **pertinente** si toutes ses méthodes sont pertinentes. De même, nous disons qu'une association ou une relation d'héritage est pertinente si elle relie des classes pertinentes. Nous déterminons ainsi un premier critère de pertinence sur la base de la pertinence des opérations lorsqu'elles sont associées aux classes préliminaires. Ceci nous amène à formaliser la dépendance entre une *BOperation* et une *BData*, par une relation que nous appelons relation de dépendance de concepts[30] et dont la définition est la suivante :

[30] Le terme "concept" est utilisé de manière générale pour désigner des instances de méta-classes B.

Définition 6.2
Une ***relation de dépendance de concepts*** *est* $\mathcal{I} \in \mathcal{D}ata \leftrightarrow \mathcal{O}$*, où :*
- $\mathcal{D}ata$ *représente l'ensemble des instances des méta-classes BVariable, BConstant et BAbstractSet issues de la spécification B,*
- $\mathcal{O}$ *représente l'ensemble des opérations de la spécification (instances de la méta-classe BOperation).*

avec,
- $(d, o) \in \mathcal{I}$ *signifie que la BData* d *est utilisée par l'opération* o*.*
  Nous notons $\mathcal{D}$ *l'ensemble des données* B *utilisées par au moins une opération.*
  i.e. $\mathcal{D} = \mathrm{dom}(\mathcal{I})$.

Cette relation de dépendance de concepts permet d'identifier les classes auxquelles chaque opération peut être associée. Par exemple, pour le cas de la machine *SecureFlightRegistration* nous avons $\mathcal{I}^{-1}[\{give_owner\}] = \{Luggage, luggage_owner\}$, ce qui signifie que l'opération *give_owner* peut être encapsulée aussi bien par la classe *Luggage* que par la classe associative *luggage_owner*. La relation de dépendance de concepts extraite à partir de la machine *SecureFlightRegistration* peut être représentée par une matrice booléenne comme suit :

	Passenger	*Luggage*	*registeredP*	*registeredB*	*luggage_owner*
list_luggage	×				×
give_owner		×			×
register_passenger	×		×		
register_luggage		×	×	×	×

FIG. 6.5 – Matrice binaire représentant la relation de dépendance de concepts issue de *SecureFlightRegistration*

Remarquons, à partir de cette représentation de la relation de dépendance de concepts, l'existence de plusieurs choix de classes possibles pour chaque opération. Ceci amène à définir d'autres critères de pertinence en vue de pouvoir retrouver la meilleure classe pour chaque opération. Pour ce faire, nous proposons le critère de maximalité de données que nous définissons comme suit :

Définition 6.3
Pour une relation de dépendance de concepts $\mathcal{I}$*, et pour une BData* d *telle que* $d \in \mathcal{D}$ *alors*

$$maximal(d) \Leftrightarrow \forall s \cdot (s \in \mathcal{D} - \{d\} \Rightarrow \mathcal{I}[\{d\}] \not\subset \mathcal{I}[\{s\}])$$

Nous appelons $max(\mathcal{I})$ *l'ensemble de toutes les données maximales de* $\mathcal{D}$*.*

En d'autres termes, la maximalité de données fait référence à un caractère de domination entre données B $-$ relativement à leurs opérations $-$ et exclut les données dites dominées. En effet, soit l'ensemble $dominate$ défini comme suit :

$$dominate = \{a, b | a \in \mathcal{D} \wedge b \in \mathcal{D} \wedge \mathcal{I}[\{b\}] \subset \mathcal{I}[\{a\}]\}$$

alors,

$$max(\mathcal{I}) = \{d | d \in \mathcal{D} \wedge d \notin \mathrm{ran}(dominate)\}$$

Par exemple, le couple $(registeredP, registeredB)$ appartient à l'ensemble $dominate$ étant donné que $\mathcal{I}[\{registeredB\}] \subseteq \mathcal{I}[\{registeredP\}]$ et indique que *registeredB* est dominé par *registeredP*. La prise en compte d'une telle classification des *BData*s permet de guider la phase de réorganisation du diagramme de structure préliminaire. Notre première règle de construction des diagrammes préliminaires traduit chaque *BData* soit en une classe, soit en une sous-classe, soit alors en une classe associative. Ainsi, restructurer le diagramme préliminaire en ne gardant que les classes correspondant aux *BData*s maximales permet de garantir que, pour une opération donnée[31] o telle que $o \in \mathrm{ran}(\mathcal{I})$, il existe au moins une classe, au niveau du diagramme de classes, susceptible de l'encapsuler. D'un autre côté, nous pensons qu'il est plus opportun de traduire les *BData*s dominées en attributs des classes associées à des *BData*s dominantes, et ce, en vue de réduire au mieux la complexité du diagramme de classes.

Nous considérons alors qu'un élément de modélisation, du diagramme préliminaire, associé à une donnée d telle que $d \in max(\mathcal{I})$, peut être vu comme pertinent en tant que tel pour le diagramme. Les autres données nécessitent une étude plus détaillée en vue de les restructurer d'une manière adéquate. Par exemple, la relation de dépendance de concepts issue de la machine *SecureFlightRegistration* (Fig. 6.5) aboutit à :

$$max(\mathcal{I}) = \{Passenger, registeredP, luggage_owner\}$$

Ce qui nous conduit à garder *Passenger* en tant que classe, *registeredP* en tant que sous-classe et finalement, *luggage_owner* en tant qu'association. Dans ce qui suit, nous allons distinguer trois cas de figure selon que la donnée maximale soit à l'origine d'une classe, d'une sous-classe ou d'une classe associative.

Classes issues de *BData*s maximales. Les classes issues de *BData*s maximales sont considérées automatiquement comme pertinentes pour le diagramme de classes. En effet, nous nous attachons à identifier les classes dominantes en vue de pouvoir réduire la complexité des diagrammes préliminaires et d'en dériver des diagrammes plus simples et où toutes les opérations sont convenablement distribuées. Par conséquent *Passenger* est systématiquement maintenue en tant que classe pertinente pour le diagramme de classes.

Sous-classes issues de *BData*s maximales. Nous avons évoqué la pertinence du lien d'héritage par rapport à la pertinence des deux classes concernées par le lien l'héritage. Ainsi, une sous-classe préliminaire correspondant à une *BData* maximale conduit, d'une part, à considérer sa super-classe comme pertinente (même si celle-ci n'est pas issue d'une donnée maximale), et d'autre part, au maintien du lien d'héritage. Par exemple, si $max(\mathcal{I}) = \{registeredP, luggage_owner\}$ alors la *BData Passenger* aurait été considérée comme pertinente vu que :

$$\begin{gathered} registeredP \subseteq Passenger \wedge \\ registeredP \in max(\mathcal{I}) \end{gathered}$$

Classes associatives issues de *BData*s maximales. Comme pour le cas de l'héritage, l'existence d'une association entre deux classes sous-entend que les deux classes sont pertinentes. Une classe associative préliminaire issue d'une donnée maximale conduit à considérer les deux extrémités de

[31] Nous excluons les opérations qui font simplement *skip* où qui réalisent des traitements non liés aux *BData*s de la spécification.

l'association comme classes pertinentes. Par exemple, la maximalité de *luggage_owner* implique la pertinence de la classe *Luggage*.

Classes associatives non maximales et liant des classes issues de *BData*s maximales. Étant donné que les classes issues de *BData*s maximales sont considérées comme pertinentes pour le diagramme de classes alors, les classes associatives non maximales et liant des classes issues de *BData*s maximales sont considérées comme pertinentes en tant qu'associations simples (puisque leurs opérations peuvent être encapsulées dans d'autres classes).

6.4.1 Restructuration préliminaire

La prise en compte des opérations représente un moyen intéressant pour la construction de diagrammes moins complexes. En effet, cette étape a pour objectif de distinguer les éléments de modélisation les plus pertinents compte tenu des opérations qu'ils sont susceptibles d'encapsuler. La phase de restructuration du diagramme préliminaire correspond alors à une distribution des opérations sur les classes estimées pertinentes. Le graphe bi-parties de la Fig. 6.6, issu de la relation $\mathcal{I}^{-1}$, représente pour chaque opération de la machine *SecureFlightRegistration* les classes pertinentes auxquelles cette opération peut être associée. Dans ce graphe la *BData registeredB* est omise car elle ne correspond pas à une classe préliminaire identifiée comme classe pertinente.

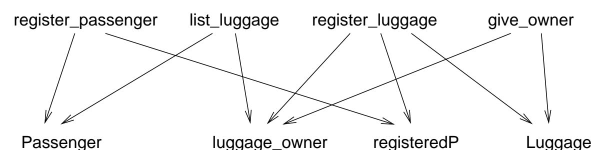


FIG. 6.6 – Graphe bi-parties représentant le(s) choix de classes pour chaque opération de *SecureFlightRegistration* (relation $\mathcal{I}^{-1}$)

Distribution des opérations privées. Les opérations privées d'une classe préliminaire C sont les opérations qui utilisent seulement la donnée B correspondante. Ces opérations sont automatiquement associées à la classe C. Dans le cadre de la Fig. 6.6 toutes les opérations sont partagées. Notons qu'une opération privée se traduit par une flèche unique de l'opération vers la classe à laquelle elle sera associée.

Opérations partagées par une super-classe et sa sous-classe. Dans le cas d'une opération partagée par une super-classe et sa sous-classe nous éliminons la possibilité d'associer l'opération à la sous-classe vu que ceci est implicitement réalisé par le mécanisme d'héritage. Par exemple, l'opération *register_passenger* peut être encapsulée par la classe *Passenger* plutôt que par la classe *registeredP* vu que celle-ci est une spécialisation de *Passenger*.

Opérations partagées par une classe associative et l'une de ses extrémités. Dans ce cas, nous n'envisageons que la possibilité d'associer les opérations partagées à la classe extrémité de l'association. Si la classe associative n'est associée à aucune opération alors elle est transformée en une association simple. Par exemple, les opérations *list_luggage*, *register_luggage* et *give_owner*

ne peuvent être associées à la classe associative *luggage_owner* étant donné qu'elles sont partagées avec les classes *Passenger* et *Luggage*. Ceci remet en cause la pertinence de la *BData luggage_owner* en tant que classe associative et permet de la transformer en une association simple entre *Passenger* et *Luggage*.

6.4.2 Application

Les trois cas de partage précédents permettent de supprimer certains liens du graphe de la Fig. 6.6 ce qui aboutit au graphe suivant :

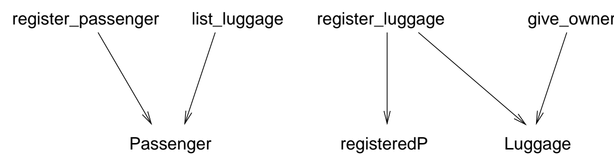


FIG. 6.7 – Graphe bi-parties représentant le(s) choix le(s) plus pertinent(s) de classes pour chaque opération de *SecureFlightRegistration* (sous ensemble de la relation $\mathcal{I}^{-1}$)

Ainsi, seules les classes *Passenger*, *Luggage* et *registeredP* sont susceptibles d'être maintenues au niveau du diagramme de classes. En effet, d'une part, la *BData registeredB* a été identifiée comme non-pertinente en tant que classe vu son caractère dominé ; et d'autre part, il est plus adéquat de transformer *luggage_owner* en une association simple car toutes ses opérations sont partagées avec la *BData Luggage*.

Par ailleurs, nous remarquons que dans ce graphe l'opération *give_owner* est une opération propre à *Luggage* vu qu'elle n'est pas liée à d'autres *BData*s et elle est par conséquent nécessairement associée à la classe *Luggage*. Ceci est également le cas des opérations *register_passenger* et *list_luggage* qui doivent être associées à la classe *Passenger*. Cependant, un choix entre *registeredP* et *Luggage* reste à établir pour l'opération *register_luggage*.

D'ores et déjà, le diagramme de classes résultant de ce processus de transformation dépend de la classe choisie pour encapsuler l'opération *register_luggage*. En effet, associer *register_luggage* à la classe *Luggage* produit uniquement deux groupements conceptuels de données et d'opérations alors que le choix de *registeredP* produit trois groupements. La Fig. 6.8 illustre ces deux cas possibles pour la machine *SecureFlightRegistration*.

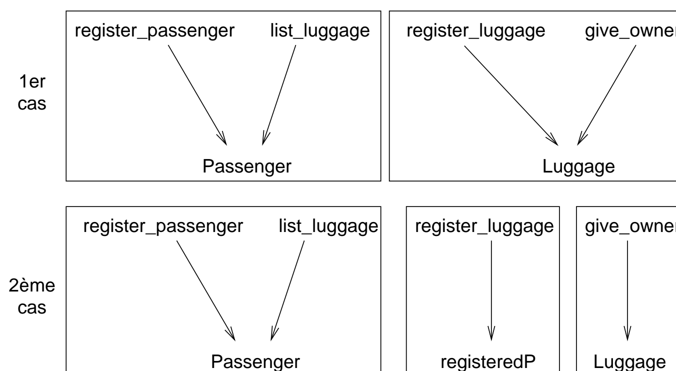


FIG. 6.8 – Groupements conceptuels de données et d'opérations issus de *SecureFlightRegistration*

Dans le premier cas, la *BData registeredP* ne peut être considérée comme une classe étant donné qu'elle ne dispose d'aucune opération. Elle est alors transformée en un attribut booléen de *Passenger* par application du schéma de transformation 5.5 page 78.

Ci-dessous, nous présentons les diagrammes de classes résultant de ces deux cas possibles. Au niveau de la Fig. 6.9, nous distinguons deux classes *Passenger* et *Luggage* encapsulant respectivement les attributs booléens *registerdP* et *registeredB* et illustrant ce qui suit :

(*i*) Un bagage doit appartenir à un et un seul passager ;

(*ii*) Un passager peut être propriétaire de plusieurs bagages ;

(*iii*) Pour un passager donné

- La valeur associée à l'attribut *registeredP*, représente l'état enregistré ou non du passager ;
- La méthode *list_luggage* permet de lister l'ensemble des bagages de ce passager ;
- L'enregistrement est réalisé par exécution de la méthode *register_passenger*.

(*iv*) Pour un bagage donné

- La valeur associée à l'attribut *registeredB*, représente l'état enregistré ou non du bagage ;
- La méthode *give_owner* permet d'identifier le propriétaire du bagage ;
- L'enregistrement est réalisé par exécution de la méthode *register_luggage*.

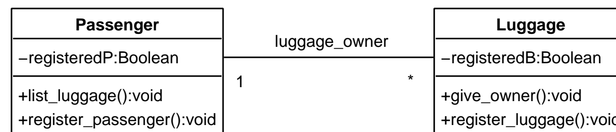


FIG. 6.9 – Restructuration du diagramme de structure préliminaire – 1er cas

Le diagramme de la Fig. 6.10 identifie les passagers enregistrés par la sous-classe *registeredP* de *Passenger*. L'indication supplémentaire pouvant être dégagée de ce diagramme est que l'enregistrement des bagages ne concerne que les passagers enregistrés. Ceci provient du fait que la méthode *register_luggage* est encapsulée par *registeredP*.

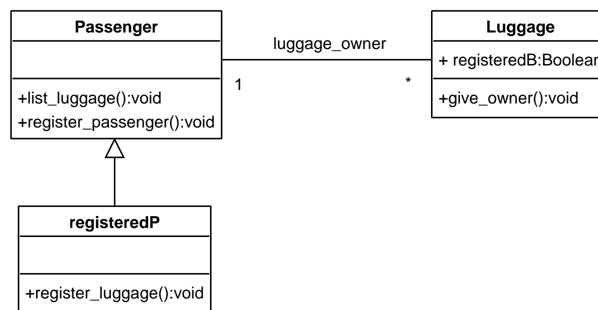


FIG. 6.10 – Restructuration du diagramme de structure préliminaire – 2ème cas

6.5 Bilan et discussion

Nous avons montré au niveau de cette étude préliminaire que la prise en compte des opérations permet de définir un premier critère de pertinence des éléments de modélisation d'une vue structurelle. Ce

critère, qu'est la maximalité de *BData*s, a constitué une base pour guider le processus de transformation et produire des diagrammes de classes simples et compréhensibles d'une manière semi-automatique. En effet, sur la base de ce critère, la transformation que nous avons proposée, à ce niveau, a été axée autour de deux étapes principales :

(*i*) L'étude des classes, sous-classes et classes associatives issues de *BData*s maximales,

(*ii*) La distribution des opérations privées et partagées sur les éléments de modélisation retenus lors de la phase précédente.

La première étape traite les dépendances entre *BData*s et *BData*s maximales en se basant sur l'inclusion entre ensembles, le domaine d'une relation et le co-domaine d'une relation. Quant à la deuxième étape, son principal objectif est l'identification de groupements conceptuels (Fig. 6.8) susceptibles d'être traduits automatiquement en classes.

Cependant, d'autres groupements conceptuels peuvent s'avérer pertinents tels que ceux présentés à la Fig. 6.11 où il s'agit d'associer *register_luggage* à la classe *Passenger*.

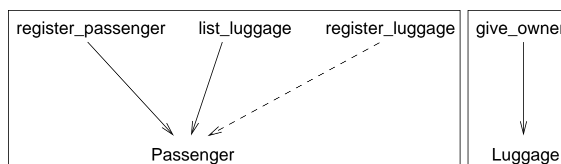


FIG. 6.11 – Autres groupements conceptuels de données et d'opérations dérivés de *SecureFlightRegistration*

Contrairement aux autres liens entre données et opérations, le lien identifié entre *register_luggage* et *Passenger* ne correspond pas à une instance de la méta-classe *Access* entre la *BData Passenger* et la *BOperation register_luggage* (car $(Passenger \mapsto register_luggage) \notin \mathcal{I}$). Ce lien est déduit de l'inclusion entre *registeredP* et *Passenger*. En effet, vu que *registeredP* peut devenir un attribut booléen de *Passenger* alors il est aussi pertinent d'encapsuler les opérations qui l'utilisent au niveau de la classe *Passenger*. Le diagramme de la Fig. 6.12 illustre ce cas de figure.

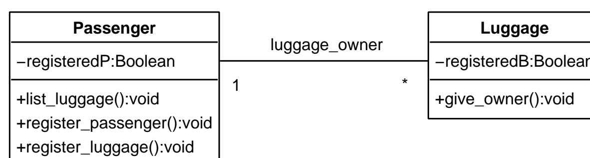


FIG. 6.12 – Autre diagramme de classes produit pour *SecureFlightRegistration*

Ainsi, la mise en évidence des dépendances entre *BData*s et *BOperation*s ainsi que les dépendances entre *BData*s elles-mêmes présente un guide important lors du choix des schémas de transformation pour la construction de vues structurelles à partir de spécifications B, et conduit par conséquent à franchir un pas important vers l'automatisation du processus de transformation. Dans le chapitre suivant, nous présenterons la partie *calculée* de notre processus de transformation où nous définissons un cadre formel pour la mise en œuvre de cette étude.

Chapitre 7

Analyse formelle de concepts : mise en œuvre

« *L'objet est démembré, analysé, de façon à acquérir une connaissance détaillée de ses parties. Mais il y a là un danger. Les parties ne peuvent être correctement comprises indépendamment de leur rapport au tout. Il est nécessaire de revenir à l'objet en tant que système complexe et de saisir la dynamique sous-jacente qui le détermine comme un tout.* »

A. Woods & T. Grant

« La raison en révolte – philosophie marxiste et science moderne », 1995.

Sommaire

7.1 Introduction 123

7.2 Identification des classes candidates 124

7.2.1 Correspondance entre spécifications B et diagrammes préliminaires 124

7.2.2 Relation d'inclusion entre éléments de modélisation préliminaires 126

7.3 Construction de groupements conceptuels de données B 129

7.3.1 Les contextes 129

7.3.2 Construction des contextes 131

7.3.3 Validation et preuve de l'algorithme de construction des contextes 136

7.4 Conclusion 139

7.1 Introduction

Dans le cadre d'une perspective calculée et pragmatique, nous présentons notre processus de production de vues statiques UML à partir de spécifications B au niveau de la Fig. 7.1. Rappelons que les approches existantes (Voisinet, 2004, Tatibouet *et al.,* 2002, Fekih *et al.,* 2004, Fekih *et al.,* 2006) suivent un chemin direct, illustré dans la partie supérieure de la Fig. 7.1 où des règles de traduction extraient le diagramme de classes directement de la structure des données de la spécification B. Vu les limites d'une telle approche, les règles que nous proposons passent par la construction de diagrammes de classes préliminaires qui sont par la suite restructurés en diagrammes de classes pertinents selon des critères particuliers que nous allons formellement définir dans ce chapitre. Cette restructuration est réalisée au moyen de groupements conceptuels, inspirés de la notion de contextes formels (voir

définition 6.1 page 110), et formant des modèles (dits modèles de contextes) traduisibles en diagrammes de classes.

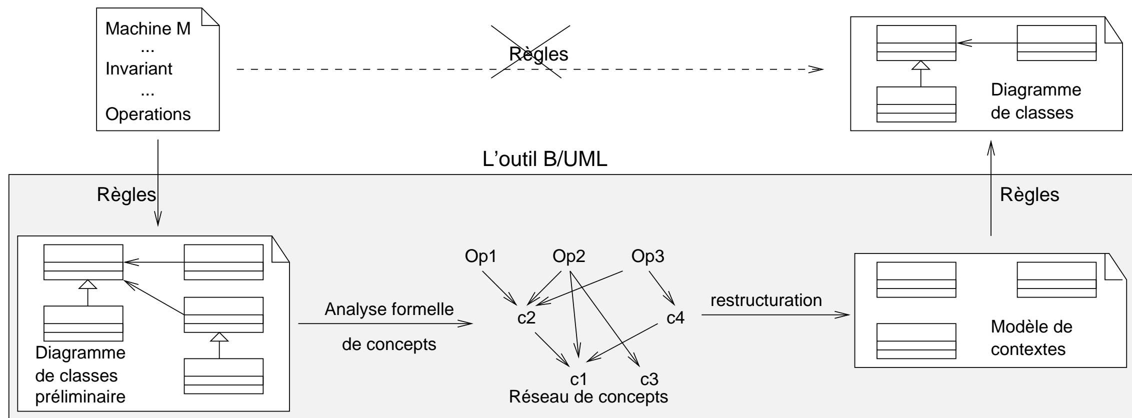


FIG. 7.1 – Approche proposée pour la construction des diagrammes de classes

Notre approche de formation de concepts pour l'automatisation du processus de dérivation de diagrammes de classes a fait l'objet des articles (Idani *et al.,* 2005b) et (Idani *et al.,* 2006d). Nous la détaillons au niveau du présent chapitre tout en mettant l'accent sur les deux étapes suivantes :

(i) identification des classes candidates ;

(ii) formation de contextes sur la base d'un réseau de concepts.

La traduction des modèles de contextes en diagrammes de classes fera l'objet du chapitre suivant. Ce faisant, nous présentons dans ce chapitre une formalisation de notre algorithme de construction de modèles de contextes ainsi que la preuve associée.

7.2 Identification des classes candidates

La phase d'identification des classes candidates se base particulièrement sur les éléments de modélisation distingués au niveau du diagramme de classes préliminaire. Ces derniers proviennent des *BVariable*, *BConstant*, *BAbstractSet* et également des *BEnumSet* et des *BPrimitiveType*.

7.2.1 Correspondance entre spécifications B et diagrammes préliminaires

Avant de définir les correspondances entre spécifications B et leurs diagrammes préliminaires, nous commençons tout d'abord par définir la fonction d'instanciation *Inst* qui associe à chaque méta-classe du méta-modèle B l'ensemble de ses instances au niveau d'une spécification B.

Définition 7.1
Soit $\mathcal{M}_{\mathrm{B}}$ l'ensemble des méta-classes du méta-modèle B, et soit $\mathcal{B}$ l'ensemble des constructions d'une spécification B, alors la relation Inst suivante :

$$Inst \in \mathcal{M}_{\mathrm{B}} \leftrightarrow \mathcal{B}$$

permet d'identifier pour chaque méta-classe du méta-modèle B, l'ensemble de ses instances au niveau d'une spécification B donnée.

Étant donnée la relation d'instanciation précédente, nous définissons la fonction $\mathcal{M}apping$ entre spécifications B et diagrammes préliminaires[32] comme suit :

Définition 7.2
Soit Element l'ensemble des éléments de modélisation du diagramme de classes préliminaire, alors :

$$\mathcal{M}apping \in (Inst[BData] \cup Inst[BPrimitiveType]) \rightarrowtail\!\!\!\!\!\!+ \; Element$$

Où :
- $Inst[BData]$ *représente l'ensemble des instances des spécialisations de la méta-classe BData, à savoir BVariable, BConstant, BAbstractSet et également des BEnumSet*
- $Inst[BPrimitiveType]$ *représente l'ensemble des instances de la méta-classe BPrimitive-Type.*

La fonction $\mathcal{M}apping$ permet de définir les correspondances entre une spécification B de départ et son diagramme de structure préliminaire (les constructions de ce dernier sont spécifiées par l'ensemble $Element$).

C'est une fonction partielle injective car, d'une part, ce ne sont pas toutes les constructions en B qui sont couvertes par cette transformation préliminaire (*e.g.* les éléments typés par un *BBasicType* via l'opérateur d'appartenance ne sont pas couverts...). Et d'autre part, certains éléments de modélisation du diagramme préliminaire ne proviennent pas de la spécification B et sont créés par certains schémas de transformation. Ceci provient précisément de la règle 6.4 (page 112) où il s'agit d'appliquer les schémas de composition des relations et fonctions. Par exemple, la relation *registered_objects* définie par $registered_objects \in (Objects \times Luggage) \rightarrow Passenger$ est transformée via le schéma de transformation 13 (page 88) en rajoutant la classe associative *Object_Luggage* (en d'autres termes $\mathcal{M}apping^{-1}[\{Object_Luggage\}]$ n'est pas défini). De plus, la règle de transformation préliminaire 6.4 traduit systématiquement *registered_objects* par une classe associative tel que présenté au niveau de la Fig. 7.2.

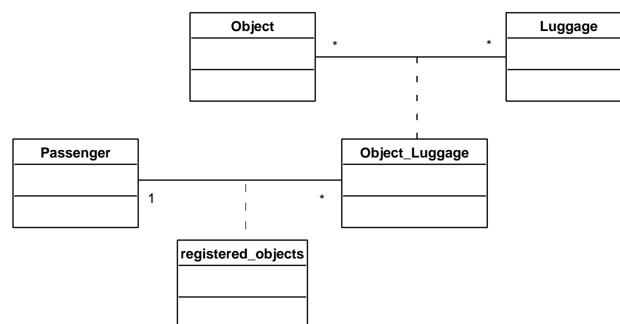


FIG. 7.2 – Transformation préliminaire de *registered_objects*

Étant donné que notre technique de formation de concepts établit une restructuration du diagramme préliminaire, alors nous définissons la projection des notions de relation de dépendance de concepts $\mathcal{I}$ (définition 6.2 page 116) et de maximalité des *BData*s (définition 6.3 page 116) sur les éléments du diagramme de structure préliminaire comme suit :

[32] Ces diagrammes sont construits par quatre règles définies au niveau du chapitre précédent.

Définition 7.3
Soit $\mathcal{I}$ une relation de dépendance de concepts, alors la projection $P_{\mathcal{I}}$ de $\mathcal{I}$ sur les éléments du diagramme préliminaire est :

$$P_{\mathcal{I}} \in \mathcal{M}apping[\mathcal{D}ata] \leftrightarrow \mathcal{O}$$

et est définie par :

$$P_{\mathcal{I}} \mathrel{\widehat{=}} \mathcal{I} \circ \mathcal{M}apping^{-1}$$

Définition 7.4
Soit $P_{\mathcal{I}}$ la projection d'une relation de dépendance de concepts $\mathcal{I}$, et soit e un élément de modélisation préliminaire ($e \in \mathrm{dom}(P_{\mathcal{I}})$), alors la maximalité de e (notée $maximal_elt(e)$) est définie par :

$$maximal_elt(e) \mathrel{\widehat{=}} maximal(\mathcal{M}apping^{-1}(e))$$

Nous appelons alors $max_elt(P_{\mathcal{I}})$ l'ensemble de tous les éléments préliminaires maximaux.

Rappelons que les trois types d'éléments de modélisation (*i.e.* classes, sous-classes et classes associatives), produits au niveau du diagramme préliminaire, sont traités distinctement lors de la phase de restructuration présentée dans le chapitre 6. Nous définissons alors la fonction *Type* suivante :

Définition 7.5
Soit $Element$ l'ensemble des éléments de modélisation du diagramme de classes préliminaire, alors :

$$Type \in Element \rightarrow \{Class, SubClass, AssociativeClass\}$$

La fonction *Type* permet, d'une part, de distinguer les constructions prises compte lors de la phase de restructuration, et d'autre part, de formaliser les différents liens entre ces constructions. Ceci est fondamental dans notre travail, car nos deux principaux facteurs de restructuration sont : (*i*) les dépendances entre *BOperation*s et *BData*s et (*ii*) les dépendances entre *BData*s elles-mêmes.

7.2.2 Relation d'inclusion entre éléments de modélisation préliminaires

L'identification des classes candidates passe également par la prise en compte des dépendances entre *BData*s elles-mêmes ; en effet, nous considérons que la pertinence d'une sous-classe implique la pertinence de sa super-classe, et la pertinence d'une association implique la pertinence de ses extrémités. . . La formalisation de ces dépendances permet d'identifier l'ensemble des classes candidates et entamer la phase d'identification des groupements conceptuels. Pour ce faire, nous définissons la relation d'inclusion $Incl$ entre les éléments de modélisation préliminaires comme suit : pour une *BData* d traduite par une classe préliminaire, $(\mathcal{M}apping(d') \mapsto \mathcal{M}apping(d)) \in Incl$ si et seulement si :

(*i*) Si d' est traduit par une classe préliminaire alors $d' \subseteq d$, et donc $\mathcal{M}apping(d')$ est une sous-classe de la classe préliminaire $\mathcal{M}apping(d)$;

(*ii*) Si d' est traduit par une classe associative préliminaire alors $\mathrm{dom}(d') \subseteq d$ ou $\mathrm{ran}(d') \subseteq d$. En d'autres termes, l'une des classes extrémités de l'association est la classe préliminaire $\mathcal{M}apping(d)$.

Définition 7.6

La relation Incl définie par :

$$Incl \in Element \leftrightarrow Element$$

est telle que $\mathrm{dom}(Incl) \subseteq \mathcal{M}apping[\mathcal{D}ata]$ *et* $\mathrm{ran}(Incl) \subseteq \mathcal{M}apping[\mathcal{D}ata]$. *Avec, pour tout* e *où* $Type(e) \in \{Class, SubClass\}$ *alors* $(e' \mapsto e) \in Incl$ *si et seulement si* :

$$\begin{aligned} & Type(e') = SubClass \wedge \mathcal{M}apping^{-1}(e') \subseteq \mathcal{M}apping^{-1}(e) \\ \vee \; & Type(e') = AssociativeClass \wedge \mathrm{dom}(\mathcal{M}apping^{-1}(e')) \subseteq \mathcal{M}apping^{-1}(e) \\ \vee \; & Type(e') = AssociativeClass \wedge \mathrm{ran}(\mathcal{M}apping^{-1}(e')) \subseteq \mathcal{M}apping^{-1}(e) \end{aligned}$$

Étant donné que l'ensemble $\mathcal{D}ata$ (définition 6.2) représente l'ensemble des instances des méta-classes *BVariable*, *BConstant* et *BAbstractSet*, alors nous excluons de la relation d'inclusion les *BEnumSet*s et les *BPrimitiveType*s. En effet, aucun schéma de transformation n'a été défini pour produire des classes à partir des *BEnumSet*s et des *BPrimitiveType*s, nous ne pouvons alors les inclure pour l'identification des classes candidates. Cependant, leur existence au niveau des diagrammes préliminaires est justifiée par le fait qu'ils seront considérés lors de la phase de restructuration et seront transformés principalement en types.

En vue d'illustrer la relation $Incl$ nous fusionnons les deux spécifications *SecureFightRegistration* (Fig. 6.1, page 113) et *SecureFlightBoarding* (Fig. 4.1, page 69) tout en rajoutant l'invariant :

$$\begin{aligned} & in_room \subseteq registeredP \;\wedge \\ & in_cabin \subseteq registeredP \end{aligned}$$

Ce qui spécifie le fait que les passagers présents dans l'avion ou dans la salle d'embarquement, sont des passagers déjà enregistrés. La spécification B ainsi construite est présentée au niveau de l'annexe E.

Notons que la fusion de deux spécifications ne conserve pas nécessairement la maximalité des *BData*s, nous passons par cette étape pour un objectif d'illustration en vue de disposer d'une structuration de données pouvant produire des vues intéressantes. Nous considérons alors qu'il s'agit d'une nouvelle spécification à partir de laquelle nous allons construire les relations $\mathcal{I}$ et $Incl$.

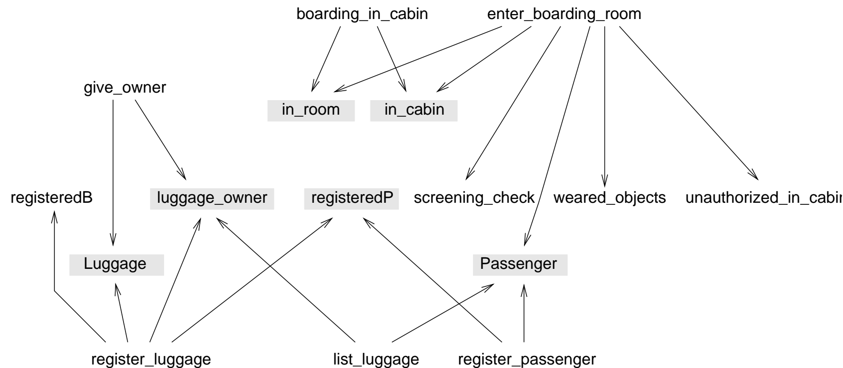


FIG. 7.3 – Graphe illustrant la relation $\mathcal{I}^{-1}$ issue de SecureFlight (Annexe E)

La relation d'inclusion $Incl$ issue de notre cas d'étude est représentée par le graphe de la Fig. 7.4. Nous nous en servons pour identifier les classes candidates (ces classes candidates correspondent aux éléments grisés du graphe) pour le diagramme de classes final. Rappelons qu'au niveau de notre étude

préliminaire, nous avons montré de manière informelle, que la maximalité d'une *BData* peut induire la pertinence d'autres *BData*s. Par exemple, la maximalité de *luggage_owner* implique la pertinence de *Luggage* et de *Passenger* en tant que classes. En d'autres termes l'existence d'un couple $(e' \mapsto e)$ d'éléments de modélisation préliminaires tel que $(e' \mapsto e) \in Incl$ et tel que $e' \in max_elt(P_{\mathcal{I}})$ implique que e' et e peuvent être deux classes candidates. Ainsi, la relation $Incl$ est une formalisation des dépendances que nous avons étudiées au niveau de la section 6.4 du chapitre 6.

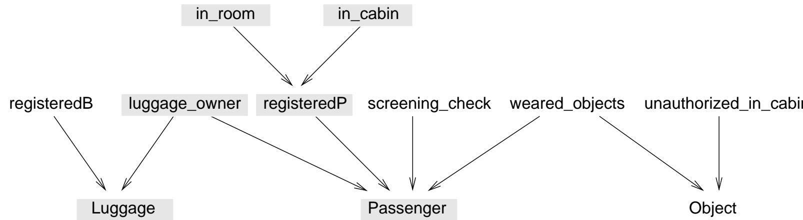


FIG. 7.4 – Graphe illustrant la relation d'inclusion $Incl$

Nous mettons l'accent, à ce niveau, sur le fait que la classe préliminaire *Result* n'est pas prise en compte par la relation $Incl$ car elle provient d'un ensemble énuméré. En effet,

$Inst^{-1}[\{\mathcal{M}apping^{-1}(Result)\}] = BEnumSet$

alors que :

(i) $\mathrm{dom}(Incl) \subseteq \mathcal{M}apping[\mathcal{D}ata]$ et

(ii) $BEnumSet \notin Inst^{-1}[\mathcal{D}ata]$.

L'élément *screening_check* n'est donc lié qu'à *Passenger* ; de ce fait, sa pertinence implique uniquement la pertinence de *Passenger* et il peut éventuellement être traduit en un attribut de *Passenger*.

Avant de présenter formellement la notion de classes candidates, nous définissons tout d'abord l'ensemble $linked$ ($linked \subseteq Element$) des éléments de modélisation qui incluent transitivement, par la relation $Incl^{+}$, d'autres éléments de modélisation issus de *BData*s maximales.

Définition 7.7

Pour un élément de modélisation préliminaire e $(e \in Element)$*,* $e \in linked$ *si l'une des expressions suivantes est établie par induction :*

(i) $maximal_elt(e)$

(ii) $\neg maximal_elt(e) \ \wedge \exists e' \cdot (e' \in Element \wedge (e' \mapsto e) \in Incl \wedge e' \in linked)$

L'appartenance d'un élément de modélisation e à l'ensemble $linked$ est une condition nécessaire pour pouvoir considérer e comme une classe candidate. Cependant, l'exclusion des éléments issus des *BPrimitiveType* et de *BEnumSet* au niveau de la relation $Incl$ implique leur exclusion de l'ensemble $linked$ et par conséquent, de l'ensemble des classes candidates. Cette contrainte imposée par les correspondances entre méta-modèles nous amène également à exclure de l'ensemble des classes candidates les classes associatives préliminaires dont l'une des extrémités provient d'un *BPrimitiveType* ou d'un *BEnumSet*. Supposons par exemple que *screening_check* est maximale alors que *Passenger* ne l'est pas. Dans ce cas, *screening_check* ne peut être considéré comme une classe candidate car son co-domaine ne peut être transformé en une classe. Nous considérons alors que les opérations qui utilisent *screening_check* peuvent être des méthodes de la classe *Passenger*. Nous définissons donc l'ensemble des classes candidates $\mathcal{C}lasses$ ($\mathcal{C}lasses \subseteq Element$) comme suit :

Définition 7.8

Soit $\mathcal{A}$ l'ensemble des classes associatives préliminaires identifiées comme maximales et dont l'une des extrémités provient d'un BPrimitiveType ou d'un BEnumSet :

$$\mathcal{A} = \{e \mid Type(e) = \mathit{AssociativeClass} \wedge maximal_elt(e) \wedge \\ (\mathit{Inst}^{-1}(\mathrm{dom}(\mathcal{M}apping^{-1}(e))) \in \{\mathit{BEnumSet}, \mathit{BPrimitiveType}\} \vee \\ \mathit{Inst}^{-1}(\mathrm{ran}(\mathcal{M}apping^{-1}(e))) \in \{\mathit{BEnumSet}, \mathit{BPrimitiveType}\})\}$$

alors l'ensemble des classes candidates est : $\mathcal{C}lasses = linked - \mathcal{A}$

L'ensemble $\mathcal{A}$ ($\mathcal{A} \subseteq Element$) définit des classes associatives préliminaires servant principalement d'intermédiaire pour l'inclusion de l'une de leurs extrémités dans l'ensemble des classes candidates. Notons que dans notre exemple $\mathcal{A} = \emptyset$ et que les classes candidates correspondent aux éléments grisés des Figs. 7.3 et 7.4.

7.3 Construction de groupements conceptuels de données B

Les classes candidates identifiées par l'ensemble $\mathcal{C}lasses$ sont susceptibles d'être transformées en classes dans le diagramme de classes final. Elles vont regrouper alors un ensemble de données B pour former leurs attributs ainsi que certaines opérations pour construire leurs méthodes. Avant d'aboutir à ces classes finales, la technique de formation de concepts que nous proposons permet de construire, sur la base du principe d'encapsulation d'attributs et de méthodes d'UML, des entités encapsulant des attributs et des méthodes provenant respectivement des données et des opérations de la spécification. Ces entités appelées **contextes** forment ensemble un modèle intermédiaire, dit **modèle de contextes**, systématiquement traduisible en un diagramme de classes.

7.3.1 Les contextes

Nous avons évoqué le concept de *groupement conceptuel de données* défini par (Godin *et al.,* 1995a) grâce à la notion de *contexte formel* et de sa propriété sous-jacente de *complétude*. C'est cette même idée de pouvoir dériver des groupements conceptuels satisfaisant certaines propriétés de pertinence qui motive la notion de contexte dans notre travail. Par analogie à la notion de classe en UML, nous pouvons dire que les données B dans un contexte ressemblent à des attributs dans une classe, et les opérations à des méthodes de la classe. Les contextes seront alors transformés en classes, les opérations des contextes seront des méthodes de ces classes, et les différentes données encapsulées dans les contextes seront réorganisées d'une manière adéquate conformément aux schémas de transformation.

Définition 7.9

Un contexte $\mathcal{F} = (\mathcal{D}', \mathcal{O}')$ issu d'une relation de dépendance de concepts $\mathcal{I}$ ($\mathcal{I} \in \mathcal{D}ata \leftrightarrow \mathcal{O}$) est une paire formée par un ensemble d'éléments de modélisation du diagramme préliminaire $\mathcal{D}'$ et un ensemble d'opérations $\mathcal{O}'$, avec les propriétés suivantes :

$$(\mathrm{P}_1)\ \ \mathcal{M}apping^{-1}[\mathcal{D}'] \subseteq \mathcal{D}ata \quad \wedge \quad \mathcal{D}' \cap \mathcal{C}lasses \neq \emptyset$$

$$(\mathrm{P}_2)\ \ \mathcal{O}' \subseteq \mathcal{O} \quad \wedge \quad \mathcal{O}' \subseteq P_{\mathcal{I}}[\mathcal{D}']$$

Les deux propriétés (P_1) et (P_2), dites **propriétés élémentaires des contextes** expriment ce qui suit :

(P_1) Chaque contexte doit contenir au moins une classe candidate. Cette classe candidate sera à l'origine d'une classe dans le diagramme de classes final ;

(P_2) Chaque opération d'un contexte utilise au moins une donnée B parmi celles originaires des éléments de modélisation contenus dans le contexte. Les opérations ne sont donc pas associées aux contextes d'une manière arbitraire.

Ces propriétés des contextes sont des propriétés élémentaires et sont, à elles seules, insuffisantes pour définir des modèles de contextes systématiquement traduisibles en classes. En effet, d'une part, les propriétés (P_1) et (P_2) n'abordent pas le cas où une opération **privée** accède également à une (ou plusieurs) *BData*(s) non maximale(s) ; et d'autre part, comme pour le cas des opérations, l'existence d'un élément de modélisation e au niveau d'un contexte doit être justifiée par l'existence d'au moins une opération du contexte qui utilise $\mathcal{M}apping^{-1}(e)$. Les attributs d'un contexte ne doivent donc pas être rattachés au contexte d'une manière arbitraire.

Dans le but de prendre en compte ces points, nous commençons tout d'abord par définir les opérations privées des classes candidates comme suit :

Définition 7.10

Soit $\mathcal{I}$ une relation de dépendance de concepts, alors l'ensemble des opérations privées $Private$ d'une classe candidate c ($c \in \mathcal{C}lasses$) est défini par :

$$Private(c) = \{o \mid \forall\, c' \cdot (c' \in \mathcal{C}lasses \wedge c' \neq c \Rightarrow o \in P_{\mathcal{I}}(c) - P_{\mathcal{I}}(c'))\}$$

Les opérations privées d'une classe candidate c sont donc les opérations qui utilisent uniquement $\mathcal{M}apping^{-1}(c)$. Les autres opérations sont dites opérations **partagées**.

Les opérations privées des classes candidates introduisent une première propriété de pertinence des contextes. En effet, vu que chaque contexte peut encapsuler une ou plusieurs classes candidates alors il doit nécessairement encapsuler leurs opérations privées. Cette propriété de pertinence des contextes vient renforcer la propriété (P_2) car celle-ci implique qu'une opération privée d'une classe candidate c ne doit être associée qu'à un contexte encapsulant c. Nous définissons alors la notion de **contexte pertinent** comme suit :

Définition 7.11

Un contexte pertinent issu d'une relation de dépendance de concepts $\mathcal{I}$ est un contexte $\mathcal{F} = (\mathcal{D}', \mathcal{O}')$ avec :

(P_3) $\forall c \cdot (c \in \mathcal{D}' \wedge c \in \mathcal{C}lasses \Rightarrow Private(c) \subseteq \mathcal{O}')$

(P_4) $\forall d \cdot (d \in \mathcal{D}' \wedge d \notin \mathcal{C}lasses \Rightarrow P_{\mathcal{I}}(d) \cap \mathcal{O}' \neq \emptyset)$

Les propriétés (P_3) et (P_4), dites **propriétés de pertinence des contextes** expriment ce qui suit :

(P_3) Les opérations privées d'une classe candidate doivent être encapsulées dans un même contexte que la classes candidate ;

(P_4) Pour chaque élément de modélisation encaspulé dans un contexte, existe au moins une opération qui l'utilise dans le même contexte.

Nous nous efforçons à ce niveau de définir un ensemble de critères de pertinence permettant l'obtention d'un modèle intermédiaire bien formé et pouvant être traduit en un diagramme de classes. Cette affirmation provient du fait que les propriétés élémentaires et de pertinence des contextes ont pour principal objectif de simuler le principe d'encapsulation d'UML. En effet, ce principe incite à créer des classes UML où :

- La structure (attributs) et le comportement (méthodes) d'un objet quelconque sont regroupés au sein d'une même entité. Ceci est reproduit par la définition même d'un contexte et traduit par la propriété (P_1).
- Les méthodes présentent l'interface de l'objet, et donc elles sont étroitement liées à la structure interne de l'objet. Ce qui permet de justifier les propriétés (P_2) et (P_3).
- Les attributs sont définis en vue d'être utilisés et modifiés par les méthodes. En effet, un attribut défini au sein d'une classe et non-utilisé par les méthodes n'a pas de sens. C'est pourquoi nous mettons en évidence la propriété (P_4). En B, une variable d'état non utilisée par les opérations peut être vue comme une erreur de modélisation et peut disparaître sans affecter le modèle B.

7.3.2 Construction des contextes

Dans ce qui suit, nous nous basons sur les propriétés élémentaires et de pertinence des contextes ainsi que sur les différentes constatations précédentes pour proposer un algorithme de construction de contextes pertinents. Nous évoquons également la validation et la preuve formelle en B de cet algorithme.

7.3.2.1 Réseaux de concepts et réseaux de concepts raffinés

La Fig. 6.8 (page 119) illustre deux ensembles de contextes. Bien que ces groupements satisfont les propriétés des contextes, ils ne sont pas complets car ils ne couvrent que les classes candidates et ne prennent pas en compte les autres éléments de modélisation préliminaires provenant de données B utilisées par les opérations. La prise en compte des autres éléments de modélisation du diagramme préliminaire est aussi fondamentale en vue de construire des contextes complets et couvrant le plus de constructions. Par ailleurs, la Fig. 6.11 (page 121) présente deux contextes ne satisfaisant pas la 2$^{\text{ème}}$ propriété de contextes mais qui permettent d'aboutir à un diagramme de classes cohérent (Fig. 6.12). En effet, l'opération *register_luggage* est associée à *Passenger* car elle utilise *registeredP* et que ce dernier est susceptible d'être traduit en un attribut booléen de la classe *Passenger*. Un tel contexte doit alors encapsuler *registeredP* parmi ses attributs.

Ces constatations montrent que la construction des contextes doit être dirigée et par la relation de dépendance de concepts $\mathcal{I}$ et par la relation d'inclusion $Incl$ entre éléments de modélisation préliminaires. Nous proposons alors de composer la projection $P_{\mathcal{I}}$ de la relation $\mathcal{I}$ sur les éléments de modélisation préliminaires, avec la relation d'inclusion $Incl$ et former ainsi un graphe d'éléments de modélisation et d'opérations (ce graphe sera appelé **réseau de concepts**) exploitable par un algorithme de construction des contextes.

Nous pouvons alors définir le réseau de concepts, comme étant une structure qui relie les opérations de la spécification B aux différents éléments de modélisation du diagramme préliminaire (par la relation $P_{\mathcal{I}}^{-1}$) et les éléments de modélisation préliminaires eux-mêmes (par la relation $Incl$) :

Définition 7.12

Soit $\mathcal{I}$ une relation de dépendance de concepts, et $Incl$ la relation d'inclusion issue d'un diagramme de structure préliminaire, alors le réseau de concepts est formé par la relation $\mathcal{J}$ suivante :

$$\mathcal{J} \in (\mathcal{O} \cup \mathcal{M}apping[\mathcal{D}ata]) \leftrightarrow \mathcal{M}apping[\mathcal{D}ata]$$

et est défini par :

$$\mathcal{J} \mathrel{\widehat{=}} P_{\mathcal{I}}^{-1} \cup Incl$$

Le réseau de concepts issu de la fusion des spécifications *SecureFlightBoarding* et *SecureFlightRegistration* est représenté au niveau de la Fig. 7.5. Dans ce graphe, les éléments grisés correspondent aux classes candidates, les arcs en pointillés illustrent la relation $Incl$ et les autres arcs représentent $P_{\mathcal{I}}^{-1}$.

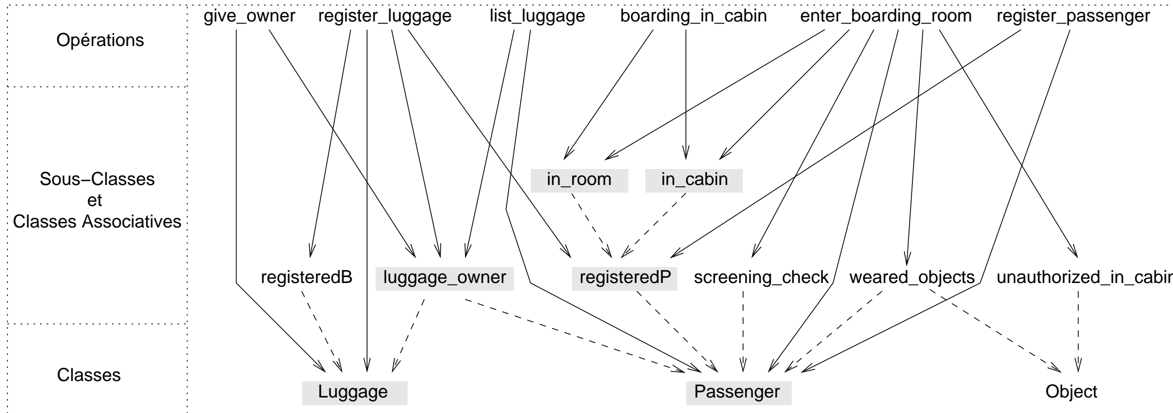


FIG. 7.5 – Relation $\mathcal{J}$ issue de la fusion de *SecureFlightBoarding* et *SecureFlightRegistration*

La construction des contextes, s'effectue par le parcours du réseau de concepts de haut en bas, tout en cherchant des groupements satisfaisant les propriétés de pertinence des contextes. La distribution des opérations sur les différentes classes candidates est réalisée en exploitant les chemins existants au niveau du réseau de concepts entre les opérations et les classes candidates. Par exemple, encapsuler l'opération *register_luggage* au niveau de la classe candidate *Passenger* se réduit, tout simplement, à l'existence d'un chemin entre *register_luggage* et *Passenger* au niveau du réseau de concepts. De par l'existence de ce chemin, la classe *Passenger* est vue comme une cible possible de l'opération *register_luggage*. Cependant, le choix des cibles a été affiné au niveau de notre étude préliminaire où nous avons traité plusieurs cas de partages (section 6.4.1 page 118) et exclu ainsi − de manière implicite − certains choix. En vue de prendre en compte ces différents cas, nous définissons une forme particulière du réseau de concepts $\mathcal{J}$ (appelée réseau de concepts raffiné $\mathcal{J}_{\mathcal{R}}$) par $\mathcal{J}_{\mathcal{R}} \subseteq \mathcal{J}$ et telle que :

$$\mathcal{J}_{\mathcal{R}} \mathrel{\widehat{=}} \mathcal{J} - \{(o \mapsto b) | (o \mapsto b) \in P_{\mathcal{I}}^{-1} \wedge \exists a \cdot ((o \mapsto a) \in P_{\mathcal{I}}^{-1} \wedge (b \mapsto a) \in Incl)\}$$

Ceci se traduit par le fait que lorsqu'une opération o est partagée par deux éléments a et b telle que b est inclus dans a (*i.e.* par la relation $Incl$) alors le lien de o vers b est supprimé de $\mathcal{J}$. Ceci

permettra d'éviter que b soit une cible possible de o. $\mathcal{J}_{\mathcal{R}}$ correspond donc à une structure de donnée exploitable principalement lors de l'identification des cibles possibles pour chacune des opérations de la spécification. Contrairement au graphe de $\mathcal{J}$ où les racines[33] sont uniquement des opérations, dans celui de $\mathcal{J}_{\mathcal{R}}$ les racines peuvent aussi correspondre à des éléments de modélisation préliminaires.

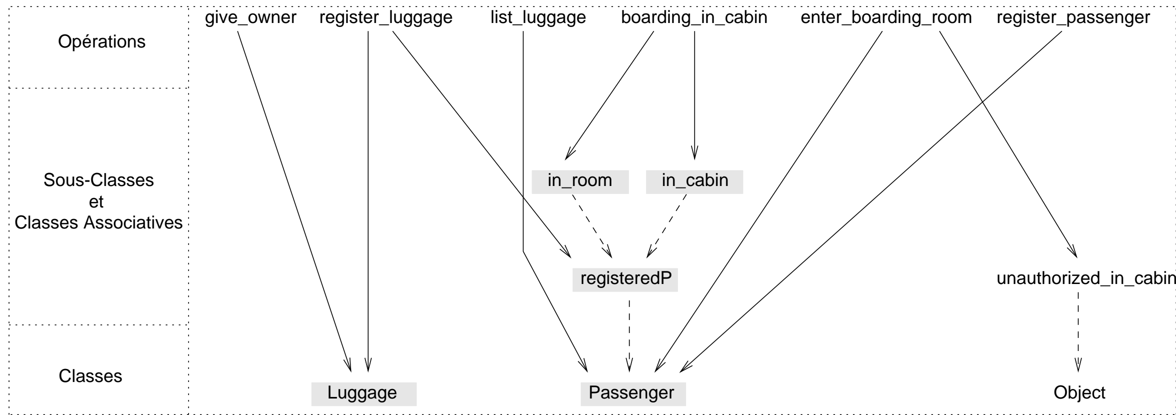


FIG. 7.6 – Sous-graphe du réseau de concepts raffiné $\mathcal{J}_{\mathcal{R}}$ où les racines sont toutes des opérations

La Fig. 7.6 illustre la partie du réseau de concepts raffiné $\mathcal{J}_{\mathcal{R}}$ calculé à partir de la relation $\mathcal{J}$ où nous ne représentons que les racines correspondant à des opérations.

7.3.2.2 Algorithme de construction des contextes

Notre algorithme de construction des contextes suit trois étapes fondamentales :

(i) le choix, pour chaque opération, d'une classe candidate unique comme cible et la formation d'un contexte pour chaque cible ;

(ii) la distribution des éléments de modélisation liés aux classes cibles et aux opérations sur les différents contextes, et

(iii) la prise en compte des éléments de modélisation non contenus dans les contextes.

Description de l'algorithme.

(i) Choisir, pour chaque opération o, une classe candidate c telle que :

$$c \in \mathcal{J}_{\mathcal{R}}^{+}[\{o\}] \cap \mathcal{C}lasses$$

Le choix de c se ramène à l'existence d'un chemin menant de o à une classe candidate c dans **le réseau de concepts raffiné** $\mathcal{J}_{\mathcal{R}}$. À partir de là, nous définissons, la fonction totale $Target$ des opérations vers les classes candidates :

$$Target \in \mathcal{O} \rightarrow \mathcal{C}lasses \;\wedge$$
$$Target \subseteq \mathcal{J}_{\mathcal{R}}^{+}$$

[33] Nous appelons racine d'un graphe orienté, tout nœud r du graphe tel qu'il n'existe aucun chemin dans le graphe qui mène d'un nœud r' (avec r' différent de r) au nœud r.

(*ii*) Former, pour chaque classe candidate c telle que $c \in \mathrm{ran}(Target)$, un contexte $\mathcal{F} = (\mathcal{D}', \mathcal{O}')$ défini par :

$$\begin{aligned} \mathcal{O}' = \quad & Target^{-1}[\{c\}] \\ \mathcal{D}' = \quad & \{c\} \cup (\mathcal{J}^{+}[\mathcal{O}'] \cap (Incl^{-1})^{+}[\{c\}] \cap \mathrm{dom}(P_{\mathcal{I}} \rhd \mathcal{O}')) \end{aligned}$$

Le contexte $\mathcal{F}$ est alors constitué de :

- La classe candidate cible c ;
- Des opérations ayant la classe candidate c pour cible ; et
- Des éléments du **réseau de concepts** $\mathcal{J}$ qui, d'une part, se trouvent sur un chemin menant des opérations de $\mathcal{O}'$ jusqu'à la cible c (*i.e.* appartenant à $\mathcal{J}^{+}[\mathcal{O}'] \cap (Incl^{-1})^{+}[\{c\}]$), et d'autre part, directement liés aux éléments de $\mathcal{O}'$ via $P_{\mathcal{I}}$ (*i.e.* appartenant à $\mathrm{dom}(P_{\mathcal{I}} \rhd \mathcal{O}')$). La Fig. 7.7 illustre cet aspect lorsque les opérations *enter_boarding_room* et *register_passenger* sont associées à la classe candidate *Passenger*. Dans ce cas, l'ensemble $\mathcal{O}'$ inclut les opérations *enter_boarding_room* et *register_passenger*, et l'ensemble $\mathcal{D}'$ inclut les éléments de modélisation préliminaires : *Passenger*, *registeredP*, *in_room*, *in_cabin*, *screening_check* et *weared_objects*.

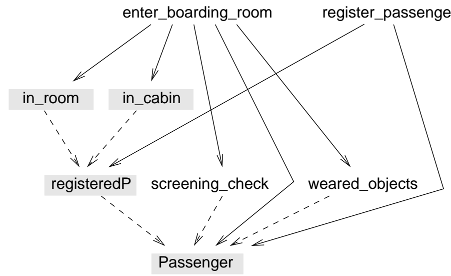


FIG. 7.7 – Illustration de l'étape (*ii*) de l'algorithme

(*iii*) Vérifier que tous les éléments de modélisation dans $\mathcal{M}apping[\mathcal{D}]$ (*i.e.* provenant de *BData*s utilisées par les opérations de la spécification (voir définition 6.2, page 116)) font partie d'au moins un contexte. Si un élément e tel que $\mathcal{M}apping^{-1}[\{e\}] \in \mathcal{D}$ et $\nexists \mathcal{F} \cdot (\mathcal{F} = (\mathcal{D}', \mathcal{O}') \wedge e \in \mathcal{D}')$ alors rajouter e dans tout contexte encapsulant au moins une opération o telle que $o \in P_{\mathcal{I}}[\{e\}]$. Ceci est le cas de *unauthorized_in_cabin*, qui n'apparaît pas dans un chemin menant de l'opération *enter_boarding_room* à une classe candidate, et peut ainsi être rattaché au contexte qui inclut cette opération.

Discussion. Bien que cet algorithme permette de construire des contextes satisfaisant aussi bien les propriétés élémentaires des contextes que les propriétés de pertinence (nous présentons la preuve associée au niveau de la section suivante), il contient un niveau de non-déterminisme portant sur le choix de la fonction $Target$ (étape (*i*)).

Toutefois, nous soulignons que nous pouvons envisager de lister tous les contextes qui permettent de lever ce non-déterminisme et fournir ainsi le moyen, pour l'utilisateur, de déterminer le bon choix entre diagrammes produits pour toutes les configuration de $Target$. En vue de remédier à cet aspect interactif nous proposons, au niveau du chapitre 9, un ensemble de mesures, sous forme de matrices simples et pondérées, permettant de classer tous ces choix et réduire encore cet aspect interactif.

Application. La première étape de l'algorithme consiste à définir la fonction totale $Target$ où il s'agit simplement de choisir une cible pour chaque opération. La liste des cibles possibles identifiées sur la base du réseau de concepts raffiné $\mathcal{J}_{\mathcal{R}}$ de la Fig. 7.6 est illustrée au niveau du tableau ci-dessous :

	Luggage	*in_room*	*in_cabin*	*registeredP*	*Passenger*
give_owner	×				
register_luggage	×			×	
list_luggage					×
boarding_in_cabin		×	×	×	×
enter_boarding_room					×
register_passenger					×

TAB. 7.1 – Illustration des différents choix de cibles pour chaque opération

Nous remarquons, à travers ce tableau, que l'interaction est passée, d'un choix entre une grande variété de transformations de B vers UML (chapitres 4 et 5), à un simple choix de cibles pour chacune des opérations de la spécification. Notons également, que le non-déterminisme à ce niveau porte uniquement sur les opérations *register_luggage* (avec 2 choix possibles) et *boarding_in_cabin* (avec 4 choix possibles). Les quatre autres opérations sont affectées de manière déterministe à leurs cibles uniques.

Soit par exemple, la fonction $Target$ suivante :

$$\begin{aligned} Target \;=\; & \{(give_owner \mapsto Luggage), \\ & (register_luggage \mapsto registeredP) \\ & (list_luggage \mapsto Passenger), \\ & (boarding_in_cabin \mapsto in_room), \\ & (enter_boarding_room \mapsto Passenger), \\ & (register_passenger \mapsto Passenger)\} \end{aligned}$$

alors l'algorithme présenté précédemment construit les 4 contextes suivants :

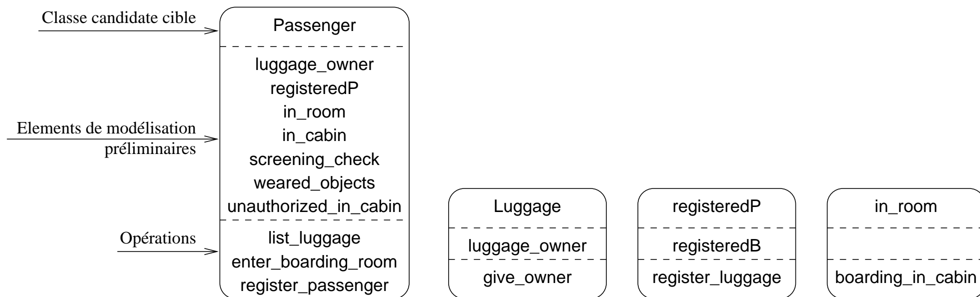


FIG. 7.8 – Modèle de contextes issu de la fusion de *SecureFlightBoarding* et *SecureFlightRegistration*

Pour des raisons de clarté nous représentons un contexte de la même manière qu'une classe. Pour un contexte $\mathcal{F} = (\mathcal{D}', \mathcal{O}')$, nous avons :

- Le premier compartiment illustre la classe candidate c choisie comme cible des opérations du contexte ($\{c\} = Target^{-1}[\mathcal{O}']$) ;
- Le second compartiment contient les autres éléments de modélisation ($\mathcal{D}' - \{c\}$) ; et,
- Le troisième compartiment présente les opérations du contexte ($\mathcal{O}'$)

Observons au niveau de ce modèle de contextes que *unauthorized_in_cabin* est encapsulé par le contexte *Passenger* bien qu'il n'appartienne à aucun chemin dans le réseau de concepts menant jusqu'à *Passenger*. Nous notons que ceci provient de l'application de la troisième étape de l'algorithme. En effet, *unauthorized_in_cabin* est associé au même contexte que l'opération *enter_boarding_room* étant donné que $\mathcal{I}[\{unauthorized_in_cabin\}] = \{enter_boarding_room\}$. Une autre application de cette troisième étape de l'algorithme est illustrée par *registeredB* au niveau du contexte *registeredP*. Bien qu'il n'y ait aucun lien sémantique entre ces deux *BData*s leur appartenance à un même contexte est justifié par l'association de l'opération *register_luggage* à la classe candidate *registeredP*. Notons aussi que *luggage_owner*, ainsi que *in_room* et *registeredP* apparaissent plusieurs fois au niveau du modèle de contextes de la Fig. 7.8. Ceci provient du fait que l'application des étapes de l'algorithme n'impliquent pas un changement de la structure du réseau de concepts. De plus les contextes sont vus comme étant des groupements d'éléments de modélisation préliminaires et d'opérations et non des classes d'un diagramme de classes. Le modèle de contextes forme donc un modèle pivot assurant la restructuration du diagramme préliminaire en un diagramme de classes dit pertinent.

Les classes du diagramme préliminaire correspondant aux contextes sont systématiquement maintenues en tant que classes dans le diagramme de classes final. Les autres éléments du diagramme préliminaire sont réorganisés en prenant en compte leur distribution au niveau des contextes. Par exemple, la Fig. 7.9 présente une transformation du contexte *Luggage* indépendante des autres contextes. Cependant, le maintien de la classe *Passenger* induit la transformation de *luggage_owner* en une association simple entre les classes *Luggage* et *Passenger* (Fig. 8.2 page 144). Ainsi, un tel modèle de contextes sert de base pour guider le choix des schémas de transformation d'une spécification B en un diagramme de classes UML. Ces transformations seront détaillées au niveau du chapitre 8.

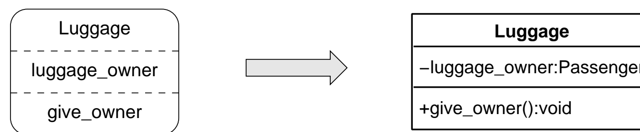


FIG. 7.9 – Exemple de transformation d'un contexte en une classe

7.3.3 Validation et preuve de l'algorithme de construction des contextes

Nous montrons à ce niveau, que les contextes formés par notre algorithme satisfont aussi bien les propriétés élémentaires des contextes (propriétés P_1 et P_2) que les propriétés de pertinence (propriétés P_3 et P_4). Nous considérons $\mathcal{G}$ un modèle de contextes formé par n contextes. En d'autres termes, n est le nombre de contextes produits par une exécution particulière de notre algorithme.

$$\mathcal{G} = \bigcup_{i=1}^{n} \mathcal{F}_i \quad avec \quad \mathcal{F}_i = (\mathcal{D}'_i, \mathcal{O}'_i)$$

Preuve de la propriété ($\mathbf{P_1}$). La propriété (P_1) atteste que dans tout contexte du modèle $\mathcal{G}$ existe au moins un élément de modélisation préliminaire correspondant à une classe candidate. Nous pouvons alors la reformuler en :

$$(P_1) \equiv \forall \mathcal{F} \cdot (\mathcal{F} \in \mathcal{G} \wedge \mathcal{F} = (\mathcal{D}', \mathcal{O}') \Rightarrow \exists c \cdot (c \in \mathcal{C}lasses \wedge c \in \mathcal{D}'))$$

Cette propriété est vérifiée par construction vu que les attributs d'un contexte contiennent la classe candidate à l'origine du contexte. En effet, nous construisons un contexte pour chaque classe candidate dans $\mathrm{ran}(Target)$, en incluant cette classe candidate dans l'ensemble des attributs du contexte. L'algorithme produit donc autant de contextes que d'éléments dans $\mathrm{ran}(Target)$, ce qui se traduit par :

$$\forall \mathcal{F}(\mathcal{F} \in \mathcal{G} \wedge \mathcal{F} = (\mathcal{D}', \mathcal{O}') \Rightarrow \exists c \cdot (c \in \mathcal{D}' \wedge c \in \mathrm{ran}(Target)))$$

Vu que $c \in \mathrm{ran}(Target) \Rightarrow c \in \mathcal{C}lasses$ (d'après la définition de la fonction $Target$ nous avons $\mathrm{ran}(Target) \subseteq \mathcal{C}lasses$), alors la formule précédente est équivalente à ($\mathbf{P_1}$).

Dans ce qui suit, nous considérons $\mathcal{F} = (\mathcal{D}', \mathcal{O}')$ un contexte particulier construit par notre algorithme ($\mathcal{F} \in \mathcal{G}$) et cl la classe candidate à l'origine de $\mathcal{F}$ ($\{cl\} = Target[\mathcal{O}']$).

Preuve de la propriété ($\mathbf{P_2}$). La propriété ($\mathbf{P_2}$) indique que chaque opération d'un contexte doit utiliser au moins un élément de modélisation encapsulé dans le même contexte. Nous pouvons alors la reformuler en :

$$(\mathrm{P}_2) \equiv \forall o \cdot (o \in \mathcal{O}' \Rightarrow \exists c \cdot (c \in \mathcal{D}' \wedge c \in P_{\mathcal{I}}^{-1}[\{o\}]))$$

La fonction totale $Target$ est une partie du réseau de concepts $\mathcal{J}$ car :

$$Target \subseteq \mathcal{J}_{\mathcal{R}}^{+} \wedge \mathcal{J}_{\mathcal{R}}^{+} \subseteq \mathcal{J}^{+} \quad \Rightarrow \quad Target \subseteq \mathcal{J}^{+}$$

De ce fait, $Target(o) = cl$ implique qu'il existe un chemin de o vers cl dans le graphe associé à $\mathcal{J}$ et que ce chemin comprend nécessairement un élément utilisé par o. Notre algorithme introduit dans l'ensemble $\mathcal{D}'$ du contexte formé pour cl tous les éléments se trouvant sur ces chemins et appartenant à $\mathrm{dom}(P_{\mathcal{I}} \rhd \{o\})$.

Preuve de la propriété ($\mathbf{P_3}$). Cette propriété désigne le fait qu'une opération privée d'une classe candidate c est toujours associée au contexte encapsulant c.

$$(\mathrm{P}_3) \equiv \forall c \cdot (c \in \mathcal{D}' \wedge c \in \mathcal{C}lasses \Rightarrow \forall o \cdot (o \in Private(c) \Rightarrow o \in \mathcal{O}'))$$

Vu que les opérations privées ne concernent que les classes candidates provenant de *BData*s maximales, alors pour une opération donnée o nous avons :

$$o \in Private(c) \quad \Rightarrow \quad c \in max_elt(P_{\mathcal{I}}) \qquad \text{où c est une classe candidate}$$

Notons qu'une classe candidate n'est pas nécessairement maximale. Ici, c est maximale, du fait qu'elle dispose d'une opération privée. Deux cas de figure sont possibles :

(*a*) la classe cl cible de l'opération o provient d'une *BData* maximale ($Target(o) \in max_elt(P_{\mathcal{I}})$), dans ce cas $o \in Private(cl)$, et l'opération o sera incluse dans l'ensemble $\mathcal{O}'$ du contexte formé pour cl.

(*b*) la classe candidate cl cible de l'opération o ne provient pas d'une *BData* maximale (c'est-à-dire $Target(o) \notin max_elt(P_{\mathcal{I}})$). Dans ce cas, il existe un chemin de o vers cl dans le réseau de concepts $\mathcal{J}$ passant par une autre classe candidate c telle que $o \in Private(c)$. Ce qui conduit à l'introduction de l'élément de modélisation c dans l'ensemble $\mathcal{O}'$ du contexte construit pour cl.

Preuve de la propriété (P_4). Cette propriété implique que pour tout élément de modélisation e attribut d'un contexte $\mathcal{F}$, il existe une opération o parmi celles encapsulées par $\mathcal{F}$ telle que $(e \mapsto o) \in P_{\mathcal{I}}$:

$$(P_4) \equiv \forall e \cdot (e \in (\mathcal{D}' - Target^{-1}[\mathcal{O}']) \Rightarrow P_{\mathcal{I}}[\{e\}] \cap \mathcal{O}' \neq \emptyset)$$

Pour un élément de modélisation e autre que la classe candidate cible cl ($cl = Target^{-1}[\mathcal{O}']$), nous avons établi la condition nécessaire suivante pour le rajout de e à l'ensemble $\mathcal{D}'$:

$$e \in \mathrm{dom}(P_{\mathcal{I}} \rhd \mathcal{O}')$$

En effet, deux cas de figure ont été identifiés respectivement au niveau des étapes (*ii*) et (*iii*) de l'algorithme :

(*ii*) L'élément e appartient à un chemin menant d'une opération o telle $o \in \mathcal{O}'$ à la classe cible cl et satisfaisant $e \in \mathcal{J}^{+}[\mathcal{O}'] \cap (Incl^{-1})^{+}[\{e\}] \cap \mathrm{dom}(P_{\mathcal{I}} \rhd \mathcal{O}')$.

(*iii*) L'élément e n'appartient à aucun chemin menant d'une opération o jusqu'à cl. Dans ce cas, l'algorithme rattache e à $\mathcal{F}$ si et seulement si $\mathcal{O}'$ contient au moins une opération qui utilise la *BData* $\mathcal{M}apping^{-1}(e)$.

Démarche de preuve par l'atelier B. Le but de cette section n'étant pas de présenter une formalisation en B de notre algorithme de formation de concepts, nous mettons seulement l'accent sur les résultats produits par l'atelier B. Cette formalisation est néanmoins détaillée dans (Idani *et al.,* 2006e, Idani *et al.,* 2006d). Nous présentons donc succinctement la démarche poursuivie en vue d'effectuer la preuve formelle des propriétés élémentaires et de pertinence des contextes. Pour ce faire, nous avons construit 4 machines B dont les opérations respectives sont : *Compute_Classes*, *Determine_Private_Operations*, *Choose_Targets* et *Build_Contexts*. Ces machines calculent respectivement et successivement l'ensemble des classes candidates, les opérations privées d'une classe, la fonction $Target$ et les contextes pertinents. Chaque machine a comme invariant les propriétés des résultats calculés aux étapes précédentes. Chaque opération contient un prédicat (ANY) qui indique comment est calculé le résultat et un autre prédicat (ASSERT) qui déclare les propriétés du résultat. Ce sont ces propriétés qui deviennent des obligations de preuve. Ci-dessous, l'exemple de l'opération *Compute_Classes* qui permet de calculer l'ensemble des classes candidates issues d'une spécification B en vérifiant que les concepts maximaux sont aussi des classes candidates :

```
Compute_Classes ≙
    ANY cls WHERE
        cls ⊆ Data ∧ cls = Incl*[maximal]
    THEN
        ASSERT   maximal ⊆ cls
        THEN     Classes := cls
        END
    END
```

Résultats produits par l'atelier B. Le tableau ci-dessous, donne le nombre des obligations de preuve produites par l'atelier B pour chacune des opérations précédentes. Nous notons que toutes ces obligations de preuves ont été prouvées (43 preuves automatiques et 11 preuves interactives). Nous avons réalisé cette preuve dans le but de garantir que pour toute spécification B conforme à notre méta-modèle, il est possible de former des contextes qui satisfont nos 4 propriétés.

	Preuves	
	Automatiques	**Interactives**
Calcul des classes candidates	7	1
Identification des opérations privées	9	1
Détermination des cibles	13	6
Construction des contextes	14	3
Total	43	11

TAB. 7.2 – Obligations de preuves produites par l'atelier B pour chaque étape de l'algorithme

7.4 Conclusion

Ce chapitre a eu pour objectif de définir un cadre formel traitant, outre la structure de la spécification B, les dépendances entre les différentes constructions en B prises en compte au niveau du méta-modèle B. Sur cette base, nous avons défini et exploité un réseau de concepts en vue de produire des modèles intermédiaires, dits modèles de contextes, satisfaisant des critères de pertinence et traduisibles en diagrammes de classes. Cette traduction sera présentée au niveau du chapitre suivant.

Cependant, chaque modèle de contextes est produit pour une configuration particulière de la fonction $Target$ et résulte, par conséquent, en un diagramme de classes particulier. Bien que la phase de choix des cibles est interactive, notre technique réduit significativement la tâche de l'analyste. En effet, il ne s'agit pas de choisir entre un ensemble de transformations interdépendantes, et gérer d'une manière manuelle la cohérence du résultat, mais plutôt de choisir pour certaines opérations une unique classe candidate cible. La mise en œuvre des schémas de transformations est par la suite réalisée d'une manière automatique. Ce faisant, cette interaction introduit une certaine souplesse et flexibilité pour la construction des diagrammes de classes. Nous notons également que notre algorithme peut lister toutes les configurations possibles, et produire ainsi une variété de représentations structurelles. Une telle démarche peut être considérée comme avantageuse, dans le sens où elle permet de donner plusieurs points de vue sur une même spécification B. Son inconvénient majeur est que le nombre de diagrammes représentant une même spécification a tendance à exploser pour des spécifications de très grande taille. Pour remédier à cette limite nous présenterons au niveau du chapitre 9 une technique d'optimisation fondée sur l'analyse formelle que nous avons menée dans le présent chapitre.

Chapitre 8

Transformation des modèles de contextes en diagrammes de classes

« *Mais dès que je perçois ou que je présuppose que cela implique un rapport à un état précédent d'où la repésentation dérive d'une règle, alors je me représente quelque chose [...], c'est-à-dire que je reconnais un objet que je dois poser dans le temps à une certaine place déterminée et qui ne peut être autrement en raison de l'état précédent.* »

Kant

« Critique de la raison pure », 1787.

Sommaire

8.1 Introduction 141

8.2 Règles de transformation 142

8.2.1 Définition 142

8.2.2 Exemple d'application 143

8.3 Des contextes aux classes 144

8.4 Transformation des attributs des contextes 147

8.4.1 Présentation 147

8.4.2 Visibilité des attributs des classes 151

8.5 Transformation des opérations des contextes 151

8.6 Synthèse 152

8.1 Introduction

Notre processus de dérivation de diagrammes de classes UML à partir de spécifications B est fondé sur un algorithme de formation de concepts permettant la construction, à partir d'un diagramme de structure préliminaire, des modèles dont les entités répondent à des critères de pertinence. Dans ce chapitre, nous présentons les règles de restructuration qui permettent une réorganisation systématique du diagramme de structure préliminaire, et ce, sur la base du modèle de contextes. Nous avons présenté les fondements de cette réorganisation dans (Idani *et al.,* 2006e) où nous avons proposé des règles plutôt informelles. Le présent chapitre est une extension de l'article (Idani *et al.,* 2006e). Nous

y exprimons de manière plus rigoureuse nos règles en les ramenant à une sélection du(des) schéma(s) de transformation adéquats dépendant de la structure du modèle de contextes.

8.2 Règles de transformation

8.2.1 Définition

Une règle de transformation est une suite d'instructions formées au moyen des principales constructions suivantes :

RÈGLE	::=	**Begin** INSTRUCTION **End**
INSTRUCTION	::=	CONDITION | CHOIX | SCHÉMA | BLOC1 | BLOC2
CONDITION	::=	**If** *"prédicat"*[a] **Then** INSTRUCTION **End If ;**
		| **If** *"prédicat"* **Then** INSTRUCTION **Else** INSTRUCTION **End If ;**
CHOIX	::=	**Case Inst**$^{-1}$(*"donnée B"*[b]) **:** *"méta-classe B"* ⇒ INSTRUCTION **;**
		OTHERS **End Case ;**
OTHERS	::=	*"méta-classe B"* ⇒ INSTRUCTION **;**OTHERS | ε
SCHÉMA	::=	**run** *"schéma"*[c] **;**
BLOC1	::=	**Identify some** TYPEDVAR **such that «***"construction B"***»**
		In INSTRUCTION **End In ;**
TYPEDVAR	::=	LIST_ID **:***"méta-classe B"* | LIST_ID **:***"méta-classe B"***,**TYPEDVAR
LIST_ID	::=	*"id"*[d] |*"id"*, LIST_ID
BLOC2	::=	**identify** *"id"* **with** *"prédicat"*
		In INSTRUCTION **End In ;**

[a] *"prédicat"* : indique une expression de condition dont l'interprétation est une valeur booléenne.
[b] *"donnée B"* : indique un élément de la spécification B instance de la méta-classe *BData*.
[c] *"schéma"* : indique une instanciation d'un schéma de transformation bien déterminé en utilisant des paramètres effectifs.
[d] *"id"* : indique un identifiant quelconque désignant une instance d'une méta-classe B.

Quatre types d'instruction sont donc considérés : CONDITION, CHOIX, SCHÉMA et BLOC. Remarquons que l'instruction de base d'une règle est l'instruction SCHÉMA dont l'objectif est d'exécuter une instanciation d'un schéma de transformation en utilisant des paramètres effectifs. Par exemple, l'instruction "**run** Schéma$_{3.3}$(*Passenger*) **;**" est une instruction SCHÉMA permettant de traduire l'ensemble abstrait *Passenger* en une classe UML par exécution du schéma de transformation 3.3 (page 74). Les autres instructions forment une imbriquation d'instructions et permettent la vérification de conditions ou la réalisation de traitements supplémentaires avant l'exécution d'une (ou plusieurs) instruction(s) SCHÉMA :

- CONDITION : permet de conditionner la réalisation d'instructions.
- CHOIX : la réalisation d'instructions dépend de la méta-classe associée à une construction B particulière. Par exemple, l'instruction :

Case Inst$^{-1}(d)$ **:**
BAbstractSet ⇒ Schéma$_{3.3}(d)$ **;**
BEnumSet ⇒ Schéma$_{3.2}(d)$ **;**
End Case ;

permet d'exécuter l'un des schémas de transformation 3.2 et 3.3 pour transformer la donnée d en un type énuméré ou en une classe, selon que d soit un *BEnumSet* ou un *BAbstractSet*.

- BLOC(1 et 2) : l'instruction BLOC sert pour la déclaration d'une nouvelle variable (désignée par *"id"*) à l'intérieur de la règle elle-même et dont la portée est restreinte aux instructions du bloc. L'instruction BLOC1 réalise un pattern matching pour identifier la valeur de "*id*" dans la construction B désignée par *"construction B"*. Par exemple,

  `Identify some` w **`:`** *ComposedTypeExp*
  `such that` « *weared_objects* $\in$ *Object* w *Passenger* » **`In`**...

  signifie que la variable "w" désigne une expression de composition de type et qu'elle est utilisée dans la construction de typage de *weared_objects* (il s'agit ici de "$\rightarrow$"). L'existence de plusieurs valeurs possibles pour w représente un cas indéterminé pour la règle. Par exemple, si *weared_objects* est définie par une relation $\leftrightarrow$ et par une fonction totale $\rightarrow$ entre les ensembles *Object* et *Passenger*, alors l'instruction précédente associe à w une valeur quelconque parmi $\leftrightarrow$ et $\rightarrow$. Quant à l'instruction BLOC2, elle introduit la variable *"id"* au moyen d'un prédicat. Dans le cadre d'une instruction BLOC2 le prédicat portant sur *"id"* ne doit pas introduire de l'indéterminisme au niveau de la règle. La variable *"id"* doit donc correspondre à un élément unique.

Notons que dans cette sous-section nous avons présenté les instructions qui nous paraissent indispensables pour la suite de notre étude. En effet, nous ne détaillons pas une grammaire complète pour l'expression des règles de traduction. D'autres instructions peuvent donc exister (*e.g.* l'instruction **`add_access_dependency`** sera utilisée et sera introduite plus tard).

8.2.2 Exemple d'application

Les Figs. 8.1 et 8.2 représentent respectivement le diagramme préliminaire et sa restructuration sur la base du modèle de contextes de la Fig. 7.8 page 135.

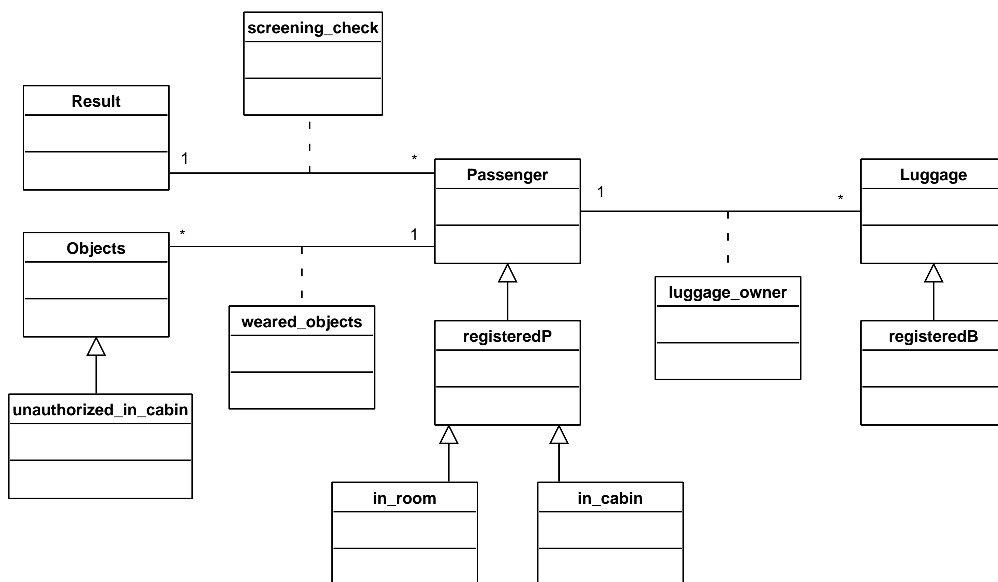


FIG. 8.1 – Diagramme de classes préliminaire issu de la machine SecureFlight

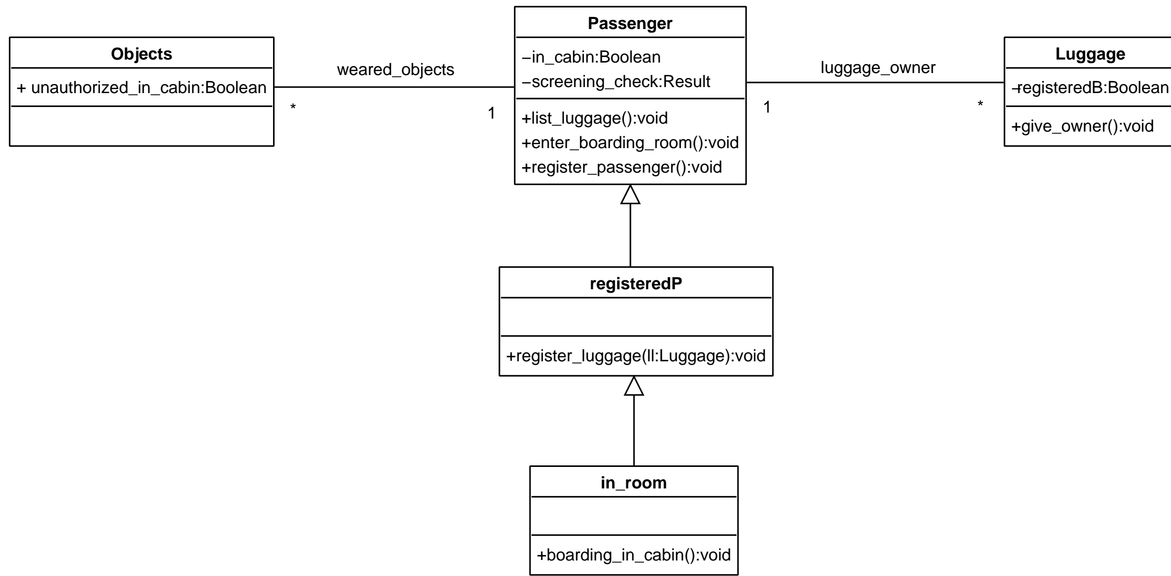


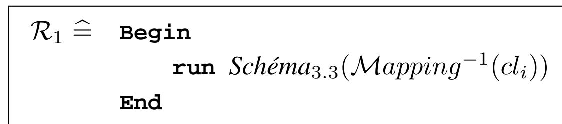
FIG. 8.2 – Diagramme de classes issu du modèle de contextes de la Fig. 7.8

Dans ce qui suit, nous considérons $\mathcal{G}$ un modèle de contextes formé par n contextes satisfaisant les propriétés (P_1), (P_2), (P_3) et (P_4), et cl_i la classe candidate à l'origine d'un contexte particulier $\mathcal{F}_i$ de $\mathcal{G}$: $cl_i = Target^{-1}[\mathcal{O}'_i]$.

8.3 Des contextes aux classes

Vu la définition de la fonction d'inclusion $Incl$ (définition 7.6, page 127), les classes candidates à l'origine des contextes de $\mathcal{G}$ proviennent des *BConstant*s, des *BVariable*s ou des *BAbstractSet*s et sont, en particulier, des classes, des sous-classes ou des classes associatives au niveau du diagramme préliminaire. Nous traitons alors distinctement et successivement ces trois cas de figure.

A. Cas des contextes $\mathcal{F}_i$ tels que $Type(cl_i) = Class$: dans ce cas le contexte $\mathcal{F}_i$ est transformé en une classe UML dans le diagramme de classes final par application du schéma de transformation 3.3. ($\mathcal{M}apping^{-1}(cl_i)$: *BData* $\xrightarrow{Schema_{3.3}}$ cl_i : *Class*)



$\mathcal{R}_1 \mathrel{\widehat{=}}$ **Begin**
    run $Schéma_{3.3}(\mathcal{M}apping^{-1}(cl_i))$
End

Dans notre cas d'étude, $\mathcal{R}_1$ est appliquée précisément sur les contextes associés aux classes préliminaires *Passenger* et à *Luggage*. Cette règle maintient donc ces deux classes au niveau du diagramme de classes final.

B. Cas des contextes $\mathcal{F}_i$ tels que $Type(cl_i) = SubClass$: la transformation de tels contextes se fait selon deux cas de figure :

(i) Il existe une classe candidate c à l'origine d'un contexte $\mathcal{F}_j$ du modèle $\mathcal{G}$ telle que $c \in Incl^+[\{cl_i\}]$: dans ce cas la classe candidate cl_i devient automatiquement une sous-classe de la classe c en ap-

pliquant le schéma de transformation 5.4. Si plusieurs classes candidates dans $Incl^+[\{cl_i\}]$ correspondent à des contextes de $\mathcal{G}$, alors le choix de la super-classe de cl_i porte sur la classe candidate la plus proche de cl_i dans le chemin formé par $Incl^+[\{cl_i\}]$.

```
R2.1 ^= Begin
        | Identify c with
        | | c ∈ Classes ∧ c ∈ Incl+[{cl_i}] ∧
        | | ∃F_j · (F_j ∈ G ∧ i ≠ j ∧ cl_j = c ∧
        | |        ∄F_k · (F_k ∈ G ∧ k ≠ j ∧ c ∈ Incl+[{cl_k}] ∧ cl_k ∈ Incl+[{cl_i}]))
        | In
        | | run Schéma_5.4(Mapping^-1(c), Mapping^-1(cl_i)) ;
        | End In ;
        End
```

Cette règle s'applique aux contextes correspondant aux classes candidates *registeredP* et *in_room*. En effet, $Incl^+[\{registeredP\}] = \{Passenger\}$ et $Incl^+[\{in_room\}] = \{registeredP, Passenger\}$. Étant donné que *Passenger* correspond également à un contexte (Fig. 7.8) alors $\mathcal{R}_{2.1}$ traduit *registeredP* en une sous-classe de *Passenger*, et *in_room* en une sous-classe de *registeredP*. Notons que si *registeredP* n'a pas donné lieu à un contexte (*i.e.* $Target(register_luggage) \neq registeredP$) alors le contexte *in_room* aurait été traduit par une sous-classe de *Passenger*.

(*ii*) Aucune classe candidate de $Incl^+[\{cl_i\}]$ n'est à l'origine d'un contexte de $\mathcal{G}$: dans ce cas, nous transformons la *BData* d à l'origine de la super-classe préliminaire de cl_i conformément au schéma 3. Quant au lien d'inclusion, nous le transformons via l'un des sous-schémas du schéma de transformation 5 selon $Inst^{-1}(d)$.

```
R2.2 ^= Begin
        | Identify some d : BData such that
        | | « Mapping^-1(cl_i) ⊆ d »
        | In
        | | run Schéma_3(d) ;
        | | Case Inst^-1(d) :
        | | | BEnumSet      ⇒ run Schéma_5.1(d, Mapping^-1(cl_i)) ;
        | | | BPrimitiveType ⇒ run Schéma_5.3(d, Mapping^-1(cl_i)) ;
        | | | BAbstractSet  ⇒ run Schéma_5.4(d, Mapping^-1(cl_i)) ;
        | | End Case ;
        | End In ;
        End
```

Lorsque plusieurs *BData*s d satisfont $\mathcal{M}apping^{-1}(cl_i) \subseteq d$, alors il s'agit d'un héritage multiple au niveau du diagramme de classes préliminaire. Le choix de la super-classe préliminaire de cl_i est indéterminé au niveau de la règle $R_{2.2}$ et peut être traité soit de manière interactive, soit en gardant toutes les super-classes. Par ailleurs, dans le cas où $Inst^{-1}(d) = BEnumSet$, le schéma de transformation 5 distingue deux transformations possibles : schéma 5.1 et 5.2. L'application de ces deux transformations dépend simplement du *TypingOperator* utilisé pour typer la *BData* à l'origine de cl_i (*i.e.* $\subset$ ou $\subseteq$). Ainsi, une autre forme de la règle $R_{2.2}$ peut être considérée où il s'agit de remplacer $\mathcal{M}apping^{-1}(cl_i) \subseteq d$ par $\mathcal{M}apping^{-1}(cl_i) \subset d$ et d'appliquer le schéma 5.2 lorsque $Inst^{-1}(d) = BEnumSet$. Pour les *BPrimitiveType*s et les *BAbstractSet*s les transformations sont identiques aussi bien pour $\subset$ que pour $\subseteq$.

Les deux cas de figure (*i*) et (*ii*) précédents sont illustrés au niveau de la Fig. 8.3 et correspondent respectivement à la partie supérieure et inférieure de cette figure. Notons que dans le second cas l'élément de modélisation préliminaire B correspond à une classe candidate telle que $B \notin ran(Target)$.

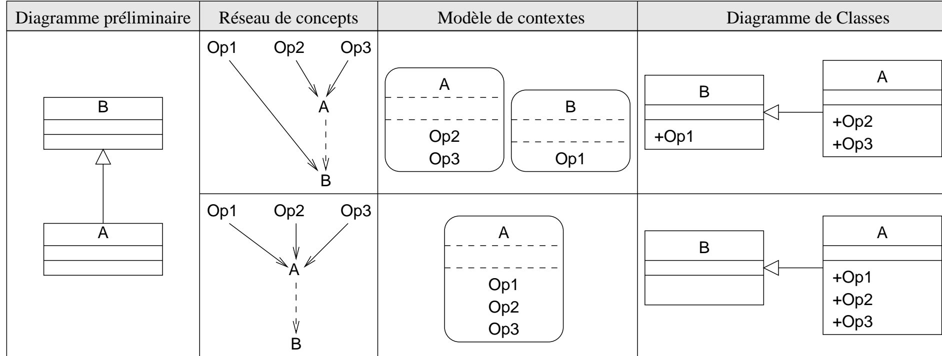


FIG. 8.3 – Illustration de la règle $\mathcal{R}_2$ avec $A \subseteq B$

C. Cas des contextes $\mathcal{F}_i$ telle que $Type(cl_i) = AssociativeClass$: Nous avons vu que dans ce cas, les extrémités de la classe candidate cl_i correspondent forcément à des *BAbstractSet* (vu la définition de l'ensemble $\mathcal{A}$ – voir définition 7.8, page 129). L'existence d'un contexte au niveau de $\mathcal{G}$ pour cl_i induit le maintient de la classe associative au niveau du diagramme de classes final. Pour ce faire, nous distinguons trois cas particuliers :

1. Les deux extrémités de la classe associative sont à l'origine de deux contextes cl_j et cl_k au niveau de $\mathcal{G}$;
2. L'une des deux extrémités ne correspond pas à un contexte de $\mathcal{G}$;
3. Les deux extrémités ne correspondent pas à des contextes de $\mathcal{G}$.

Dans le premier cas, les classes candidates cl_j et cl_k sont transformées par la règle $\mathcal{R}_1$ en des classes dans le diagramme de classes final. La classe candidate cl_i est alors simplement maintenue en tant que classe associative entre les classes cl_j et cl_k, et ce, par application du schéma 9.4. Dans les deux autres cas, l'application du schéma 9.4 n'est possible qui si les classes préliminaires extrémités de cl_i sont déjà transformées en classes dans le diagramme de classes final. La Fig. 8.4 illustre l'application de cette règle sur le deuxième cas particulier.

$\mathcal{R}_3 \mathrel{\widehat{=}}$ **Begin**
  Identify some d, d' : *BAbstractSet*, w : *ComposedTypeExp* **such that**
    « $\mathcal{M}apping^{-1}(cl_i) \in d\ w\ d'$ »
  In
    If $\nexists \mathcal{F}_j \cdot (\mathcal{F}_j \in \mathcal{G} \wedge j \neq i \wedge cl_j = d)$ **Then**
      run $Schéma_{3.3}(d)$ **;**
    End If ;
    If $\nexists \mathcal{F}_k \cdot (\mathcal{F}_k \in \mathcal{G} \wedge k \neq i \wedge cl_k = d')$ **Then**
      run $Schéma_{3.3}(d')$ **;**
    End If ;
    run $Schéma_{9.4}(\mathcal{M}apping^{-1}(cl_i), d, d', w)$ **;**
  End In ;
End

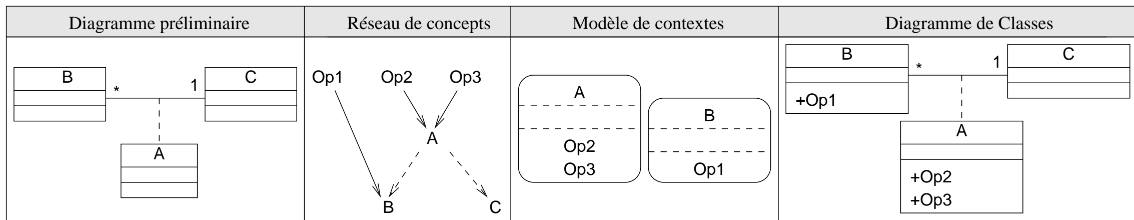


FIG. 8.4 – Illustration de la règle $\mathcal{R}_3$ avec $A \in B \to C$

8.4 Transformation des attributs des contextes

8.4.1 Présentation

Naturellement, les attributs des contextes sont traduits en des attributs de classes. Les schémas de transformation qui permettent de transformer des *BData*s en des attributs de classes sont : 4.1, 5.5, 9.1, 9.2 et 9.3. Vu que les diagrammes préliminaires ne prennent pas en compte des *BData*s typées par un opérateur d'appartenance, alors nous ne considérons pas à ce niveau le schéma 4.1. La transformation de cas particulier de *BData*s fais donc partie des transformations non couvertes par notre algorithme et devra être réalisée de manière interactive. Quant aux trois autres schémas, ils portent principalement sur les sous-ensembles et relations de la spécification B et produisent soit des attributs booléens soit des attributs typés par le domaine ou le co-domaine de relations.

D'un autre côté, transformer une *BData* en un attribut nécessite l'existence d'une classe pouvant encapsuler cette *BData*. Les premières règles qui génèrent des classes pour le diagramme final sont $\mathcal{R}_1$, $\mathcal{R}_2$ et $\mathcal{R}_3$. Celles-ci sont exécutées d'une manière séquentielle et produisent pour chaque contexte du modèle $\mathcal{G}$ une ou plusieurs classes. Une manière intuitive de procéder est de traduire chaque élément de modélisation préliminaire e d'un contexte $\mathcal{F}_i$ en un attribut de la classe cl_i (*e.g. in_cabin* et *screening_check* faisant partie du contexte *Passenger* et devenus des attributs privés de la classe *Passenger*).

Cependant, des éléments de modélisation tels que *unauthorized_in_cabin* ne peuvent être transformés d'une manière pareille. Cette contrainte provient du fait que la classe susceptible d'encapsuler ces éléments en tant qu'attributs ne correspond pas à la classe produite pour le contexte qui les contient. En effet, bien que *unauthorized_in_cabin* figure au niveau du contexte *Passenger*, aucun schéma de transformation n'existe pour transformer *unauthorized_in_cabin* en un attribut de la classe *Passenger*. La transformation de *unauthorized_in_cabin* en un attribut nécessite l'existence de la classe *Object* (contrainte posée par le schéma 5.5). Cette migration de *unauthorized_in_cabin* du contexte *Passenger* vers la classe *Object* induit une visibilité publique de l'attribut *unauthorized_in_cabin* pour qu'il soit accessible à l'opération *enter_boarding_room*.

Ainsi, la traduction des attributs des contextes, peut conduire à la création de nouvelles classes, et nécessite la prise en compte de plusieurs contraintes, telles que la visibilité des attributs. En vue de formaliser les règles de transformation des attributs des contextes, nous considérons l'ensemble $\mathcal{K}$ des classes du diagramme de classes final, et la relation $\mathcal{B}_{\mathcal{K}}$ qui associe à chaque classe du diagramme de classes final la donnée B correspondante :

$$\mathcal{B}_{\mathcal{K}} \in \mathcal{K} \leftrightarrow \mathcal{B}$$

Les règles ci-dessous sont appliquées pour chaque élément de modélisation e de chaque contexte $\mathcal{F}_i$ du modèle de contextes $\mathcal{G}$. Soit cl_i la classe candidate à l'origine du contexte qui encapsule e.

A. Cas des éléments e **tels que** $Type(e) = Class$: ces éléments proviennent précisément de l'étape (*iii*) de notre algorithme où e n'a pas été choisi comme cible d'une ou plusieurs opérations. La classe préliminaire e ne correspond donc pas à un contexte et apparaît dans tout contexte dont les opérations incluent au moins une opération de $P_{\mathcal{I}}[\{e\}]$. Étant donné que $Type(e) = Class$, alors e ne peut être transformée en un attribut d'une classe. Nous le transformons donc en une classe dans le diagramme de classes final avec une dépendance ≪*access*≫ de la classe cl_i vers la classe e.

```
R4 ≙  Begin
          If  e ∉ K  Then
              run  Schéma3.3(Mapping^-1(e)) ;
          End If ;
          add_access_dependency from  cl_i  to  e ;
      End
```

Une seule classe est produite pour e même si e appartient à plusieurs contextes (ceci est réalisé au moyen de la condition $e \notin \mathcal{K}$). L'instruction `add_access_dependency` au niveau de la règle $\mathcal{R}_4$ permet de rajouter une dépendance stéréotypée par ≪*access*≫ entre la classe produite pour le contexte encapsulant e et la classe e. Cette règle est illustrée au niveau de la Fig. 8.5 où B correspond à une classe préliminaire et est encapsulé par les contextes A et C.

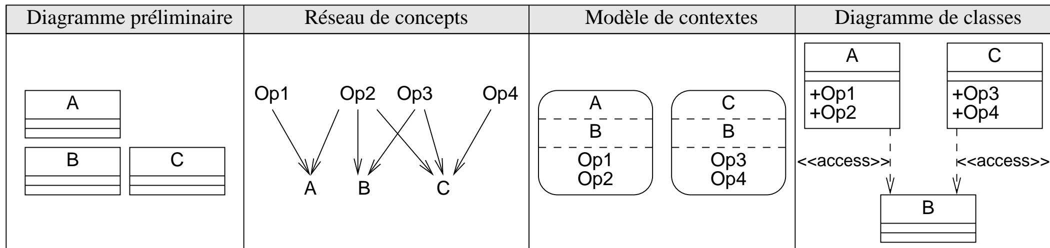


FIG. 8.5 – Illustration de la règle $\mathcal{R}_4$

B. Cas des éléments e **tels que** $Type(e) = SubClass$: dans ce cas, il s'agit d'appliquer le schéma de transformation 5.5 tout en vérifiant l'existence d'une classe convenable pour encapsuler e. Plusieurs cas sont traités à ce niveau :

1. La classe candidate cl_i à l'origine du contexte encapsulant e correspond à l'une des super-classes de e dans le diagramme de classes préliminaire ($cl_i \in Incl^+[\{e\}]$) : dans ce cas e est traduit par un attribut booléen (privé) de la classe du diagramme de classes final produite pour cl_i.

```
R5.1 ≙ Begin
       |   If  cl_i ∈ Incl+[{e}]  Then
       |   |   run  Schéma5.5(B_K(cl_i), Mapping^-1(e)) ;
       |   End If ;
       End
```

2. La classe candidate cl_i ne fais pas partie de $Incl^+[\{e\}]$ alors que l'une des super-classes préliminaires de e est traduite par une classe cl dans le diagramme de classes final : dans ce cas e devient

un attribut booléen (public) de cl avec une dépendance ≪access≫ de la classe associée à cl_i vers la classe cl.

$\mathcal{R}_{5.2} \widehat{=}$ **Begin**
| **Identify** cl **with**
| | $cl \in Incl^{+}[\{e\}] \wedge \mathcal{M}apping^{-1}(cl) \in \mathrm{ran}(\mathcal{B}_{\mathcal{K}}) \wedge$
| | $\forall cl' \cdot (cl' \in Incl^{+}[\{e\}] \wedge \mathcal{M}apping^{-1}(cl') \in \mathrm{ran}(\mathcal{B}_{\mathcal{K}}) \Rightarrow cl' \notin Incl^{+}[\{cl\}])$
| **In**
| | **run** $Schéma_{5.5}(\mathcal{B}_{\mathcal{K}}(cl), \mathcal{M}apping^{-1}(e))$ **;**
| **End In ;**
End

3. Aucune super-classe préliminaire de e n'est traduite par une classe dans le diagramme de classes final : dans cas, nous appliquons le schéma de transformation 3 pour traduire convenablement la super-classe préliminaire immédiate de e avant de transformer e par l'un des sous-schémas du schéma de transformation 5.

$\mathcal{R}_{5.3} \widehat{=}$ **Begin**
| **Identify some** d **:** $BData$ **such that**
| | « $\mathcal{M}apping^{-1}(e) \subseteq d$ »
| **In**
| | **run** $Schéma_3(d)$ **;**
| | **Case Inst**$^{-1}(d)$ **:**
| | | $BEnumSet \Rightarrow$ **run** $Schéma_{5.1}(d, \mathcal{M}apping^{-1}(e))$ **;**
| | | $BPrimitiveType \Rightarrow$ **run** $Schéma_{5.3}(d, \mathcal{M}apping^{-1}(e))$ **;**
| | | $BAbstractSet \Rightarrow$ **run** $Schéma_{5.5}(d, \mathcal{M}apping^{-1}(e))$ **;**
| | **End Case ;**
| **End In ;**
End

Les règles $\mathcal{R}_{5.1}$, $\mathcal{R}_{5.2}$ et $\mathcal{R}_{5.3}$ ne définissent aucune visibilité pour e lorsque e est transformé en un attribut dans une classe. Cette distinction entre attributs privés et publics fera l'objet de la section 8.4.2.

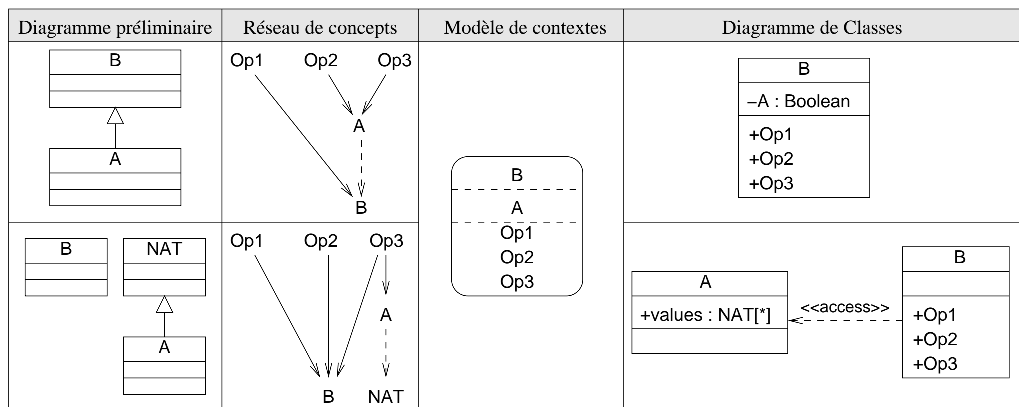


FIG. 8.6 – Illustration de la règle $\mathcal{R}_5$

Notons également, que dans le cas où $Inst^{-1}(c) \in \{BPrimitiveType, BEnumeSet\}$, l'application de l'un des schémas 5.2 et 5.3 traduit e en une classe. Ce faisant, une dépendance ≪*access*≫ est requise entre la classe nouvellement créée et la classe e en vue de mettre en évidence l'accès des opérations de cl_i

à la classe e. La Fig. 8.6 illustre ces propos. Dans ce cas la classe A est vue comme une classe utilitaire ou comme une classe disposant d'une instance unique.

C. Cas des éléments e **tels que** $Type(e) = AssociativeClass$: dans ce cas e est transformé soit en une association simple, soit en un attribut d'une classe, soit en une classe associative. Ces transformations correspondent à l'application de l'un des schémas 9.1, 9.2, 9.3 et 9.4. La première et la troisième transformations ne sont possibles que si les deux classes préliminaires extrémités de e correspondent à des classes dans le diagramme de classes final. Quant à la deuxième transformation, elle dépend de la classe susceptible d'encapsuler e en tant qu'attribut. Nous distinguons alors plusieurs spécialisations de cette règle :

```
R6 ≙  Begin
      | Identify some c1, c2 : BAbstractSet, w : ComposedTypeExp such that
      | |  « Mapping⁻¹(e) ∈ c1 w c2 »
      | In
      | | If Mapping⁻¹(cl_i) ∈ {c1, c2} Then
      | | | If {c1, c2} ⊆ ran(B_K) Then
      | | | | run Schéma_9.1(Mapping⁻¹(e), c1, c2, w) ;
      | | | Else
      | | | | If c1 ∈ ran(B_K) ∧ Mapping⁻¹(cl_i) = c1 Then
      | | | | | run Schéma_9.2(Mapping⁻¹(e), c1, c2, w) ;
      | | | | End If ;
      | | | | If c2 ∈ ran(B_K) ∧ Mapping⁻¹(cl_i) = c2 Then
      | | | | | run Schéma_9.3(Mapping⁻¹(e), c1, c2, w) ;
      | | | | End If ;
      | | | End If ;
      | | Else
      | | | If {c1, c2} ⊆ ran(B_K) Then
      | | | | run Schéma_9.4(Mapping⁻¹(e), c1, c2, w) ;
      | | | | add_access_dependency from cl_i to e ;
      | | | End If ;
      | | End If ;
      | End In ;
      End
```

– Les deux extrémités de la classe associative préliminaire e correspondent à des classes dans le diagramme de classes final telles que l'une d'entre elles provient d'un contexte encapsulant e (il s'agit de cl_i dans $\mathcal{R}_6$). Dans ce cas, la règle $\mathcal{R}_6$ traduit la classe associative e en une association simple entre les classes du diagramme de classes final ($\mathcal{K}$) issues des extrémités de e, et ce, en sélectionnant le schéma 9.1. Ceci est le cas de la relation *luggage_owner* transformée en une association simple entre les classes *Luggage* et *Passenger*.
– L'une des extrémités de e correspond à cl_i, alors qu'il n'existe aucune classe produite pour l'autre extrémité de e dans le diagramme de classes final. Dans ce cas, la classe associative préliminaire e est transformée en un attribut de la classe produite par $\mathcal{R}_1$ à partir de cl_i. Par exemple, *screening_check* encapsulé par le contexte *Passenger* ne peut être traduit en une association car *Result* n'est pas identifié comme une classe au niveau du diagramme de classes final. La relation *screening_check* est donc traduite par un attribut de type *Result* au niveau de la classe *Passenger*.

– cl_i ne correspond à aucune extrémités de e. Ce cas provient de l'étape (*iii*) de l'algorithme où e a été rattaché au contexte $\mathcal{F}_i$ produit pour cl_i vu qu'il contient au moins une opération qui l'utilise. Dans ce cas, si les deux extrémités de e sont transformées en classes au niveau du diagramme de classes final par application de l'une des règles précédentes, alors nous transformons e en une classe associative avec une une dépendance ≪*access*≫ de la classe produite pour cl_i vers e. Dans le cas contraire, aucune transformation n'est envisagée par notre approche pour e et par conséquent e ne sera pas représenté dans le diagramme.

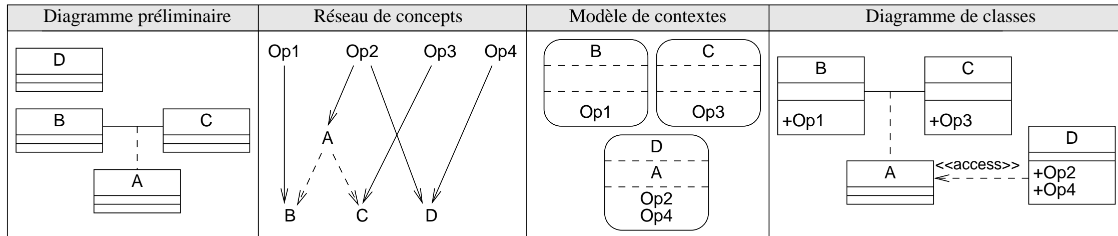


FIG. 8.7 – Illustration de la règle $\mathcal{R}_6$ − Cas 3.

8.4.2 Visibilité des attributs des classes

Vu que les attributs des classes proviennent des attributs des contextes, leur visibilité au niveau des classes est étroitement liée aux contextes à partir desquels ils proviennent. Nous considérons alors qu'un attribut d'une classe cl ($cl \in \mathcal{K}$) est privé s'il provient du contexte à partir duquel la classe cl a été créée. Dans le cas où l'attribut a subi une migration vers une autre classe, nous considérons sa visibilité publique (*e.g.* l'attribut *unauthorized_in_cabin* initialement encapsulé par *Passenger* est transformé en un attribut public de la classe *Object*).

La visibilité publique d'un attribut d d'une classe cl désigne le fait que les opérations utilisant d sont encapsulées dans d'autres classes. Cet accès externe nécessite soit l'existence d'une association entre la classe cl et la classe cl', produite à partir du contexte contenant initialement d, soit alors une dépendance stéréotypée par ≪*access*≫ de cl' vers cl. Par exemple, au niveau du diagramme de la Fig. 8.2, l'attribut public *registeredB* de la classe *Luggage* est utilisé par la méthode *register_luggage* de la sous-classe *registeredP*. Cet accès est réalisable via l'association *luggage_owner*.

8.5 Transformation des opérations des contextes

Les opérations des contextes sont systématiquement traduites en méthodes au niveau des classes générées par les règles $\mathcal{R}_1$, $\mathcal{R}_2$ et $\mathcal{R}_3$. Plus précisément, une opération o d'un contexte $\mathcal{F}_i$ de $\mathcal{G}$ est traduite par une méthode au niveau de la classe cl_i. Cette transformation est réalisée avec le schéma de transformation 16. Nous ne présentons pas de schémas de transformation ni des règles de traduction pour la prise en compte des paramètres, nous évoquons à ce niveau, simplement l'heuristique suivante : « Seuls les paramètres dont le type est différent du nom de la classe sont pris en compte au niveau de la transformation des paramètres ». Notons que ces paramètres peuvent également provenir de substitutions ANY tels que, par exemple, le paramètre ll de la méthode *register_luggage* encapsulée par la classe *registeredP*. Ce paramètre est typé par *Luggage*. La Fig. 8.8 présente le cas où la méthode *register_luggage* est encapsulée par la classe *Luggage*, dans ce cas nous lui associons le paramètre *pp* de type *Passenger*.

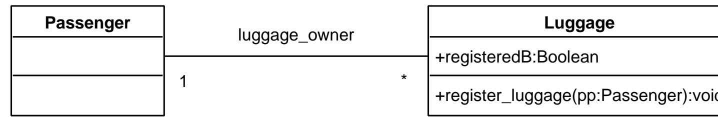


FIG. 8.8 – Exemple de prise en compte des paramètres des méthodes

8.6 Synthèse

La formation des contextes présente un moyen intéressant pour guider le processus de dérivation des diagrammes de classes. En effet, le choix d'un schéma de transformation parmi d'autres dépend de la structure des contextes, et est formalisé par les règles de traduction des contextes. Nous avons montré sur la base des travaux existants (Fekih *et al.,* 2004, Fekih *et al.,* 2006, Tatibouet *et al.,* 2002, Voisinet, 2004) que la proposition d'un ensemble de règles pour une traduction directe de B vers UML peut aboutir à des résultats complexes et incomplets. Dans la première étape de notre approche, nous suivons une démarche similaire en construisant, via des règles simples et intuitives un diagramme préliminaire. Ensuite, nous nous servons du modèle de contextes pour réorganiser ce diagramme préliminaire et en dériver un diagramme simple et plus complet.

Chapitre 9

Optimisation et prise en compte des développements structurés

« *La raison est dans le mental le juge qui mesure, compare, classe, organise, relie suivant des principes rationnels. L'intelligence est différente de la simple raison raisonneuse. L'intelligence est cette faculté particulière qui permet de voir, de voir des liens, de relier.* »

S. Carfantan

« Philosophie et spiritualité », 2002.

Sommaire

9.1 Introduction . . . 153
9.2 Matrices de liens . . . 154
9.2.1 Matrices de liens simples . . . 155
9.2.2 Matrices de liens pondérées . . . 157
9.2.3 Pertinence des modèles de contextes . . . 158
9.2.4 Évaluation . . . 161
9.3 Prise en compte des raffinements . . . 162
9.3.1 Représentation du plus haut niveau d'abstraction . . . 163
9.3.2 Premier niveau de raffinement . . . 164
9.3.3 Derniers niveaux de raffinement . . . 164
9.3.4 Mise en relation des vues produites pour les raffinements . . . 165
9.4 Conclusion . . . 167

9.1 Introduction

Nous avons axé notre travail de dérivation de vues structurelles autour de deux approches complémentaires : l'approche *interprétée*, fondée sur l'élaboration d'un méta-modèle B, et l'approche *calculée*, basée sur une technique de formation de concepts. Bien que la mise en œuvre de ces approches permette un potentiel d'automatisation satisfaisant du processus de transformation et présente ainsi un progrès important par rapport aux travaux existants, nous proposons, au niveau de ce chapitre, deux améliorations majeures portant sur :

- ◇ l'évaluation et le classement des diagrammes pouvant être construits par notre approche à partir d'une même spécification B, et
- ◇ la prise en compte de développements B structurés par raffinements successifs.

Ces deux points ont fait l'objet respectivement des articles (Idani *et al.,* 2006c, Idani *et al.,* 2006d) et ont pour principale finalité l'optimisation de la phase de choix de cibles pour chacune des opérations de la spécification. Le premier point est abordé en proposant une suite de métriques élaborées de manière empirique. Au niveau du second point, nous proposons un ensemble de critères permettant la préservation, au niveau concret, des choix de modélisation faits au niveau abstrait. Bien que ces deux optimisations soient développées indépendamment l'une de l'autre, elles peuvent être appliquées ensemble pour mieux appréhender un développement B construit par raffinements. Finalement, nous mettons l'accent sur le fait que les techniques que nous proposons dans ce chapitre se basent sur les modèles de contextes vu que ces derniers présentent un cadre formel pour la dérivation des diagrammes de classes.

9.2 Matrices de liens

L'analyse formelle de concepts proposée au niveau du chapitre 7 nous a permis de ramener l'interaction avec l'analyste du choix entre un ensemble de schémas de transformations interdépendants à un simple choix de cibles pour chaque opération. Cependant, chaque ensemble de choix résulte en une configuration particulière de la fonction $Target$ et produit donc un diagramme de classes particulier. Par exemple, nous proposons au niveau de la Fig. 9.1 une autre configuration de la fonction $Target$ (une première configuration a été introduite à la page 135) dérivée de la spécification *SecureFlight* (Annexe E) avec le modèle de contextes et le diagramme de classes qui en découlent :

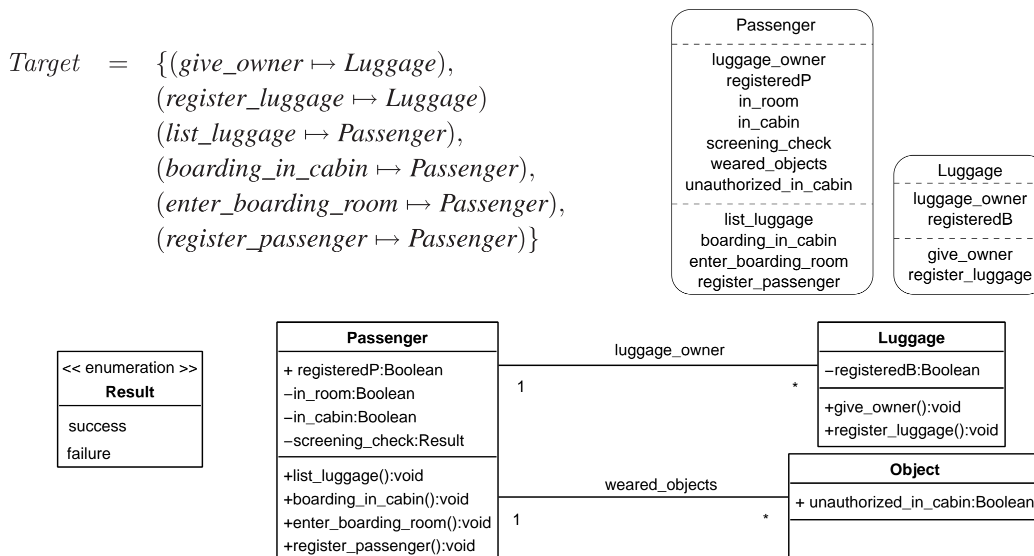


FIG. 9.1 – Modèle de contextes formé uniquement des contextes *Passenger* et *Luggage*

Les différences que nous pouvons observer entre ce diagramme de classes est celui présenté au niveau de la Fig. 8.2 (page 144) provient plus précisément des cibles choisies respectivement pour les opérations *register_luggage* (*i.e. Luggage* ou *registeredP*) et *boarding_in_cabin* (*i.e. in_room* ou *registeredP*) au moyen du réseau de concepts raffiné $\mathcal{J}_{\mathcal{R}}$.

Au niveau du tableau 7.1 (page 135) présentant un récapitulatif des cibles possibles de chacune des opérations de la spécification B de départ, nous remarquons que les opérations *register_luggage* et *boarding_in_cabin* disposent respectivement de 2 et 4 cibles possibles. Ainsi, le nombre total d'instances de la fonction $Target$ est égal à $2 \times 4 = 8$. Ce qui correspond au nombre total de diagrammes de classes pouvant être construits par notre technique pour cette même spécification B.

Ce faisant, ce nombre a tendance à exploser dans le cadre de spécifications de taille importante introduisant ainsi une difficulté certaine au niveau du choix de la meilleure représentation. En effet, l'application de notre technique sur une spécification "réelle" (extraite du projet EDEMOI) dont la taille est de l'ordre de 222 lignes de code B, permet de dénombrer 65 diagrammes de classes possibles. Il devient alors important de proposer une technique d'évaluation et de classement de ces diagrammes. Pour ce faire, nous proposons un ensemble de matrices simples et pondérées, permettant de mesurer la pertinence des modèles de contextes.

Nous avons proposé une première définition de pertinence à la page 115 se rattachant à la pertinence d'une *BOperation* par rapport à une classe. Sur la base de cette définition nous avons orienté notre processus de formation des modèles de contextes. En revanche, l'existence de plusieurs modèles possibles nous conduit à préciser davantage cette notion de pertinence et à la mesurer en prenant en compte aussi bien les *BOperation*s que les *BData*s traduits en des éléments de modélisation encapsulés au niveau des contextes.

Le but des matrices de liens est donc d'évaluer la pertinence de chaque donnée B (*BData*) par rapport aux contextes auxquels elle peut être attachée. Nous nous basons pour ce faire sur les liens d'utilisation entre ces *BData*s et les opérations du contexte :

1. **Pré-condition** : seulement la pré-condition de l'opération fait référence à la *BData*,
2. **Lecture** : l'opération nécessite seulement un accès en lecture à la valeur de la *BData*,
3. **Modification** : l'opération accède et modifie la *BData*.

La distinction entre les différents types d'utilisation (*i.e.* Pré-condition, Lecture et Modification) a pour objectif d'affiner la relation de dépendance de concepts $\mathcal{I}$ (définition 6.2 page 116). En effet, ces liens sont formalisés de manière identique au niveau de $\mathcal{I}$. Dans ce qui suit, nous proposons une première mesure de pertinence via des matrices boolénnes simples énoncées sans aucune distinction entre types d'utilisation, ensuite, nous transformons ces matrices en matrices pondérées et nous montrons leur intérêt pour guider le choix d'une représentation parmi d'autres.

9.2.1 Matrices de liens simples

Étant donnée une *BData* d avec $\mathcal{I}[\{d\}] = \{o_1, o_2, \ldots, o_n\}$, nous définissons une matrice (dite matrice de liens et désignée par $\mathcal{L}(d)$) de taille 2^n et présentant un calcul de propositions logiques exprimées sur les n opérations utilisant d. Soit, par exemple, la *BData* d telle que $\mathcal{I}[\{d\}] = \{o_1, o_2, o_3\}$, alors une matrice de liens simple $\mathcal{L}(d)$ définie par une combinaison OR des opérations o_1, o_2 et o_3 est celle de la Fig. 9.2. Dans cette représentation, les trois premières lignes correspondent aux différentes configurations possibles des opérations de la *BData* d. Quant à la quatrième ligne, elle permet de mesurer la pertinence de d pour chacune des configurations de ses opérations. Il s'agit ici d'une valeur booléenne calculée pour une proposition OR définie sur les opérations de d. En d'autres termes, d est considérée comme pertinente pour tout contexte encapsulant au moins une opération de celles qui l'utilisent.

$\mathcal{L}(d) \,\widehat{=}$

	0	1	2	3	4	5	6	7
o_1		X		X		X		X
o_2			X	X			X	X
o_3					X	X	X	X
pertinence	0	1	1	1	1	1	1	1

FIG. 9.2 – Matrice de liens associée à une *BData* d avec $\mathcal{I}[\{d\}] = \{o_1, o_2, o_3\}$

Remarquons que les matrices booléennes définies à partir d'une combinaison OR d'opérations (*e.g.* Fig. 9.2), présentent des métriques élémentaires permettant de décider si une *BData* est pertinente ou non pour un contexte $\mathcal{F} = (\mathcal{D}, \mathcal{O})$. En effet, elles reprennent la propriété (P_4) des contextes (voir page 130) et permettent de considérer que d est pertinent pour $\mathcal{F}$ si, et seulement si :

$$d \in \mathcal{D} \wedge \mathcal{I}[\{d\}] \cap \mathcal{O} \neq \emptyset$$

En d'autres termes, pour toute *BData* d encapsulée par un contexte $\mathcal{F}$, nous avons $\mathcal{L}_{\mathcal{F}}(d) = 1$. Par exemple, étant donné $\mathcal{I}[\{d\}] = \{o_1, o_2, o_3\}$ et un contexte $\mathcal{F}$ tel que $\mathcal{F} = (\{\ldots, d, \ldots\}, \{o_1, o_3\})$ alors $\mathcal{L}_{\mathcal{F}}(d)$ vaut 1 et correspond à la colonne 5 de la matrice illustrée par la Fig. 9.2.

Au niveau des propriétés de pertinence des contextes (voir page 130), nous distinguons une autre propriété portant sur les données B, il s'agit de la propriété (P_3) prenant en compte la notion d'opérations privées. Cette propriété amène à définir les matrices de liens par une combinaison AND des opérations privées ($Private(d)$) et une combinaison OR des autres opérations (*i.e.* $\mathcal{I}[\{d\}] - Private(d)$) d'une *BData* d.

Soit par exemple, la *BData* d telle que $\mathcal{I}[\{d\}] = \{o_1, o_2, o_3\}$ et $Private(d) = \{o_1\}$ alors $\mathcal{L}(d)$ est comme suit :

$\mathcal{L}(d) \,\widehat{=}$

	0	1	2	3	4	5	6	7
o_1	$\perp$	X	$\perp$	X	$\perp$	X	$\perp$	X
o_2			X	X			X	X
o_3					X	X	X	X
Pertinence	0	1	0	1	0	1	0	1

FIG. 9.3 – Matrice de liens illustrant la prise en compte des opérations privées

Le symbole $\perp$ associé à une opération privée o indique que o n'est pas considérée dans la configuration et qu'un tel cas de figure ne peut être envisagé. Ceci signifie que seuls les contextes $\mathcal{F} = (\mathcal{D}, \mathcal{O})$ tels que $d \in \mathcal{D}$ et $o_1 \in \mathcal{O}$ sont pertinents. Ce qui est, toujours vrai, vu que les contextes formés par notre algorithme de formation de contextes satisfont toutes les propriétés de pertinence.

Ces matrices simples sont élaborées en vue d'illustrer, de manière générale, les configurations d'opérations considérées par notre algorithme de construction de contextes. C'est pourquoi elles ne prennent pas en compte les types d'utilisation présentés précédemment. L'association de poids aux différents liens existants au niveau de la relation de dépendance de concepts $\mathcal{I}$ permet d'évaluer les différentes configurations illustrées par les matrices de liens simples et conduit, par conséquent, à une évaluation des modèles de contextes.

9.2.2 Matrices de liens pondérées

Au niveau de cette sous-section, nous proposons une estimation de la pertinence des données B prenant en compte outre les opérations avec lesquelles ces données sont encapsulées, les priorités associées aux différents liens d'utilisation. Par exemple, nous pouvons penser qu'une modification dispose d'un poids plus important que celui d'un accès en lecture, et qu'un accès en lecture peut avoir un poids plus grand que celui associé à un accès en pré-condition. Ainsi, l'objectif des matrices de liens pondérées est de fournir plus de précision quant aux liens existants entre une opération et une donnée B.

Un poids m est une mesure donnée empiriquement en vue d'associer une priorité à un type d'utilisation particulier. Soit, par exemple, la *BData* d avec $\mathcal{I}[\{d\}] = \{o_i, o_j\}$, et telle que o_i et o_j accèdent à d respectivement en modification et en lecture. Si m_i et m_j sont les poids respectifs de o_i et de o_j par rapport à d, alors nous pouvons considérer que $m_i > m_j$. Par conséquent, un contexte $\mathcal{F}_1 = (\{\ldots, d, \ldots\}, \{\ldots, o_i, \ldots\})$ sera préféré à un contexte $\mathcal{F}_2 = (\{\ldots, d, \ldots\}, \{\ldots, o_j, \ldots\})$ et un contexte $\mathcal{F}_3 = (\{\ldots, d, \ldots\}, \{\ldots, o_i, \ldots, o_j, \ldots\})$ sera considéré comme le plus approprié pour d en comparaison avec $\mathcal{F}_1$ et $\mathcal{F}_2$.

Définition 9.1 *Soit d une BData et soit o une BOperation, alors nous notons $m(d, o)$ le poids associé à d étant donnée l'opération o. Si $o \notin \mathcal{I}[\{d\}]$ alors $m(d, o)$ vaut 0. Nous considérons p, r et w les poids associés respectivement à un (ou plusieurs) accès en pré-condition, en lecture et en écriture, alors $m(d, o)$ correspond au* ***maximum*** *des poids associés à chacune des occurrences de d dans o (dans ce cas, $m(d, o) \in \{p, r, w\}$).*

Au niveau de la définition précédente nous avons choisi d'associer à $m(d, o)$ le maximum des poids associés aux différentes occurrences de d dans o ; cependant, d'autres calculs de $m(d, o)$ peuvent être considérés. Par exemple, nous pouvons lui associer la **somme** des poids associés à chacune des occurrences de d dans o.

Nous pouvons, notamment, considérer que si d apparaît n fois dans la pré-condition de o alors le poids d'un accès en pré-condition associé à d par rapport à o vaut $n \times p$. Le même raisonnement peut être appliqué aux autres types d'accès et le calcul de $m(d, o)$ peut être réalisé selon la définition 9.1 en choisissant le maximum des poids ainsi calculés ou alors en effectuant la somme de ces poids.

Dans la suite de notre étude, nous n'allons pas considérer toutes ces possibilités, nous allons nous limiter au cas de calcul de $m(d, o)$ de la définition 9.1 tout en traitant de manière identique l'existence d'un ou de plusieurs accès de même type. La pertinence d'une *BData* d est donc calculée sur la base des différents poids associés aux opérations de $\mathcal{I}[\{d\}]$, et ce, pour chaque configuration de ces opérations :

Définition 9.2 *Soit d une BData telle que $\mathcal{I}[\{d\}] = \{o_1, o_2, \ldots, o_n\}$ et soit $\mathcal{O} \in \mathbb{P}(\mathcal{I}[\{d\}])$ une configuration particulière des opérations utilisant d. Alors la pertinence $\mathcal{P}$ de d étant donné $\mathcal{O}$ est :*

$$\mathcal{P}(d, \mathcal{O}) = \begin{cases} \dfrac{\sum\limits_{o \in \mathcal{O}} m(d, o)}{\sum\limits_{o \in \mathcal{I}[\{d\}]} m(d, o)}, \ \textit{si Private}(d) \subseteq \mathcal{O} \\ 0, \ \textit{si Private}(d) \nsubseteq \mathcal{O} \end{cases}$$

Supposons, par exemple, $\mathcal{I}[\{d\}] = \{o_1, o_2, o_3\}$ avec $Private(d) = \emptyset$, et soit :

$$\mathcal{W}(d) = \sum_{o \in \mathcal{I}[\{d\}]} m(d, o) = m_1 + m_2 + m_3$$

alors la matrice de liens pondérée associée à d et désignée par $\mathcal{L}^m(d)$ est comme suit :

$\mathcal{L}^m(d) \triangleq$

	0	1	2	3	4	5	6	7
o_1		m_1		m_1		m_1		m_1
o_2			m_2	m_2			m_2	m_2
o_3					m_3	m_3	m_3	m_3
Pertinence	0	$\frac{m_1}{\mathcal{W}(d)}$	$\frac{m_2}{\mathcal{W}(d)}$	$\frac{m_1+m_2}{\mathcal{W}(d)}$	$\frac{m_3}{\mathcal{W}(d)}$	$\frac{m_1+m_3}{\mathcal{W}(d)}$	$\frac{m_2+m_3}{\mathcal{W}(d)}$	1

FIG. 9.4 – Matrice de liens pondérée déterminée pour une combinaison OR des opérations de d

Remarquons que dans le cadre des matrices pondérées la mesure de la pertinence d'une *BData* atteint la valeur 1 uniquement dans le cas où toutes les opérations sont encapsulées avec cette *BData*. Cependant, notre technique de formation de concepts prend en compte plusieurs facteurs pour la construction des contextes, et peut aboutir à des groupements conceptuels où des données B ne sont pas regroupées avec toutes leurs opérations. Nous sommes alors amené à calculer toutes les mesures et choisir les contextes qui maximisent cette valeur de pertinence.

Comme pour les matrices simples, nous distinguons également le cas où la *BData* d dispose d'opérations privées. Dans ce cas une pertinence égale à 0 est associée aux configurations qui excluent les opérations privées de d. Soit par exemple $Private(d) = \{o_1\}$, alors la matrice ci-dessus est transformée comme suit :

$\mathcal{L}^m(d) \triangleq$

	0	1	2	3	4	5	6	7
o_1	$\perp$	m_1	$\perp$	m_1	$\perp$	m_1	$\perp$	m_1
o_2			m_2	m_2			m_2	m_2
o_3					m_3	m_3	m_3	m_3
Pertinence	0	$\frac{m_1}{\mathcal{W}(d)}$	0	$\frac{m_1+m_2}{\mathcal{W}(d)}$	0	$\frac{m_1+m_3}{\mathcal{W}(d)}$	0	1

FIG. 9.5 – Matrice de liens pondérée prenant en compte les opérations privées

9.2.3 Pertinence des modèles de contextes

Nous proposons d'encoder les pertinences d'une *BData* d sous une forme matricielle car ces pertinences sont calculées pour toutes les configurations possibles des opérations utilisant d. De ce fait, les matrices de liens pondérées couvrent toutes les configurations d'opérations présentes au niveau des contextes. Notre objectif étant de mesurer la pertinence des modèles de contextes, nous proposons donc de déterminer sur la base des mesures précédentes les pertinences de chaque *BData* par rapport aux différents contextes de chaque modèle de contextes. Soit $\mathcal{F}_i = (\mathcal{D}_i, \mathcal{O}_i)$ un contexte et soit d une *BData* telle que $d \in \mathcal{D}_i$, nous désignons par $\mathcal{P}_{\mathcal{F}_i}(d)$ la mesure de pertinence de d dans le contexte $\mathcal{F}_i$. En d'autres termes $\mathcal{P}_{\mathcal{F}_i}(d) = \mathcal{P}(d, \mathcal{O}_i)$. La pertinence de d dans un modèle de contextes $\mathcal{G}$ (notée $\mathcal{P}_{\mathcal{G}}(d)$) est calculée sur la base des pertinences de d dans chaque contexte de $\mathcal{G}$:

Définition 9.3 *Soit $\mathcal{G} = \{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n\}$ un modèle de contextes constitué de n contextes, et soit d une BData, alors la pertinence $\mathcal{P}_{\mathcal{G}}$ de d au niveau du modèle de contextes $\mathcal{G}$ est :*

$$\mathcal{P}_{\mathcal{G}}(d) = \sum_{i=1}^{n} (\mathcal{P}_{\mathcal{F}_i}(d))$$

Le calcul de $\mathcal{P}_{\mathcal{G}}$ pour chaque *BData* permet de définir une pertinence moyenne de $\mathcal{G}$. Le classement des modèles de contextes pouvant être produits par notre algorithme s'effectue donc sur la base de cette pertinence moyenne.

Dans le but d'illustrer le calcul de la pertinence de chaque modèle de contextes, nous prenons l'exemple de la machine *SecureFlightRegistration* (page 113) dont le réseau de concepts ($\mathcal{J}$) et le réseau de concepts raffiné $\mathcal{J}_{\mathcal{R}}$ sont les suivants :

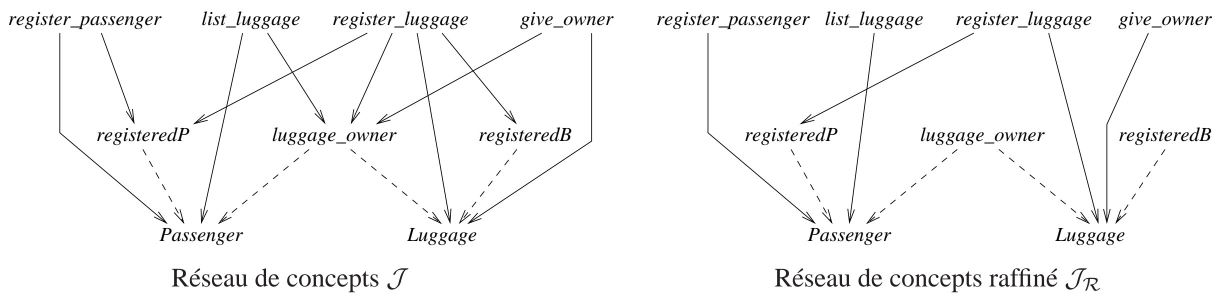


FIG. 9.6 – Réseau de concepts $\mathcal{J}$ et réseau de concepts raffiné $\mathcal{J}_{\mathcal{R}}$ issus de *SecureFlightRegistration*

Notre technique de formation de concepts se base sur ces deux structures et produit les trois modèles de contextes (a), (b) et (c) de la Fig. 9.7. Notons que ces modèles de contextes correspondent aux choix possibles de la cible de l'opération *register_luggage*. En effet, remarquons à partir du réseau de concepts raffiné $\mathcal{J}_{\mathcal{R}}$ que seule l'opération *register_luggage* dispose de plusieurs cibles possibles.

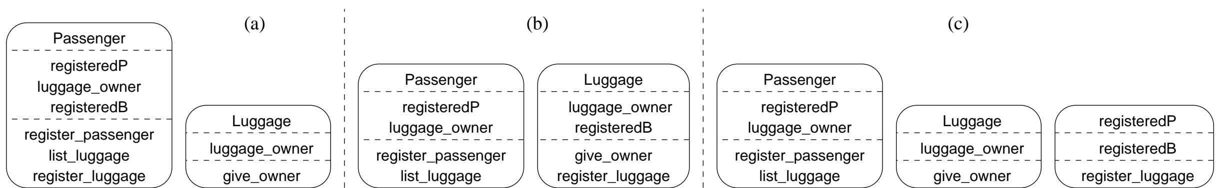


FIG. 9.7 – Modèles de contextes dérivés de *SecureFlightRegistration*

Pour montrer l'utilité des matrices de liens dans le cadre de ces trois modèles de contextes, nous prenons l'exemple des *BData*s *registeredP* et *Luggage* où :

$$\mathcal{I}[\{registeredP\}] = \{register_passenger, register_luggage\}$$
$$\mathcal{I}[\{Luggage\}] = \{give_owner, register_luggage\}$$

Les opérations *register_passenger*, *register_luggage* et *give_owner* accèdent soit en modification soit en pré-condition aux *BData*s *registeredP* et *Luggage*. Nous considérons donc w et p les poids associés respectivement à un accès en modification et à un accès en pré-condition, et soit $\mathcal{W} = p + w$, alors les matrices de liens pondérées associées à *registeredP* et *Luggage* sont :

$\mathcal{L}^m(registeredP) \widehat{=}$

register_passenger		w		w
register_luggage			p	p
Pertinence	0	$\frac{w}{\mathcal{W}}$	$\frac{p}{\mathcal{W}}$	1

$\mathcal{L}^m(Luggage) \widehat{=}$

give_owner		p		p
register_luggage			p	p
Pertinence	0	$\frac{p}{\mathcal{W}}$	$\frac{p}{\mathcal{W}}$	1

FIG. 9.8 – Matrices de liens pondérées associées à *registeredP* et *Luggage*

Remarquons que les modèles de contextes (a) et (c) sont les plus appropriés pour *registeredP* car $\mathcal{P}_{(a)}(registeredP) = \mathcal{P}_{(c)}(registeredP) = 1$. Cependant, ces deux modèles de contextes ne sont pas les plus appropriés pour *Luggage* car $\mathcal{P}_{(a)}(Luggage) = \mathcal{P}_{(c)}(Luggage) = \frac{p}{\mathcal{W}}$ alors que $\mathcal{P}_{(b)}(Luggage) = 1$. La Fig. 9.9 présente les pertinences $\mathcal{P}_{\mathcal{G}}$ calculées pour chaque *BData* issue de la spécification *SecureFlightRegistration* étant donnés les poids suivants : $w = 0.80$, $r = 0.40$ et $p = 0.20$.

Contexte (a)	$\mathcal{P}_{Passenger}$	$\mathcal{P}_{Luggage}$	$\mathcal{P}_{(a)}$
Passenger	100%		100%
Luggage		50 %	50 %
registeredP	100%		100%
registeredB	100%		100%
luggage_owner	60%	40 %	100%
Pertinence moyenne = 90 %			

Contexte (b)	$\mathcal{P}_{Passenger}$	$\mathcal{P}_{Luggage}$	$\mathcal{P}_{(b)}$
Passenger	100%		100%
Luggage		100 %	100 %
registeredP	80%		80%
registeredB		100%	100%
luggage_owner	40%	60 %	100%
Pertinence moyenne = 96 %			

Contexte (c)	$\mathcal{P}_{Passenger}$	$\mathcal{P}_{Luggage}$	$\mathcal{P}_{registeredP}$	$\mathcal{P}_{(c)}$
Passenger	100%			100%
Luggage		50 %		50 %
registeredP	80%		20%	100%
registeredB			100%	100%
luggage_owner	40%	40 %		80%
Pertinence moyenne = 86 %				

FIG. 9.9 – Pertinences $\mathcal{P}_{\mathcal{G}}$ calculées pour les modèles de contextes (a), (b) et (c)

Dans le but de classer les modèles de contextes dérivés de la spécification *SecureFlightRegistration* sur la base des métriques de la Fig. 9.9, nous calculons une pertinence moyenne pour chacun de ces modèles de contextes. Cette mesure identifie le modèle (b) comme étant le modèle de contextes le plus pertinent vu qu'il présente une pertinence moyenne maximale (96%).

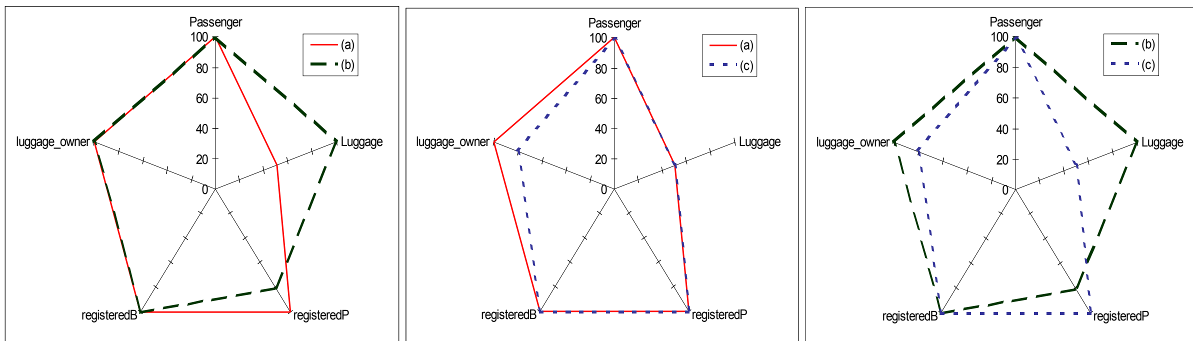


FIG. 9.10 – Comparaison des modèles de contextes (a), (b) et (c)

La Fig. 9.10 compare, deux à deux, les trois modèles de contextes (a), (b) et (c) et montre que le modèle de contextes (b) est le plus adéquat pour la majorité des *BData*s. Notons que bien que ce modèle de contextes est le moins pertinent pour *registeredP*, il peut être considéré comme acceptable car $\mathcal{P}_{(b)}(registeredP) = 80\%$.

9.2.4 Évaluation

L'approche présentée au niveau de cette section, permet de sélectionner les modèles de contextes estimés les plus convenables sur la base d'un ensemble de poids. Ces derniers sont associés de manière empirique aux différents types d'utilisation (modification, lecture et pré-condition). Dans le but d'évaluer cette technique nous l'avons appliquée sur la spécification *SecureFlight*R4 extraite du projet EDEMOI et pour laquelle notre algorithme de formation de concepts produit 65 modèles de contextes. La Fig. 9.11 illustre les pertinences moyennes calculées pour tous ces modèles de contextes et distingue 4 modèles dont la pertinence moyenne est égale à 95%. Nous illustrons au niveau de la Fig. 9.12, un diagramme de classes généré par notre outil prototype à partir de l'un de ces 4 modèles de contextes.

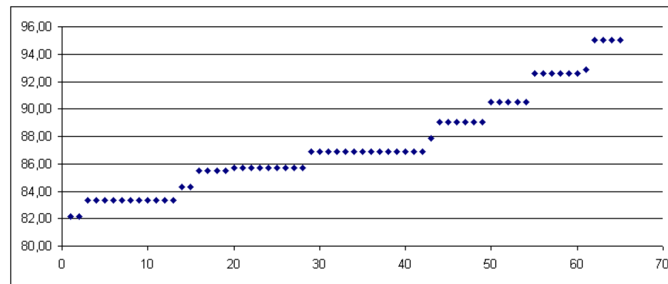


FIG. 9.11 – Classement des 65 modèles de contextes issus de la spécification *SecureFlight*

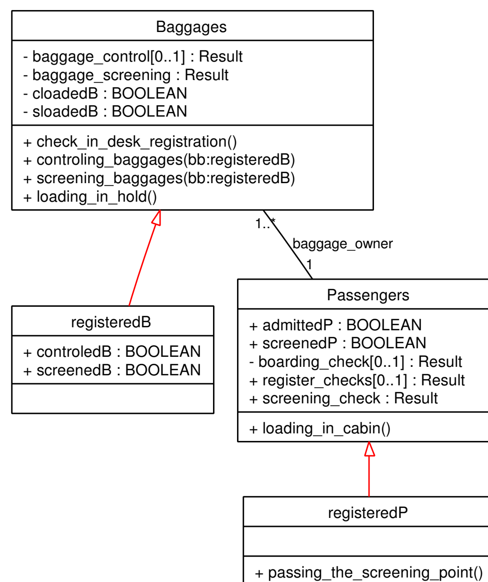


FIG. 9.12 – Diagramme de classes issu de l'un des modèles de contextes dont la pertinence moyenne est égale à 95%

Ainsi, l'utilisation des matrices de liens, nous a permis d'améliorer notre technique de dérivation de diagrammes de classes en offrant un potentiel d'automatisation plus important. La seule activité laissée à l'analyste est de choisir les différentes pondérations des types de liens entre opérations et *BData*s et de choisir parmi les diagrammes dont la pertinence moyenne est maximale. Toutefois, les poids que nous avons choisis portent plus particulièrement sur les trois types d'utilisation que nous avons identifiés au niveau de la relation de dépendance de concepts $\mathcal{I}$. Nous pensons que l'exploitation d'autres métriques notamment l'association de métriques aux liens de dépendances entre *BData*s elles-mêmes (exploitation de la relation d'inclusion $Incl$) peuvent être des améliorations intéressantes de cette technique.

Finalement, les expérimentations menées avec la spécification *SecureFlight* montrent que cette technique nous permet de passer de spécifications B de petite taille (quelques dizaines de lignes de code B) à des spécifications de taille moyenne (quelques centaines de lignes de code B). Le tableau 9.1 donne une vue d'ensemble sur le nombre de diagrammes de classes qui peuvent être construits pour chaque étape de raffinement de *SecureFlight*. Ce nombre augmente avec le nombre de classes candidates et d'opérations, parce qu'il permet une plus grande variété de choix de la fonction $Target$.

	Lignes de code B	**Diagrammes de classes**
SecureFlight	80	1
*SecureFlight*R1	142	2
*SecureFlight*R2	158	10
*SecureFlight*R3	192	15
*SecureFlight*R4	222	65

TAB. 9.1 – Nombre de diagrammes de classes construits pour les développements de *SecureFlight*

Aujourd'hui, la spécification B du métro sans conducteur de Paris (METEOR) est considérée la spécification la plus grande (plusieurs milliers de lignes de code B $\simeq$ 100.000 lignes). Le passage vers de telles spécifications pourrait faire l'objet d'une explosion importante du nombre de configurations et contraindre fortement l'élaboration des matrices de liens. C'est pourquoi cette approche vise les premières phases d'un développement en B où les spécifications sont réalisées avec un haut niveau d'abstraction et un niveau de détails maniable. Dans un tel contexte, et dans le but de faire face à la grande variété d'instances de la fonction $Target$ nous proposons une technique prenant en compte les raffinements en B pour inférer les caractéristiques des modèles de contextes issus de spécifications abstraites au niveau des modèles de contextes issus des spécifications concrètes.

9.3 Prise en compte des raffinements

À ce niveau nous nous intéressons principalement au mécanisme de raffinement permettant de développer des spécifications d'une manière incrémentale en y introduisant plus de détails et en précisant de plus en plus son comportement. Cette section présente comment une telle technique de développement peut être prise en compte pour optimiser les résultats produits par notre algorithme de formation de concepts. Pour ce faire, nous allons appliquer notre approche sur les différents niveaux de raffinements de SecureFlight (Fig. 9.13 et Tab. 9.1) développés au niveau du projet EDEMOI.

Le niveau le plus abstrait (*SecureFlight*) spécifie les objets (ensemble abstrait *Objects*) qui peuvent ou non être chargés dans la soute à bagages ou transportés dans l'avion. L'ensemble *Objects* disparaît

lors du premier raffinement (*SecureFlight*R1) où il s'agit de considérer les passagers (ensemble abstrait *Passengers*) et leurs bagages (*Baggages*). Les trois autres niveaux de raffinement gardent ces notions et introduisent de nouvelles opérations.

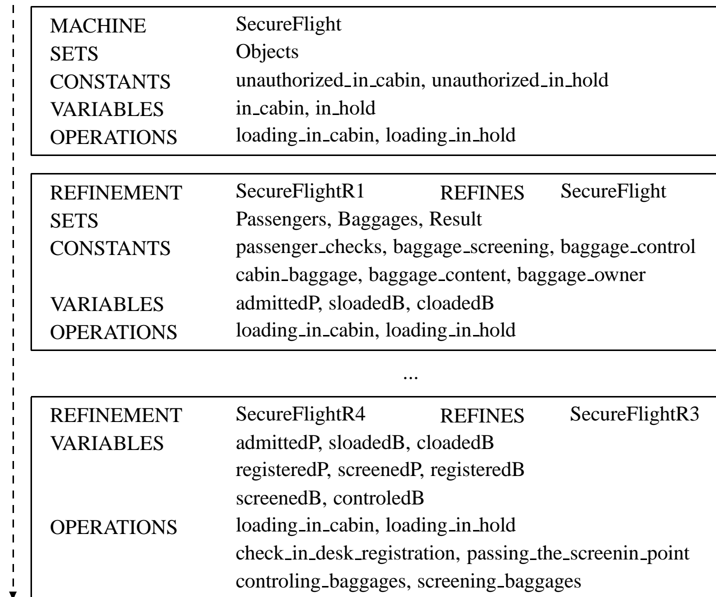


FIG. 9.13 – Développements en B de SecureFlight

En B, un raffinement hérite certaines déclarations du niveau abstrait précédent telles que, par exemple, la définition des ensembles, des constantes concrètes et des variables concrètes ou de même nom. Dans notre démarche de prise en compte des raffinements nous passons par une phase d'aplatissement (ou dépliage) de spécifications. Cette phase collecte les informations (déclarations, type, etc) concernant les ensembles, les constantes concrètes et les variables, et les insère au niveau concret pour être analysées. L'algorithme utilisé pour former les contextes à partir d'une simple machine abstraite peut alors être réutilisé pour les raffinements et l'application des règles de transformation construisant des classes à partir des contextes devient possible. Notons que les résultats présentés au niveau du tableau 9.1 sont dérivés de l'application de notre approche de formation de concepts sur des spécifications aplaties.

9.3.1 Représentation du plus haut niveau d'abstraction

Le diagramme de classes de la Fig. 9.14 est produit pour le niveau abstrait *SecureFlight*. Il s'agit d'une seule classe *Objects* décrite par 4 attributs booléens : *in_cabin*, *in_hold*, *unauthorized_in_cabin* et *unauthorized_in_hold*.

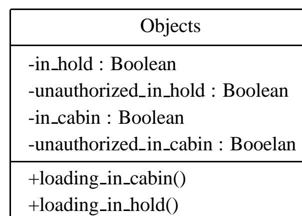


FIG. 9.14 – Diagramme de classes issu du plus haut niveau d'abstraction de SecureFlight

Les méthodes de cette classe correspondent aux opérations de la spécification et sont : *loading_in_cabin* et *loading_in_hold*. Charger un objet o en soute revient à appeler la méthode *loading_in_hold* de l'instance o de la classe *Objects*. Cette action permet de mettre à vrai l'attribut *in_hold* de l'instance o et ne sera réalisée que si l'attribut *unauthorized_in_hold* est à faux.

9.3.2 Premier niveau de raffinement

L'application de notre algorithme à une spécification aplatie correspondant au premier niveau de raffinement permet la construction de deux diagrammes de classes différents. En effet, l'opération *loading_in_hold* dispose de deux cibles possibles : *Passengers* et *admittedP*. La Fig. 9.15 illustre le cas où : $Target(\textit{loading_in_hold}) = \textit{AdmittedP}$, ce qui identifie *AdmittedP* comme une sous-classe de la classe *Passengers*.

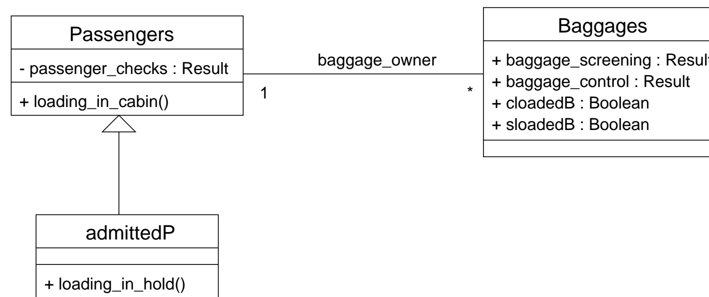


FIG. 9.15 – Diagramme de classes produit pour le raffinement SecureFlightR1

9.3.3 Derniers niveaux de raffinement

L'application de l'approche que nous proposons à l'aplatissement du troisième niveau de raffinement produit 15 diagrammes de classes illustrant chacun une certaine vision statique d'une même spécification. Ce nombre explose avec l'introduction de plus de *BData*s et d'opérations au niveau de la spécification. En effet, pour le dernier niveau de raffinement notre technique aboutit à 65 diagrammes de classes différents.

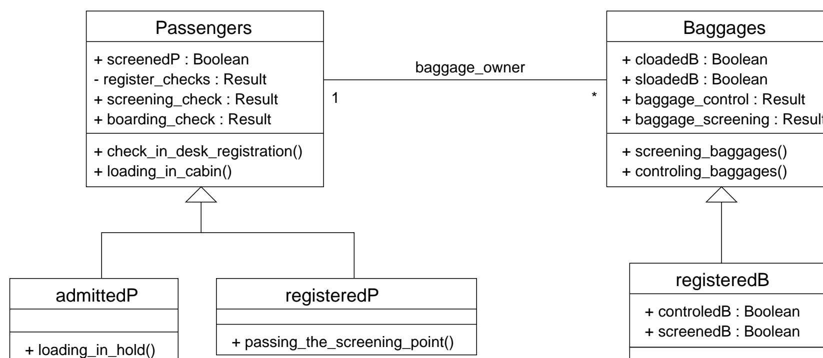


FIG. 9.16 – Diagramme de classes issu du dernier niveau de raffinement

Pour ce dernier niveau, nous avons choisi d'illustrer une représentation (Fig. 9.16) issue des 4 modèles de contextes (Fig. 9.11) dont la pertinence moyenne est égale à 95% et où les classes identifiées

dans les niveaux précédents (*Passengers* et *Baggages*) sont structurées en sous-classes. D'un point de vue "documentation", ce diagramme permet, par exemple, de mettre l'accent sur le fait que l'opération *Passing_the_screening_point* n'a de sens que pour les passagers enregistrés (*registeredP*), ou que seulement les bagages enregistrés (*registeredB*) sont contrôlés (*controlledB*) ou scannés (*screenedB*).

9.3.4 Mise en relation des vues produites pour les raffinements

L'utilisation des matrices de liens pondérées répond à la difficulté d'examiner toutes les représentations issues d'une même spécification B en vue de sélectionner celle qui paraît la plus intéressante pour un objectif de documentation. Rappelons que les métriques considérées pour le classement des diagrammes sont purement empiriques et sont appliquées à des spécifications aplaties. De ce fait, l'unique application des matrices de liens peut aboutir à des diagrammes pertinents quand ces derniers sont considérés indépendamment des différents niveaux d'abstraction. En effet, cette technique ne garantit pas l'homogénéité des diagrammes dont la pertinence moyenne est maximale et qui sont issus des différents aplatissements. Cette homogénéité est, à notre avis, primordiale pour pouvoir documenter de manière adéquate un développement par raffinements dans sa globalité. Nous proposons alors de sélectionner les vues graphiques qui correspondent à des choix de représentation cohérents pour tous les niveaux de raffinement. Pour ce faire, nous proposons deux critères de cohérence : la **préservation des cibles** et la **préservation des contextes**.

Avant de présenter nos deux critères de préservation des cibles et des contextes nous considérons les hypothèses suivantes :

1. M et M' deux spécifications B correspondant à des raffinements successifs et telle que $M \sqsubseteq M'$,
2. $\mathcal{I}$ et $\mathcal{I}'$ les relations de dépendance de concepts issues respectivement de M et de M',
3. $Target$ et $Target'$ les choix de cibles établis pour chacun de ces deux niveaux de raffinement, et
4. $\mathcal{C}lasses$ et $\mathcal{C}lasses'$ les ensembles de classes candidates identifiés respectivement pour M et M'.

Préservation des cibles. La préservation des cibles est une condition nécessaire pour pouvoir étudier la préservation des modèles de contextes. Il s'agit de mettre en relation les choix de cibles pour les deux niveaux de raffinement. La fonction $Target$ est dite préservée par $Target'$ si toutes les classes candidates partagées par les deux niveaux de raffinement et choisies comme cibles au niveau abstrait apparaissent aussi en tant que cibles des mêmes opérations au niveau concret :

$$\boxed{(Target \rhd \mathcal{C}lasses') \subseteq Target'}$$

Par exemple, les diagrammes 9.15 et 9.16 satisfont ce critère car les opérations *loading_in_cabin* et *loading_in_hold* ont les mêmes cibles dans les deux diagrammes. Ceci est également le cas pour les diagrammes 9.14 et 9.15 car les cibles des deux opérations disparaissent lors du raffinement.

Préservation entre contextes. Soient $\mathcal{F} = (\mathcal{D}, \mathcal{O})$ et $\mathcal{F}' = (\mathcal{D}', \mathcal{O}')$ deux contextes dérivés respectivement de M et M', alors nous définissons la notion de préservation entre contextes comme suit :

$$\boxed{\begin{array}{lll} Preserve(\mathcal{F}', \mathcal{F}) \mathrel{\widehat{=}} & \mathcal{O} \subseteq \mathcal{O}' & \wedge \\ & Target[\mathcal{O}] = Target[\mathcal{O}'] & \wedge \\ & \mathcal{D} \cap \mathrm{dom}(P_{\mathcal{I}'}) \subseteq \mathcal{D}' & \end{array}}$$

La préservation entre contextes indique les trois propriétés suivantes :

- le contexte le plus concret inclut toutes les opérations du contexte abstrait,
- les classes candidates à l'origine des deux contextes sont les mêmes, et
- les attributs du contexte abstrait apparaissant au niveau concret sont également des attributs du contexte concret.

Préservation des modèles de contextes. Soient $\mathcal{G}$ et $\mathcal{G}'$ deux modèles de contextes issus respectivement de M et de M' et satisfaisants le critère de préservation des cibles. Nous disons que $\mathcal{G}$ préserve $\mathcal{G}'$ si, et seulement si :

$$\boxed{\forall \mathcal{F} \cdot (\mathcal{F} \in \mathcal{G} \Rightarrow \exists \mathcal{F}' \cdot (\mathcal{F}' \in \mathcal{G}' \wedge \mathit{Preserve}(\mathcal{F}', \mathcal{F})))}$$

En d'autres termes, un modèle de contextes est préservé si tous ses contextes sont préservés. Ceci est le cas pour les diagrammes 9.15 et 9.16. Cependant cette propriété n'est pas vérifiée pour les diagrammes 9.14 et 9.15 car aucun contexte du niveau concret ne correspond au contexte ($Objects$) du niveau abstrait.

Préservation faible des modèles de contextes. Nous définissons une forme plus faible de préservation des modèles de contextes qui ne s'applique pas à tous les contextes de $\mathcal{G}$ mais seulement aux contextes impliqués au niveau de la préservation des cibles. Il s'agit donc uniquement des contextes $\mathcal{F} = (\mathcal{D}, \mathcal{O})$ et $\mathcal{F}' = (\mathcal{D}', \mathcal{O}')$ avec $\mathcal{F} \in \mathcal{G}$ et $\mathcal{F}' \in \mathcal{G}'$ et telle que $\mathit{Target}[\mathcal{O}] = \mathit{Target}[\mathcal{O}']$. Cette propriété de préservation faible des contextes permet de considérer cohérents les diagrammes 9.14 et 9.15.

Discussion. La Fig. 9.17 illustre le nombre de diagrammes de classes produits avec la prise en compte du critère de préservation des modèles de contextes. La vue sélectionnée lors du premier niveau de raffinement (*SecureFlight*R1) correspond au diagramme produit pour $\mathit{Target}(\mathit{loading_in_hold}) = \mathit{admittedP}$ et $\mathit{Target}(\mathit{loading_in_cabin}) = \mathit{Passengers}$ (Fig. 9.15).

Spécification	diagrammes	préservation des contextes
SecureFlight	1	
SecureFlightR1	2	
SecureFlightR2	10	target(loading_in_hold) = admittedP target(loading_in_cabin) = Passengers
SecureFlightR3	15	target(passing_the_screening_point) = Passengers
SecureFlightR4	65	target(check_in_desk_registration) = Passengers

FIG. 9.17 – Optimisation du nombre de vues produites pour les différents raffinements

L'application de notre technique de formation de concepts permet de produire 10 diagrammes différents pour le deuxième niveau de raffinement (*SecureFlight*R2) dont seulement 2 permettent de préserver la vue sélectionnée au niveau précédent. Ces 2 vues correspondent à 2 choix possibles pour la cible de l'opération *passing_the_screening_point*. Nous faisons ici le choix d'associer cette opération à la classe candidate *Passengers* et d'observer les contextes qui préservent ce choix pour les niveaux suivants.

Nous remarquons que pour le $3^{\text{ème}}$ niveau de raffinement (*SecureFlight*R3) un seul diagramme parmi les 15 possibles permet de préserver le modèle de contextes construit pour le niveau précédent. Finalement, pour le dernier niveau de raffinement nous avons pu distinguer 3 modèles de contextes (parmi les 65 modèles possibles) satisfaisant les critères de préservation des modèles de contextes, et ce, étant donné $Target(\textit{check_in_desk_registration}) = \textit{Passengers}$.

Cet exemple montre donc que l'application systématique de ces critères de préservation lors d'un développement par raffinements restreint significativement le choix des représentations UML à chaque niveau de raffinement. L'analyste peut donc examiner plus attentivement ces choix qui lui garantissent par ailleurs une cohérence entre les représentations des divers niveaux.

9.4 Conclusion

Dans ce chapitre, nous avons exploré deux techniques pour compenser la forte interaction avec l'analyste lors de l'instanciation de la fonction $Target$. D'un côté la prise en compte des critères de préservation des modèles de contextes permet de sélectionner des vues garantissant une cohérence par rapport aux raffinements, et d'un autre côté, l'utilisation des matrices de liens permet d'évaluer et donc trier, sur la base de mesures effectives, les différents modèles de contextes. Aussi, l'utilisation conjointe de ces techniques peut-elle réduire significativement le nombre de diagrammes tout en assurant une évaluation empirique de leur pertinence.

Par ailleurs, ces deux techniques sont fondées sur les modèles de contextes car ces derniers présentent une base formelle pour la construction des diagrammes de classes. Nous pensons que des solutions alternatives permettant d'évaluer directement les diagrammes de classes (Chidamber *et al.,* 1994) peuvent être envisagées pour étudier la pertinence de ces diagrammes pour la documentation des spécifications formelles.

Nous pensons aussi que nos techniques d'optimisation peuvent être adaptées et exploitées par d'autres approches de dérivation de diagrammes de classes à partir de spécifications, notamment l'approche de (Fekih *et al.,* 2006). En effet, l'application des règles de (Fekih *et al.,* 2006) à la machine *SecureFlightRegistration* identifie un nombre de classes allant de 2 à 4 classes. Vu qu'aucun critère n'est identifié pour dispatcher les 4 opérations de la spécification sur ces classes, alors (Fekih *et al.,* 2006) aboutit à un nombre de diagrammes de classes allant de $2^4 = 16$ à $4^4 = 256$. Aussi, les diagrammes construits par cette technique sont-ils difficiles à gérer de manière interactive lorsque les spécifications sont de taille importante.

Quatrième partie

Construction de vues comportementales à partir de spécifications B

Akram Idani, Yves Ledru. "**Dynamic Graphical UML views from Formal B Specifications**".

In *International Journal of Information and Software Technology*.

Volume 48, n°3. Pages 154 - 169. Elsevier, 2006.

Chapitre 10

Vues comportementales issues de spécifications B

« *Il n'y a qu'une méthode pour inventer, qui est d'imiter. Il n'y a qu'une méthode pour bien penser, qui est de continuer quelque pensée ancienne et éprouvée.* »

Alain
« Propos », 1868 – 1951.

Sommaire

10.1 Introduction 172
10.2 Notions d'état et de transition 173
10.2.1 Présentation informelle 173
10.2.2 Formalisation 174
10.3 Vues comportementales concrètes 175
10.3.1 Exploration du comportement d'une spécification B 176
10.3.2 Représentation concrète structurée du raffinement 177
10.3.3 Limites des vues concrètes 179
10.4 Vues comportementales abstraites 180
10.4.1 Le gestionnaire de processus 180
10.4.2 Graphes d'états concrets et explosion d'états 181
10.4.3 Focalisation sur une variable particulière 182
10.4.4 Mise en évidence sous forme concurrente de toutes les variables d'état . . 183
10.4.5 Représentation graphique d'une propriété invariante 185
10.4.6 Représentation du comportement d'un processus 185
10.5 Bilan 187

10.1 Introduction

Dans ce chapitre nous abordons une autre facette, certes complémentaire, de la documentation de spécifications B. Il s'agit précisément de proposer une stratégie qui permette de construire des diagrammes d'états/transitions illustrant le comportement d'une spécification B (Idani *et al.,* 2006a). Jusque là, nous nous sommes intéressés uniquement à la dérivation de diagrammes structurels. Toutefois, l'intérêt porté à la dérivation de diagrammes de classes est grandement lié à la complexité de la structure des données au niveau d'une spécification B. En effet, plus cette structure est complexe, plus il est opportun d'en dériver un diagramme de classes. De ce fait, nous allons aborder deux points de vue distincts pour la documentation des aspects comportementaux :

- ◇ Produire des vues comportementales indépendamment d'une quelconque vue structurelle ;
- ◇ Dériver des diagrammes d'états/transitions pour illustrer le comportements des instances des classes d'un diagramme de classes issu de la spécification B.

Dans le premier point de vue, les diagrammes comportementaux sont principalement liés à l'évolution des variables d'états. Par exemple, la Fig. 10.1, extraite de la machine RESERVATION (page 21) illustre le comportement de cette spécification en explicitant les évolutions de la variable d'état $places$.

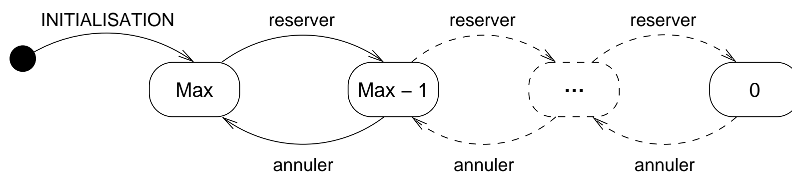


FIG. 10.1 – Illustration de l'évolution de l'état de la machine RESERVATION

Au niveau de ce diagramme les états sont des valuations possibles (permises par l'invariant) de la variable $places$ et les transitions entre états correspondent à des appels d'opérations ($reserver$ et $annuler$). À l'initialisation $places$ prend la valeur de la constante Max. Un tel diagramme permet de visualiser certaines propriétés du système telles que :

- L'*atteignabilité* : tous les états permis par l'invariant sont accessibles ;
- L'*absence de deadloack* : à partir de n'importe quel état du diagramme il existe une transition permettant un changement d'état.

D'autres diagrammes comportementaux peuvent être utilisés pour illustrer le comportement de la spécification RESERVATION. Par exemple, dans la Fig. 10.2, ci-dessous, nous avons choisi d'illustrer trois états essentiels ($places = Max$, $places = 0$ et $places \in 1..(Max - 1)$). Ces états correspondent à des prédicats satisfaisant l'invariant et mettent l'accent sur :

- deux états limites ($places = Max$ et $places = 0$) qui correspondent aux valeurs des limites de l'intervalle $0..Max$ et
- tous les états intermédiaires (entre 1 et $Max - 1$).

À la différence du diagramme de la Fig. 10.1 illustrant un comportement concret, ce diagramme présente une vue abstraite sur l'évolution de l'état de la variable $places$. Dans ce chapitre nous présentons une série de vues abstraites pouvant être extraites d'un même diagramme de comportement concret. Le

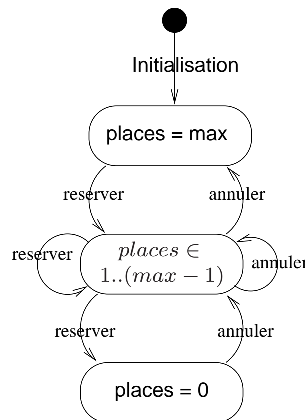


FIG. 10.2 – Diagramme d'états / transitions abstrait dérivé de la machine RESERVATION

principe, la démarche de construction ainsi que l'utilité de chaque vue seront discutés au fur et à mesure de leur présentation. Nous montrons également qu'une combinaison des techniques de construction de vues structurelles et comportementales permet de présenter un ensemble de diagrammes formant une documentation plus complète d'une même spécification. Ce faisant, les techniques utilisées et proposées pour la production des diagrammes comportementaux abstraits seront présentées et discutées au niveau du chapitre suivant.

10.2 Notions d'état et de transition

Dijkstra (Dijkstra, 1976) définit l'espace d'états d'une variable de la manière suivante : "*The state space describes the amount of freedom of the system*; *it just has nowhere else to go*". L'exemple du registre à 8 cases est généralement donné afin d'illustrer cette notion. Chaque case du registre peut contenir des valeurs de 0 à 9. Chaque valeur supportée par la case représente son état à un moment donné. Ceci étant, le registre dans son ensemble est dans l'un des 100.000.000 états différents. Cette notion d'espace d'états est aussi rattachée aux variables d'états des spécifications B. Il s'agit, plus précisément de leur domaine de définition. Ainsi, représenter tous les états possibles de ces variables sous une forme graphique s'avère très contraignant. Il faut donc chercher à partitionner cet espace et définir des sous-ensembles d'états significatifs, satisfaisant certaines conditions et pouvant ainsi être représentés de façon plus expressive. Cette évidence nous amène à considérer la mise en valeur du contraste entre l'ensemble d'états concrets d'une spécification B et la proposition d'un ensemble d'états abstraits.

10.2.1 Présentation informelle

Rappelons qu'en B, la partie statique d'une machine abstraite est constituée, entre autres, de l'état interne (caractérisé par l'ensemble des variables d'états v) et des invariants $I(v)$ définissant les espaces d'états associés aux variables. Quant à la partie opérationnelle de la machine, elle décrit comment cet état évolue. Nous définissons alors d'une manière informelle la sémantique de nos diagrammes d'états/transitions comme suit :

- Un **état** S est exprimé par un prédicat $P(v)$ satisfaisant l'invariant $I(v)$. En d'autres termes : S est tel que : $P(v) \Rightarrow I(v)$. L'état S détermine ainsi un sous-espace particulier de valuations des

variables v. Notons que dans la suite de notre travail, nous allons considérer des états mutuellement exclusifs ;

- Une **transition** entre un état source et un état cible désigne un appel d'une opération[34]. Une transition $S_1 \xrightarrow{o} S_2$ exprime le fait que l'opération o peut, d'une part, être déclenchée à partir de l'état S_1, et d'autre part, atteindre l'état S_2 ;
- Une **transition gardée** associe une condition à la transition. Une transition $S_1 \xrightarrow{[c]o} S_2$ exprime le fait que si la condition c est vraie, alors l'opération o peut être déclenchée à partir de l'état S_1 et peut atteindre l'état S_2, sinon l'opération n'est pas déclenchable à partir de S_1 ;
- Une **transition non-déterministe** est une transition qui décrit plusieurs comportements possibles sans préciser lequel des états cibles est effectivement atteint. Une transition $S_1 \xrightarrow{[c]o} S_2$ est dite non-déterministe s'il existe une autre transition $S_1 \xrightarrow{[c']o} S_3$ telle que :
  – les états S_2 et S_3 sont différents, et
  – les conditions c et c' sont vraies simultanément.
  Par exemple, à la Fig. 10.2 la transition $reserver$ déclenchée à partir de l'état $places \in 1..(max-1)$ est une transition non-déterministe car elle permet, d'une part, d'atteindre l'état $places = 0$, et d'autre part, de rester dans le même état source ;
- L'initialisation est une transition particulière déclenchée à partir d'un pseudo-état initial.

10.2.2 Formalisation

Une spécification B peut être caractérisée par l'ensemble des traces finies observables. Soit $init$ la substitution d'initialisation et soit $\{o_1, o_2, \ldots, o_n\}$ l'ensemble des opérations (ou événements s'il s'agit d'un système B) de la spécification B. Nous appelons **trace** toute séquence T de la forme :

$$T \mathrel{\widehat{=}} init, t_1, t_2, \ldots, t_m$$

où $t_i \in \{o_1, o_2, \ldots, o_n\}$ et telle qu'il existe une séquence de valeurs $val_0, val_1, \ldots, val_m$ des variables d'états v avec :

– val_0 est une valeur initiale possible,
– t_i est déclenchée à partir de val_i,
– val_{i+1} est une valeur atteignable par t_i à partir de val_i.

Nous rattachons la sémantique des traces en B à nos diagrammes d'états/transitions de la manière suivante : une trace T est une séquence de la forme $init, t_1, t_2, \ldots, t_m$ où $t_i \in \{o_1, o_2, \ldots, o_n\}$ et telle qu'il existe une séquence de la forme $(q_0, val_0), (q_1, val_1), \ldots, (q_n, val_n)$ avec :

– q_i est une occurrence d'un état abstrait S_j,
– q_0 est l'état initial,
– pour chaque paire (q_i, val_i), $P_j(v)$ est vrai (P_j est le prédicat d'état associé à l'état S_j),
– t_{i+1} est une transition de q_i vers q_{i+1}, déclenchée à partir de val_i et permettant d'atteindre la valeur val_{i+1}.

Nous considérons le système B de la Fig. 10.3 représentant un protocole de communication entre un émetteur et un récepteur liés par un canal de communication. Notons qu'à ce niveau nous supposons que l'émetteur et le récepteur connaissent tous les deux l'état du canal.

[34] En B événementiel on parlera d'une occurrence d'événements et non un déclenchement ou appel d'opérations.

```
SYSTEM
   COMMUNICATION
SETS
   Status = {empty, full}
VARIABLES
   channel
INVARIANT
   channel ∈ Status
INITIALISATION
   channel := empty
EVENTS
   send =
         SELECT
            channel = empty
         THEN
            channel := full
         END;
   receive =
            SELECT
               channel = full
            THEN
               channel := empty
            END
END
```

FIG. 10.3 – Système COMMUNICATION

Intuitivement, nous pouvons modéliser le comportement de cette spécification par le diagramme d'états/transitions concret ci-dessous. À l'initialisation le canal est à l'état *empty*, il y a alors émission d'informations. Cette émission fait passer le canal de l'état *empty* à l'état *full*. Ainsi, la réception peut être effectuée, ce qui remet l'état du canal à *empty*, et ainsi de suite.

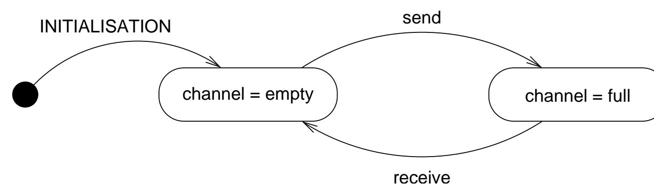


FIG. 10.4 – Diagramme d'états/transition issu du système COMMUNICATION

10.3 Vues comportementales concrètes

Un diagramme d'états/transitions concret (tel que celui de la Fig. 10.4) est un graphe d'états dont les états correspondent chacun à une valuation particulière de toutes les variables d'états. Nous formalisons ce graphe par le couple $G = (N, T)$ où N représente l'ensemble des états concrets et T l'ensemble des transitions entre les états de N. Un état concret S_v ($S_v \in N$) portant sur les variables d'état v telle que $v = \{v_1, \ldots, v_n\}$, est exprimé par une suite de conjonctions de prédicats d'égalité entre variables et valeurs : $S_v \mathrel{\widehat{=}} \bigwedge_{i=1}^{n}(v_i = val_j)$ où val_j est une valeur possible de v_i.

10.3.1 Exploration du comportement d'une spécification B

La construction des diagrammes d'états/transitions concrets est simplement réalisée par un outil d'animation de spécifications B. En effet, ces outils permettent de dérouler le comportement de la spécification et d'observer les évolutions des variables d'état. Il s'agit d'explorer le comportement de la spécification, en avançant d'état en état et en choisissant les transitions que l'on souhaite déclencher : "*on peut considérer l'animation comme une sorte de debugger fonctionnant au niveau de la spécification*" (Le Guennec, 2001). Nous distinguons trois outils d'animation de spécifications B :

– l'animateur de l'atelier B (ClearSy, 2002a),
– ProB (Leuschel *et al.,* 2003), et
– BZ Testing Tool (Legeard *et al.,* 2002b, Legeard *et al.,* 2002a)

Nous avons mené une étude détaillée de ces outils en collaboration avec Ansem Ben Cheikh (Ben Cheikh, 2004) et nous en dégageons le tableau comparatif suivant :

	`Animateur de l'atelier B`	`ProB`	`BZ Testing Tool`
Génération de diagrammes	Ne produit aucun diagramme.	Génére des diagrammes complets, lisibles et compréhensibles.	Produit des diagrammes partiels avec peu d'informations.
Choix des opérations à exécuter	Ne propose pas de choisir une opération parmi un ensemble d'opérations.	Identifie les opérations pouvant être exécutées à partir d'un état donné et propose à l'utilisateur d'en choisir une.	Propose toutes les opérations (exécutables ou non).
Prise en compte des ensembles indéterminés (*BAbstractSet*)	Lorsqu'il s'agit de traiter des éléments d'ensembles indéterminés l'utilisateur doit rajouter une hypothèse sur l'existence de ces éléments.	Un cardinal par défaut est défini par l'utilisateur comme étant une option de l'outil.	Les ensembles indéterminés ne sont pas acceptés. Tous les ensembles doivent être énumérés.
Traitement du non-déterminisme des opérations	L'animation est stoppée et l'outil propose une liste de valeurs laissant à l'utilisateur le choix d'une valeur particulière.	Propose toutes les exécutions possibles d'une même opération à partir de l'état courant et laisse à l'utilisateur le choix d'une exécution particulière. L'état cible est par la suite calculé.	Propose tous les états atteignables d'une même opération à partir de l'état courant et laisse à l'utilisateur le choix d'un résultat particulier.
Traitement des constantes non-valuées	L'utilisateur doit spécifier une valeur.	Propose des valeurs par défaut pour chaque constante. Lorsqu'il s'agit d'ensembles, un nombre maximum d'éléments (défini par l'utilisateur) est utilisé.	Propose un ensemble de valeurs possibles sans aucune restriction lorsqu'il s'agit d'ensembles.

Remarquons à partir de ce tableau, que chacun de ces animateurs résout un problème spécifique et de manière différente des autres animateurs. Les fondements et usages de ces outils peuvent, en somme, être énoncés comme suit :

L'animateur de l'atelier B : Basé sur une démarche de preuve passant par le prouveur de l'atelier B et sollicitant fortement l'utilisateur. Il est destiné à des vérifications locales et non à une exploration exhaustive du comportement de la spécification B. Son usage reste limité à des spécifications de tailles assez restreintes. L'utilisation de cet outil pour parcourir les états du système COMMUNICATION est réalisée en exécutant les séquences suivantes : (*i*) $init, send, receive$, (*ii*) $init, send, send$ et (*iii*) $init, receive$. La première séquence permet d'atteindre successivement les valeurs $empty$, $full$ et $empty$ de la variable d'état *channel*, tandis que les deux autres séquences échouent car la garde du dernier événement n'est pas établie par son état source.

ProB : Construit autour d'une technique de model-checking (Clarke *et al.,* 1999) qui s'apparente aux techniques de résolution de contraintes en *Programmation Logique*. Il a été développé pour un dépliage automatique de l'ensemble du graphe d'états d'une spécification B. Les restrictions apportées aux ensembles, notamment la prise en compte uniquement d'ensembles finis permettent de réaliser une animation complètement valuée et produire des graphes d'accessibilité de manière systématique. Cet outil produit donc automatiquement le diagramme de la Fig. 10.4.

BZ Testing Tool : Fondé sur des techniques de *Programmation Logique avec Contraintes Ensemblistes* (Peureux, 2002) et développé sur la base du solveur de contraintes CLPS-B (Bouquet *et al.,* 2002). Il est utilisé en particulier pour la génération automatique de scénarios de tests fonctionnels. Dans cette approche, l'état d'une machine B n'est pas représenté par un ensemble de variables valuées, mais par un ensembles de contraintes[35] portant sur les variables d'états. Nous n'évoquerons alors pas cet outil lors de la présentation des diagrammes d'états/transitions.

10.3.2 Représentation concrète structurée du raffinement

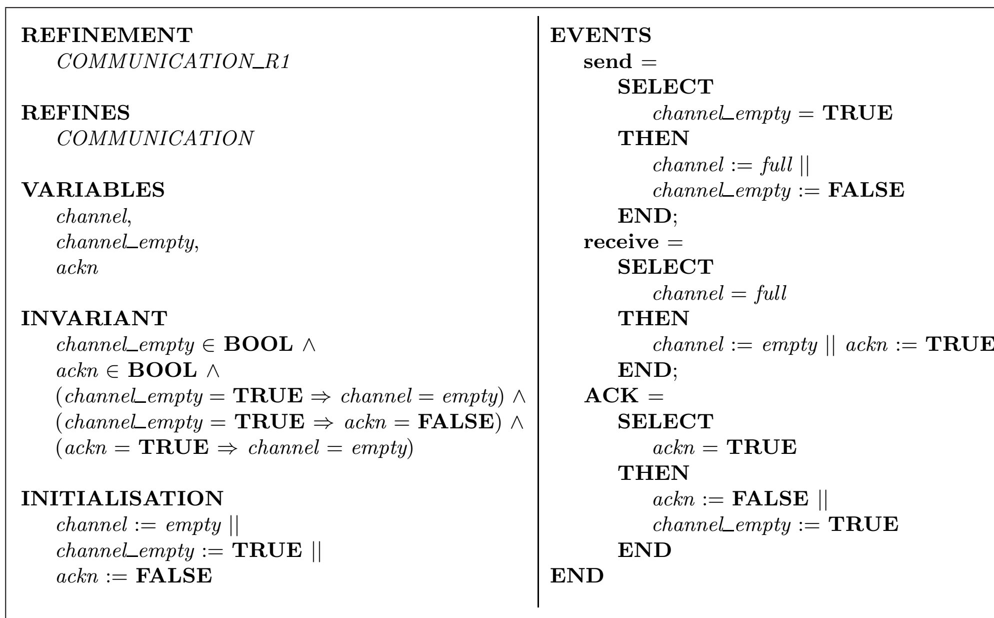
Nous présentons au niveau de la Fig. 10.5 un raffinement du système COMMUNICATION où il s'agit d'affiner l'observation du canal. En effet, dans le système concret l'émetteur ne peut plus accéder à l'état du canal (c'est pourquoi nous introduisons la variable locale $channel_empty$). D'un autre côté, si l'émetteur sait quand le canal passe de l'état $empty$ à l'état $full$, puisque c'est lui qui envoie les messages, il ne peut pas savoir quand le canal est vidé par le récepteur. Il faut donc mettre en place un mécanisme d'acquittement que nous modélisons ici par la variable $ackn$ et l'événement ACK.

Le diagramme d'états/transitons de la Fig. 10.6 représente une exploration exhaustive du comportement de COMMUNICATION_R$_1$. Il peut être construit aussi bien avec l'animateur de l'atelier B qu'avec ProB. Remarquons que les états au niveau de ce diagramme illustrent uniquement les trois combinaisons de valeurs atteignables par les événements de la spécification. En effet, l'état :

$$\begin{array}{c} channel = empty \wedge \\ channel_empty = FALSE \wedge \\ ackn = FALSE \end{array}$$

est permis par l'invariant mais n'est pas pris en compte au niveau du diagramme d'états/transitions concret de la Fig. 10.6 car aucun événement ne permet de l'atteindre.

[35] Une comparaison entre animation valuée et animation contrainte est présentée dans (Py *et al.,* 2000).



```
REFINEMENT
    COMMUNICATION_R1

REFINES
    COMMUNICATION

VARIABLES
    channel,
    channel_empty,
    ackn

INVARIANT
    channel_empty ∈ BOOL ∧
    ackn ∈ BOOL ∧
    (channel_empty = TRUE ⇒ channel = empty) ∧
    (channel_empty = TRUE ⇒ ackn = FALSE) ∧
    (ackn = TRUE ⇒ channel = empty)

INITIALISATION
    channel := empty ||
    channel_empty := TRUE ||
    ackn := FALSE

EVENTS
  send =
    SELECT
      channel_empty = TRUE
    THEN
      channel := full ||
      channel_empty := FALSE
    END;
  receive =
    SELECT
      channel = full
    THEN
      channel := empty || ackn := TRUE
    END;
  ACK =
    SELECT
      ackn = TRUE
    THEN
      ackn := FALSE ||
      channel_empty := TRUE
    END
END
```

FIG. 10.5 – Raffinement du Système COMMUNICATION

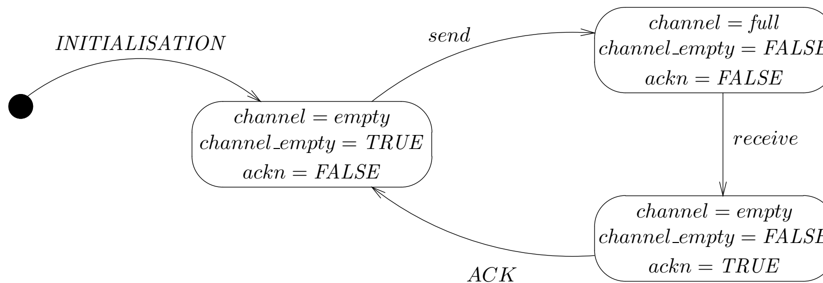


FIG. 10.6 – Diagramme d'états/transitions concret issu du raffinement du système COMMUNICATION

Une propriété intéressante du raffinement est que le comportement du système concret respecte le comportement du système abstrait. Cette propriété peut être représentée par le diagramme d'états/transitions hiérarchique suivant[36] :

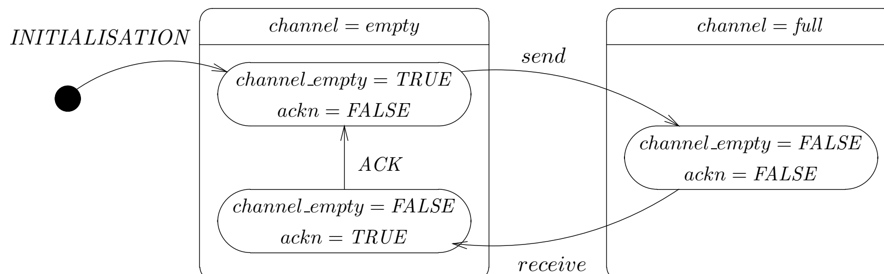


FIG. 10.7 – Diagramme d'états/transitions hiérarchique concret issu de COMMUNICATION_R1

[36] Nous n'aborderons pas dans la suite la démarche de construction d'une telle représentation structurée du raffinement. En effet, le but de ce chapitre est simplement l'exploration d'une palette de représentations comportementales pouvant être dérivées à partir de spécifications B.

L'intérêt d'une telle représentation est d'illustrer dans un même diagramme deux (ou plusieurs) automates correspondant chacun à un niveau de raffinement. Dans cette représentation, les états $empty$ et $full$ de la variable abstraite $channel$ sont vus comme des états composites. Ce faisant, nous pouvons observer que les événements $send$ et $receive$ permettent de transiter entre ces deux états comme pour le diagramme de la Fig. 10.4 issu du système abstrait.

10.3.3 Limites des vues concrètes

Nous avons montré à travers les diagrammes de cette section qu'une notation sous forme de machines d'états peut être intéressante pour représenter le comportement d'une spécification B. Nous avons également montré qu'une représentation structurée du raffinement peut aussi être utilisée à des fins de documentation. Ces représentations concrètes ont été construites grâce à l'utilisation d'outils d'animation (plus précisément ProB et l'animateur de l'atelier B). Ces derniers permettent de calculer les valuations atteignables des variables d'états après l'activation d'un événement dans un système B (ou l'appel d'une opération en B opérationnel). La construction de ces diagrammes a été aisée pour plusieurs raisons :

- l'espace d'états du système COMMUNICATION et son raffinement reste assez réduit (2 à 4 états),
- le nombre d'événements se limite à 3 ($send$, $receive$ et ACK),
- les spécifications sont déterministes.

Cependant, le succès des spécifications B est étroitement lié à leur capacité à modéliser des états complexes avec des structures de données abstraites (ensembles abstraits, relations et fonctions) pouvant résulter en un espace d'états immense (voire infini). Ces états complexes sont souvent associés à des comportements non-déterministes, et ce, précisément au niveau des premiers modèles (*i.e.* les plus abstraits) d'un développement en B construit par raffinements. Dans un tel contexte, les diagrammes de comportements concrets ne sont pas les mieux adaptés pour répondre à un besoin de documentation graphique des spécifications. C'est pourquoi nous présentons une technique d'abstraction de graphes d'accessibilité produisant des vues comportementales plus lisibles. Un exemple de diagramme d'états/transitions abstrait produit à partir du graphe d'états concret de la Fig. 10.6 est présenté au niveau de la figure ci-dessous :

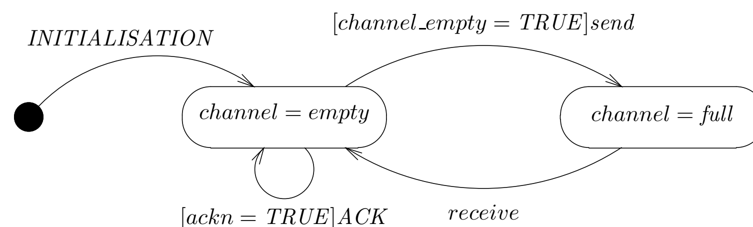


FIG. 10.8 – Diagramme d'états/transitions abstrait issu de COMMUNICATION_R$_1$

Dans ce diagramme, nous nous focalisons uniquement sur la variable $channel$, initialement définie au niveau du système B abstrait. Notons que les gardes des transitions, à ce niveau, correspondent aux gardes des événements de la spécification COMMUNICATION_R$_1$. Nous pouvons observer à partir de ce diagramme et celui de la Fig. 10.4 que, d'une part, le raffinement introduit l'événement ACK et que, d'autre part, cet événement n'a pas d'effets sur l'état du canal. D'un autre côté nous remarquons que l'acquittement n'est réalisé que si le canal est vide.

10.4 Vues comportementales abstraites

Dans cette section, nous allons présenter une palette de vues abstraites extraites d'une spécification plus complexe que celle du canal de communication et dans laquelle l'aspect comportemental est prépondérant. Il s'agit d'une spécification d'un gestionnaire de processus[37] comportant des ensembles abstraits, un espace d'états infini et des opérations non-déterministes. Les vues que nous allons présenter dans cette section ont pour objectif de montrer que chaque abstraction identifie une perspective particulière susceptible d'être utile pour les intervenants dans un projet B.

10.4.1 Le gestionnaire de processus

La spécification ci-dessous modélise un gestionnaire de processus simplifié et a pour but de gérer l'allocation d'une ressource unique (il s'agit ici d'un processeur) à un ensemble de processus. L'allocation de la ressource doit s'effectuer sans violation du principe d'exclusion mutuelle où l'exécution d'un processus exclut l'exécution concurrente d'autres processus ($\text{card}(active) \leq 1$).

```
MACHINE
    SCHEDULER
SETS
    Process
VARIABLES
    active,
    ready,
    waiting
INVARIANT
    active ⊆ Process ∧
    ready ⊆ Process ∧
    waiting ⊆ Process ∧
    card(active) ≤ 1 ∧
    (ready ∩ waiting = ∅ ) ∧
    (active ∩ waiting = ∅ ) ∧
    (active ∩ ready = ∅ ) ∧
    ((active = ∅ ) ⇒ (ready = ∅ ))
INITIALISATION
    active, ready, waiting := ∅ , ∅ , ∅

OPERATIONS

    NEW(pp) =
        PRE
            pp ∈ Process ∧
            pp ∉ (ready ∪ waiting ∪ active)
        THEN
            waiting := waiting ∪ {pp}
        END;

    READY(rr) =
        PRE
            rr ∈ Process ∧ rr ∈ waiting
        THEN
            waiting := (waiting - {rr}) ||
            IF active = ∅ THEN
                active := {rr}
            ELSE
                ready := ready ∪ {rr}
            END
        END;

    SWAP =
        PRE
            active ≠ ∅
        THEN
            waiting := waiting ∪ active ||
            IF ready = ∅ THEN
                active := ∅
            ELSE
                ANY pp WHERE
                    pp ∈ Process ∧ pp ∈ ready
                THEN
                    active := {pp} ||
                    ready := ready - {pp}
                END
            END
        END
END
```

FIG. 10.9 – Spécification B du gestionnaire de processus

Au niveau de cette spécification, l'ensemble de tous les processus est modélisé par l'ensemble $Process$. Les sous-ensembles $waiting$, $ready$ et $active$ de l'ensemble $Process$ modélisent les différents états d'un processus. En effet, un processus p peut être dans l'un des états suivants :

[37] La spécification du SCHEDULER et sa description est extraite de (Dick *et al.,* 1993, Peureux, 2002).

– "En attente" (*i.e.* $p \in waiting$), c'est-à-dire présent dans le système mais non candidat à l'attribution de la ressource,
– "Prêt" (*i.e.* $p \in ready$), c'est-à-dire s'apprête à recevoir la ressource,
– "Actif" (*i.e.* $p \in active$), c'est-à-dire possesseur de la ressource.

La gestion de l'ensemble des processus s'effectue grâce aux trois opérations *NEW*, *READY* et *SWAP* qui réalisent précisément les tâches suivantes :

– *NEW* : crée un nouveau processus et met son état à $waiting$ (en attente) ;
– *READY* : active un processus en attente s'il n'existe aucun processus actif, sinon elle met le processus à l'état prêt ;
– *SWAP* : retire la ressource au processus actif, le met en attente et alloue la ressource à l'un des processus prêts.

Les propriétés invariantes de la machine SCHEDULER désignent ce qui suit :

(i) Un processus ne peut être que dans un seul état à la fois,
(ii) Il existe au plus un seul processus actif,
(iii) Si un processus est à l'état prêt alors il existe un processus actif.

Bien que cette spécification reste relativement simple, elle présente les difficultés suivantes :

– Un espace d'états non borné : vu que l'ensemble $Process$ est indéfini. En conséquence, les variables $waiting$ et $ready$, ainsi que les paramètres possibles des opérations *NEW* et *READY* ne sont pas limités.
– Structure de données complexe : car les variables $waiting$ et $ready$ sont typés en tant qu'ensembles. Elles présentent donc une structure de données abstraite.
– Non-déterminisme : provient précisément de l'opération *SWAP* qui active de manière non-déterministe un des processus prêts.

10.4.2 Graphes d'états concrets et explosion d'états

Le graphe d'états concret produit à partir de la spécification du gestionnaire de processus dépend du nombre de processus introduits dans le système. En effet, si nous définissons $Process = \{p_1, p_2\}$, nous obtenons 10 états concrets accessibles (voir Fig. 10.10) alors que l'ajout d'un troisième processus produit 35 états accessibles. Aussi, bien que nous introduisions un nombre très réduit de processus, les graphes concrets produits sont-ils complexes et difficiles à comprendre. Par ailleurs, nous remarquons que les transitions du diagramme de la Fig. 10.10 sont toutes des transitions déterministes. Ceci provient du fait que le non-déterminisme n'apparaît que s'il existe au moins deux processus prêts alors que l'introduction de deux processus seulement produit des états concrets où au plus un seul processus est prêt à la fois.

En vue d'aborder ces deux aspects, à savoir l'explosion de la taille des diagrammes concrets et le non-déterminisme, nous allons nous servir d'autres prédicats d'états que nous appelons prédicats d'états abstraits. Ces derniers ne portent pas sur des valuations particulières des variables de la spécification mais présentent des sous-ensembles des espaces d'état associés à chacune de ces variables.

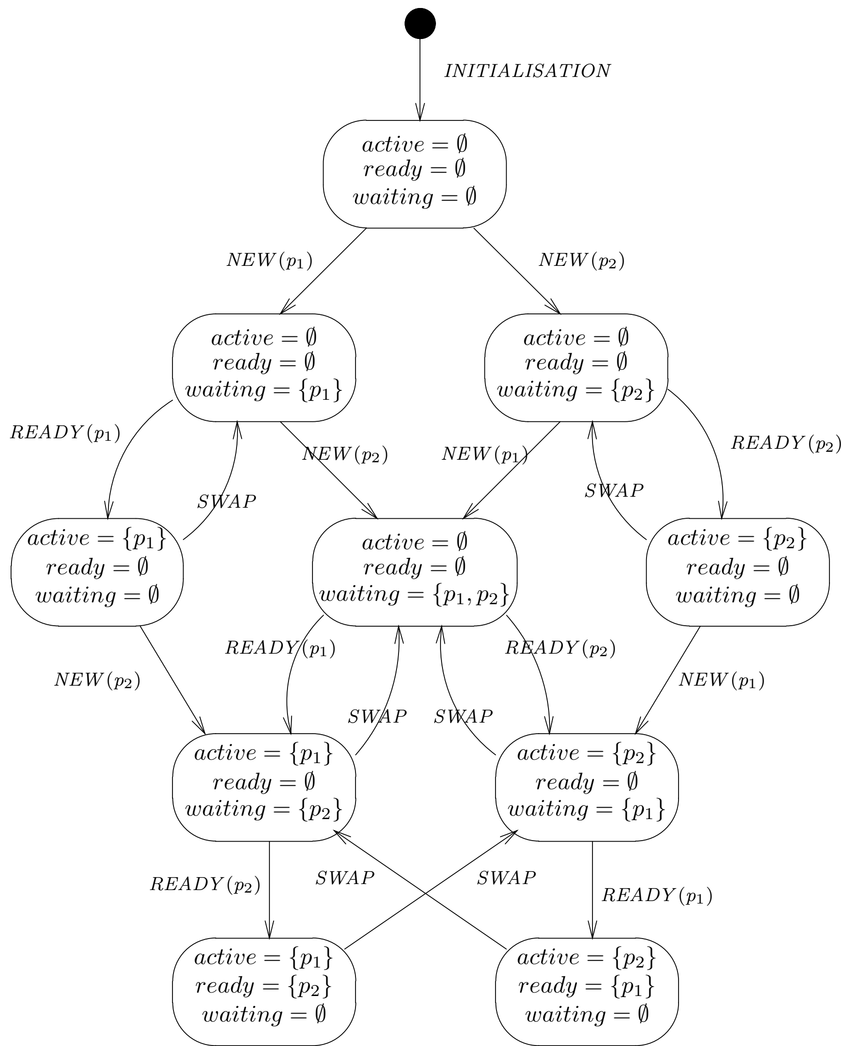


FIG. 10.10 – Graphe d'états concret produit pour $Process = \{p_1, p_2\}$

10.4.3 Focalisation sur une variable particulière

La Fig. 10.11 porte particulièrement sur l'évolution de la variable $active$ pour n'importe quel nombre de processus dans le système. Ce diagramme illustre une paire d'états abstraits dénotant le caractère vide/non-vide de l'ensemble des processus actifs. Ce faisant, notons que l'état caractérisé par $active \neq \emptyset$ correspond à une infinité d'états concrets car l'ensemble $active$ peut contenir n'importe quel élément de l'ensemble abstrait $Process$.

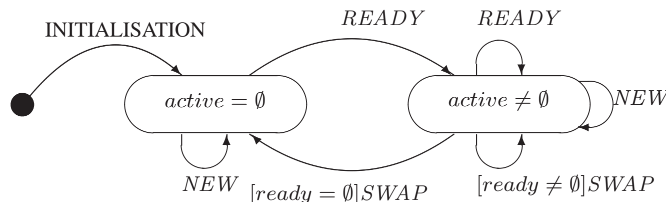


FIG. 10.11 – Diagramme d'états/transition abstrait associé à la variable $active$

Dans ce diagramme, les transitions *NEW* et *READY* ne sont pas paramétrées pour des raisons de clarté. Nous représentons donc par une seule transition l'ensemble des transitions correspondant aux diverses valeurs des paramètres. Ces derniers peuvent être considérés implicitement comme des variables d'états. La sémantique sous-jacente dénote, alors, le fait que l'appel de chacune des opérations *NEW* et *READY* avec n'importe quel paramètre satisfaisant leurs pré-conditions amène à l'état cible de la transition correspondante. Quant à l'opération *SWAP*, elle est à l'origine des deux transitions dont l'état source est $active \neq \emptyset$. Une condition est associée à chaque transition *SWAP* en vue d'exprimer que l'activation d'une transition *SWAP* à partir de l'état $active \neq \emptyset$ est déterministe. D'un point de vue documentation, ce diagramme illustre clairement certaines propriétés de la machine SCHEDULER :

– Toutes les transitions étiquetées par *READY* aboutissent à l'état $active \neq \emptyset$. Ceci correspond à l'exigence suivante d'un gestionnaire de processus : un processus demandant son activation n'est bloqué que s'il existe un autre processus actif.
– *SWAP* n'est déclenchée qu'à partir de l'état $active \neq \emptyset$. En effet, l'appel de *SWAP* a pour unique objectif de désactiver le processus actif.
– La seule manière de regagner l'état $active = \emptyset$ à partir de l'état $active \neq \emptyset$ est d'exécuter un nombre de fois suffisant l'opération *SWAP*.
– *NEW* n'a aucun effet sur l'ensemble des processus actifs car elle correspond à une transition réflexive au niveau de chacun des états $active = \emptyset$ et $active \neq \emptyset$.

10.4.4 Mise en évidence sous forme concurrente de toutes les variables d'état

Des diagrammes spécifiques à chacune des variables d'états tels que celui de la Fig. 10.11 peuvent être construits également pour les variables $ready$ et $waiting$ et illustrer ainsi leur caractère vide/non-vide. Au niveau de la Fig. 10.12 nous regroupons tous ces diagrammes au sein d'un même diagramme d'états/transitions dans le but de représenter une évolution concurrente de ces trois variables d'état. Nous remarquons que ces évolutions ne sont pas indépendantes les unes des autres. En effet, ce qui était une condition de la transition *SWAP* au niveau du diagramme d'états/transitions de la variable $active$, est devenu une dépendance entre états au niveau du diagramme concurrent.

Remarquons à partir de ce diagramme concurrent, qu'à l'initialisation, les trois ensembles de processus sont vides. Les états $waiting = \emptyset$, $ready = \emptyset$ et $active = \emptyset$ sont alors les points d'entrée de chaque automate. À partir de ces trois états initiaux, seule la transition *NEW* est activable. En effet, les transitions sortantes de ces états initiaux à savoir *SWAP* et *READY* sont conditionnées respectivement par $active \neq \emptyset$ et $waiting \neq \emptyset$. D'un autre côté, l'effet de la transition *NEW* ne porte que sur la variable $waiting$ car dès qu'un processus est introduit dans le système son état est mis en attente.

Notons que toutes les transitions *READY* sont gardées par $waiting \neq \emptyset$. En effet, il s'agit d'une condition nécessaire pour pouvoir sélectionner un processus en attente ($pp \in waiting$). L'effet de cette transition sur la variable $ready$, lorsqu'il n'y a aucun processus prêt, dépend également de l'état de $active$. En effet, si aucun processus n'est actif, l'appel de l'opération *READY*(pp) permet directement d'activer le processus pp. Dans le cas contraire, pp est mis à l'état prêt en attendant son activation.

Ce diagramme concurrent peut être déployé en un seul diagramme d'états, et ce, en réalisant le produit cartésien des états tout en prenant compte des gardes des transitions. Ceci résulte dans le diagramme de la Fig. 10.13.

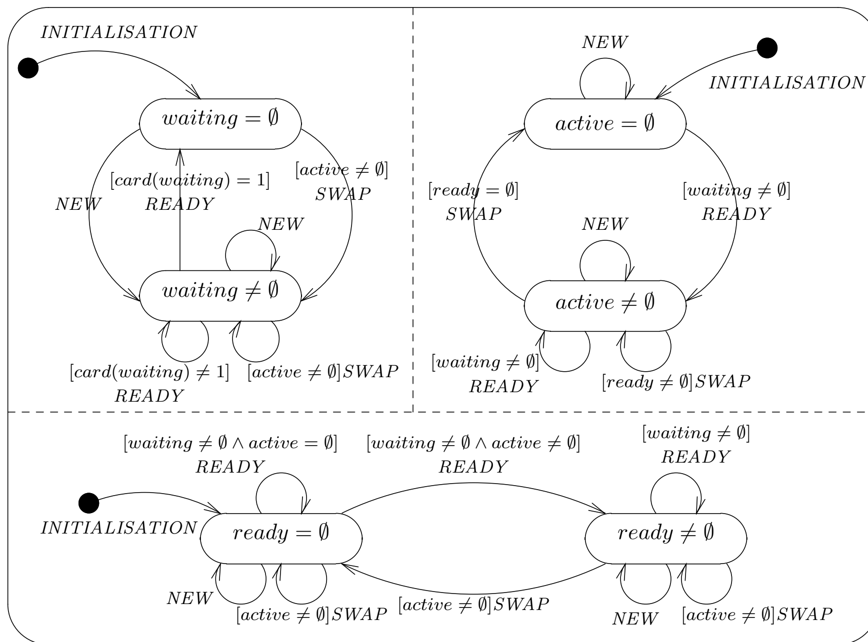


FIG. 10.12 – Diagramme d'états/transitions concurrent portant sur les variables $active$, $ready$ et $waiting$

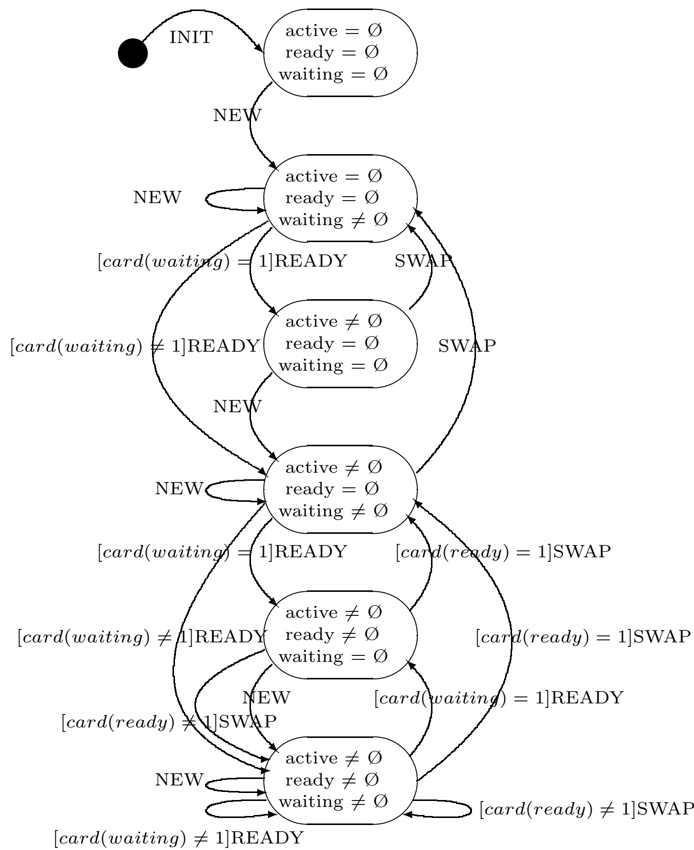


FIG. 10.13 – Diagramme d'états/transitions déplié portant sur les variables $active$, $ready$ et $waiting$

10.4.5 Représentation graphique d'une propriété invariante

Bien que le produit cartésien des différents états des automates du diagramme concurrent produise huit états distincts, le diagramme déplié ne porte que sur six de ces huit états. Les deux autres états ne sont ni atteignables ni permis par l'invariant :

$$active = \emptyset \Rightarrow ready = \emptyset$$

Nous avons présenté, d'une part, le diagramme de la Fig. 10.12 où il s'agit de se focaliser sur le comportement de chacune des variables séparément, et d'autre part, le diagramme de la Fig. 10.13 où il s'agit de présenter la combinaison du comportement de toutes ces variables. Cependant, la propriété invariante ci-dessus fait référence uniquement aux variable $active$ et $ready$. Il nous paraît alors opportun de construire un diagramme permettant de se focaliser uniquement sur ces deux variables.

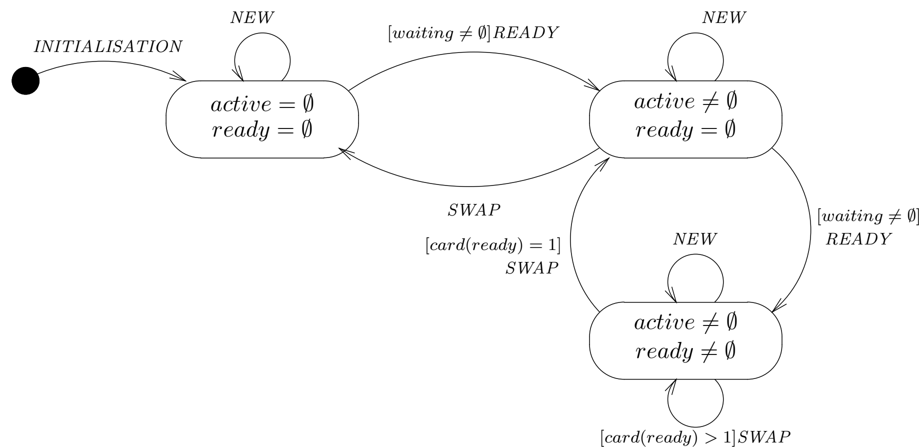


FIG. 10.14 – Diagramme d'états/transitions associé à l'invariant : $active = \emptyset \Rightarrow ready = \emptyset$

L'intérêt de cette représentation est qu'elle met l'accent sur une évolution synchronisée des deux variables $active$ et $ready$ et permet de montrer comment la propriété invariante précédente est préservée durant cette évolution. Notons également que ce diagramme comporte uniquement trois états étant donné que le quatrième (*i.e.* $active = \emptyset \wedge ready \neq \emptyset$) n'est pas permis par cet invariant.

10.4.6 Représentation du comportement d'un processus

Les diagrammes que nous avons présentés jusqu'ici mettent l'accent, plus particulièrement, sur les variables d'état. Au niveau de ces diagrammes nous nous sommes attaché précisément à une représentation abstraite du mécanisme interne à la spécification du gestionnaire de processus.

Cependant, dans les chapitres précédents nous avons montré comment une spécification B peut être représentée par une ou plusieurs vues structurelles sous forme de diagrammes de classes UML. De ce fait, nous pensons que l'élaboration de vues comportementales en relation avec les vues structurelles peut conduire à une documentation plus complète.

Le diagramme de classes de la Fig. 10.15 résulte de l'application de notre technique de construction de diagrammes de classes sur la machine SCHEDULER. Dans ce diagramme, une seule classe représentant

l'ensemble des processus a été identifiée. Les attributs booléens de cette classe dénotent les états de chacune de ses instances. En effet, une instance p_i de la classe $Process$ dont l'attribut $waiting$ est égal à $true$ représente un processus particulier identifié par p_i et dont l'état est : "en attente". Nous formalisons ceci par le prédicat d'état $p_i \in waiting$. Finalement, les méthodes de cette classe permettent d'agir sur ces attributs booléens et réaliser ainsi l'évolution des états de chaque processus.

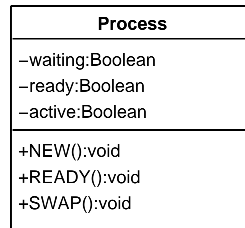


FIG. 10.15 – Diagramme de classes issu de la machine SCHEDULER

Représenter le comportement de la classe $Process$ ne se ramène pas à l'étude des variables d'états de la machine SCHEDULER sur leur plage de valeurs, mais aux états particuliers de chacun des processus mis en jeu dans le système. Nous adoptons alors à ce niveau une approche différente, où il s'agit de représenter le cycle de vie d'un processus p_i étant donnée la spécification B du gestionnaire de processus :

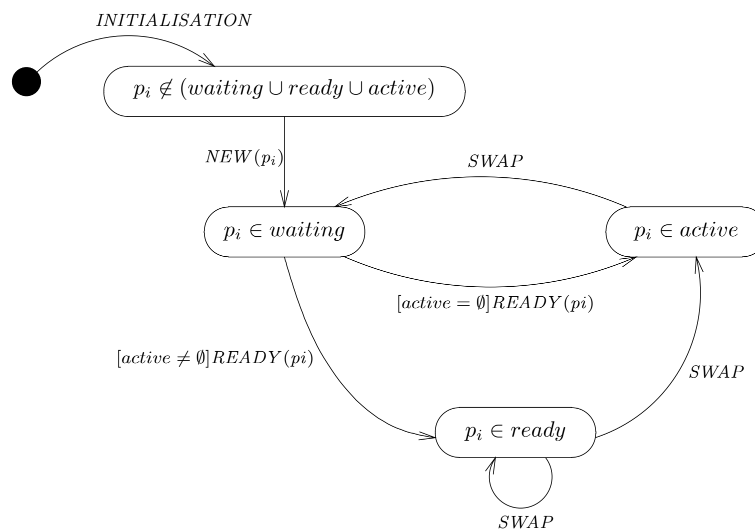


FIG. 10.16 – Automate du cycle de vie d'un processus p_i

Au niveau de ce diagramme, quatre états sont identifiés :

- L'état initial où p_i n'appartient à aucune variable d'état représente le cas où tous les attributs booléens de l'instance p_i de la classe $Process$ valent $false$;
- Les trois autres états définis par l'appartenance de p_i à $waiting$, $ready$ ou $active$ montrent les cas où l'un des attributs de l'instance p_i vaut $true$.

Contrairement aux diagrammes précédents où tous les appels possibles d'opérations sont pris en compte à partir de chaque état atteignable, ce diagramme ne s'attache qu'aux opérations aboutissant au changement de l'état du processus p_i. Autrement dit, des opérations telles que *NEW*(p_j) ou *READY*(p_j),

avec $i \neq j$, ne sont pas considérées. Par ailleurs, au niveau de l'état $p_i \in ready$, *SWAP* présente un comportement non-déterministe justifiant l'existence de deux transitions avec une même étiquette. Notons que les traces issues de ce diagramme ne correspondent pas strictement aux traces de la spécification car certaines opérations ne sont pas illustrées alors qu'elles agissent sur les variables d'état de la spécification B. Ces opérations sont donc omises des traces issues du diagramme de la Fig. 10.16 alors qu'elles influencent le cycle de vie d'un processus particulier p_i.

Cependant, l'intérêt de cette représentation est justifié par le fait qu'elle permet de mettre en évidence graphiquement les propriétés suivantes de la spécification du gestionnaire de processus :

- *Absence de blocage* : après avoir été activé un processus ne doit pas stopper le système. En effet, la transition *SWAP* peut être déclenchée à partir de l'état $p_i \in active$; ce qui remet le processus actif à l'état $waiting$ et permet l'activation d'autres processus. Dans le but de vérifier cette propriété, il suffit de vérifier que chaque état est source d'au moins une transition.
- *Absence de famine* : si un processus est prêt alors il existe une exécution dans laquelle ce processus obtient la ressource. Cette propriété n'est pas vérifiée à cause du non-déterminisme de l'opération *SWAP*. En effet, nous remarquons à partir du diagramme du cycle de vie d'un processus que *SWAP* peut boucler indéfiniment dans l'état $p_i \in ready$.

10.5 Bilan

Dans ce chapitre nous avons présenté une revue de représentations graphiques issues de spécifications B sous forme de diagrammes d'états/transitions. Ces représentations permettent d'appréhender une machine B (ou un système B) selon différentes vues grandement liées aux invariants associés aux états (concrets ou abstraits).

L'intérêt porté à cette diversité dans les représentations est justifié par la nécessité d'expliciter différents points de vue potentiellement utiles pour les intervenants dans un projet B. D'un côté, les vues **internes** telles que celles s'attachant à la mise en évidence des variables d'états (*e.g.* Figs 10.11 − 10.14) permettent de représenter la construction du système, de comprendre son mécanisme interne et de vérifier ainsi si certains objectifs visés sont atteints. Ces diagrammes ont pour but d'aider les analystes lors de la phase de spécification. D'un autre côté, les vues **externes** telles que celles du cycle de vie d'un processus (*e.g.* Fig. 10.16) sont identifiées pour définir les usages du système. Ces représentations externes sont principalement destinées à des fins de communication car elles permettent d'exprimer les besoins des utilisateurs sur la base d'une spécification existante. Ces deux catégories complémentaires de vues délimitent, d'une part, ce que fait le système et, d'autre part, comment ce système est réalisé.

Nous avons également présenté un exemple de diagrammes permettant d'explorer la hiérarchie raffiné/raffinant (Fig. 10.7) et dont le principal objectif est de visualiser une structuration hiérarchisée des niveaux de raffinement.

Finalement, sur la base des différentes observations étayant ces différents diagrammes nous avons montré l'utilité de ces derniers pour une meilleure compréhension des spécifications. De plus, lesdits diagrammes nous ont permis d'expliciter graphiquement certaines propriétés intrinsèques aux spécifications ce qui motive leur importance pour un objectif de documentation.

Chapitre 11

Techniques de dérivation de vues comportementales

« *A picture is worth a thousand words – isn't it ?*
And hence graphical representation is by its nature universally superior to text ? – isn't it ?
Why then isn't the anecdote itself expressed graphically ?
Perhaps anecdotes don't lead themselves to purely graphical representation. »

Mariam Petre
« *why looking isn't always seeing* » *Communication of the* ACM (Vol. 38, N°6)

Sommaire

11.1 Introduction **190**
11.2 Outils de construction de vues comportementales **190**
11.2.1 Animation basée sur les suites de test 190
11.2.2 Abstraction de graphes d'accessibilité 193
11.2.3 L'approche `GeneSyst` 196
11.2.4 Propriétés des transitions 200
11.3 Identification des états abstraits **200**
11.3.1 Recours aux données de la spécification 201
11.3.2 Recours aux besoins informels de l'utilisateur 202
11.3.3 Définition d'un catalogue de patrons d'états abstraits 203
11.4 Dérivation de diagrammes d'états/transitions sur la base des vues structurelles **206**
11.4.1 Formalisation 207
11.4.2 Prédicats d'états associés aux classes 208
11.4.3 Application et discussion 211
11.5 Bilan et conclusion **212**

11.1 Introduction

Partant du constat qu'une même spécification B peut être représentée par plusieurs diagrammes d'états/transitions, nous allons nous attacher, dans ce chapitre, aux outils théoriques et effectifs permettant de construire ces vues comportementales. Nous avons présenté, au niveau du chapitre précédent, trois outils d'animation de spécifications B qui se limitent à la production de vues comportementales concrètes. Nous avons alors mis l'accent sur des graphes d'états particuliers dits graphes d'accessibilité et où les états sont des combinaisons de valeurs atteignables par une ou plusieurs séquences d'opérations (ou d'événements lorsqu'il s'agit de systèmes B). Notons que plusieurs travaux de recherche se basent sur ces graphes d'accessibilité pour vérifier des propriétés d'un système. Par exemple, vérifier que le graphe est un modèle d'une certaine formule logique − on parle alors de *model-checking* − ou bien calculer différentes relations d'équivalence − par exemple la bi-simulation − entre graphes d'accessibilité et indiquer des comportements similaires. Ces méthodes de vérification sont maintenant bien connues dans le cas où le graphe d'accessibilité est fini.

Dans le cadre de notre travail, nous nous situons dans un contexte de *vérification externe* (Gaudel, 1991) de spécifications impliquant amplement l'utilisateur. Nous utilisons alors ces graphes concrets pour produire différentes abstractions visant, d'une part, à exprimer différentes facettes du système, et d'autre part, à définir une base semi-formelle pour l'étude de la conformité des spécifications par rapport aux besoins informels de l'utilisateur.

Ce faisant, la technique d'abstraction de graphes d'accessibilité que nous proposons dans ce chapitre, vient compléter l'outil **`GeneSyst`** (Bert *et al.,* 2005) qui permet, grâce à une technique de preuve, d'exhiber des systèmes de transitions étiquetés à partir de spécifications B événementielles.

Nous évoquons alors dans ce chapitre un ensemble de techniques conduisant à l'automatisation du processus de dérivation des vues comportementales à partir de spécifications B, ainsi qu'un appui méthodologique permettant d'aider le développeur lors de l'identification des abstractions d'états.

11.2 Outils de construction de vues comportementales

11.2.1 Animation basée sur les suites de test

La difficulté intrinsèque à l'animation des spécifications est l'identification d'une séquence finie d'interactions avec le système B aboutissant à un ensemble significatif de comportements. Ceci est un problème assez connu dans le domaine du test de spécifications et plus précisément lors de l'élaboration des séquences[38] de test (Behnia, 2000, Peureux, 2002, Petit, 2003). Nous n'allons pas présenter les fondements théoriques des techniques de test de spécifications B, ni les problèmes qui surgissent de ce domaine de recherche ; nous nous contentons tout simplement de mentionner le fait que ces travaux s'attachent en grande partie à garantir la qualité des tests et que cette qualité dépend de la pertinence du choix des données mises en jeu au niveau des séquences de test.

De ce fait, il nous paraît judicieux de représenter de manière graphique le résultat de l'animation d'un ensemble de séquences de test jugées significatives pour le test de spécifications B. Par exemple, pour le cas de la machine SCHEDULER, plusieurs séquences de test ont été proposées dans la littérature

[38] Appelées aussi suites de test ou cas de test.

telles que celles calculées par l'outil **CASTING** (Aertryck *et al.,* 1997) et celles proposées par J. Dick et A. Faivre dans (Dick *et al.,* 1993). Dans ce qui suit, nous allons nous baser sur ces deux références en vue de montrer qu'une technique d'animation basée sur des suites de test peut s'avérer efficace pour la dérivation de diagrammes de comportements intéressants.

Le tableau 11.1 présente la suite de tests extraite de (Aertryck *et al.,* 1997) où principalement cinq séquences de test sont présentées.

Test seq 1	Test seq 2	Test seq 3	Test seq 4	Test seq 5
$INIT_1$	$INIT_1$	$INIT_1$	$INIT_1$	$INIT_1$
$NEW_1(p_1)$	$NEW_1(p_1)$	$NEW_1(p_1)$	$NEW_1(p_1)$	$NEW_1(p_1)$
$NEW_2(p_2)$	$NEW_2(p_2)$	$NEW_2(p_2)$	$NEW_2(p_2)$	$READY_4(p_1)$
$READY_1(p_1)$	$READY_1(p_1)$	$READY_1(p_1)$	$READY_1(p_1)$	$SWAP_4$
$READY_2(p_2)$	$READY_2(p_2)$	$READY_2(p_2)$	$SWAP_3$	
$NEW_3(p_3)$	$NEW_3(p_3)$	$SWAP_2$		
$READY_3(p_3)$	$SWAP_1$			

TAB. 11.1 – Suite de tests produite par **CASTING**

Nous avons réalisé l'animation de ces cinq séquences de test aussi bien par ProB que par l'animateur de l'atelier B. Nous représentons au sein d'une même machine à états (Fig. 11.1) une visualisation graphique du résultat de cette animation. Au niveau de ce diagramme, chaque état correspond à une valuation particulière du triplet $(active, ready, waiting)$.

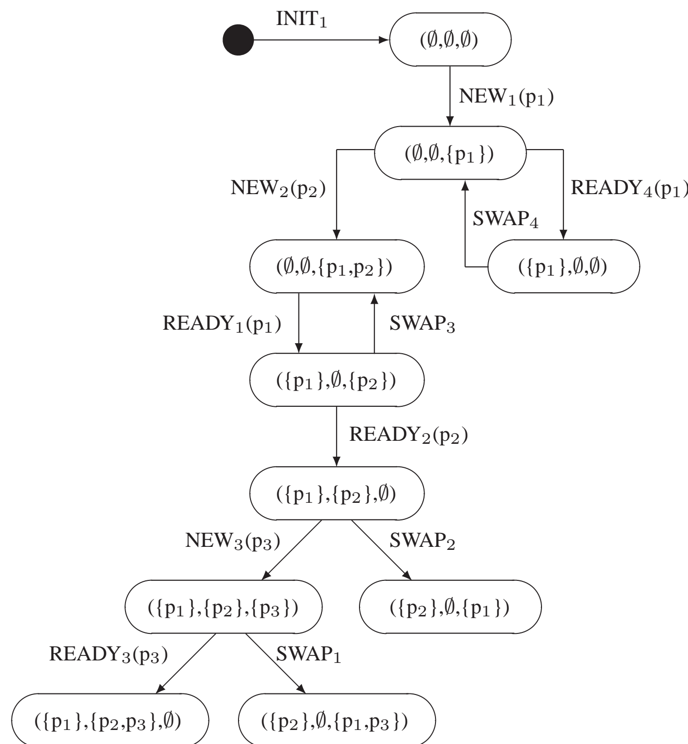


FIG. 11.1 – Représentation graphique de la suite de tests produite par **CASTING** et des états résultants

Ce diagramme n'est autre qu'un sous graphe du graphe d'accessibilité (Fig. 11.3) produit pour un système contenant trois processus ($Process = \{p_1, p_2, p_3\}$). Néanmoins, il est important de noter que ce

diagramme présente une structure arborescente due au fait que les suites de test tendent vers l'exploration d'un nombre maximum d'états différents. Cette même constatation résulte également de la suite de tests proposée par J. Dick et A. Faivre (Dick *et al.,* 1993) et qui consiste en une séquence unique de tests. Toutefois, le résultat de la visualisation de l'animation de cette séquence est un diagramme linéaire étant donné que les auteurs ont exclu l'exploration d'un même état plusieurs fois. Nous ne présentons pas graphiquement le résultat de cette animation car des diagrammes tels que celui de la Fig. 11.1 ne montrent pas un aperçu pertinent d'un point de vue documentation.

En revanche, il paraît intéressant de représenter ces suites de test en utilisant une abstraction particulière parmi celles présentées au niveau du chapitre 10. Nous présentons alors les diagrammes de la Fig. 11.2 où il s'agit de diagrammes d'états/transitions abstraits construits à partir de l'animation de ces deux suites de tests. Notons que l'abstraction utilisée au niveau de ces diagrammes est analogue à celle proposée au niveau du diagramme déplié de la Fig. 10.13 (page 184).

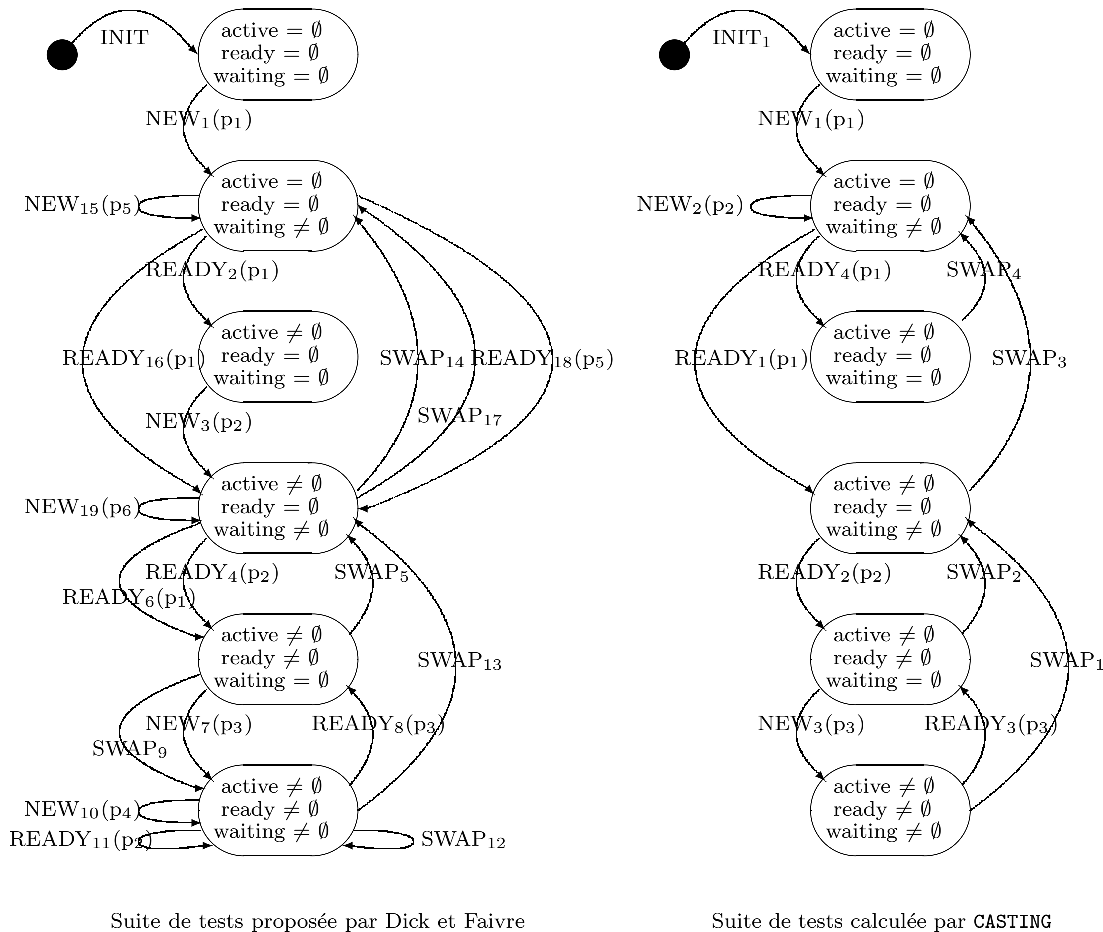


FIG. 11.2 – Représentation graphique abstraite de deux suites de tests

La partie gauche de la Fig. 11.2 présente le diagramme abstrait construit à partir de la suite de tests proposée par J. Dick et A. Faivre. En comparaison avec le diagramme de la Fig. 10.13 (page 184), nous remarquons que cette abstraction couvre la plupart des transitions. Ce résultat provient du fait que la

technique de génération des suites de tests proposée par J. Dick et A. Faivre se base sur le parcours d'un graphe d'états abstrait fondé sur les mêmes abstractions que notre diagramme déplié. Cependant, deux transitions ne sont pas couvertes par cette abstraction, à savoir les transitions *SWAP* et *READY* suivantes :

$$\begin{pmatrix} active \neq \emptyset \\ ready = \emptyset \\ waiting = \emptyset \end{pmatrix} \xrightarrow{\textit{SWAP}} \begin{pmatrix} active = \emptyset \\ ready = \emptyset \\ waiting \neq \emptyset \end{pmatrix}$$

$$\begin{pmatrix} active \neq \emptyset \\ ready = \emptyset \\ waiting \neq \emptyset \end{pmatrix} \xrightarrow{\textit{READY}} \begin{pmatrix} active \neq \emptyset \\ ready \neq \emptyset \\ waiting \neq \emptyset \end{pmatrix}$$

Par contre, le diagramme de la partie droite de la Fig. 11.2 est issu d'une suite de tests construite sur la base d'une technique différente. Nous constatons que bien que ce diagramme couvre moins de transitions, il inclut une des deux transitions manquantes : il s'agit ici de $SWAP_4$.

En conséquence, un couplage des outils d'animation de spécifications et des techniques de génération de tests peut être très utile pour découvrir et distinguer les transitions entre états abstraits. Vu que plusieurs techniques de génération de tests sont basées sur des abstractions semblables, elles peuvent alors fournir une aide intéressante lors de l'identification des états abstraits. Réussir une meilleure couverture des transitions résultera alors de l'utilisation des mêmes abstractions d'états aussi bien au niveau de la représentation graphique qu'au niveau de la génération de tests.

11.2.2 Abstraction de graphes d'accessibilité

Nous proposons, à ce niveau, une démarche alternative permettant de construire les diagrammes présentés au niveau du chapitre précédent. Il s'agit d'appliquer une technique d'abstraction en partant d'un diagramme d'états concret (voir Fig 10.10 page 182 et Fig. 11.3). Une telle démarche passe donc par une étape de calcul de tous les états atteignables (ou concrets) de la spécification B en explorant exhaustivement son espace d'états. Pour ce faire, nous limitons notre étude à des systèmes finis et nous nous servons de l'outil ProB (Leuschel *et al.,* 2003) qui produit automatiquement des graphes d'accessibilité par *model-checking*. Ensuite, et par une technique de preuve nous identifions pour chaque prédicat d'état abstrait, les états concrets qui le satisfont. Nous ramenons alors la dérivation de diagrammes d'états/transitions à un algorithme d'abstraction de graphes d'accessibilité.

Nous désignons par $G = (N, T)$ un diagramme d'états/transitions concret issu d'une spécification B, où N désigne l'ensemble des états concrets de G, et T l'ensemble des transitions entre ces états. Rappelons qu'un état concret S_v ($S_v \in N$) portant sur les variables d'état v telle que $v = \{v_1, \ldots, v_n\}$, est exprimé par une suite de conjonctions de prédicats d'égalité entre variables et valeurs :

$$S_v \mathrel{\widehat{=}} \bigwedge_{i=1}^{n} (v_i = val_j)$$

Où val_j est une valeur possible de v_i.

Soit S_v un état concret défini par le prédicat $P(v)$ et soit $S_{abstrait}$ un état abstrait défini par le prédicat d'état abstrait R (*e.g.* $active \neq \emptyset$), alors nous dirons que S_v satisfait $S_{abstrait}$ (noté $S_v \vdash S_{abstrait}$) si et seulement si : $P(v) \Rightarrow R$.

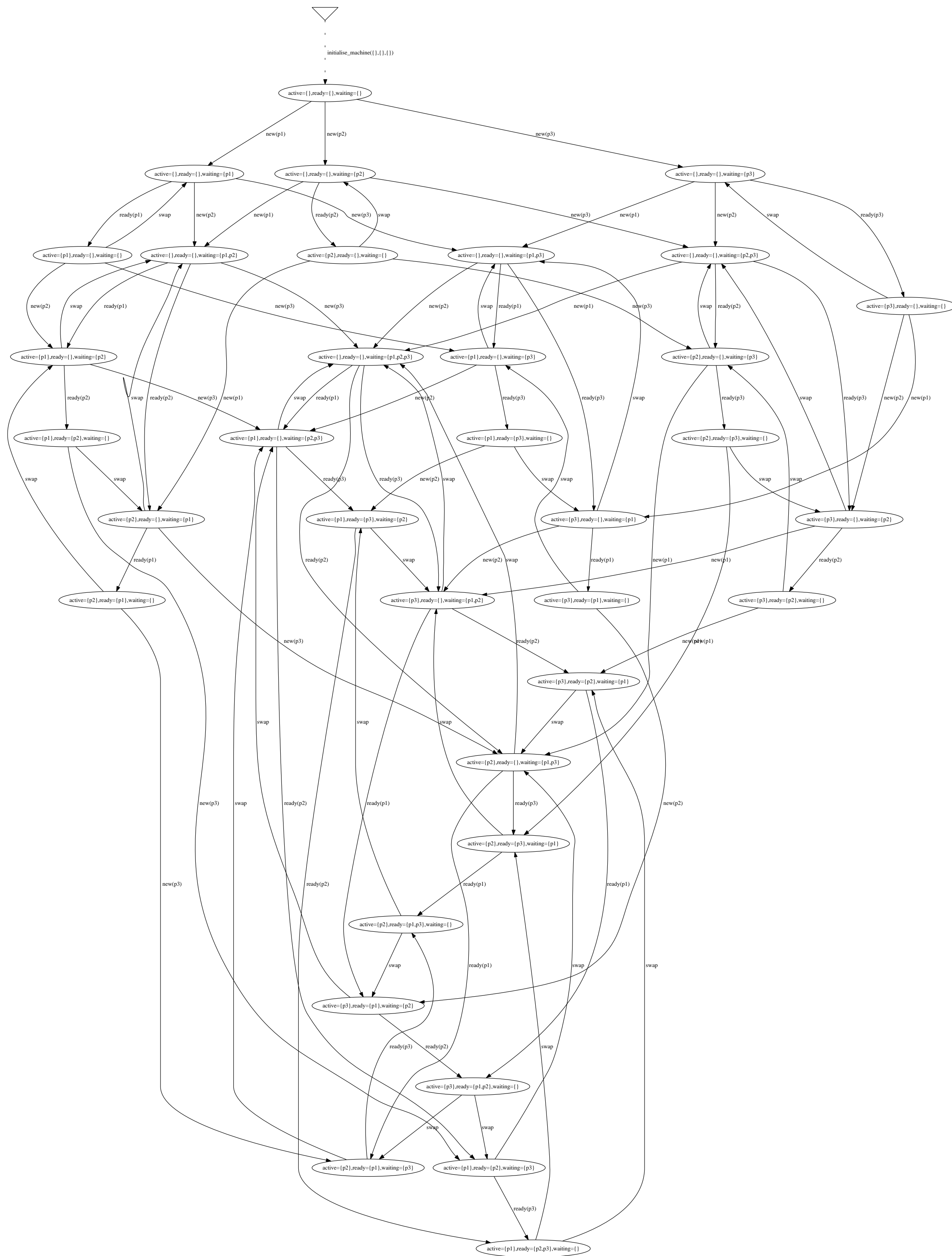


FIG. 11.3 – Graphe d'accessibilité produit par ProB étant donné $Process = \{p_1, p_2, p_3\}$

Notre algorithme d'abstraction que nous présentons ci-dessous, prend en entrée (*i*) un diagramme d'états/transitions concret $G = (N, T)$ et (*ii*) un ensemble d'états abstraits, et transforme G comme suit :

```
Pour chaque état abstrait S_abstrait faire
|     Ajouter S_abstrait dans N
|     Pour chaque état concret S_v tel que
|     |     S_v ∈ N − {S_abstrait} et S_v ⊢ S_abstrait
|     faire
|     |     Pour chaque état S tel que
|     |     |     S ∈ N − {S_v}
|     |     faire
|     |     |     Pour chaque transition t tel que
|     |     |     |     t ∈ T ∧ t = S --o--> S_v
|     |     |     faire
|     |     |     |     remplacer t par S --o--> S_abstrait ;
|     |     |     fin faire ;
|     |     |     Pour chaque transition t tel que
|     |     |     |     t ∈ T ∧ t = S_v --o--> S
|     |     |     faire
|     |     |     |     remplacer t par S_abstrait --o--> S ;
|     |     |     fin faire ;
|     |     fin faire ;
|     |     Transformer les transitions S_v --o--> S_v en S_abstrait --o--> S_abstrait ;
|     |     Supprimer S_v de N ;
|     fin faire ;
|     Éliminer les transitions redondantes ;
fin faire ;
```

Le résultat de cet algorithme est donc une représentation symbolique construite par un regroupement d'états concrets. Un état n'est plus une valuation particulière de variables mais plutôt un ensemble de valuations décrites par un prédicat dit prédicat d'état abstrait. Nous nous servons alors de deux outils pour automatiser la construction de ces diagrammes :

- l'outil ProB pour la production de diagrammes concrets (*e.g.* Figs. 10.10 et 11.3), et
- le prouveur de l'atelier B pour identifier les états concrets satisfaisant les prédicats d'abstraction

L'utilisation de l'atelier B à ce niveau est restreinte au prouveur `force 0`. En effet, vu que nous partons d'états entièrement valués, la preuve $P(v) \Rightarrow R$ est assez triviale. Cependant, l'explosion de la taille des diagrammes d'états concrets produits par ProB influence fortement cette technique car il en résulte une augmentation du nombre de preuves réalisées par l'atelier B. Les deux illustrations ci-dessous montrent les preuves générées par l'atelier B à partir des graphes concrets des figures 10.10 et 11.3, et ce, pour le prédicat d'état abstrait $active = \emptyset$. Notez que ce résultat passe par la construction de machines intermédiaires et donc les preuves que nous montrons ici incluent les preuves des invariants de typage. Nous remarquons que pour le cas d'un graphe d'accessibilité produit pour 2 processus (10 états concrets) l'outil identifie 6 états ne satisfaisant pas le prédicat $active = \emptyset$. Vu que les preuves sont triviales, l'échec de la preuve correspond au fait que le prédicat d'état abstrait est faux pour ces états concrets. Par contre dans le cadre d'un graphe produit pour 3 processus (35 états) l'outil identifie 27 états ne satisfaisant pas ce prédicat et réalise un total de 123 preuves.

Cette technique d'abstraction présente une solution automatique pour la construction de diagrammes d'états/transitions abstraits moyennant l'identification des prédicats d'abstraction. Elle permet donc de

réduire la taille des graphes d'états concrets jugés illisibles et d'en produire des diagrammes compréhensibles. Par exemple, étant donnés les deux prédicats d'états abstraits $active = \emptyset$ et $active \neq \emptyset$, notre technique construit automatiquement le diagramme de la Fig. 10.11 page 182 sans les gardes des transitions, et ce, aussi bien pour deux processus (Fig. 10.10) que pour trois processus (Fig. 11.3).

```
...
End of Proof
   State1               Proved 3       Unproved 0
   ...                  ...            ...
   State10              Proved 3       Unproved 1
TOTAL for State_SCHEDULER_0      Proved 33       Unproved 6
```

FIG. 11.4 – Preuves générées par l'atelier B pour $active = \emptyset$ et $Process = \{p_1, p_2\}$

```
...
End of Proof
   State1               Proved 3       Unproved 0
   ...                  ...            ...
   State35              Proved 3       Unproved 1
TOTAL for State_SCHEDULER_0      Proved 123       Unproved 27
```

FIG. 11.5 – Preuves générées par l'atelier B pour $active = \emptyset$ et $Process = \{p_1, p_2, p_3\}$

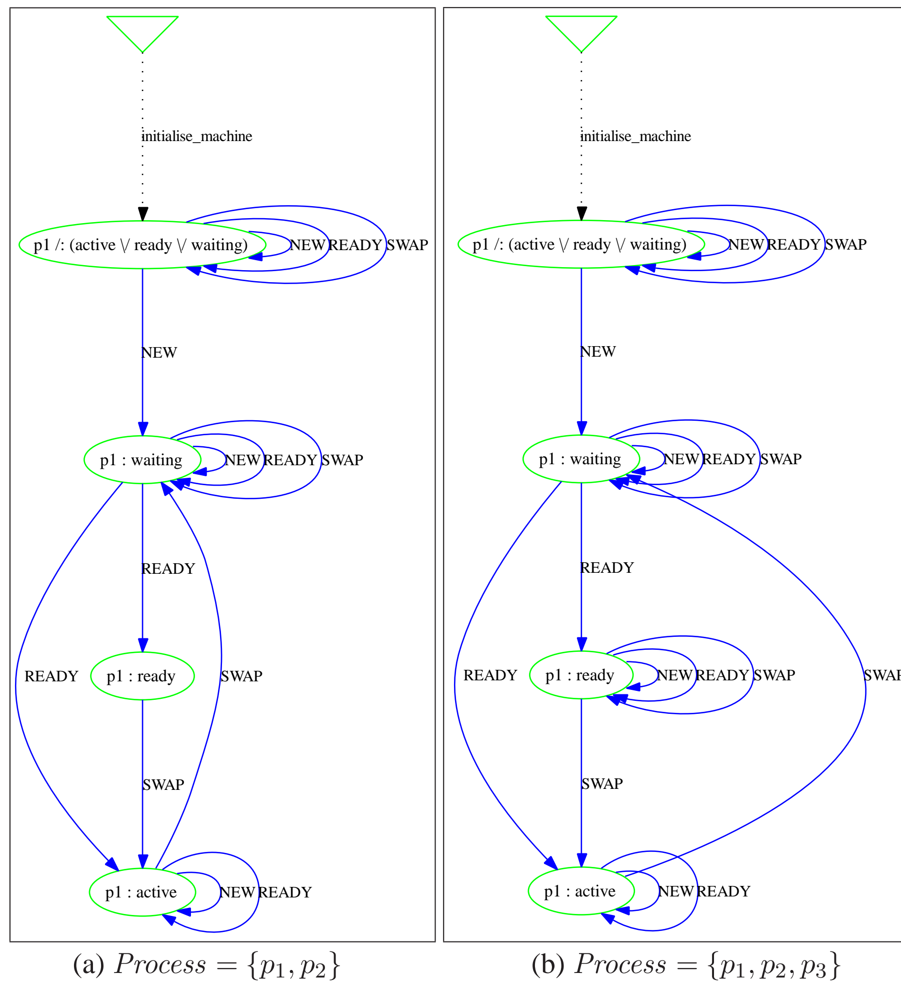
La principale limite de cette approche est qu'elle dépend fortement du diagramme d'états/transitions concret. En effet, si un comportement particulier n'apparaît pas au niveau du graphe concret, il ne sera pas inclus au niveau du diagramme abstrait (*e.g.* transitions réflexives sur l'état $p_1 \in ready$ au niveau de la Fig. 11.6).

La Fig. 11.6 présente les diagrammes abstraits produits par cette technique à partir des Figs. 10.10 et 11.3 et s'attachant au cycle de vie du processus p_1. Nous ne distinguons pas à ce niveau les paramètres en vue de traiter de manière uniforme les opérations. Nous observons que la différence entre ces deux diagrammes se situe au niveau de l'état $p_1 \in ready$ où les transitions réflexives *NEW*, *READY* et *SWAP* apparaissent quand le diagramme est produit à partir du graphe de la Fig. 11.3. Ceci est justifié par le fait que l'appartenance de p_1 à l'ensemble $ready$ sous-entend l'existence d'au moins un processus actif. Ce dernier est forcément p_2 dans le cas où $Process = \{p_1, p_2\}$. Les opérations *NEW* et *READY* ne sont donc pas déclenchables à partir de l'état $p_1 \in ready$ pour un système ne contenant que deux processus. Toutefois, dans le cas d'un système contenant trois processus ces deux opérations sont déclenchables et n'ont aucun effet sur l'état $p_1 \in ready$. Finalement, le comportement non-déterministe de l'opération *SWAP* ne peut être exhibé que si le nombre de processus est supérieur ou égal à 3 (plus précisément quand au moins deux processus sont à l'état $ready$ et un processus à l'état $active$).

Une solution concevable permettant de surmonter cette limite serait d'augmenter de plus en plus la taille du diagramme concret jusqu'à ce que le graphe abstrait atteigne un point fixe. Ce qui n'est pas aisé à réaliser de manière automatisée. D'autres techniques sont donc requises en vue d'aborder des systèmes infinis d'une manière plus efficace.

11.2.3 L'approche `GeneSyst`

Dans le cadre des travaux portant sur l'explicitation du comportement de spécifications et dans le but de remédier aux inconvénients des techniques basées sur l'animation (plus particulièrement celles liées



(a) $Process = \{p_1, p_2\}$ (b) $Process = \{p_1, p_2, p_3\}$

FIG. 11.6 – Diagrammes abstraits illustrant le cycle de vie du processus p_1 construits par notre prototype

aux espaces d'états infinis), D. Bert et F. Cave (Bert *et al.*, 2000) proposent une technique de preuve pour la génération d'un système de transitions étiqueté symbolique sans passer par un dépliage exhaustif des états concrets de la spécification. Cette technique se base sur des systèmes B événementiels à partir desquels chaque transition est réécrite en une obligation de preuve de la manière suivante :

1. Une transition simple $S_1 \xrightarrow{e} S_2$ correspond à la preuve suivante : si l'invariant rattaché à S_1 est vrai, et si l'événement e est déclenché alors l'état atteint par e établit l'invariant associé à S_2 ;
2. Une transition gardée $S_1 \xrightarrow{[c]e} S_2$ correspond au rajout de la condition c aux prémisses de la preuve précédente ;
3. Une transition non-déterministe menant d'un état S_1 à l'un des états S_2 et S_3 signifie que l'état atteint par la transition établit l'invariant associé à S_2 ou l'invariant associé à S_3.

Vu que cette approche se base sur des spécifications B événementielles, alors son application sur des spécifications B opérationnelles telles que celle du gestionnaire de processus passe par une étape de transformation des opérations en événements. Rappelons que ces derniers sont sous la forme :

$$Ev \mathrel{\widehat{=}} Garde(Ev) \Longrightarrow Action(Ev)$$

Lorsqu'une opération o est paramétrée, l'obligation de preuve produite pour l'événement qui lui est associé devient quantifiée par un quantificateur universel portant sur les paramètres de o.

Cette approche, actuellement outillée (Potet *et al.,* 2004, Bert *et al.,* 2005), ramène les obligations de preuve précédentes à la preuve de propriétés de déclenchabilité d'événements et d'atteignabilité d'états. En effet, dans (Potet *et al.,* 2004) une transition est définie par $(E, (D, A, Ev), F)$ où E et F sont les prédicats d'états abstraits source et cible de la transition, et D et A sont respectivement la condition de déclenchabilité de Ev à partir de E et la condition d'atteignabilité de F par Ev à partir de E, avec :

– l'événement Ev est déclenchable à partir de E si la garde de Ev est établie par tout élément de E qui satisfait D :

$$E \Rightarrow (D \Leftrightarrow Garde(Ev))$$

– à partir de tout état de E tel que Ev est déclenchable, alors l'état après exécution de $Action(Ev)$ est F si le prédicat A est vrai en entrée :

$$E \wedge Garde(Ev) \Rightarrow (A \Leftrightarrow \exists x' \cdot (prd_x(Action(Ev)) \wedge [x := x']F))$$

Sur cette base, l'algorithme utilisé au niveau de l'outil **`GeneSyst`** pour la génération de systèmes de transitions étiquetés symboliques cherche à déterminer ces conditions pour chaque événement Ev de la spécification et pour chaque couple de prédicats d'états abstraits E et F. Ceci est réalisé en prouvant les formules[39] ci-dessous :

Obligations de preuve	**Conditions**	**Interprétations**
(1) $E \Rightarrow Garde(Ev)$	$D \Leftrightarrow true$	Ev est déclenchable à partir de E
(2) $E \Rightarrow \neg Garde(Ev)$	$D \Leftrightarrow false$	Ev n'est jamais déclenchable à partir de E
(3) $E \wedge Garde(Ev) \Rightarrow \langle Action(Ev)\rangle F$	$A \Leftrightarrow true$	F peut être atteint par Ev à partir de E
(4) $E \wedge Garde(Ev) \Rightarrow [Action(Ev)]\neg F$	$A \Leftrightarrow false$	F n'est jamais atteint par Ev à partir de E

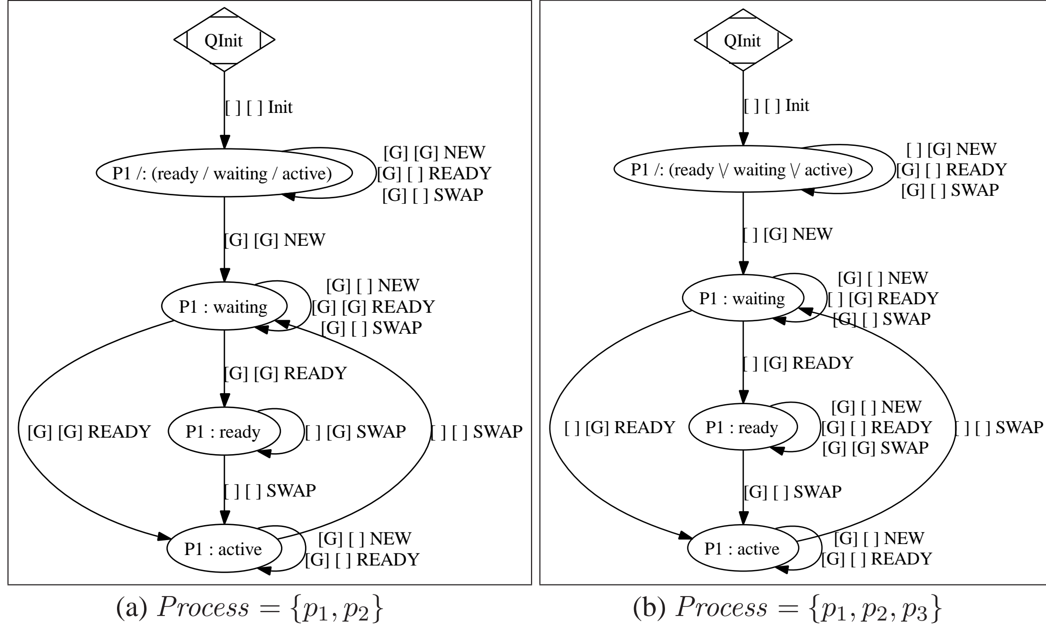
Bien que cette approche soit applicable à des systèmes infinis, certaines difficultés, principalement liées à la preuve, sont encore à considérer. Par exemple, l'échec des preuves (1) et (2) (respectivement (3) et (4)) ne permet pas de conclure quant à la déclenchabilité de Ev à partir de E (respectivement l'atteignabilité de F par Ev à partir de E). En effet, cet échec peut provenir d'une insuffisance du prouveur et de son incapacité à démontrer automatiquement certains théorèmes, et non d'une contradiction au niveau de l'obligation de preuve. En vue de prendre en compte ce problème, l'approche **`GeneSyst`** rajoute une information au niveau de chaque transition indiquant l'obligation de preuve non satisfaite.

La Fig. 11.7 présente les diagrammes du cycle de vie de p_1 produits par cette approche pour les deux systèmes finis où $Process = \{p_1, p_2\}$ et $Process = \{p_1, p_2, p_3\}$. Notons que dans ces diagrammes les conditions précédant chaque nom d'événement correspondent aux conditions de déclenchabilité et d'atteignabilité de l'événement. La mention "[G]" associée à une condition de déclenchabilité (respectivement d'atteignabilité) indique que la transition est déclenchable à partir d'un sous-ensemble de l'état source (respectivement un sous-ensemble de l'état cible est atteignable par la transition).

Observons que pour le cas d'un système contenant trois processus le diagramme abstrait résultant de la technique **`GeneSyst`** est similaire à celui produit par notre technique d'abstraction de graphes concrets. Cependant, l'outil n'élimine pas la transition réflexive *SWAP* au niveau de l'état $p_1 \in ready$

[39] La notation $\langle Action(Ev)\rangle F$ équivaut à $\exists x' \cdot (prd_x(Action(Ev)) \wedge [x := x']F)$ (se référer à (Potet *et al.,* 2004)).

pour le cas où $Process = \{p_1, p_2\}$. Cette surcharge du diagramme abstrait provient de l'échec de la preuve automatique des obligations de preuves associées à la condition d'atteignabilité de l'état $p_1 \in ready$ par *SWAP* à partir de ce même état.



(a) $Process = \{p_1, p_2\}$ (b) $Process = \{p_1, p_2, p_3\}$

FIG. 11.7 – Diagrammes abstraits illustrant le cycle de vie du processus p_1 construits par l'outil **GeneSyst**

Une autre difficulté, liée également à la complexité de la preuve, réside dans le coût d'une telle approche en terme de temps de calcul. Le tableau 11.2 présente une comparaison des temps de traitement de l'outil **GeneSyst** et de notre prototype d'abstraction de graphes d'états/transitions concrets pour la construction des Figs. 10.11, 11.6 et 11.7.

		Approche par model-checking		**Approche**
Diagramme	**États**	pour 2 processus	pour 3 processus	**GeneSyst**
Fig. 10.11 page 182	2	$\simeq$ 3 sec	$\simeq$ 8 sec	$\simeq$ 23 sec
Cycle de vie de p_1	4	Fig. 11.6(a) $\simeq$ 7 sec	Fig. 11.6(b) $\simeq$ 13 sec	Fig. 11.7(a)/(b) $\simeq$ 1 mn 20 sec

TAB. 11.2 – Temps de calcul de l'approche par model-checking et de l'approche **GeneSyst**

Ces résultats montrent une dualité des approches basées sur l'animation et la preuve provenant principalement de leur applicabilité sur des systèmes finis et infinis. Cette caractéristique procure de nombreux avantages notamment ceux de pouvoir décrire et analyser différents aspects comportementaux de la spécification d'un côté, et d'un autre côté, de disposer de techniques complémentaires pour une meilleure élaboration des diagrammes d'états/transitions abstraits. En effet, l'identification de dissemblances par une confrontation des diagrammes produits par chacune de ces techniques et pour une même abstraction, permet la construction de vues plus pertinentes et moins complexes.

En conclusion, les techniques d'animation aident à explorer les spécifications et ont vocation à identifier de manière effective les transitions possibles. Néanmoins, la taille de l'espace d'états à explorer reste leur principale contrainte. Ces techniques peuvent s'avérer très utiles au niveau des premières étapes de

construction d'un diagramme d'états/transitions abstrait où l'analyste recherche les bonnes abstractions d'états. Ensuite, les techniques de preuves peuvent être employées en vue d'examiner les transitions à partir de preuves formelles et de manière indépendante de l'espace d'états. Toutefois, les principales contraintes de ces techniques sont, d'une part, la lourdeur des preuves, et d'autre part, la nécessité, quelquefois, de vérifier de manière interactive les obligations de preuves qui n'ont pas pu être démontrées automatiquement.

11.2.4 Propriétés des transitions

Dans le cadre de notre approche d'abstraction de graphes d'accessibilité, une transition t entre deux états S_1 et S_2 ($t = S_1 \xrightarrow{o} S_2$) reflète l'existence d'au moins un appel de l'opération o permettant la transition entre deux états concrets e_1 et e_2 et telle que e_1 satisfait S_1 ($e_1 \vdash S_1$) et e_2 satisfait S_2 ($e_2 \vdash S_2$). Cette observation permet d'affiner les propriétés de déclenchabilité et d'atteignabilité proposées par l'approche **`GeneSyst`** sur la base des notions d'états concrets et d'états abstraits et proposer ainsi une vision plus précise des transitions entre états abstraits :

Soit t une transition entre deux états abstrait S_1 et S_2 ($t = S_1 \xrightarrow{o} S_2$), alors :

(i) t est dite toujours déclenchable à partir de S_1 si tous les états concrets satisfaisant l'état abstrait S_1 sont sources d'au moins une transition étiquetée par o. Dans le cas contraire, t est dite éventuellement déclenchable à partir de S_1 et sera étiquetée par o précédé du stéréotype ≪*possibly*≫.

(ii) S_2 est dit toujours atteignable par t à partir de S_1 si toutes les transitions étiquetées par o et sortant d'un état concret satisfaisant S_1, permettent d'atteindre au moins un état concret satisfaisant S_2. Dans le cas contraire, l'état S_2 est dit éventuellement atteignable par t à partir de S_1 et la transition t sera étiquetée par o suivi du stéréotype ≪*possibly*≫.

(iii) t est dite toujours réalisable si elle est toujours déclenchable à partir de S_1, et l'état $\mathcal{E}_2$ est toujours atteignable par t à partir de S_1. Dans le cas contraire, t est dite éventuellement réalisable et sera représentée par des pointillés.

11.3 Identification des états abstraits

Au niveau des techniques précédentes, nous n'avons pas abordé un processus automatique d'identification des états abstraits. En effet, dans ces techniques, le choix des prédicats d'abstraction est laissé à la propre appréciation de l'analyste. La principale finalité d'une telle préférence est d'offrir une plus grande souplesse au niveau de la construction des diagrammes d'états/transitions et permettre ainsi à l'analyste de se focaliser sur les propriétés qu'il juge opportun de mettre en évidence graphiquement. L'outil **`GeneEtat`**[40] (Hamdane, 2003, Hamdane, 2002), développé au sein de notre équipe, permet de mettre en œuvre certaines techniques d'identification d'états abstraits. Dans cette même optique, nous présentons au niveau de cette section, un ensemble de moyens permettant de pourvoir à l'identification de ces états abstraits.

[40] L'outil **`GeneEtat`** est actuellement en cours d'expérimentation.

11.3.1 Recours aux données de la spécification

La spécification peut contenir plusieurs assertions (*e.g.* invariants, pré-conditions, ...) pouvant traduire des états abstraits intéressants. Par exemple, nous avons montré comment l'invariant suivant : $active = \emptyset \Rightarrow ready = \emptyset$, peut être pris en compte au niveau d'une vue comportementale abstraite (Fig. 10.14 page 185).

Par ailleurs, au niveau d'un diagramme abstrait, les états reflètent des regroupements d'états concrets et indiquent, par conséquent, un partitionnement de l'espace d'états de la spécification. Une idée analogue surgit de la détermination d'une garde pour un événement (respectivement d'une pré-condition pour une opération) car il s'agit de délimiter un sous-espace d'états à partir duquel l'événement (respectivement l'opération) peut être déclenché. Il serait donc intéressant de considérer les gardes comme des états abstraits et de dégager ainsi une vue comportementale se focalisant sur l'étude de ces gardes (*i.e.* Fig. 10.4, page 175).

À partir d'un prédicat P portant sur les variables d'état v nous pouvons définir le sous-espace des états correspondants E_v. En d'autres termes, soit S_v l'espace d'états associé aux variables v et défini par l'invariant $\mathcal{I}$ de la spécification (*e.g.* $S_v = \{val|\mathcal{I}(val)\}$), alors nous définissons l'ensemble E_v suivant :

$$E_v = \{val|val \in S_v \wedge P(val)\}$$

Ainsi, si P correspond à la garde d'un événement e alors E_v représente l'ensemble des états à partir desquels e peut être déclenché. Cependant, considérer les gardes comme des états abstraits présente les deux inconvénients suivants :

- L'invariant peut ne pas être entièrement couvert vu que les sous-espaces d'états délimités par les gardes couvrent partiellement l'espace d'états délimité par l'invariant. Cette limite peut être traitée de deux manières : (*i*) considérer uniquement les états définis par les gardes et conserver tous les autres états en tant qu'états concrets, et ce, dans le cadre d'une approche par animation ou alors et dans le cadre d'une approche par preuve (*ii*) définir un état abstrait particulier délimité par E'_v comme suit :

$$E'_v = S_v - \bigcup E_v$$

- Les espaces d'états abstraits peuvent se chevaucher vu que les gardes peuvent ne pas être disjointes. De ce fait, considérer qu'une garde est un état peut aller à l'encontre de l'aspect intuitif d'un diagramme d'états/transitions car cela indique qu'un système peut être à plusieurs états à la fois. Nous proposons alors dans ce cas de traiter l'intersection des espaces d'états associés aux gardes et d'en dériver un espace d'états à part entière. C'est-à-dire, ayant $E_v^3 = E_v^1 \cap E_v^2$ nous définissons les trois sous-espaces d'états abstraits suivants : E_v^3, $E_v^1 - E_v^3$ et $E_v^2 - E_v^3$.

En vue d'illustrer ces propos, nous prenons l'exemple des deux opérations $reserver$ et $annuler$ de la machine RESERVATION. Celles-ci, étant pré-conditionnées respectivement par les prédicats $places > 0$ et $places < max$, avec la propriété invariante $place \in 0..max$, conduisent à l'identification des espaces d'états suivants :

$$\begin{array}{l} E^1_{places} = \{x|(x \in 0..max) \wedge (x > 0)\} \\ E^2_{places} = \{x|(x \in 0..max) \wedge (x < max\} \\ E'_{places} = \{x|(x \in 0..max)\} - (E^1_{places} \cup E^2_{places}) = \emptyset \end{array}$$

Vu que $E^1_{places} \cap E^2_{places} = 1..max - 1$, alors nous pouvons considérer les trois états abstraits ci-dessous et construire le diagramme de la Fig. 10.2 page 173 :

$$\begin{array}{lcl} S_1 \widehat{=} places \in (E^2_{places} - E^1_{places}) & & S_1 \widehat{=} places = 0 \\ S_2 \widehat{=} places \in (E^1_{places} - E^2_{places}) & \Leftrightarrow & S_2 \widehat{=} places = max \\ S_3 \widehat{=} places \in (E^1_{places} \cap E^2_{places}) & & S_3 \widehat{=} places \in 1..max - 1 \end{array}$$

11.3.2 Recours aux besoins informels de l'utilisateur

La pertinence et la lisibilité des descriptions abstraites construites par les différentes techniques que nous avons présentées sont fondamentales dans le cadre de notre travail. En effet, notre principal objectif est l'élaboration d'une documentation graphique servant d'outil de communication avec des clients ou des autorités de certification qui ne seraient pas familiarisés avec la méthode B. Ce faisant, notre travail est une étape aval d'un développement formel et vise, entre autres, la mise en valeur graphique de l'adéquation de la spécification par rapport aux exigences. Dans un tel contexte, le cahier des charges et son analyse forment un support important pour l'identification des meilleures abstractions. En effet, l'étude des exigences distingue, de manière souvent informelle, toutes les propriétés que le système doit satisfaire et peut par conséquent fournir des indications sur le choix des abstractions à mettre en valeur.

A. Davis (Davis, 1993) définit la phase de recensement et d'étude des besoins par deux activités principales : l'analyse du problème et la description du produit. La première activité se termine par une compréhension radicale des besoins permettant d'entreprendre la seconde activité et donc entamer la description de la solution. Ces activités de spécification amont produisent différents artefacts garantissant une bonne compréhension des exigences. La Fig. 11.8, extraite de (Davis, 1993) présente ces différents artifacts.

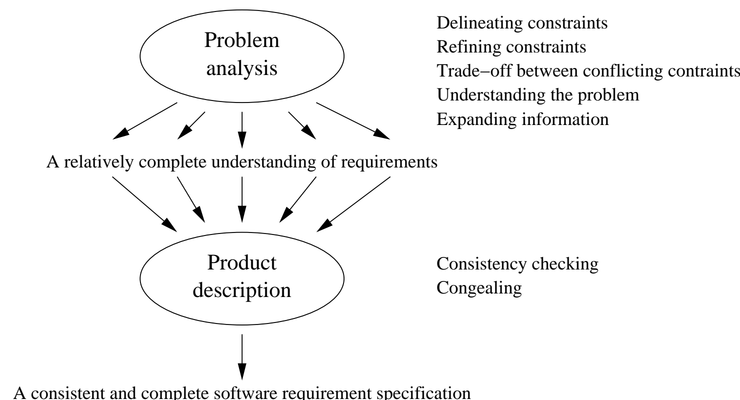


FIG. 11.8 – Ingénierie des besoins vue par A. Davis (Davis, 1993)

Nous pensons que le recours à ces artifacts permet de distinguer les vues graphiques les plus adéquates pour un objectif de documentation et d'approbation de la spécification par rapport aux exigences. Nous citons, en guise d'exemple, l'exigence indiquant qu'au plus un seul processus est actif à un instant donné.

Cette exigence peut conduire à définir des prédicats d'états abstraits portant sur $\mathrm{card}(active)$ tels que ceux illustrés au niveau de la Fig. 11.9 où nous montrons à un niveau abstrait comment un tel besoin

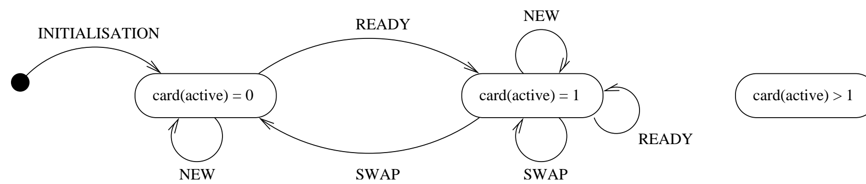


FIG. 11.9 – Illustration d'une exigence

est pris en compte lors de l'évolution de la spécification. Cette figure montre que l'état $\mathrm{card}(active) > 1$ ne peut être atteint par la spécification.

11.3.3 Définition d'un catalogue de patrons d'états abstraits

Nous avons présenté plusieurs diagrammes d'états/transitions abstraits s'attachant à la mise en évidence du caractère vide/non-vide des ensembles $active$, $ready$ et $waiting$. L'intérêt que nous portons à cette distinction particulière des états abstraits est justifié par la nécessité de prendre en compte le caractère indéterminé de ces ensembles. Par conséquent, il nous semble intéressant d'associer aux variables ayant une structure de donnée typique, telles que les ensembles abstraits, des prédicats d'abstraction typiques. Par exemple, nous pouvons penser que vu qu'un ensemble abstrait e peut inclure un ensemble indéterminé d'éléments, alors il est convenable de considérer des assertions qui correspondent à un nombre indéterminé de valeurs possibles telles que $e \neq \emptyset$.

Il nous paraît donc judicieux de définir une bibliothèque de patrons de prédicats d'abstraction dans le but d'aider à manipuler ces structures de données potentielles. Dans ce qui suit, nous n'allons pas présenter une liste exhaustive de prédicats d'abstraction, nous allons cependant présenter quelques exemples de patrons d'états abstraits tirés des structures les plus couramment utilisées.

Patron "Intervalle". Ce patron est appliqué dans le cadre de variables délimitées par des intervalles. Soit v une variable d'état telle que $v \in i..j$, dans ce cas, le patron "intervalle" suggère deux états aux limites de l'intervalle et un état représentant toutes les valuations intermédiaires :

$$S_1 \widehat{=} v = i$$
$$S_2 \widehat{=} v = j$$
$$S_3 \widehat{=} v \in (i+1)..(j-1)$$

Ce partitionnement $-$ décrit par les états S_1, S_2 et S_3 $-$ du domaine de définition de la variable v permet de mettre l'accent sur le comportement de la spécification quand v atteint ses valeurs extrêmes. Par exemple, nous constatons à partir du diagramme de la Fig. 10.2 page 173 que quand il n'y a plus de places disponibles ($places = max$), il n'est plus possible de faire appel à $reserver$.

Patron "Énumération". Ce patron porte sur des variables appartenant à des ensembles énumérés. Dans ce cas, chaque élément de l'énumération correspond à un état particulier et peut définir une abstraction intéressante. Par exemple, la Fig. 10.8 page 179 illustre les états du canal de communication en termes de valuations de la variable $channel$ tout en omettant les autres variables d'état. Ceci peut également être appliqué à des variables booléennes tels que $ackn$ et $channel_empty$. La Fig. 11.10 représente l'application de ce patron sur la variable $ackn$ et permet de mettre en évidence le fait que l'acquittement (transition *ACK*) est réalisé après chaque réception de messages.

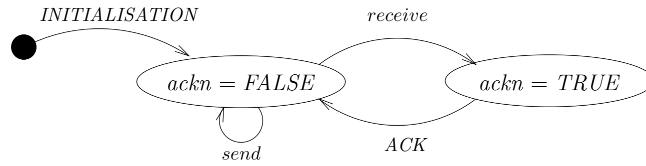


FIG. 11.10 – Application du patron "Énumération"

Patron "Inclusion". Il s'agit ici précisément de variables incluses dans des ensembles abstraits. Soit v une variable d'état telle que $v \subseteq E$ (où E est un ensemble abstrait), alors nous proposons les deux états abstraits disjoints, ci-dessous, portant sur un élément particulier $elem$ de v :

$$S_1 \widehat{=} elem \in E - v$$
$$S_2 \widehat{=} elem \in E \cap v \equiv elem \in v$$

L'application de ce patron sur la variable *registeredP* définie au niveau de la machine *SecureFlightRegistration* (page 113) peut conduire aux états suivants du passager Tom :

Non-enregistré $\widehat{=}$ $Tom \in Passenger - registeredP$
Enregistré $\widehat{=}$ $Tom \in Passenger \cap registeredP \equiv Tom \in registeredP$

Dans le cadre d'une technique d'abstraction de graphes d'accessibilité où $Passenger = \{Tom\}$, ces états permettent de produire le diagramme suivant où nous constatons que Tom doit être enregistré pour pouvoir enregistrer ses bagages (Fig. 11.11).

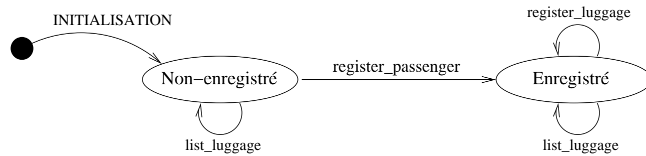


FIG. 11.11 – Application du patron "Inclusion"

L'application de ce patron rappelle le diagramme du cycle de vie du processus p_1. En effet, les prédicats d'états abstraits utilisés pour représenter l'évolution de p_1 peuvent être considérés comme le résultat d'une application simultanée du patron "Inclusion" sur les trois variables $active$, $ready$ et $waiting$ tout en remplaçant l'intersection des états[41] $p_1 \notin active$, $p_1 \notin ready$ et $p_1 \notin waiting$ par l'état particulier $p_1 \notin (waiting \cup ready \cup active)$. En effet, le patron inclusion produit les états suivants :

- $p_1 \in active$ et $p_1 \notin active$
- $p_1 \in ready$ et $p_1 \notin ready$
- $p_1 \in waiting$ et $p_1 \notin waiting$

Notons qu'il a été possible de combiner l'application de ce patron à ces trois variables car celles-ci sont parfaitement disjointes (*e.g.* un processus est dans un seul état à la fois). Ceci nous amène à définir une forme particulière du patron "Inclusion" comme suit : soit E un ensemble abstrait et soient $v_1, \ldots, v_n$ des variables d'états telles que :

(a) $\forall i \cdot (i \in 1..n \Rightarrow v_i \subseteq E)$, et

(b) $\forall i, j \cdot (i \in 1..n \wedge j \in 1..n \wedge i \neq j \Rightarrow v_i \cap v_j = \emptyset)$

[41] L'état abstrait $p_1 \notin waiting$ est équivalent à $p_1 \in Process - waiting$...

alors pour chaque i appartenant à $1..n$ nous dérivons un état S_i avec $S_i \widehat{=} elem \in v_i$ et nous rajoutons l'état particulier S_{n+1} avec :

$$S_{n+1} \widehat{=} elem \in E - \bigcup_{i=1}^{n} v_i$$

Le patron "Inclusion" permet de se focaliser sur l'évolution d'un élément particulier $elem$ de l'ensemble abstrait E et d'illustrer le passage de $elem$ d'un sous-ensemble à un autre sous-ensemble. Des formes hiérarchiques peuvent alors apparaître provenant plus particulièrement de l'inclusion entre variables elles-mêmes. En effet, soient v_1, v_2 deux variables d'état définies comme suit : $v_1 \subseteq E$ et $v_2 \subseteq v_1$, alors l'application du patron "Inclusion" produit les quatre états suivants pour $elem \in E$:

$$S_1 \widehat{=} elem \in E - v_1 \qquad S_3 \widehat{=} elem \in v_1 - v_2$$
$$S_2 \widehat{=} elem \in v_1 \qquad S_4 \widehat{=} elem \in v_2$$

Ce faisant, nous remarquons que S_2 englobe les états S_3 et S_4. Nous ne considérons alors que les trois états disjoints S_1, S_3 et S_4 en vue de pouvoir appliquer l'une des techniques de construction de diagrammes d'états/transitions abstraits, et ensuite nous illustrons S_2 par un état composite. Ce cas de figure s'applique aux variables $registeredP$ et in_room car elles sont définies telles que : $registeredP \subseteq Passenger$ et $in_room \subseteq registeredP$. Nous gardons alors les états *Enregistré* et *Non-enregistré* présentés ci-dessus et nous considérons les états :

Dans le hall central $\widehat{=}$ $Tom \in registeredP - in_room$
Dans la salle d'embarquement $\widehat{=}$ $Tom \in in_room$

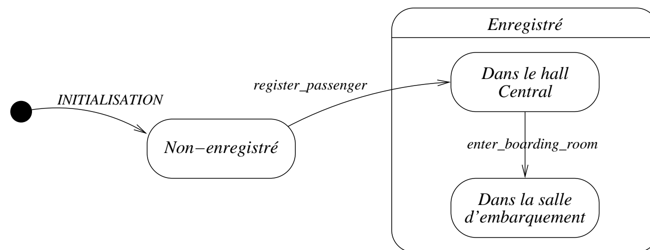


FIG. 11.12 – Vue hiérarchique produite par application du patron "Inclusion"

Patron "Relations fonctionnelles". Ce patron s'inspire des autres patrons et permet d'appliquer des abstractions similaires en se focalisant sur le domaine et le co-domaine de relations. Soit R une relation entre deux ensembles E_1 et E_2 ($R \in E_1 \leftrightarrow E_2$), alors nous distinguons les cas suivants :

	$\mathbf{E_1 = i..j}$	$\mathbf{E_2 = i..j}$
Prédicats d'états	$R^{-1}(elem) = i$	$R(elem) = i$
	$R^{-1}(elem) = j$	$R(elem) = j$
	$R^{-1}(elem) \in (i+1)..(j-1)$	$R(elem) \in (i+1)..(j-1)$

TAB. 11.3 – Utilisation du patron "Intervalle"

	$\mathbf{E_1 = \{val_1, \ldots, val_n\}}$	$\mathbf{E_2 = \{val_1, \ldots, val_n\}}$
Prédicats d'états	$R^{-1}(elem) = val_1$ $\ldots$ $R^{-1}(elem) = val_n$	$R(elem) = val_1$ $\ldots$ $R(elem) = val_n$

TAB. 11.4 – Utilisation du patron "Énumération"

	$\mathbf{E_1 \subseteq E}$	$\mathbf{E_2 \subseteq E}$
Prédicats d'états	$R^{-1}(elem) \in E - \mathrm{dom}(R)$ $R^{-1}(elem) \in \mathrm{dom}(R)$	$R(elem) \in E - \mathrm{ran}(R)$ $R(elem) \in \mathrm{dom}(R)$

TAB. 11.5 – Utilisation du patron "Inclusion"

Soit, par exemple[42], les deux constantes $ground$ et top définies telle que :

$$ground \in \mathbb{N} \wedge$$
$$top \in \mathbb{N} \wedge$$
$$ground < top$$

et soit la variable abstraite *floor* définie par : $floor \in LIFT \rightarrow ground..top$ (où $LIFT$ est un ensemble abstrait). Alors l'application de ce patron peut conduire à l'identification des trois états ci-dessous indiquant la situation d'un élément particulier *lift* de l'ensemble $LIFT$:

$$S_1 \widehat{=} floor(lift) = ground$$
$$S_2 \widehat{=} floor(lift) = top$$
$$S_3 \widehat{=} floor(lift) \in (ground + 1)..(top - 1)$$

11.4 Dérivation de diagrammes d'états/transitions sur la base des vues structurelles

Au niveau des sections précédentes nous nous sommes intéressé aux outils de dérivation de diagrammes d'états/transitions à partir de spécifications B et nous avons montré leur utilité pour un objectif de visualisation des aspects comportementaux. Cependant, bien que nous ayons préconisé certaines techniques pour l'identification d'un ensemble d'états abstraits, la détermination des meilleures abstractions reste une perspective ouverte. En effet, vu que nous ne ciblons ni un domaine applicatif particulier ni un ensemble de modèles "métier" et que les techniques que nous proposons ont pour vocation d'être générales, alors les abstractions présentées restent limitées aux constructions les plus couramment utilisées. Par conséquent, nous avons exploré de manière indépendante les aspects structurels et les aspects comportementaux d'une variété de spécifications B.

Cependant, nous avons montré, au niveau du chapitre précédent, l'intérêt de construire des diagrammes d'états/transitions pour représenter, en particulier, le comportement de classes issues de la spécification B (*i.e.* classe *Process* au niveau de la Fig. 10.15 page 186). Ainsi, et dans le but d'aboutir à une documentation formée par des diagrammes cohérents dans leur ensemble et présentant aussi bien les aspects structurels que les aspects comportementaux d'une même spécification B, nous ciblons, dans cette section, une mise en relation des différentes techniques de dérivation des diagrammes de classes et

[42] Cet exemple est extrait de (Abrial, 1996) page 359 où il s'agit d'une spécification d'un système de contrôle d'ascenseurs.

des diagrammes d'états/transitions. Par ailleurs, nous notons que cette mise en relation de vues structurelles et comportementales a fait l'objet de l'article (Idani, 2006).

Les spécifications B traitées jusqu'ici se divisent en deux catégories selon que l'aspect structurel ou comportemental est prédominant. Ce faisant, en vue d'illustrer le couplage des différentes techniques que nous proposons, nous allons considérer une spécification B intégrant des aspects structurels et comportementaux. Il s'agit de la spécification d'un contrôleur d'accès (Fig D.5, page 258) dont la tâche est de gérer l'accès de personnes à des bâtiments selon un ensemble d'autorisations[43]. Nous n'allons pas détailler cette spécification à ce niveau, nous allons cependant la présenter progressivement en interprétant les diagrammes qui en découlent. Le diagramme de classes de la Fig. 11.13, ci-dessous, est issu de la spécification du contrôleur d'accès par application de notre technique de dérivation de diagrammes de classes. Dans ce diagramme quatre classes ont été identifiées : *PERSONNE*, *BATIMENT*, *CARTE* et *VALIDE*.

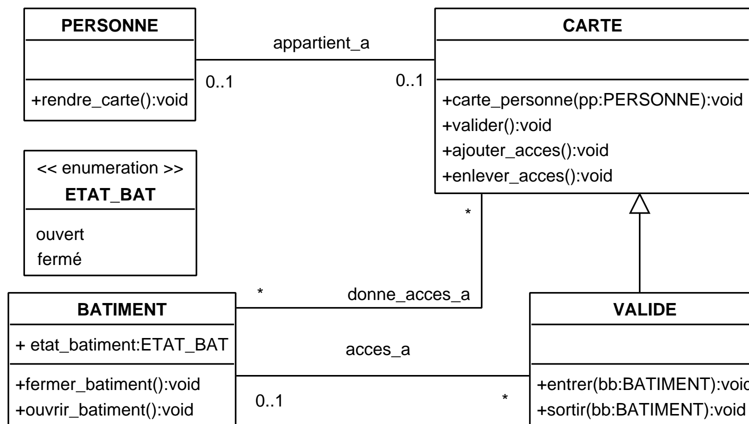


FIG. 11.13 – Diagramme de classes issu de la machine ACCESSCONTROL

L'association *donne_acces_a* modélise l'ensemble des bâtiments pour lesquels une carte donnée peut être utilisée, tandis qu'un lien *acces_a* entre une carte C et un bâtiment B désigne le fait que la carte C a été effectivement utilisée pour accéder au bâtiment B (en d'autres termes, la personne qui détient la carte C est actuellement à l'intérieur du bâtiment B). Les deux méthodes +*carte_personne*(*pp* : *PERSONNE*) et +*rendre_carte*() permettent respectivement de créer et de supprimer un lien *appartient_a* entre une carte donnée et une personne passée en paramètres. Les méthodes +*ouvrir_batiment*() et +*fermer_batiment*() de la classe *BATIMENT* permettent d'ouvrir ou de fermer un bâtiment en modifiant la valeur de l'attribut *etat_batiment*. Celle-ci, prend ses valeurs de l'ensemble énuméré *ETAT_BAT* = $\{$*ferme*, *ouvert*$\}$. La création et la suppression d'un lien *acces_a* entre une instance de la classe *VALIDE* et une instance de la classe *BATIMENT* se fait à travers les méthodes +*entrer*(*bb* : *BATIMENT*) et +*sortir* (*bb* : *BATIMENT*) de la classe *VALIDE*.

11.4.1 Formalisation

Rappelons que les différents éléments de modélisation du diagramme de classes de la Fig. 11.13 proviennent des données et opérations de la spécification B de départ et satisfont les liens que nous avons

[43] Cette spécification B est inspirée de (Potet, 1998)

définis entre méta-modèles B et UML. De ce fait, les correspondances structurelles et sémantiques entre B et UML délimitées par les schémas de transformation du chapitre 4 nous procurent une base formelle pour l'identification des prédicats d'états. En effet, soient $\mathcal{M}_\mathrm{B}$ et $\mathcal{M}_\mathrm{UML}$ les méta-modèles respectifs de B et de UML, alors nous définissons les correspondances entre les deux méta-modèles par la relation $\mathcal{H}$ suivante :

$$\mathcal{H} \in \mathcal{M}_\mathrm{UML} \leftrightarrow \mathcal{M}_\mathrm{B}$$

Par exemple, la correspondance établie de la méta-classe *BAbsractSet* vers la méta-classe *Class* (Fig. 4.5 page 74) est formalisée par $(\mathit{BAbstractSet} \mapsto \mathit{Class}) \in \mathcal{H}$ et signifie qu'il existe un schéma de transformation pouvant transformer un ensemble abstrait B en une classe UML. Par analogie à la relation d'instanciation *Inst* (Définition 7.2 page 125) nous définissons la relation $\mathit{Inst}_\mathrm{UML}$ comme suit :

Définition 11.1
Soit $\mathcal{M}_\mathrm{UML}$ l'ensemble des méta-classes du méta-modèle UML, et soit $\mathcal{K}$ l'ensemble des construction d'un diagramme de classes, alors la relation $\mathit{Inst}_\mathrm{UML}$ suivante :

$$\mathit{Inst}_\mathrm{UML} \in \mathcal{M}_\mathrm{UML} \leftrightarrow \mathcal{K}$$

permet d'identifier pour chaque méta-classe du méta-modèle UML, l'ensemble de ses instances au niveau d'un diagramme de classes.

Sur cette base, nous pouvons préciser la relation $\mathcal{B}_\mathcal{K}$, définie entre une spécification B (représentée par $\mathcal{B}$) et le diagramme de classes final (représenté par $\mathcal{K}$), de la manière suivante :

$$\mathcal{B}_\mathcal{K} \in \mathcal{K} \leftrightarrow \mathcal{B} \text{ avec,}$$
$$\forall (a \mapsto b) \cdot ((a \mapsto b) \in \mathcal{B}_\mathcal{K} \Rightarrow (\mathit{Inst}_\mathrm{UML}^{-1}(a) \mapsto \mathit{Inst}^{-1}(b)) \in \mathcal{H})$$

11.4.2 Prédicats d'états associés aux classes

En UML, un état associé à une classe définit une situation donnée durant la vie de l'objet instance de la classe (Muller *et al.,* 2001) et est caractérisé par :

(*i*) la conjonction des valeurs des attributs de l'objet ; par exemple, un objet instance de *BATIMENT* est dans l'état *Fermé* si la valeur de son attribut *etat_batiment* est égale à *fermé*.

(*ii*) l'existence ou non de liens entre l'objet considéré et d'autres objets ; par exemple, une instance de la classe *BATIMENT* est à l'état *vide* si elle n'est pas liée par le lien *acces_a* à aucune instance de *CARTE*.

Toutefois, une classe $\mathcal{C}$ $(\mathit{Inst}_\mathrm{UML}^{-1}(\mathcal{C}) = \mathit{Class})$ peut être dérivée par application de plusieurs schémas de transformation, ce qui nécessite une distinction de différents cas de figures dépendant des éléments B à l'origine de $\mathcal{C}$ $(\mathit{Inst}^{-1}[\mathcal{B}_\mathcal{K}[\{\mathcal{C}\}]])$. Parmi ces schémas nous allons seulement nous focaliser sur les schémas 1 (page 71) et 3.3 (page 74) car ils produisent des classes à partir d'éléments B (instances des méta-classes *BMachine* et *BAbstractSet*) pour lesquels un comportement peut être associé.

Classes produites pour des *BMachine*. Dans le cadre de classes issues de machines B (c'est-à-dire $\mathit{Inst}^{-1}[\mathcal{B}_\mathcal{K}[\{\mathcal{C}\}]] = \{\mathit{BMachine}\}$), la seule caractérisation possible des états porte sur la conjonction des valeurs des attributs. En effet, la caractérisation des états par l'existence ou non de liens avec d'autres

classes ne peut être envisagé car les schémas de transformation ne construisent que des liens de dépendance pour les classes issues de *BMachine*. De ce fait, l'étude du comportement de ces classes revient à l'étude du comportement de la spécification B de départ, et se ramène aux différents types d'abstraction présentés précédemment. La Fig. 11.14 illustre deux classes issues de machines B avec un exemple de diagrammes d'états/transitions exprimés à partir de leurs attributs.

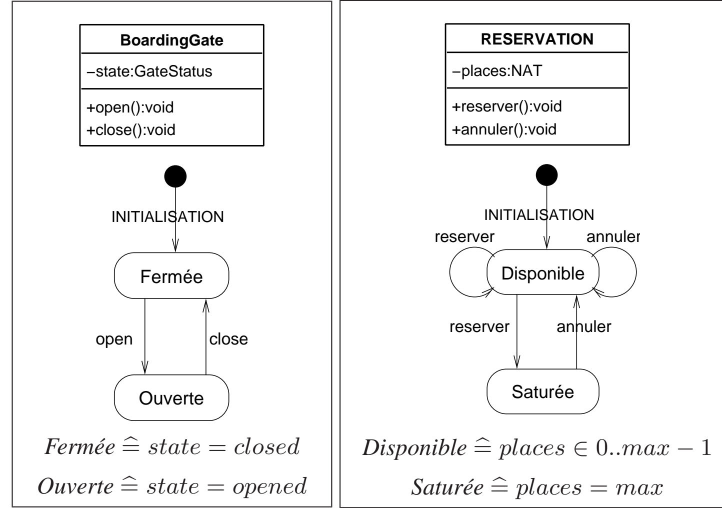


FIG. 11.14 – Classes issues de machines B avec leurs diagrammes d'états/transitions

Classes produites pour des *BAbstractSet*. Dans le cadre d'une classe $\mathcal{C}$ issue d'un ensemble abstrait (c'est-à-dire $Inst^{-1}[\mathcal{B}_{\mathcal{K}}[\{\mathcal{C}\}]] = \{BAbstractSet\}$), les prédicats d'états sont exprimés sur la base d'un élément $elem$ du *BAbstractSet* correspondant à $\mathcal{B}_{\mathcal{K}}(\mathcal{C})$ et nous distinguons deux cas de caractérisation des états de $\mathcal{C}$.

A. Expression de prédicats d'états à partir des attributs des classes

Dans le cas où la classe est produite à partir d'ensembles abstraits, seuls les schémas 5.5 (page 78), 9.2 (page 82) et 9.3 (page 82) permettent de lui associer un attribut. Dès lors, un attribut t d'une classe $\mathcal{C}$ telle que $Inst^{-1}[\mathcal{B}_{\mathcal{K}}[\{\mathcal{C}\}]] = \{BAbstractSet\}$ provient soit d'un ensemble abstrait avec :

$$\mathcal{B}_{\mathcal{K}}(t) \subseteq \mathcal{B}_{\mathcal{K}}(\mathcal{C})$$

soit d'une relation avec :

$$\mathcal{B}_{\mathcal{K}}(\mathcal{C}) \in \{\mathrm{dom}(\mathcal{B}_{\mathcal{K}}(t)), \mathrm{ran}(\mathcal{B}_{\mathcal{K}}(t))\}$$

Dans le premier cas, l'attribut considéré est un attribut booléen (vu l'inclusion), et dans le second cas, l'attribut est typé soit par le domaine soit par le co-domaine de la relation.

(*i*) **Cas d'un attribut booléen** : deux états peuvent être associés à la classe $\mathcal{C}$ selon la valeur logique de l'attribut t (S^t_{true} et S^t_{false}). Soit $elem$ tel que $elem \in \mathcal{B}_{\mathcal{K}}(\mathcal{C})$, alors :

$$\begin{array}{lcl} S^t_{true} & \widehat{=} & elem \in \mathcal{B}_{\mathcal{K}}(t) \\ S^t_{false} & \widehat{=} & elem \notin \mathcal{B}_{\mathcal{K}}(t) \end{array}$$

Par exemple, dans le cas où le sous-ensemble *VALIDE* de *CARTE* est transformé en un attribut booléen de la classe *CARTE*, alors nous pouvons associer les états *valide* et *invalide*, à la classe CARTE en les définissant à partir des prédicats B correspondant aux valeurs logiques de son attribut booléen *VALIDE*, et ce, de la manière suivante :

$$\textit{valide} \mathrel{\widehat{=}} carte \in \textit{VALIDE}$$
$$\textit{invalide} \mathrel{\widehat{=}} carte \notin \textit{VALIDE}$$

(ii) **Cas d'un attribut typé par un ensemble énuméré** : des états élémentaires S_i^t sont exprimés pour chaque valeur val_i de l'ensemble énuméré. Étant donné que $\mathcal{B}_{\mathcal{K}}(t)$ est une relation alors :

– Si le type de t correspond au co-domaine de $\mathcal{B}_{\mathcal{K}}(t)$:

$$S_i^t \mathrel{\widehat{=}} \mathcal{B}_{\mathcal{K}}(t)(elem) = val_i$$

– Si le type de t correspond au domaine de $\mathcal{B}_{\mathcal{K}}(t)$:

$$S_i^t \mathrel{\widehat{=}} (\mathcal{B}_{\mathcal{K}}(t))^{-1}(elem) = val_i$$

Par exemple, les états *Fermé* et *Ouvert*, associés à la classe *BATIMENT* et exprimés à partir des valeurs possibles de l'attribut *etat_batiment*, sont définis par les prédicats d'états suivants :

$$\textit{Fermé} \mathrel{\widehat{=}} etat_batiment(batiment) = ferme$$
$$\textit{Ouvert} \mathrel{\widehat{=}} etat_batiment(batiment) = ouvert$$

B. Expression de prédicats d'états à partir des liens entre classes

Les classes peuvent être liées par un lien d'héritage, une association ou une classe associative. Dans le cadre d'un lien d'héritage les états sont exprimés de la même manière que pour un attribut booléen (*i.e.* les états *valide* et *invalide* de la classe *CARTE* peuvent être considérés étant donnée la sous-classe *VALIDE* de *CARTE*).

Lorsqu'il s'agit d'une association ou d'une classe associative $\mathcal{R}$ entre deux classes $\mathcal{C}_1$ et $\mathcal{C}_2$ telle que $\mathcal{B}_{\mathcal{K}}(\mathcal{C}_1)$ et $\mathcal{B}_{\mathcal{K}}(\mathcal{C}_2)$ sont respectivement le domaine et le co-domaine de la relation $\mathcal{B}_{\mathcal{K}}(\mathcal{R})$, les prédicats d'états associés à chacune des classes $\mathcal{C}_1$ et $\mathcal{C}_2$ seront exprimés par $S_\exists$ et $S_\nexists$ selon qu'il existe ou non de liens $\mathcal{R}$ entre les instances de $\mathcal{C}_1$ et de $\mathcal{C}_2$:

– Pour $\mathcal{C}_1$:

$$S_\exists \mathrel{\widehat{=}} elem \in \mathrm{dom}(\mathcal{B}_{\mathcal{K}}(\mathcal{R}))$$
$$S_\nexists \mathrel{\widehat{=}} elem \notin \mathrm{dom}(\mathcal{B}_{\mathcal{K}}(\mathcal{R}))$$

– Pour $\mathcal{C}_2$:

$$S_\exists \mathrel{\widehat{=}} elem \in \mathrm{ran}(\mathcal{B}_{\mathcal{K}}(\mathcal{R}))$$
$$S_\nexists \mathrel{\widehat{=}} elem \notin \mathrm{ran}(\mathcal{B}_{\mathcal{K}}(\mathcal{R}))$$

Par exemple l'existence ou non d'un lien *acces_a* entre une instance quelconque $carte$ de la classe *CARTE* et une instance quelconque $batiment$ de la classe *BATIMENT* permet de définir les états *In* et *Out* de *CARTE* ainsi que les états *vide* et *occupé* de *BATIMENT* comme suit :

Pour la classe *CARTE* :	Pour la classe *BATIMENT* :
In $\mathrel{\widehat{=}} carte \in \mathrm{dom}(acces_a)$	*vide* $\mathrel{\widehat{=}} batiment \in \mathrm{ran}(acces_a)$
Out $\mathrel{\widehat{=}} carte \notin \mathrm{dom}(acces_a)$	*occupé* $\mathrel{\widehat{=}} batiment \notin \mathrm{ran}(acces_a)$

11.4.3 Application et discussion

Soient les états *Fermé* et *Ouvert* (définis au niveau de la section 11.4.2) associés à la classe *BATIMENT*, et un graphe d'accessibilité construit pour *PERSONNE* = {*Tom*} ; *CARTE* = {*C*} ; *BATIMENT* = {*B*}, alors le diagramme d'états/transitions produit par notre technique pour la classe *BATIMENT* est celui de la Fig. 11.15. Ce diagramme d'états/transitions illustre le fait que l'instance B de la classe *BATIMENT* peut être soit à l'état *Fermé* soit à l'état *Ouvert*, que les transitions entre ses états sont réalisées uniquement via les opérations *ouvrir_batiment* et *fermer_batiment* et qu'à l'initialisation le bâtiment est fermé. La transition *ouvrir_batiment*(B) est une transition toujours réalisable à partir de l'état *Fermé* contrairement à la transition *fermer_batiment*(B) qui est éventuellement déclenchable à partir de l'état *Ouvert*. En effet, un bâtiment ne peut être fermé que s'il est vide. Cependant, l'état *Fermé* est toujours atteignable par la transition *fermer_batiment*(B) à partir de l'état *Ouvert*.

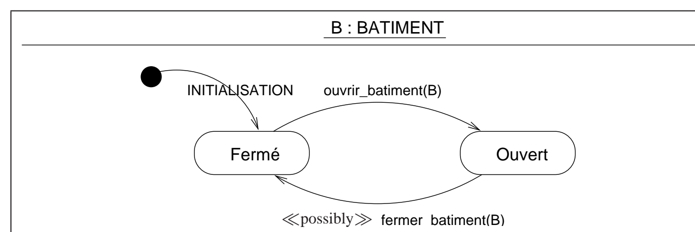


FIG. 11.15 – Diagramme d'états/transitions associé à la classe "BATIMENT"

Cependant, nous remarquons à partir du diagramme de la Fig. 11.13, que l'état de la classe *BATIMENT* peut également être caractérisé par l'existence ou non des liens *acces_a* et *donne_acces_a*. Il nous semble alors plus adéquat, d'illustrer sous forme d'un diagramme concurrent toutes les paires d'états de la classe *BATIMENT*. Nous considérons donc que les états concurrents correspondent aux différentes combinaisons des états abstraits. Il s'agit à ce niveau d'identifier les relations possibles entre les différents diagrammes d'états/transitions énoncés pour chaque paire d'états d'une classe donnée.

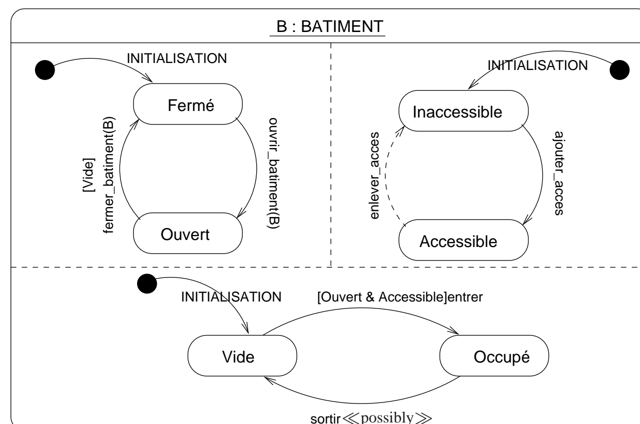


FIG. 11.16 – Diagramme d'états/transitions concurrent pour à la classe "BATIMENT"

Par exemple, pour la classe *BATIMENT*, nous avons identifié 6 états élémentaires : deux états exprimés à partir des attributs (*Fermé*/*Ouvert*) et quatre états exprimés à partir de l'existence ou non de liens entre les instances de la classe *BATIMENT* et les instances de la classe *CARTE* (*Accessible*/*Inaccessible*

et *Occupé*/*Vide*). Un objet *BATIMENT* est simultanément dans les états (*Fermé* ou *Ouvert*) et (*Accessible* ou *Inaccessible*) et (*Occupé* ou *Vide*). Vu que chaque paire d'états élémentaires donne lieu à un diagramme d'états/transitions alors la construction du diagramme d'états/transitions concurrent se fait par composition des différents diagrammes d'états/transitons élémentaires.

Le diagramme d'états/transitions concurrent associé à la classe *BATIMENT* est présenté au niveau de la figure 11.16. Dans ce diagramme, les relations de dépendance entre états concurrents est définie par les gardes de certaines transitions. Par exemple, la transition *entrer* de *Vide* à *Occupé* n'est déclenchable que si le bâtiment est simultanément *Vide*, *Ouvert* et *Accessible*.

Il y a là une ouverture sur l'étude des conditions de déclenchement dans des diagrammes concurrents, ainsi que sur l'introduction d'événements "partagés" dans ces diagrammes (Stoddart *et al.,* 1998).

11.5 Bilan et conclusion

Au niveau du présent chapitre nous avons exploré un ensemble de techniques complémentaires permettant d'automatiser le processus de dérivation de diagrammes d'états/transitions à partir de spécifications B. Ces techniques sont classées en deux catégories selon qu'elles sont applicables à des systèmes finis ou infinis, en effet, leurs fondements sont respectivement :

- L'animation des spécifications pour un calcul exhaustif des états concrets et des transitions entre ces états, et
- La preuve des propriétés de déclenchabilité d'événements et d'atteignabilité d'états pour l'explicitation d'un comportement abstrait sans avoir recours à un dépliage effectif du comportement de la spécification.

Dans le cadre de notre travail, nous nous sommes principalement basé sur l'animation des spécifications B vu que la contribution d'une telle technique à l'exhibition du comportement de ces spécifications répond parfaitement à notre objectif de documentation. En effet, tous les diagrammes d'états/transitions présentés au niveau du présent chapitre et du chapitre 10 ont pu être construits de manière automatique par notre outil prototype d'abstraction de graphes d'accessibilité.

Nous avons également présenté un ensemble de moyens pour l'identification des prédicats d'états et extraire ainsi des diagrammes d'états/transitions utiles et pragmatiques. Il s'agit notamment d'avoir recours aux données de la spécifications de départ, ou aux exigences et aux différents artifacts produits lors de l'analyse des besoins, ou encore de définir un catalogue de patrons d'états abstraits dépendants des modèles utilisés et des systèmes les plus couramment spécifiés.

Cependant, ces différentes voies n'offrent pas des fondements théoriques formels pour l'identification des états abstraits de manière systématique car ils ont pour objectif d'assurer une certaine flexibilité lors de la construction d'une documentation comportementale. Nous pensons que cette flexibilité est indispensable pour aborder des points de vues spécifiques à chaque intervenant dans un projet B (analyste, utilisateur, client, programmeur, autorité de certification, ...).

Ce faisant, nous nous sommes basé sur le cadre conceptuel proposé au niveau du chapitre 4 pour fournir une base formelle précise pour la mise en relation de vues structurelles et comportementales. Dans ce contexte-ci l'identification des prédicats d'états est systématique et a pour objectif de se focaliser sur le comportement des classes issues de la spécification B.

Conclusion générale

> « *Maintenant l'abstrait s'était matérialisé, l'être enfin compris avait aussitôt perdu de son pouvoir de rester invisible,* [...]
> *tout ce qui avait paru, jusque-là, incohérent à mon esprit,*
> *devenait intelligible, se montrait évident, comme une phrase, n'offrant aucun sens tant qu'elle reste décomposée en lettres disposées au hasard, exprime,*
> *si les caractères se trouvent replacés dans l'ordre qu'il faut,*
> *une pensée que l'on ne pourra plus oublier.* »
>
> *M. Proust*
> *« Sodome et gomorrhe », 1922.*

Face à la complexité croissante des systèmes informatiques, la **modélisation** et la **documentation**, deux ingrédients étroitement liés dans le processus de développement de logiciels, se veulent de plus en plus indispensables (The Standish Group, 1998, The Standish Group, 2001). Aussi, est-il fondamental de définir un cadre méthodologique cohérent intégrant aussi bien modélisation que documentation. Cependant, les travaux de recherche menés jusqu'à présent se sont principalement penchés sur les outils et méthodes de modélisation – justifiant ainsi leur diversité (informelles, semi-formelles et formelles) – sans fournir les outils nécessaires à la construction d'une documentation adéquate. Celle-ci, reste donc liée au système de façon informelle bien qu'elle soit issue d'une activité conjointe au développement et malgré son importance lors des phases de développement aval, et plus particulièrement lors des phases de validation et de certification. Dans le présent travail de thèse, notre intérêt a porté précisément sur la méthode B qui est une méthode formelle utilisée pour modéliser des systèmes et vérifier la correction de leur conception au moyen de raffinements successifs. La documentation en B est généralement fournie sous forme de textes ou d'annotations avec parfois des schémas mettant en évidence quelques traits de la spécification formelle.

Notre contribution a été la proposition d'outils théoriques et effectifs pour la documentation de spécifications formelles en B. Nous avons ciblé particulièrement l'aide à la compréhension de spécifications pour une assistance à la validation externe des développements B. Pour ce faire, la documentation que nous produisons est construite par un ensemble de vues structurelles et comportementales développées selon le standard de facto UML. L'idée d'intégrer des notations formelles et semi-formelles largement connues et utilisées dans un contexte industriel (*e.g.* B et UML) a été, néanmoins, étudiée depuis plusieurs années (Fraser *et al.,* 1991), et ce, compte tenu de l'efficacité d'une telle intégration. En effet, les inconvénients des méthodes formelles peuvent être surmontés grâce aux apports des méthodes semi-formelles et réciproquement.

Méthodes semi-formelles. Dans (Dupuy-Chessa, 2000), S. Dupuy distingue trois qualificatifs principaux des vues produites par les méthodes semi-formelles : *synthétiques*, *structurantes* et *intuitives*. En effet, les avantages de ces méthodes peuvent être synthétisés dans les points suivants :

– Être adaptées à la plupart des acteurs de développement ;
– Offrir des mécanismes de structuration riches en diagrammes ;
– Présenter une vision abstraite mais claire du système via des notations graphiques ;
– Mieux faire percevoir les besoins par les concepteurs ;
– Faciliter la communication avec les utilisateurs finaux du système.

Cependant, ces méthodes présentent un inconvénient majeur dû au manque de sémantique précise des notations utilisées. La construction de systèmes sur la base de ces méthodes peut alors conduire, parfois, à des modèles ambigus et provoquer des malentendus ainsi que des difficultés d'interprétation. Nous citons à titre indicatif (Simons *et al.,* 1999, Reggio *et al.,* 1999, Fecher *et al.,* 2005) où nous avons retrouvé une revue de nombreuses lacunes de la notation UML.

Méthodes formelles. Les principaux points forts des méthodes formelles sont : précision, abstraction et formalisme permettant raisonnement, vérification, tests et preuves (Craigen *et al.,* 1995). En effet, les avantages de ces méthodes peuvent être énoncés succinctement ainsi :

– Les propriétés des systèmes peuvent être établies par raisonnement ;
– Les intuitions sont démontrables par une argumentation stricte ;
– L'implantation d'un programme est conforme à sa spécification ;
– Possibilité de génération automatique de code ou des cas de test à partir des spécifications ;
– Différents niveaux d'abstraction sont établis et sont clairement définis.

Ceci étant, certains points faibles de ces méthodes sont à mettre en évidence. Nous citons d'un côté, le niveau de formation nécessaire à leur maîtrise, et d'un autre côté, le fait que les notations mathématiques abstraites sont peu intuitives et donc difficiles à comprendre (Hall, 1990, Gaudel, 1991, Bowen *et al.,* 1995).

Bilan de nos contributions

Les atouts et les insuffisances, listés précédemment, des méthodes formelles et semi-formelles ont motivé également l'ensemble des travaux de couplage de UML et B. Nous avons distingué, dans ce contexte, trois catégories de couplage : (*i*) dérivation de UML vers B, (*ii*) développement conjoint UML/B, et finalement (*iii*) dérivation de B vers UML.

Nous avons axé notre travail de thèse autour de la dérivation de B vers UML et nous avons proposé des techniques conduisant à un progrès important, en comparaison avec les approches existantes et plus particulièrement les approches fondées sur des règles de traduction (Fekih *et al.,* 2004, Voisinet, 2004, Fekih *et al.,* 2006). Ce progrès peut être circonscrit dans deux critères majeurs pour notre objectif de validation externe de spécifications : (*i*) automatisation et (*ii*) lisibilité de la documentation graphique. En effet, la prospection des approches basées sur des règles de traduction nous a permis de constater que ces deux critères sont traités avec discordance étant donné que l'amélioration de l'un réduit significativement l'autre. Ce faisant, notre apport, se concrétise par un compromis entre automatisation et lisibilité, aussi bien pour la construction de diagrammes de classes que pour la construction de diagrammes d'états/transitions.

Contributions pour la construction de diagrammes de classes. Lors de la dérivation de diagrammes de classes, nous ne nous sommes pas basé sur une suite de règles de traduction directes de B vers UML

car une telle approche présente de nombreuses limites dues principalement au manque de formalisation des approches existantes et à la grande variété de constructions structurelles pouvant être dérivées à partir d'un modèle B. C'est pourquoi nous avons proposé de passer par la construction de modèles intermédiaires (dits modèles de contextes) satisfaisant des propriétés de pertinence formellement définies avant l'élaboration de règles de traduction. Nous soulignons alors que l'avantage de nos règles est leur précision car elles sont, d'une part, déterminées sur la base de la formalisation des modèles de contextes ; et d'autre part, délimitées par les correspondances que nous avons établies entre méta-modèles B et UML. Par ailleurs, l'analyse formelle de concepts que nous avons proposée nous a permis non seulement de disposer d'une technique semi-automatique pour la construction de diagrammes de classes, mais également de prouver et donc de démontrer que les modèles de contextes construits par notre algorithme de formation de concepts satisfont bien les propriétés de pertinence fixées au départ. Lesquelles propriétés sont énoncées de manière analogue au concept d'encapsulation d'attributs et de méthodes d'UML. Ce cadre théorique, de par sa nature homogène et précise, a fait l'objet de techniques d'optimisation dont l'objectif est, d'une part, de réduire au mieux l'aspect interactif, et d'autre part, de prendre en compte des développements B construits par raffinements. Finalement, les expérimentations que nous avons menées sur une étude de cas concrète nous ont permis de montrer que notre approche de dérivation de diagrammes de classes ainsi que les optimisations qui soutiennent cette approche permettent de passer de spécifications de tailles réduites, à des spécifications de tailles plus réelles et franchir ainsi un pas vers le passage en vraie en grandeur.

Contributions pour la construction de diagrammes d'états/transitions. Dans l'optique de construire, à partir de spécifications B, des diagrammes d'états/transitions, nous avons adopté, dans un premier temps, une approche exploratoire où nous avons cherché à montrer l'importance de distinguer entre vues concrètes et abstraites du comportement de la spécification. Cette distinction a permis d'expliciter, outre une sémantique de traces d'un modèle B[44], le comportement de la spécification selon plusieurs perspectives susceptibles d'être utiles pour différents intervenants d'un projet B. Dans un second temps, et sur la base du constat qu'une même spécification B peut être représentée selon différents points de vue, dépendant de l'abstraction choisie, nous avons proposé d'explorer les traces de la spécification pour en dériver les diagrammes abstraits. Contrairement aux approches (Fekih *et al.,* 2004, Voisinet, 2004, Fekih *et al.,* 2006) qui se limitent à une analyse syntaxique de la spécification pour la construction de diagrammes d'états/transitions, notre approche couvre un plus grand nombre de représentations graphiques et est grandement automatisée moyennant l'identification des meilleures abstractions. Par ailleurs, vu la complémentarité entre animation et preuve, nous pensons que notre outil d'abstraction de graphes d'accessibilité et l'outil `GeneSyst` (Potet *et al.,* 2004, Bert *et al.,* 2005) peuvent être utilisés conjointement pour une meilleure appréhension des vues comportementales.

Retombées sur les autres techniques de couplage. Bien que notre travail s'inscrive au niveau de la troisième catégorie du couplage entre B et UML, nous pensons que nos apports théorique et pratique contribuent également à enrichir les approches qui se situent au niveau des deux autres catégories :

- *Retombées sur une dérivation de* UML *vers* B : en général les travaux de dérivation de UML vers B n'exploitent pas un retour vers les modèles UML après la génération du modèle B par application des règles de traduction. En effet, dès que les spécifications formelles B sont corrigées en cas

[44] Cette sémantique de trace est initialement définie par M.-L. Potet et N. Stouls dans (Potet *et al.,* 2004).

d'inconsistance ou alors raffinées pour la prise en compte de contraintes liées au système et non spécifiées en UML, la traduction inverse n'est plus garantie. Nous pensons qu'un tel retour vers le modèle UML ne manque pas d'importance car il permet de visualiser directement au niveau du modèle UML de départ les changements apportés aux spécifications B. Nous n'avons pas effectué des expérimentations concrètes sur des spécifications B générées à partir d'un modèle UML. Néanmoins, nous envisageons de mener une telle étude lors de nos futurs travaux.

⋆ *Retombées sur un développement conjoint* UML/B : à ce jour, ce développement conjoint se base sur les règles de traduction de UML vers B, ce qui contraint l'évolution directe du modèle B aux constructions orientées objets issues de UML. Nous pensons que les schémas de traduction que nous avons définis par des correspondances du méta-modèle B vers le méta-modèle d'UML, peuvent être exploités et transformés en des *opérations de construction*[45] en vue de garantir une évolution directe du modèle B sans que celui-ci ne soit fortement contraint par des constructions orientées objets.

Expérimentations, Résultats & Discussion

La technique de formation de concepts qui fonde notre approche de construction de diagrammes de classes est presque totalement automatique. En effet, l'intervention de l'analyste n'est indispensable qu'au moment du choix des "cibles" pour les opérations qui disposent de plusieurs cibles possibles. Nous avons développé un outil prototype (Idani *et al.,* 2005b), support de l'approche, que nous avons expérimenté sur plusieurs études de cas de tailles variables et nous avons obtenu des résultats que nous considérons comme satisfaisants. Le tableau ci-dessous donne la taille de chaque étude de cas en termes de nombre de *BData*s, de nombre de *BOperation*s et de lignes de code B. Les colonnes "couverture des données" et "couverture des opérations" indiquent les proportions de données et d'opérations effectivement représentées au niveau des diagrammes produits par notre outil. Remarquons que dans la plupart des cas nous obtenons une couverture égale à 100%.

Machine B	BDatas	BOperations	lignes de code	couverture des opérations	couverture des données
BookStore	5	6	119	100%	100%
TravelAgency	23	10	296	100%	100%
Parking1	22	20	364	20%	64%
Parking2	25	24	462	21%	64%
SecureFlight0	5	2	80	100%	100%
SecureFlight1	10	2	142	100%	100%
SecureFlight2	12	3	158	100%	100%
SecureFlight3	15	4	192	100%	100%
SecureFlight4	17	6	222	100%	100%

Certaines spécifications, comme *Parking*1 et *Parking*2 donnent des résultats moins satisfaisants vu la couverture partielle des concepts et des opérations. Cette constatation est justifiée par le fait que ces spécifications présentent un grand nombre de données qui ne sont ni des ensembles ni des relations, mais qui sont principalement des éléments d'ensembles. Ceci montre que notre outil prototype de dérivation

[45] La notion d'opérations de construction est définie par (Ossami *et al.,* 2004). Pour plus de détails voir le chapitre 2.

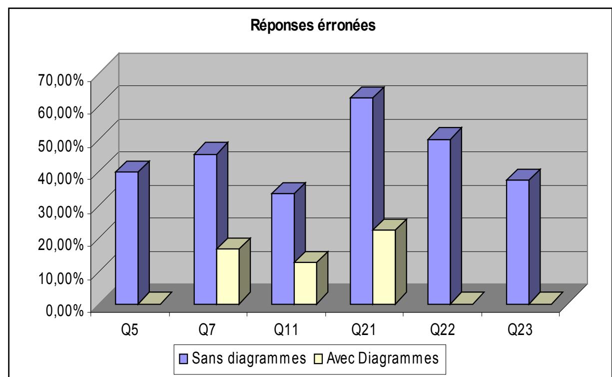
de diagrammes de classes privilégie un certain type de spécifications B en entrée. En effet, les spécifications qui ne sont que partiellement couvertes correspondent plutôt à des systèmes dont la spécification comportementale est plus largement développée que la structuration des données. De ce fait, nous distinguons spécifications formelles orientées données et spécifications formelles orientées comportements. Bien que nos outils permettent de traiter distinctement ces deux familles de spécifications, nous pensons que pour garantir une plus grande couverture de spécifications, celles-ci doivent intégrer aussi bien le caractère structurel que le caractère comportemental (*e.g.* la spécification du contrôleur d'accès Annexe D, Fig. D.5 page 258).

Cependant, ces résultats portent uniquement sur la couverture des données et d'opérations, et ne donnent, par conséquent, qu'une évaluation quantitative de nos résultats. Pour une évaluation qualitative, nous avons mené une étude empirique auprès de deux groupes[46] d'étudiants A et B (un total de 17 étudiants) en Master 1 − Informatique (spécialité : Génie Logiciel) et venant de clôturer un cours semestriel de modélisation formelle en B. Cette étude a été réalisée au moyen d'un questionnaire (Annexe D) et s'est déroulée en 2 étapes traitant chacune une spécification B, et telles que dans chaque étape, la spécification est documentée pour un groupe et non documentée pour l'autre groupe. Au final, chaque étudiant a disposé de deux listes de questions sur deux spécifications B différentes, dont une seule a été étayée par des diagrammes. Les deux spécifications traitées correspondent respectivement à un gestionnaire de processus (*i.e.* SCHEDULER page 254) et à un contrôleur d'accès (*i.e.* ACCESSCONTROL page 258).

Compréhension des spécifications. Le tableau ci-dessous représente les proportions de réponses erronées et correctes aux questions posées et permet d'observer une baisse du taux d'erreurs de 26.14% à 15.60% lorsque les questions sont accompagnées de diagrammes.

Réponses			
Sans Diagrammes		**Avec Diagrammes**	
Erronées	**Correctes**	**Erronées**	**Correctes**
$40 = 26.14\%$	$113 = 73.86\%$	$22 = 15.60\%$	$119 = 84.40\%$

Nous avons également pu constater que pour certaines questions les diagrammes ont permis de réduire considérablement ce taux d'erreurs. Le graphe ci-dessous, illustre les taux d'erreurs enregistrés pour les questions dont une différence de 2 erreurs au minimum a été constatée entre les réponses basées sur des diagrammes et les réponses sans diagrammes[47].

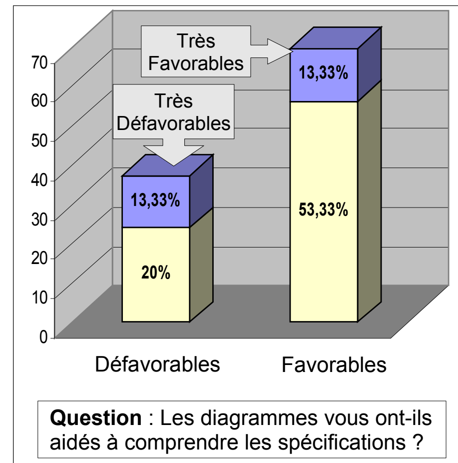


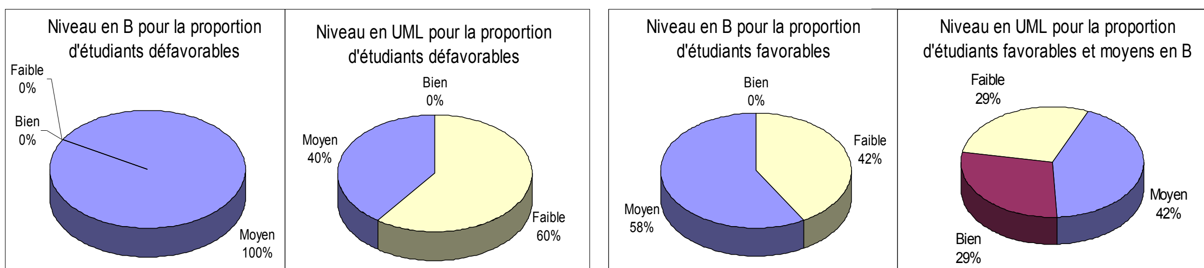
[46] Les étudiants sont répartis sur les deux groupes selon leurs notes d'examen et de telle sorte qu'il n'y ait pas d'écart entre les groupes.

[47] Les autres questions disposent de taux d'erreurs presque équivalents.

Nous avons aussi observé que parfois les diagrammes conduisent à des réponses incorrectes vu qu'ils ne visualisent pas toute la spécification et restent incomplets (*e.g.* le diagramme de classes de la Fig. D.6 (page 259) et la question **Q19**). Un recours à la spécification est donc obligatoire pour pouvoir répondre avec certitude sur certaines questions.

Perception des étudiants. Nous mettons l'accent au niveau du graphe ci-contre[48] sur les proportions d'étudiants d'avis favorables et d'avis non favorables au fait que les diagrammes ont servi pour la compréhension des spécifications. Nous remarquons qu'une grande majorité des étudiants interrogés (*i.e.* 66.66%) ont été convaincus de l'utilité des vues graphiques pour mieux répondre aux questions qui leur ont été posées, alors que 33.33% approuvent plutôt l'opinion contraire. En vue d'expliquer ce pourcentage, nous notons que 100% des étudiants de cette dernière partition estiment "Moyen" leur niveau en B (voir figure ci-dessous). Ce niveau de maîtrise de B, jugé correct de ces étudiants, permet, entre autres, de justifier leur opinion défavorable sur notre question. Cette constatation est confirmée également par la partition d'étudiants ayant adhéré à notre opinion et dont un grand nombre (42%) évaluent leur niveau en B comme "Faible", d'où l'importance pour ces derniers de disposer d'éléments d'aide à la compréhension de spécifications. Par ailleurs, le fait que les étudiants dont le niveau en B est estimé "Moyen" se départagent entre les partitions favorable et défavorable ne remet pas en cause notre constatation car nous pensons que c'est le niveau de maîtrise d'UML qui a été l'élément déterminant dans cette distribution. En effet, nous notons que 60% des étudiants dont l'avis est défavorables jugent comme "Faible" leur niveau en UML et que les autres se veulent de niveau "Moyen". En revanche, sur les 66.66% d'étudiants favorables, nous avons constaté que 71% présentent un niveau en UML allant de "Moyen" à "Bien". Cette observation nous amène donc à songer qu'une certaine maîtrise des diagrammes de UML, précisément des diagrammes de classes et d'états/transition, est requise en vue de pouvoir étudier l'incidence des représentations graphiques sur la compréhension des spécifications B.





Finalement, les deux études quantitatives et qualitatives que nous avons menées aussi bien en interne via l'expérimentation de notre outil sur des spécifications variées, qu'en externe auprès d'une population de personnes ne maîtrisant pas nécessairement B, présentent des résultats encourageants. Nous sommes bien conscient que dans le cadre de notre étude empirique, la taille de l'échantillon (17 étudiants) est

[48] Il s'agit là de la question **Q27** de l'annexe D. Les parties en gris foncé correspondent aux étudiants très défavorables et très favorables.

assez restreinte, et que certaines questions, comme le fait de fournir aux étudiants des spécifications commentées, restent à débattre. Cependant, ce premier questionnaire et cette expérimentation prototype nous permettent d'envisager une étude sur des bases plus rigoureuses (Singer, 2006) et une population plus vaste d'étudiants et d'ingénieurs.

Perspectives

Lors de notre investigation, nous avons abordé notre problématique de recherche dans une optique de complétude et d'homogénéité. Pour ce faire, nous avons exploré plusieurs voies interdépendantes d'un côté, et d'un autre côté, orientées vers deux types de diagrammes : diagrammes structurels et diagrammes comportementaux. Bien que ces voies aient débouché sur des contributions importantes en comparaison avec les approches existantes, elles présentent certaines limites et conduisent, par conséquent, vers de nombreuses perspectives prometteuses pour un objectif d'aide à la validation de spécifications formelles.

A. Perspectives liées à la dérivation de diagrammes de classes

Les perspectives liées à la dérivation de diagrammes de classes portent sur quatre axes principaux : (*i*) le cadre conceptuel à partir duquel nous avons défini nos transformations, (*ii*) la technique de formation de concepts, (*iii*) les optimisations, et (*iv*) la prise en compte de développements modulaires.

- ⋆ **Vers une intégration par méta-modélisation automatisée et riche.** Les schémas de transformation que nous avons définis sur la base de correspondances entre méta-modèles présentent une source exploitable par les outils de transformation de modèles préconisés par l'approche MDA (Bézivin *et al.,* 2001, OMG, 2003). Une telle démarche a été étudiée par S.-K. Kim et ses co-auteurs (Kim *et al.,* 2005) lors de leur tentative d'explicitation d'une traçabilité entre UML et Object-Z et a montré son efficacité lorsque chaque construction du modèle formel de départ est transformée de manière unique. Dans le cadre de nos transformations, le recours a une telle démarche peut s'avérer aisé car cela ne nécessitera qu'une implantation adéquate, au niveau des outils MDA existants, des méta-modèles B et UML, ainsi que du catalogue de schémas de transformation déjà mis au point. L'apport d'une telle démarche serait de disposer d'un cadre transformationnel réutilisable et facile à enrichir pour la couverture de plus de constructions en B et la prise en compte d'autres concepts UML (*e.g.* composition, héritage multiple ...). En revanche, l'aspect interactif serait prépondérant car pour certaines constructions B, plusieurs transformations peuvent exister.

- ⋆ **Vers une analyse formelle de concepts élargie.** La technique de formation de concepts que nous avons proposée et que nous avons consolidée par une formalisation dont les fondements sont inspirées de l'AFC (Bernhard *et al.,* 1999), a permis d'étudier nos transformations avec rigueur et de manière potentiellement automatisable. Ce faisant, nous avons pu établir quelques traits de conciliation entre approches par méta-modélisation (ou *interprétées*) et approches par traduction directe (ou *compilées*). Plusieurs travaux de transformation de modèles n'exploitent pas, par exemple, l'existence de facteurs de pertinence lorsque diverses transformations peuvent exister pour une même construction du modèle source. Ces travaux adoptent en général une approche restrictive où plusieurs hypothèses sont énoncées au départ. Par exemple, dans (Lemesle, 2000) une classe dans un modèle UML est uniquement traduite par une table dans un modèle relationnel, alors que certains voudront, par exemple, exclure les classes abstraites de la transformation en tables,

et d'autres chercheront un mélange des deux selon le besoin. Nous pensons que notre analyse formelle de concepts peut être adaptée et élargie, en perspective, pour toucher des techniques de transformation de modèles relevant d'autres domaines et aider ainsi à mieux cerner l'existence d'une multitude de transformations. Dans cette même optique, nous pensons qu'une telle adaptation pourrait servir également pour une documentation en UML d'autres formalismes, notamment du code source. Aujourd'hui, l'AFC est principalement utilisée pour la restructuration de code, sans que l'intérêt primordial ne soit l'aide à la compréhension ni l'assistance à la validation externe via des modèles UML.

⋆ **Vers une optimisation assistée et éprouvée.** Plusieurs traits liés à l'utilisation des matrices de liens peuvent être améliorés pour une meilleure optimisation de notre technique de construction de diagrammes de classes. Nous avons déjà mentionné la proposition d'autres types de pondérations telles que, par exemple, l'attribution de priorités aux liens de dépendances entre *BData*s elles-mêmes (*i.e.* relation *Incl*). Dans ce cadre, nous projetons de répertorier, au moyen d'expérimentations, les mesures susceptibles de produire des classifications pertinentes des diagrammes de classes construits pour une même spécification. Ceci permet de proposer plus de choix de métriques à l'analyste lors de la phase d'optimisation et aboutir par conséquent à diverses évaluations des diagrammes de classes. Étant donné que l'approbation des métriques par l'analyste à ce niveau est fondamentale, nous songeons alors à déterminer d'autres techniques d'optimisation, ne se limitant pas à l'utilisation de matrices de liens, mais portant sur une évaluation directe des diagrammes de classes. Il s'agit précisément, d'expérimenter des métriques déjà approuvées telles que les “CK metrics” proposées dans (Chidamber *et al.,* 1994) pour l'évaluation de modèles orientés objets.

⋆ **Vers une prise en compte efficace des liens de composition.** Actuellement, les machines composées par les clauses INCLUDES, SEES et USES passent par une phase d'aplatissement avant d'en dériver des diagrammes de classes. Par conséquent, notre algorithme de formation de concepts traite les données B issues des différentes machines de manière identique. Cette démarche ne paraît pas très efficace pour un objectif de documentation car les données B n'ont pas été initialement rattachées à une machine unique. Pour aborder ce point, nous pensons que la composition de machines B peut se ramener à une composition de réseaux de concepts ; et donc un ajustement de notre algorithme de formation concepts à des réseaux composites peut améliorer grandement notre technique.

B. Perspectives liées à la dérivation de diagrammes d'états/transitions

Notre technique d'abstraction de graphes d'accessibilité présente une limite assez connue par les techniques d'animation et de model-checking : être destinée en particulier aux systèmes finis. Dans le cadre de ces systèmes, notre technique s'avère bien adaptée en comparaison avec les techniques existantes car l'exploration exhaustive du comportement de la spécification est aisée et assez bien outillée aujourd'hui. Cependant, pour aborder des systèmes infinis l'approche **`GeneSyst`** (Potet *et al.,* 2004, Bert *et al.,* 2005) est mieux adaptée et peut être complétée par notre technique d'animation pour combler certains points faibles liés principalement aux preuves et à la surcharge de diagrammes. L'une des perspectives déjà citées et ayant pour visée de tirer profit des apports complémentaires de l'animation et la preuve, est l'intégration de notre outil au sein de l'outil **`GeneSyst`**. Par ailleurs, nous pensons que dans cette même perspective, d'autres techniques sont exploitables et peuvent être également d'une grande

utilité ; nous évoquons, à titre d'exemple, l'animation symbolique (Bouquet, 2005). Une autre perspective qui ne manque pas d'importance pour notre objectif de dérivation de diagrammes d'états/transitions est l'automatisation du procédé d'identification des états abstraits. En effet, à ce jour nous nous sommes limité à un guide méthodologique portant sur : (*i*) le recours aux données de la spécification, (*i*) le recours aux besoins informels de l'utilisateur, et (*iii*) la définition d'un catalogue de patrons d'états abstraits. Dans ce contexte, nous pensons que l'interprétation abstraite (Cousot *et al.,* 2003) peut amener des bénéfices intéressants.

C. Vers une démarche pour les SI

D'une part, les travaux de dérivation de UML vers B cherchent à surmonter les limites d'UML grâce au langage B ; et d'autre part, les travaux de dérivation de B vers UML tentent de combler les failles de B grâce à la notation UML. Que l'on exploite un sens de dérivation ou l'autre, ou que l'on fasse de l'UML étendu par du B (Attiogbé *et al.,* 2003) ou du B orienté objet (Malioukov, 1998), le modèle obtenu reste sujet aux critiques classiques de chacune de ces notations et les cultures de développement restent, par conséquent, inchangées pour des systèmes de grande taille, et plus particulièrement les systèmes d'information. En effet, pour ces derniers, UML (ou toute autre méthode semi-formelle) restera largement diffusé alors que B (ou toute autre méthode formelle) ciblera encore les parties critiques où sûreté et sécurité sont fondamentales. Dans une telle conjoncture, garantir ce "va-et-vient" entre B et UML, de manière unifiée et dans un environnement homogène, est d'une extrême importance car cela permettrait d'harmoniser les parties formelles et semi-formelles du système d'information et d'étudier, de manière fondée, leurs concordances.

Bibliographie

Abrial J.-R., *The B-book : assigning programs to meanings*, Cambridge University Press, 1996.

Aertryck L. V., Benveniste M., Metayer D. L., « CASTING : A Formally Based Software Test Generation Method », *ICFEM'97, First IEEE International Conference on Formal Engineering Methods*, Hiroshima, 1997.

Amálio N., Polack F., « Comparison of Formalisation Approaches of UML Class Constructs in Z and Object-Z », *ZB 2003 : Formal Specification and Development in Z and B : Third International Conference of B and Z Users*, vol. 2651 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 339-358, 2003.

André P., Méthodes formelles à Objets pour le développement du logiciel : études et propositions, PhD thesis, Université de Rennes 1, Juillet, 1995.

André P., Barbier F., , Royer J.-C., « Une expérimentation de développement formel à objets », *Technique et Science Informatique*, vol. 14, n° 8, pp. 973-1005, Octobre, 1995. Hermés-Lavoisier.

Attiogbé C., Poizat P., Salaün G., « Integration of Formal Datatypes within State Diagrams », *Fundamental Approaches to Software Engineering (FASE'2003)*, vol. 2621 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 341-355, 2003.

Badeau F., Bert D., Boulmé S., Métayer C., Potet M.-L., Stouls N., Voisin L., « Traduction de B vers des langages de programmation », *TSI - Approches formelles pour le développement de logiciels*, vol. 23, pp. 879-903, 2004.

Baldwin C. Y., Clark K. B., *Design Rules : The Power of Modularity*, The MIT Press, 2000.

Behm P., Benoit P., Faivre A., Meynadier J.-M., « METEOR : A successful application of B in a large project », *FM'99 : World Congress on Formal Methods*, vol. 1709 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 369-387, 1999.

Behm P., Desforges P., Mejia F., *Application de la méthode B dans l'industrie ferroviaire*, Lavoisier, chapter 3 of "Application des techniques formelles au logiciel", pp. 59-88, 1997. Ofta editor.

Behnia S., Test de modèles formels en B : cadre théorique et critères de couvertures, PhD thesis, Institut National Polytechnique de Toulouse, October, 2000.

Ben Cheikh A., Graphical Visualisation of formal B specifications with Animation Tools, Rapport de stage, École Plytechnique de Tunisie, 2004. www-lsr.imag.fr/users/Akram.Idani/ansem_report.pdf.

Bernhard G., Rudolf W., *Formal concept analysis*, Springer, 1999.

Bert D., Bouquet F., Ledru Y., Vignes S., « Validation of Regulation Documents by Automated Analysis of Formal Models », *International Workshop on Regulations Modelling and their Validation*

and Verification (REMO2V'06), In conjunction with CAiSE'06, Presses Universitaires de Namur, Luxembourg, Juin, 2006.

Bert D., Cave F., « Construction of Finite Labelled Transition Systems from B Abstract Systems », *IFM'2000 : Second International Conference of Integrated Formal Methods*, vol. 1945 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 235-254, 2000.

Bert D., Potet M.-L., « Spécifications en B (support de cours) », 2004, École des Jeunes Chercheurs en Programmation.

Bert D., Potet M.-L., Stouls N., « GeneSyst : a tool to reason about behavioral aspects of B event specifications. Application to security properties », *Proceedings of 4th International Conference of B and Z users*, vol. 3455 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 299-318, 2005.

Blanc X., *MDA en action*, Eyrolles, 2005.

Booch G., Rumbaugh J., Jacobson I., *The Unified Modeling Language User Guide*, Addison Wesley Longman Publishing Co., Inc., 2002.

Boulton R., Gordon A., Gordon M., Harrison J., Herbert J., Tassel J. V., « Experience with embedding hardware description languages in HOL », *in* V. Stavridou, T. F. Melham, R. T. Boute (eds), *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design : Theory, Practice and Experience*, vol. A-10 of *IFIP Transactions A : Computer Science and Technology*, North-Holland, Nijmegen, The Netherlands, pp. 129-156, 1993.

Bouquet F., *« Sémantique interprétative de spécifications formelles en animation symbolique et génération de tests à partir de modèles »*, Hdr, Université de Franche-Comté, 2005.

Bouquet F., Legeard B., Peureux F., « CLPS-B - A Constraint Solver for B. », *8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 2280 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 188-204, 2002.

Bowen J. P., Hinchey M. G., « Seven More Myths of Formal Methods », *IEEE Software*, vol. 12, n° 4, pp. 34-41, July, 1995.

Bruel J.-M., « Integrating Formal and Informal Specification Techniques. Why ? How ? », *Proceedings of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98)*, IEEE Computer Society Press, Florida, pp. 50-57, January, 1999.

Bézivin J., Gerbé O., « Towards a Precise Definition of the OMG/MDA Framework », *International Conference on Automated Software Engineering*, IEEE Computer Society Press, San Diego, USA, pp. 273-280, 2001.

Casset L., « Development of an Embedded Verifier for Java Card Byte Code Using Formal Methods », *FME'02, Formal Methods Europe*, vol. 2391 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 290-309, 2002.

Chen Y., Miao H., « From an Abstract Object-Z Specification to UML Diagrams », *Journal of Information & Computational Science*, vol. 1, n° 2, pp. 319-324, 2004. Binary Information Press. http ://www.joics.com.

Cheung K., Chow K., Cheung T., « Extending Formal Specification To Object-Oriented Models Through Level-View Structured Schemas », *31st International Conference on Technology of Object-Oriented Language and Systems*, IEEE Computer Society Press, pp. 118-126, 1999.

Chidamber S. R., Kemerer C. F., « A Metrics Suite for Object Oriented Design », *IEEE Transactions on Software Engineering*, vol. 20, n° 6, pp. 476-493, 1994.

Clarke E. M., Grumberg O., Peled D. A., *Model Checking*, MIT Press, 1999.

ClearSy, *Animateur : Manuel Utilisateur*, Version 3.6, 2002a.

ClearSy, *Manuel de Référence du langage B*, Version 1.8.5, 2002b.

Cousot P., Cousot R., « Parsing as Abstract Interpretation of Grammar Semantics », *Theoretical Computer Science*, vol. 290, pp. 531-544, 2003.

Craigen D., Gerhart S., Ralston T., « Formal Methods Reality Check : Industrial Usage », *IEEE Transactions on Software Engineering*, vol. 21, n° 2, pp. 90-98, 1995.

Davis A., *Software Requirements : Objects, Functions, and States*, Prentice-Hall, 1993.

Dick J., Faivre A., « Automating the Generation and Sequencing of Test Cases from Model-Based Specifications », *in* J. C. P. Woodcock, P. G. Larsen (eds), *FME'93 : Industrial Strength, Formal Methods*, vol. 670 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 268-284, 1993.

Dick J., Loubersac J., « Integrating structured and formal methods : A visual approach to VDM », *in* A. van Lamsweerde, A. Fugetta (eds), *Proceedings of European Software Engineering Conference (ESEC'91)*, vol. 550 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 37-59, 1991.

Dijkstra E. W., « Guarded Commands, Nondeterminacy and Formal Derivation of Programs », *Communication of the ACM*, vol. 18, n° 8, pp. 453-457, 1975.

Dijkstra E. W., *A Discipline of Programming*, Prentice Hall, 1976.

Dürr E., van Katwijk J., « VDM++ : a formal specification language for object-oriented designs », *Proceedings of the seventh international conference on Technology of object-oriented languages and systems*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, pp. 63-77, 1992.

Dupuy-Chessa S., Couplage de notations semi-formelles et formelles pour la spécification des systèmes d'information, PhD thesis, Université Joseph Fourier, Septembre, 2000.

Essamé D., « La méthode B et l'ingénierie système », *TSI, (Technique et Science Informatiques)*, vol. 23, n° 7, pp. 929-938, 2004. Hermès.

Etienne J.-F., Delahaye D., Donzeau-Gouge V. V., « Modeling airport security regulations in Focal », *International Workshop on Regulations Modelling and their Validation and Verification (REMO2V'06), In conjunction with CAiSE'06*, Presses Universitaires de Namur, Luxembourg, Juin, 2006.

Facon P., Laleau R., « Des spécifications informelles aux spécifications formelles : compilation ou interprétation ? », *Actes du 13ème congrès INFORSID*, 1995.

Fecher H., Schönborn J., Kyas M., Roever W.-P., « 29 New Unclarities in the Semantics of UML 2.0 State Machines », *Formal Methods and Software Engineering : 7th International Conference on Formal Engineering Methods, ICFEM 2005*, vol. 3785 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 52-65, 2005.

Fekih H., Jemni L., Merz S., « Transformation des spécifications B en des diagrammes UML », *in* J. Julliand (ed.), *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2004)*, pp. 131-148, Juin, 2004.

Fekih H., Jemni L., Merz S., « Transformation of B Specifications into UML Class Diagrams and State Machines », *21st Annual ACM Symposium on Applied Computing*, ACM, pp. 1840-1844, Avril, 2006.

Fowler M., *UML Distilled - Third Edition - A Brief Guide to the Standard Object Modeling Language*, Addison Wesley Professional, 2003.

Fowler M., *UML 2.0*, CampusPress, 2004.

Fraser M. D., Kumar K., Vaishnavi V. K., « Informal and formal requirements specification laguages : Bridging the gap », *IEEE Transactions on Software Engineering*, vol. 17, n° 5, pp. 454-465, 1991.

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns : Catalogue de modèles de conception réutilisables*, Vuibert, 1999.

Ganter B., Wille R., « Conceptual scaling », *in* F. Roberts (ed.), *Applications of combinatorics and graph theory to the biological and social sciences*, Springer-Verlag, New York, USA, pp. 139-167, 1989.

Gaudel M.-C., « Advantages and limits of formal approaches for ultra-high dependability », *IWSSD'91 : Proceedings of the 6th international workshop on Software specification and design*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 237-241, 1991.

Gaudel M.-C., « Advantages and Limits of Formal Approaches for Ultra-High Dependability », *in* B. Randell, J. C. Laprie, H. Kopetz, B. Littlewood (eds), *Predictably Dependable Computing Systems*, Springer Verlag, pp. 241-251, 1995. ESPRIT Basic Research Series.

Godin R., Mineau G., Missaoui R., Mili H., « Méthodes de classification conceptuelle basées sur les treillis de Galois et applications », *Intelligence Artificielle*, vol. 9, n° 2, pp. 105-137, 1995a.

Godin R., Mineau G., Missaoui R., St-Germain M., Faraj N., « Applying Concept Formation Methods to Software Reuse », *International Journal of Knowledge Engineering and Software Engineering*, vol. 5, n° 1, pp. 387-410, 1995b.

Godin R., Missaoui R., Alaoui H., « Incremental concept formation algorithms based on Galois (concept) lattices », *Computational Intelligence*, vol. 11, pp. 246-267, 1995c.

Groh B., Eklund P. W., « Algorithms for Creating Relational Power Context Families from Conceptual Graphs », *ICCS '99 : Proceedings of the 7th International Conference on Conceptual Structures*, Springer-Verlag, London, UK, pp. 389-400, 1999.

Habrias H., *Spécifications avec B "The art of progress is to preserve order amid change and to preserve change amid order (Alfred North Whitehead)"*, Université de Nantes, I.U.T. de Nantes, Département informatique, 2006.

Hall A., « Seven Myths of Formal Methods », *IEEE Software*, vol. 7, n° 5, pp. 11-19, September, 1990.

Hallerstede S., « Parallel hardware design in B », *ZB 2003 : Third International Conference of B and Z Users*, vol. 2651 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 101-102, 2003.

Hamdane S., Génération de systèmes de transition étiquetés à partir de la description d'un système d'évènements décrits avec le langage B, Rapport de licence, Université Joseph Fourrier, Grenoble-1, France, 2002.

Hamdane S., Système de transitions d'un système abstrait : méthode de calcul des états, Rapport de maîtrise, Université Joseph Fourrier, Grenoble-1, France, 2003.

Hammad A., Tatibouët B., Voisinet J.-C., Wu W., « From a B Specification to UML StateChart Diagrams », *International Conference on Formal Engineering Methods*, vol. 2495 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 511-522, 2002.

Harel D., « STATECHARTS : a visual formalism for complex systems », *Science of Computer Programming*, 1987.

Hazem L., Levy N., Marcano-Kamenoff R., « UML2B : un outil pour la génération de modèles formels », *in* J. Julliand (ed.), *AFADL'2004 - Session Outils*, 2004.

Hoare C. A. R., « An axiomatic basis for computer programming », *Communication of the ACM*, vol. 12, n° 10, pp. 576-580, 1969.

Idani A., « Couplage de spécifications B et de descriptions UML pour l'aide aux développements formels des systèmes d'information : Approche par méta-modélisation », *Actes du 24ème congrés INFORSID*, Tunisie, Juin, 2006.

Idani A., Ledru Y., « Object Oriented Concepts Identification from Formal B Specifications », *Proceedings of 9th International Workshop on Formal Methods for Industrial Critical Systems(FMICS'04)*, vol. 133 of *ENTCS*, Elsevier, pp. 159-174, 2005a.

Idani A., Ledru Y., « Dynamic Graphical UML Views from Formal B Specifications », *International Journal of Information and Software Technology*, vol. 48, n° 3, pp. 154-169, Mars, 2006a. Elsevier.

Idani A., Ledru Y., « Object Oriented Concepts Identification from Formal B Specifications », *International Journal of Formal Methods in System Design*, 2006b. Numéro Spécial FMSD/FMICS2004, à paraître.

Idani A., Ledru Y., Bert D., « Derivation of UML Class Diagrams as Static Views of Formal B Developments », *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005*, vol. 3785 of *Lecture Notes in Computer Science*, Springer-Verlag, Manchester, UK, pp. 37-51, November, 2005b.

Idani A., Ledru Y., Bert D., « A Reverse-Engineering Approach to Understanding B Specifications with UML Diagrams », *Proceedings of 30th Annual IEEE/NASA Software Engineering Workshop (SEW-30)*, IEEE Computer Society Press, USA, April, 2006c.

Idani A., Ledru Y., Bert D., « Analyse formelle de concepts pour la génération de diagrammes de classes UML à partir de spécifications B », *Technique et Sciences Informatique*, 2006d. Numéro Spécial TSI/AFADL2006, version étendue en cours de soumission.

Idani A., Ledru Y., Bert D., « Analyse formelle de concepts pour la génération de diagrammes de classes UML à partir de spécifications B », *Actes de la 7ème conférence AFADL - Approches Formelles dans l'Assistance au Développement de Logiciels*, pp. 9-23, 2006e.

Junior J. C. F., Ingénierie des Systèmes d'Information : une approche de multi-modélisation et de méta-modélisation, PhD thesis, Université de Grenoble 1, 1997.

Kim S.-K., Burger D., Carrington D., « An MDA Approach Towards Integrating Formal and Informal Modeling Languages », *Formal Methods*, vol. 3582 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 448-464, 2005.

Kim S.-K., Carrington D., « Visualization of Formal Specifications », *Sixth Asia-Pacific Software Engineering Conference (APSEC'99)*, IEEE Computer Society Press, pp. 102-110, 1999.

Kim S.-K., Carrington D., Duke R., « A Metamodel-based transformation between UML and Object-Z », *IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, IEEE Computer Society Press, pp. 112-122, 2001.

Laleau R., *« Conception et développement formels d'applications bases de données »*, Hdr, Université d'Evry, 2002.

Laleau R., Mammar A., « An Overview of a Method and Its Support Tool for Generating B Specifications from UML Notations », *15th IEEE International Conference on Automated Software Engineering*, pp. 269-272, 2000. IEEE Computer Society Press.

Laleau R., Polack F., « Specification of integrity-preserving operations in information systems by using a formal UML-based language », *Journal of Information and Software Technology*, vol. 43, n° 12, pp. 693-704, November, 2001. Elsevier.

Laleau R., Polack F., « Coming and Going from UML to B : A Proposal to Support Traceability in Rigorous IS Development. », *2nd International Conference of B and Z Users*, vol. 2272 of *Lecture Notes in Computer Science*, Springer, pp. 517-534, 2002.

Laleau R., Vignes S., Ledru Y., Lemoine M., Bert D., Donzeau-Gouge V., Dubois C., , Peureux F., « Adopting a situational requirements engineering approach for the analysis of civil aviation security standards », *Software Process : Improvement and Practice*, 2006. Accepted for publication in February 2006.

Lano K., « Z^{++} », *in* S. Stepney, R. Barden, D. Cooper (eds), *Object Orientation in Z*, Workshops in Computing, Springer, pp. 105-112, 1992.

Lano K., *Formal Object-Oriented Development*, Springer-Verlag New York, Inc., USA, 1995.

Lano K., *The B Language and Method : A Guide to Practical Formal Development*, Springer, 1996.

Lano K., Androutsopolous K., Clark D., « Refinement Patterns for UML », *Electronic Notes in Theoretical Computer Science*, vol. 137, n° 2, pp. 131-149, July, 2005.

Lano K., Clark D., Androutsopoulos K., « UML to B : Formal Verification of Object-Oriented Models », *Integrated Formal Methods*, vol. 2999 of *Lecture Notes in Computer Science*, Springer, pp. 187-206, 2004.

Le Guennec A., Génie logiciel et méthodes formelles avec UML, Spécification, Validation et Génération de tests, PhD thesis, Université de Rennes 1, Juin, 2001.

Lecourt D., *Dictionnaire d'histoire et philosophie des sciences*, PUF, 1999.

Ledang H., Traduction systématique de spécifications UML en B, PhD thesis, Université de Nancy 2, 2002.

Ledang H., Souquières J., Charles S., « Argo/UML+B : un outil de transformation systématique de spécification UML en B », *AFADL'2003*, pp. 3-18, January, 2003.

Ledru Y., Laleau R., Lemoine M., Vignes S., Bert D., Donzeau-Gouge V., Dubois C., Peureux F., « An attempt to combine UML and formal methods to model airport security », *18th International Conference on Advanced Information Systems Engineering (CAiSE'06)*, Luxembourg, Juin, 2006.

Legeard B., Peureux F., Utting M., « Automated boundary testing from Z and B », *FME'02, Formal Methods Europe*, vol. 2391 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002a.

Legeard B., Peureux F., Utting M., « A Comparison of the LIFC/B and TTF/Z Test-Generation Methods », *ZB 2002 : 2nd International Conference of B and Z Users*, vol. 2272 of *Lecture Notes in Computer Science*, Springer-Verlag, Grenoble, France, pp. 309-329, January, 2002b.

Lemesle R., Techniques de Modélisation et de Méta-modélisation, PhD thesis, École centrale de Nantes, Octobre, 2000.

Leuschel M., Butler M., « ProB : A Model Checker for B », *FME 2003 : Formal Methods Europe*, vol. 2805 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 855-874, 2003.

Malioukov A., « An Object-Based Approach to the B Formal Method », *Second International B Conference on Recent Advances in the Development and Use of the B Method*, Springer-Verlag, London, UK, pp. 162-181, 1998.

Mammar A., Un environnement formel pour le développement d'applications bases de données, PhD thesis, CNAM-Paris, Novembre, 2002.

Mammar A., Laleau R., « UML2SQL : Un environment intégré pour le développement d'implémentations relationnelles à partir de diagrammes UML », *in* J. Julliand (ed.), *AFADL'2004 - Session Outils*, 2004.

Mammar A., Laleau R., « From a B formal specification to an executable code : application to the relational database domain », *Journal of Information and Software Technology*, vol. 48, n° 4, pp. 253-279, Avril, 2005. Elsevier.

Marcano R., Spécification formelle à objets en UML/OCL et B : une approche transformationnelle, PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2002.

Mariano G., Évaluation de logiciels critiques développés par la méthode B : Une approche quantitative, PhD thesis, Université de Valenciennes, Décembre, 1997.

Marvie R., Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants, PhD thesis, Université des Sciences et Technologies de Lille, 2002.

Meyer E., Développements formels par objets : utilisation conjointe de B et d'UML, PhD thesis, Université de Nancy 2, Mars, 2001.

Minsky M. L., *Semantic Information Processing*, The MIT Press, 1968.

Muller P.-A., Geartner N., *Modélisation Objet avec UML*, Eyrolles, 2001.

Nguyen H. P., Dérivation de spécifications formelles B à partir de spécifications semi-formelles, PhD thesis, CNAM-Paris, Décembre, 1998.

OMG, *MDA Guide Version 1.0.1*, 2003. www.omg.org/docs/omg/03-06-01.pdf.

OMG, *Unified Modeling Language : Superstructure (version 2.0)*, Object Management Group, August, 2005. www.omg.org/docs/formal/05-07-04.pdf.

Ossami D. D. O., Jacquot J.-P., Souquières J., « Consistency in UML and B Multi-view Specifications », *5th International Conference on Integrated Formal Methods*, vol. 3771 of *Lecture Notes in Computer Science*, Springer, pp. 386-405, 2005.

Ossami D. D. O., Souquières J., Jacquot J.-P., « Opérations de construction de spécifications multi-vues UML et B », *in* J. Julliand (ed.), *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2004)*, pp. 115-130, Juin, 2004.

Petit D., Génération automatique de composants logiciels sûrs à partir de spécifications formelles B, PhD thesis, Université de Valenciennes, Décembre, 2003.

Peureux F., Génération de tests aux limites à partir de spécifications B en Programmation Logique avec Contraintes ensemblistes, PhD thesis, Université de Franche-Comté, Décembre, 2002.

Pons C., « Heuristics on the Definition of UML Refinement Patterns. », *SOFSEM*, vol. 3831 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 461-470, 2006.

Potet M.-L., *Une approche B pour le contrôle d'accés*, Mars, 1998. www-lsr.imag.fr/afadl2000/EtudeDeCas/Contributions/Solutions_B/Potet.ps.

Potet M.-L., *« Spécifications et développements formels : Étude des aspects compositionnels dans la méthode B »*, Hdr, INPG, 2002.

Potet M.-L., Stouls N., « Explicitation du contrôle de développement B événementiel », *in* J. Julliand (ed.), *AFADL'2004*, pp. 13-27, June, 2004.

Pouzancre G., « How to Diagnose a Modern Car with a Formal B Model ? », *ZB 2003 : Formal Specification and Development in Z and B*, vol. 2651 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 98-100, 2003.

Pouzancre G., Pitzalis J.-P., « Modélisation en B événementiel des fonctions mécaniques, électriques et informatiques d'un véhicule », *TSI, (Technique et Science Informatiques)*, vol. 22, n° 1, pp. 119-128, 2003. Hermès.

Py L., Legeard B., Tatibouët B., « Évaluation de spécifications formelles en programmation logique avec contraintes ensemblistes – application à l'animation de spécifications formelles B », *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2000)*, pp. 21-35, 2000.

Reggio G., Wieringa R., « Thirty one Problems in the Semantics of UML 1.3 Dynamics », *OOPSLA'99 workshop : Rigorous Modelling and Analysis of the UML - Challenges and Limitations*, 1999.

Sahraoui H., Lounis H., Melo W., Mili H., « A Concept Formation Based Approach to Object Identification in Procedural Code », *Automated Software Engineering*, vol. 6, pp. 119-142, 1999.

Satpathy M., Harrison R., Snook C., Butler M., « A Comparative Study of Formal and Informal Specifications through an Industrial Case Study », *Proceedings of FSCBS'01 : IEEE/IFIP Joint Workshop on Formal Specification of Computer Based Systems*, pp. 133-137, 2001.

Sekerinski E., « Graphical Design of Reactive Systems », *B '98 : Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, Springer-Verlag, London, UK, pp. 182-197, 1998.

Simons A. J. H., Graham I., « 30 Things that go wrong In Object modelling with UML 1.3 », *in* H. Kilov, B.Rumpe, I. Simmonds (eds), *Behavioral Specifications of Businesses and Systems*, Kluwer Academic, chapter 17, pp. 237-257, 1999.

Singer J., *Evaluating tools in ASE (or Everything I've learned in 20+ years taught in 1.5 hours)*, 2006. ASE'2006 Tutorial.

Smith G., *The Object-Z Specification Language*, Advances in Formal Methods Series, Kluwer Academic Publishers, 1995.

Snelting G., Tip F., « Reengineering class hierarchies using concept analysis », *SIGSOFT Software Engineering Notes*, vol. 23, n° 6, pp. 99-110, 1998.

Snook C., Butler M., « U2B-A tool for translating UML-B models into B », *in* J. Mermet (ed.), *UML-B Specification for Proven Embedded Systems Design*, 2004.

Snook C., Butler M., « UML-B : Formal modeling and design aided by UML », *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, n° 1, pp. 92-122, 2006.

Stoddart B., Dunne S., Galloway A., Shore R., « Abstract State Machines : Designing Distributed Systems with State Machines and B », *in* D. Bert (ed.), *B'98 : Recent Advances in the Development and Use of the B Method, Second International B Conference*, vol. 1393 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 226-242, 1998.

Tatibouet B., Hammad A., Voisinet J., « From an abstract B specification to UML class diagrams », *2nd IEEE International Symposium on Signal Processing and Information Technology*, 2002.

The Standish Group, *The CHAOS Report*, 1998. http ://www.standishgroup.com.

The Standish Group, *Extreme CHAOS*, 2001. http ://www.standishgroup.com.

Tilley T., Cole R., Becker P., Eklund P. W., « A Survey of Formal Concept Analysis Support for Software Engineering Activities », *Formal Concept Analysis*, vol. 3626 of *Lecture Notes in Computer Science*, Springer, pp. 250-271, 2005.

Valtchev P., Missaoui R., Lebrun P., « A partition-based approach towards constructing galois (concept) lattices », *Discrete Mathematics*, vol. 256, n° 3, pp. 801-829, 2002.

Voisinet J.-C., Contribution au processus de développement d'applications spécifiées à l'aide de la méthode B par validation utilisant des vues UML et traduction vers les langages à objets, PhD thesis, Université de Franche-Comté, Septembre, 2004.

Warmer J. B., Kleppe A. G., *The Object Constraint Language : Precise Modeling With UML*, Addison-Wesley Object Technology Series, Addison-Wesley Professional, 1998.

Wildmoser M., Nipkow T., « Certifying Machine Code Safety : Shallow versus Deep Embedding », *in* K. Slind, A. Bunker, G. Gopalakrishnan (eds), *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, vol. 3223, pp. 305-320, 2004.

Wille R., *Lattice Theory as Algebra*, University of Calgary, 1980.

Annexe A

Nos outils prototypes

A.1 Outil de dérivation de diagrammes de classes

A.1.1 Présentation

L'interface principale (Fig. A.1) de notre outil de génération de diagrammes de classes à partir de machines B présente trois menus principaux en excluant le menu **`Help`** :

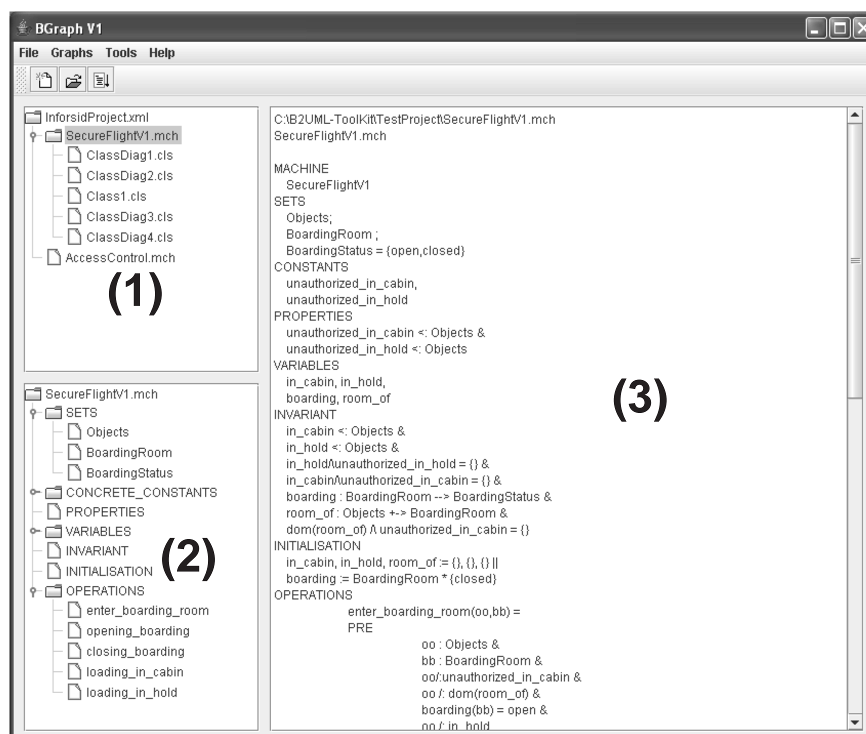


FIG. A.1 – Interface principale de notre outil de dérivation de diagrammes de classes

1. Menu **`File`** : ce menu est utilisé pour créer un projet, ouvrir un projet deja existant, ouvrir une machine dans le projet courant ou quitter l'application.
2. Menu **`Graphs`** : ce menu sert a afficher les différentes structures de données utilisées par notre

algorithme pour la génération de diagrammes de classes : le graphe de dépendance de concepts (relation $\mathcal{I}$, définition 6.2 page 116) et le réseau de concepts $\mathcal{J}$ (définition 7.12 page 132).

3. Menu **`Tools`** : ce menu permet la génération de diagrammes de classes préliminaires, le choix des cibles, le calcul des contextes et la dérivation d'un diagramme de classes après avoir choisi les cibles des opérations. Les interfaces de choix des cibles et de visualisation du diagramme de classes sont présentées au niveau de la Fig. A.3.

L'outil contient également, deux listes : une première liste dans la zone **(1)** de l'interface montrant l'arborescence du projet, et une seconde liste dans la zone **(2)** contenant l'arborescence de la machine. La zone **(3)** de l'interface sert à afficher la machine B à partir de laquelle seront dérivés les diagrammes de classes.

Dans sa version actuelle, notre outil de dérivation de diagrammes de classes (Idani *et al.,* 2005b) à partir de spécifications B, met en œuvre une version antérieure de notre algorithme de formation de concepts. Dans cette version antérieure, nous ne distinguons pas entre différents cas de partages (chapitre 6). Le réseau de concepts $\mathcal{J}$ est donc utilisé aussi bien pour l'étape d'identification des cibles que pour l'étape de construction des contextes. Par conséquent l'outil identifie plus de cibles que lors de l'utilisation du réseau de concepts raffiné $\mathcal{J}_{\mathcal{R}}$.

A.1.2 Architecture

Trois modules principaux forment notre outil de dérivation de diagrammes de classes :

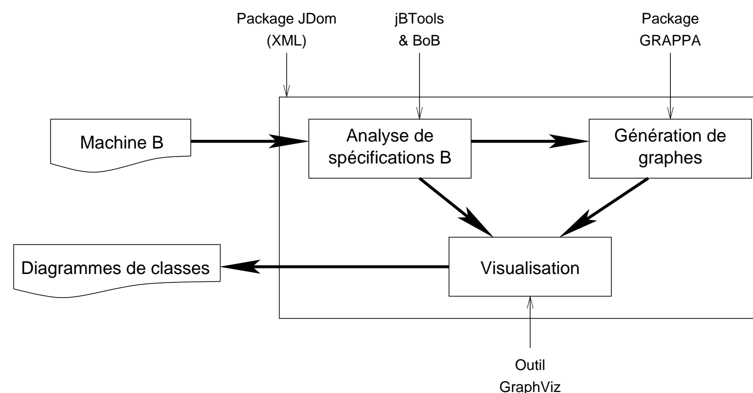


FIG. A.2 – Architecture de l'outil de dérivation de diagrammes de classes

(*i*) le module d'analyse de spécifications B : sert principalement à récupérer une instance du méta-modèle proposé pour B. Ce module est basé sur l'analyseur syntaxique fourni par la plate-forme jBTools[49] et la boîte à outils B [50] développée au LSR/IMAG.,

(*ii*) le module de génération de graphes : permet de réaliser les différents traitements (calcul de $\mathcal{I}$, de $\mathcal{J}$, des contextes, etc) pour l'identification de concepts orientés objets à partir de l'analyse syntaxique des spécifications B. Ce module est développé autour du package GRAPPA[51] qui est un paquetage

[49] lifc.univ-fcomte.fr/ tatibouet/JBTOOLS/
[50] www-lsr.imag.fr/users/Nicolas.Stouls/Productions/BoB/bobUtil.tar.gz
[51] www.research.att.com/ john/Grappa

JAVA destiné à la manipulation de graphiques. Il est est utilisé car il simplifie l'inclusion des possibilités d'affichage des graphiques dans des applications et des applets Java.

(*iii*) le module de visualisation : récupère la sortie du module précédent sous format “dot” et au moyen de l'outil GraphViz[52] réalise l'affichage des différents graphiques dans un frame Java. L'utilisation de GraphViz est principalement destinée au calcul des positions des nœuds et arcs des différents graphiques, ce qui permet un affichage optimal dans le frame Java sans nécessiter l'intervention de l'utilisateur pour aménager le graphe. Le module de visualisation permet aussi d'exporter les graphiques dans des formats externes exploitables pour documenter les spécifications.

Pour donner a l'utilisateur la possibilité de sauvegarder les différents graphes et diagrammes de classes, nous lui permettons de créer des projets. À chaque projet, il peut associer plusieurs machines et pour chaque machine, il peut générer les figures possibles. Pour gérer ces projets l'outil produit des fichiers XML dans lesquels sont sauvegardées toutes les données se rattachant aux machines B et aux figures que l'utilisateur souhaite sauvegarder.

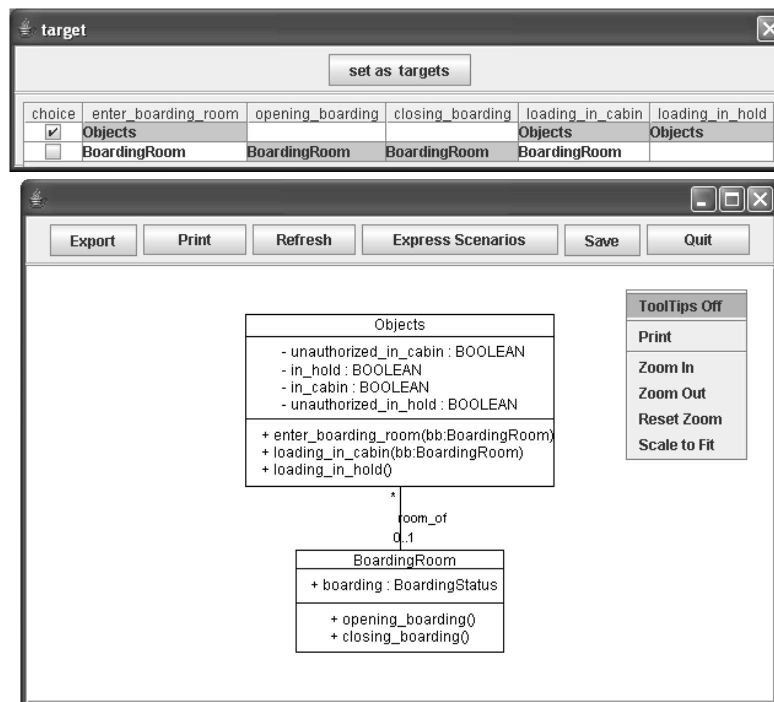


FIG. A.3 – Interface de visualisation d'un diagramme de classes correspondant à un choix particulier de cibles

A.1.3 Quelques résultats

La spécification *SecureFlightV*1 présentée au niveau de l'interface de l'outil (Fig. A.1) modélise des bagages et des passagers dans un aéroport ainsi que leur embarquement. Cette machine utilise trois ensembles : (*i*) L'ensemble de tous les objets (*Objects*) transportés par les passagers, (*ii*) l'ensemble des salles d'embarquement (*BoardingRoom*) et (*iii*) l'ensemble énuméré *BoardingStatus* représentant les

[52] www.graphviz.org

états *open* ou *closed* de la porte d'embarquement. Nous distinguons, également, quatre sous-ensembles d'objets : *unauthorized_in_cabin*, *unauthorized_in_hold*, *in_cabin* et *in_hold*. Certains objets peuvent être considérés comme dangereux, comme par exemple, les bombes et les armes. Ces objets sont désignés par les ensembles *unauthorized_in_cabin* (ensembles d'objets qui ne doivent pas être embarqués avec les passagers dans la cabine comme les armes et les ciseaux) et *unauthorized_in_hold* (ensembles d'objets qui ne doivent pas être embarqués dans la soute à bagages comme les bombes). Comme les bagages embarqués dans la soute ne sont pas accessibles aux passagers durant le vol, certains objects peuvent être considérés comme dangereux dans la cabine mais sont autorisés dans la soute (*e.g.* les ciseaux). Les ensembles *in_cabin* et *in_hold* représentent respectivement les ensembles d'objets qui ont été effectivement chargées dans la cabine et dans la soute. Les deux variables *boarding* et *room_of* sont des relations. La première relation determine l'état de la salle d'embarquement, tandis que la deuxième associe à chaque objet embarqué sa salle d'embarquement. Finalement, cinq opérations ont été associées à cette machine, à savoir :

1. *enter_boarding_room* : permet de faire entrer un objet autorisé dans la cabine (*authorized_in_cabin*) dans une salle d'embarquement déjà ouverte.
2. *opening_boarding* : ouvre une salle d'embarquement fermée.
3. *closing_boarding* : ferme une salle d'embarquement ouverte.
4. *loading_in_hold* : charge un objet *authorized_in_hold* dans la soute.
5. *loading_in_cabin* : charge un objet *authorized_in_cabin* dans la cabine.

La fenêtre de choix des cibles illustrée au niveau de la Fig. A.3 montre les cibles identifiées par l'outil pour chacune des opérations de la spécification : il s'agit des deux classes candidates *Objects* et *BoardingRoom*. L'utilisateur clique simplement sur la cellule correspondant à la cible choisie pour chaque opération. Il peut également choisir le bouton radio d'une ligne donnée pour sélectionner la classe correspondante en tant que cible de toutes les opérations qui peuvent lui être associées. Remarquons que les opérations *opening_boarding*, *closing_boarding* et *loading_in_hold* disposent d'une seule cible possible, alors que les opérations *enter_boarding_room* et *loading_in_cabin* disposent de deux cibles possibles. Ainsi l'outil produit quatre diagrammes de classes différents (Fig. A.4) dépendant des choix faits pour ces deux opérations.

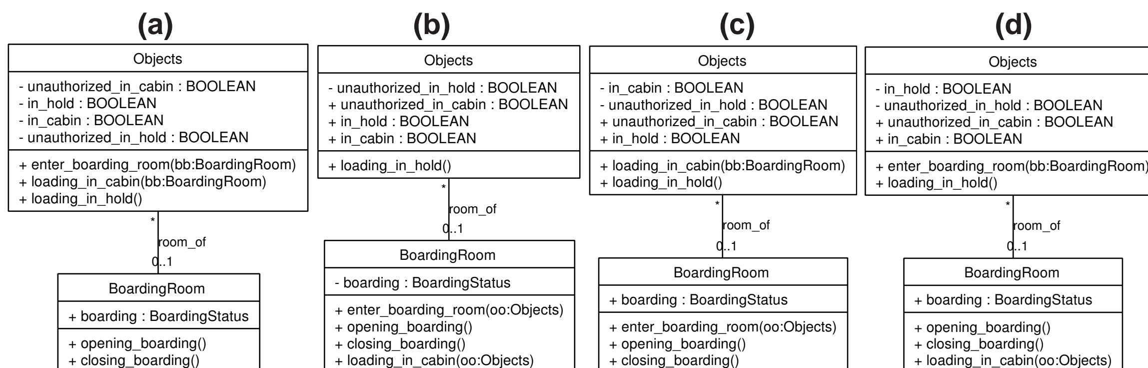


FIG. A.4 – Diagrammes de classes générés par notre outil pour l'exemple *SecureFlightV*1

Les paramètres des méthodes qui figurent dans chacune de classes sont dérivés par une analyse syntaxique du corps de l'opération B. Plus particulièrement de l'analyse des instances de la méta-

classe *BPrecondition* associées aux différentes opérations. Nous prenons l'exemple de l'opération *enter_boarding_room* qui dispose de deux paramètres *oo* et *bb* tels que $oo \in Objects$ et $bb \in BoardingRoom$. Quand *enter_boarding_room* est associée à la classe candidates *Objects*, l'outil la transforme en une méthode paramétrée par $bb : BoardingRoom$, et quand elle est associée à *BoardingRoom* l'outil la transforme en une méthode paramétrée par $oo : Objects$.

Pour les quatre configurations de cibles possibles, la fonction totale *boarding* est transformée en un attribut de la classe *BoardingRoom* de type *BoardingStatus*, et les sous-ensembles *in_cabin*, *in_hold*, *unauthorized_in_cabin* et *unauthorized_in_hold* sont transformés en attributs booléens de la classe *Objects*. Seule la visibilité des ces attributs varie d'un diagramme à l'autre. Quant à la fonction partielle *room_of*, elle est transformée en une association entre les classes *Objects* et *BoardingRoom*.

A.2 Outil d'abstraction de graphes d'accessibilité

A.2.1 Présentation

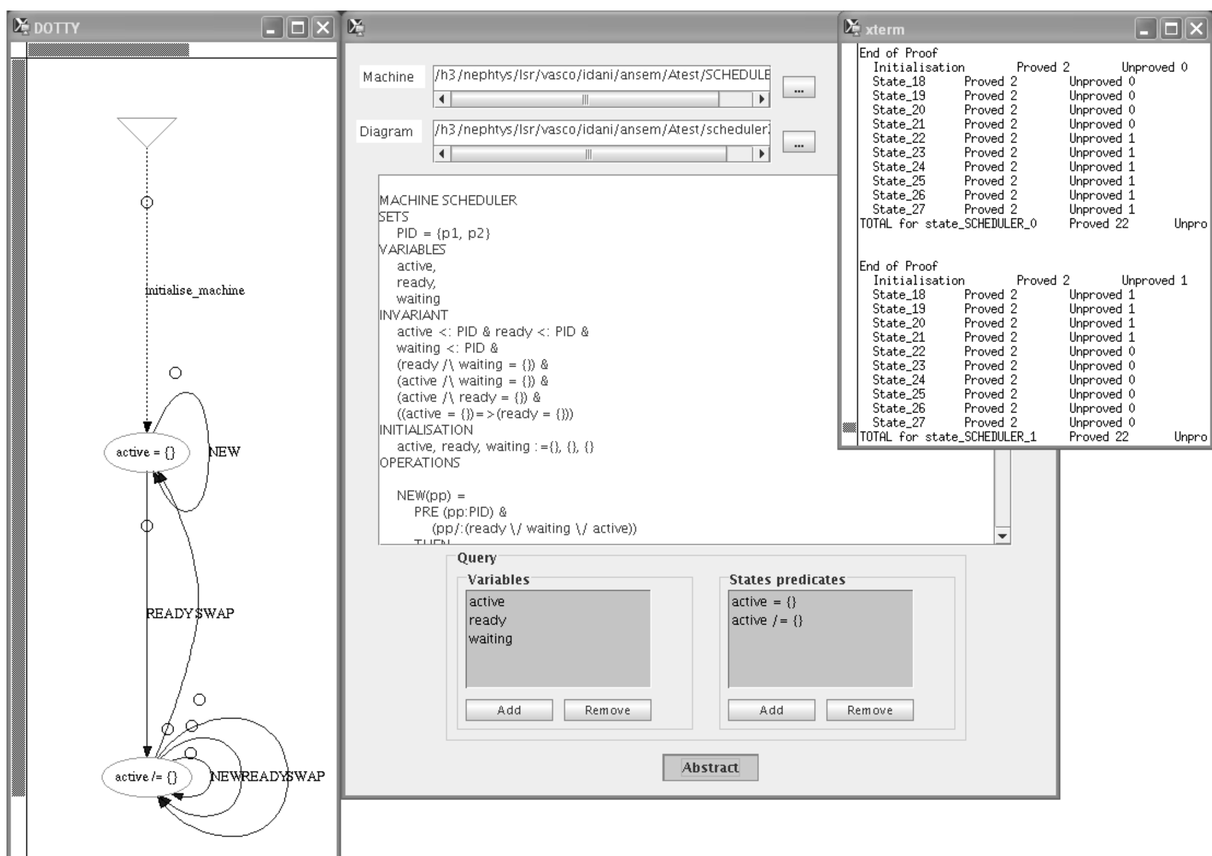


FIG. A.5 – Interfaces de notre outil d'abstraction de graphes d'accessibilité

Notre outil d'abstraction de graphes d'accessibilité est composé de 3 interfaces :

1. Une interface principale (celle qui apparaît au centre de la Fig. A.5) permettant à l'utilisateur

d'ouvrir une machine abstraite B, d'indiquer le graphe d'accessibilité[53] correspondant et de saisir les prédicats d'états abstraits. Dans la Fig. A.5 il s'agit des prédicats : $active = \emptyset$ et $active \neq \emptyset$.

2. Une interface de visualisation du résultat de la preuve réalisée par l'atelier B : pour chaque prédicat d'état abstrait saisi par l'utilisateur et pour chaque état concret du graphe d'accessibilité, l'outil fait appel au prouveur de l'atelier B pour vérifier si l'état concret satisfait le prédicat d'état abstrait (Voir algorithme d'abstraction de graphes).
3. Une interface de visualisation du graphe d'états/transitions abstrait correspondant à l'outil "dotty" fourni avec GraphViz. Cette interface est présentée au niveau de la partie gauche de la Fig. A.5.

A.2.2 Scénario de fonctionnement

Le diagramme de la Fig. A.6 présente un scénario de fonctionnement de notre outil de génération de diagrammes d'états/transitions à partir de spécifications B. L'outil prend en entrée, la machine abstraite concernée par la transformation (que nous notons $\mathcal{M}$), un graphe d'accessibilité au format "dot" et un ensemble de prédicats d'états abstraits.

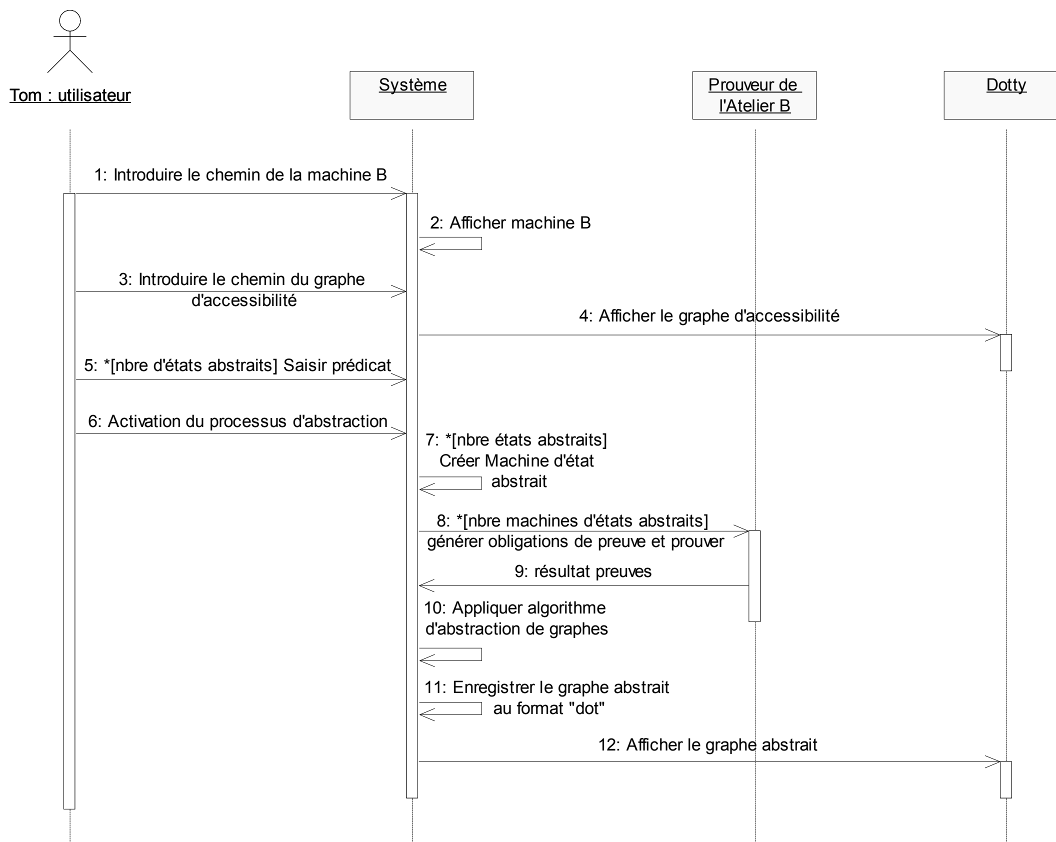


FIG. A.6 – Scénario de fonctionnement de l'outil d'abstraction de graphes d'accessibilité

Pour des systèmes finis, ProB (Leuschel *et al.,* 2003) permet d'explorer exhaustivement le comportement de la spécification et de générer un graphe d'accessibilité dont les nœuds sont des valuations des variables d'état de la spécification et les transitions sont des appels d'opérations. Sur la base de ce graphe

[53] Ce graphe doit être au format "dot" et peut être produit à l'aide de l'outil ProB (Leuschel *et al.,* 2003)

et d'un ensemble de prédicats d'états abstraits l'algorithme d'abstraction que nous avons proposé permet de dériver un diagramme d'états-transitions en regroupant les états concrets satisfaisant les prédicats d'états abstraits. Pour identifier les états concrets qui satisfont un prédicat d'état abstrait, l'outil interagit avec le prouveur de l'atelier B. Ceci est réalisé au moyen de machines B (que nous désignons par $\mathcal{M}_i$) que l'outil génére pour chaque prédicat d'état abstrait $\mathcal{E}_i$ avec :

1. L'invariant de $\mathcal{M}_i$ correspond à l'invariant de la machine $\mathcal{M}$ augmenté par le prédicat $\mathcal{E}_i$
2. À chaque état concret e tel que $e \mathrel{\widehat{=}} (v_1 = val_1, \ldots, v_n = val_n)$ est associée une opération o dans $\mathcal{M}_i$ avec $o \mathrel{\widehat{=}}$ **BEGIN** $v_1 := val_1 \parallel \ldots \parallel v_n := val_n$ **END**

Ci-dessous, nous présentons la machine B produite par notre outil pour le prédicat d'état abstrait $active = \emptyset$ et pour un système contenant deux processus uniquement ($p1$ et $p2$). Vu que tous les états concrets du graphe d'accessibilité correspondent à des valuations satisfaisant l'invariant $\mathcal{I}$ de $\mathcal{M}$ alors toutes les opérations de $\mathcal{M}_i$ établissent $\mathcal{I}$. Ce faisant, la conjonction $\mathcal{I} \wedge \mathcal{E}_i$ permet de départager les états qui satisfont $\mathcal{E}_i$ et ceux qui ne le satisfont pas. Le résultat de la preuve générée par l'atelier B pour la machine **state_SCHEDULER_0** ci-dessous est présenté au niveau de la Fig. A.5 et montre 4 états concrets satisfaisant le prédicat d'abstraction $active = \emptyset$. L'outil interprète cette preuve et regroupe ces 4 états au niveau d'un seul état abstrait étiqueté par $active = \emptyset$.

MACHINE
  state_SCHEDULER_0
SETS
  $PID = \{p1,\ p2\}$
VARIABLES
  active,
  ready,
  waiting
INVARIANT
  $active \subseteq PID \wedge ready \subseteq PID \wedge waiting \subseteq PID \wedge$
  $(ready \cap waiting = \emptyset\ ) \wedge$
  $(active \cap waiting = \emptyset\ ) \wedge$
  $(active \cap ready = \emptyset\ ) \wedge$
  $((active = \emptyset\ ) \Rightarrow (ready = \emptyset\ )) \wedge$
  $active = \emptyset$
INITIALISATION
  $active,\ ready,\ waiting := \emptyset\ ,\ \emptyset\ ,\ \emptyset$
OPERATIONS
  State_18 = **BEGIN** $active := \emptyset \parallel ready := \emptyset \parallel waiting := \emptyset$ **END**;
  State_19 = **BEGIN** $active := \emptyset \parallel ready := \emptyset \parallel waiting := \{p1\}$ **END**;
  State_20 = **BEGIN** $active := \emptyset \parallel ready := \emptyset \parallel waiting := \{p2\}$ **END**;
  State_21 = **BEGIN** $active := \emptyset \parallel ready := \emptyset \parallel waiting := \{p1,p2\}$ **END**;
  State_22 = **BEGIN** $active := \{p1\} \parallel ready := \emptyset \parallel waiting := \emptyset$ **END**;
  State_23 = **BEGIN** $active := \{p1\} \parallel ready := \emptyset \parallel waiting := \{p2\}$ **END**;
  State_24 = **BEGIN** $active := \{p2\} \parallel ready := \emptyset \parallel waiting := \{p1\}$ **END**;
  State_25 = **BEGIN** $active := \{p2\} \parallel ready := \emptyset \parallel waiting := \emptyset$ **END**;
  State_26 = **BEGIN** $active := \{p1\} \parallel ready := \{p2\} \parallel waiting := \emptyset$ **END**;
  State_27 = **BEGIN** $active := \{p2\} \parallel ready := \{p1\} \parallel waiting := \emptyset$ **END**
END

Annexe B

Méta-modèle d'UML : quelques extraits

Dans cette annexe nous nous limitons aux points fondamentaux d'UML[54] (version 2.0) sur lesquels nous nous sommes basé pour la définition de nos schémas de transformation. Les éléments du méta-modèle d'UML que nous présentons à ce niveau présentent principalement la spécification des diagrammes de classes.

B.1 Classifier

L'élément de construction représenté par la méta-classe abstraite **`Classifier`** (Fig. B.1) représente un élément fondamental en UML permettant une classification des éléments de modélisation. Cet élément décrit en particulier des éléments ayant des caractéristiques communes. Dans le cadre du présent travail, nous avons considéré en particulier les spécialisations **`Class`** (Fig. B.2) et **`DataType`** (Fig. B.4) de **`Classifier`** car au niveau de la partie interprétée de notre travail nous nous sommes intéressé uniquement aux transformations de constructions en B en des éléments d'un diagramme de classes. Les caractéristiques concernées par cette classification sont représentées par la méta-classe abstraite **`Feature`** qui se spécialise en caractéristiques structurelles (**`StructuralFeature`**) et caractéristiques comportementales (**`BehavioralFeature`**). Les caractéristiques structurelles et comportementales auxquelles nous nous sommes intéressé dans la présente thèse portent en particulier sur les classes et les classes associatives d'un diagramme de classes, et correspondent respectivement aux attributs et aux méthodes.

B.2 Structural Feature

Au niveau de la Fig. B.1 une caractéristique structurelle apparaît comme un élément typé (méta-classe **`TypedElement`**) dont le type est spécifié par la méta-classe **`Type`** (Fig. B.3), et que celle-ci se spécialise en types primitifs (méta-classe **`PrimitiveType`**) et en types énumérés (**`Enumeration`**). Par ailleurs, une troisième spécialisation de cette méta-classe provient de l'héritage entre **`Classifier`** et **`Type`** (Fig. B.4) et de l'héritage entre **`Class`** et **`Classifier`** (Fig. B.2). En effet, une classe dans un diagramme de classes sert aussi de type (ou type abstrait). Par analogie à ces trois spécialisations de la méta-classe **`Type`** nous avons construit la partie de notre méta-modèle destinée à la spécification de types en B (Fig. 3.11 page 65), où un type de base en B (*BBasicType*) se spécialise en *BAbstractSet*, *BEnumSet*

[54] La spécification complète du méta-modèle d'UML 2.0 se trouve à l'url suivante : www.omg.org

et *BPrimitiveType*. Les projections suivantes des constructions B en constructions UML : (*i*) *BAbstractSet* ⟶ **`Class`**, (*ii*) *BEnumSet* ⟶ **`Enumeration`**, et (*iii*) *BPrimitiveType* ⟶ **`PrimitiveType`** permettent donc de récupérer des types en UML pour les *BData*s transformées en caractéristiques structurelles (*i.e.* les attributs de classes).

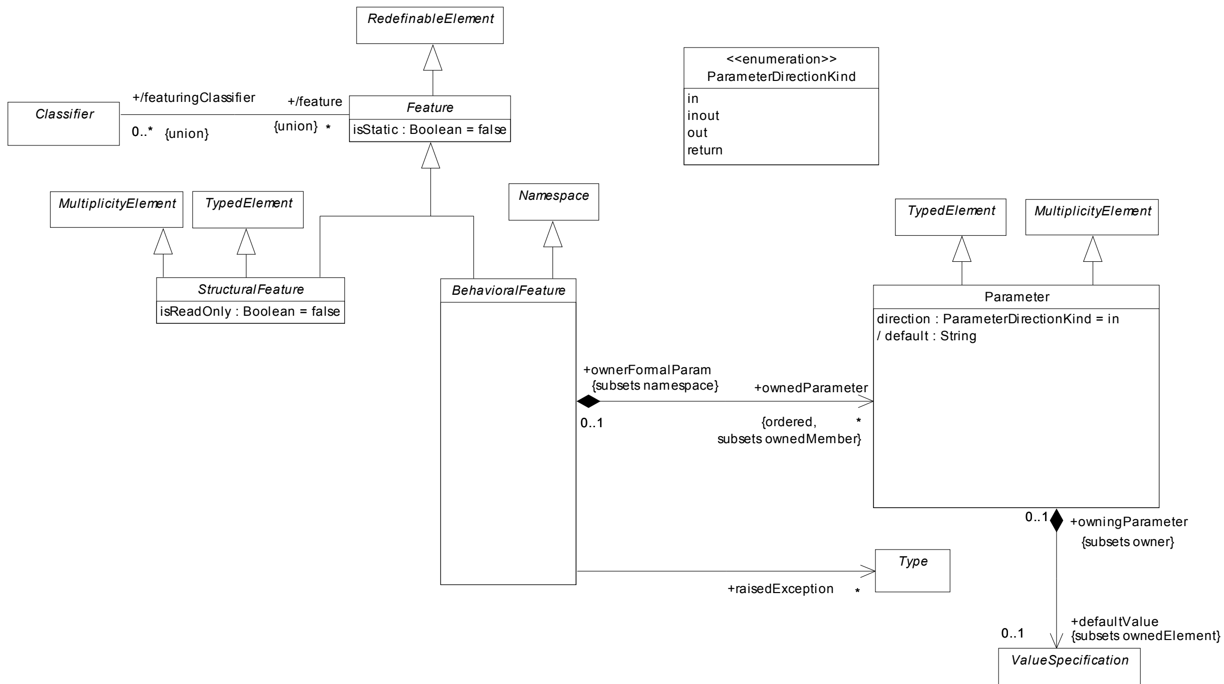


FIG. B.1 – Features diagram of the Kernel package

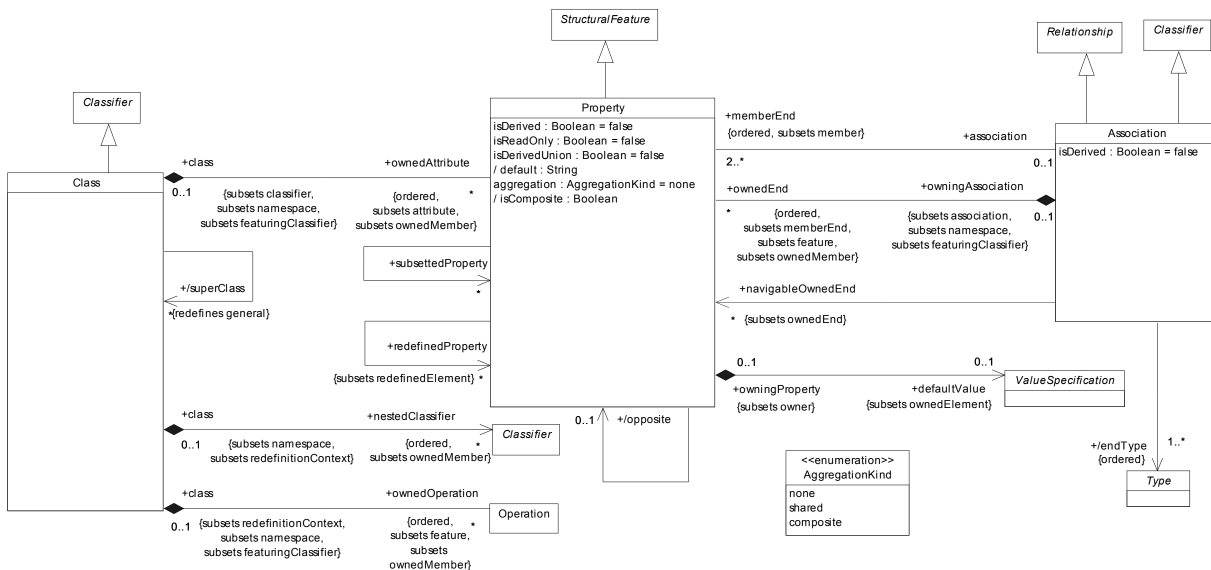


FIG. B.2 – Classes diagram of the Kernel package

La spécialisation **`Property`** de la méta-classe *StructuralFeature* (Fig. B.2) est sollicitée au niveau de deux transformations principales dans le cadre de notre travail : (*i*) représenter des attributs

de classes (nom de rôle **`+ownedAttribute`**), et/ou (*ii*) représenter des extrémités d'associations (rôle **`+memberEnd`**). Une **`Property`** reliée à une association via **`+memberEnd`** désigne une extrémité de l'association. La multiplicité **`2..*`** à ce niveau montre qu'une association dispose d'au moins deux extrémités d'association. On parle alors d'associations binaires (deux extrémités uniquement) et d'associations n-aires (plus que deux extrémités). Les multiplicités d'un attribut ou d'une extrémité d'association sont spécifiées via la méta-classe **`ValueSpecification`** qui, d'une part, compose la **`Property`**, et d'autre part, reliée à la méta-classe **`MultiplicityElement`** par **`+upperValue`** et **`+lowerValue`** (voir Fig. B.3).

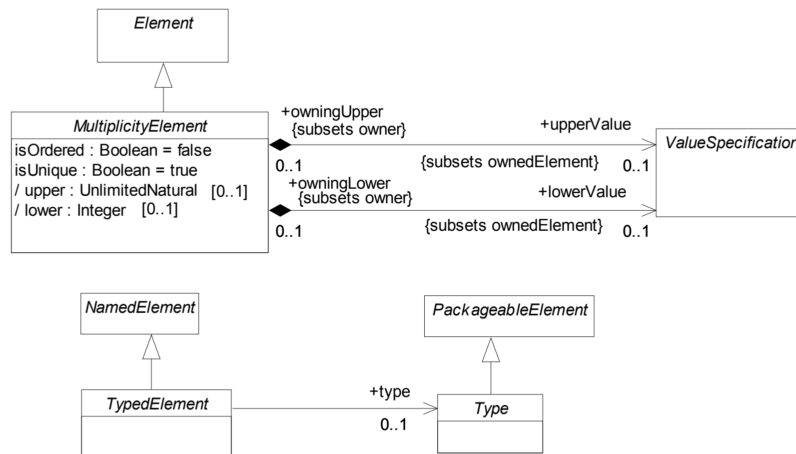


FIG. B.3 – Multiplicities diagram of the Kernel package

B.3 Data Types

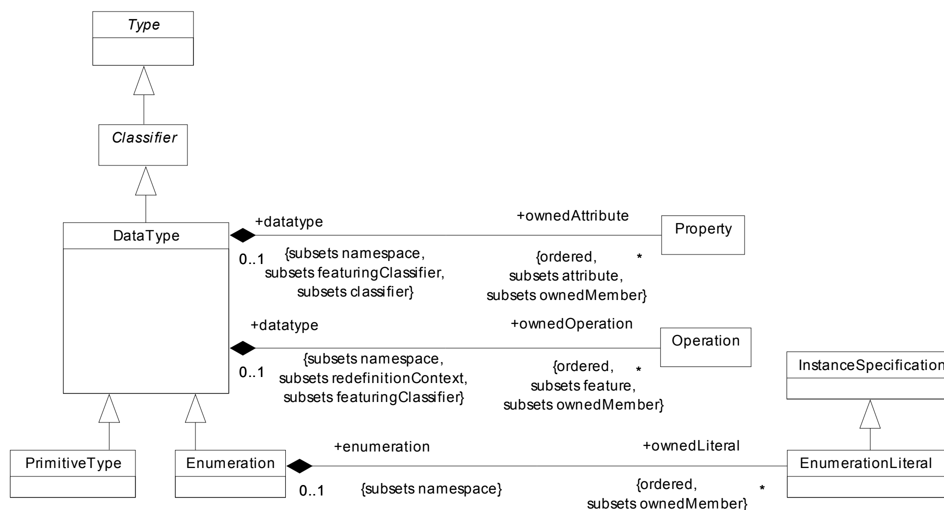


FIG. B.4 – Data types diagram of the Kernel package

En UML, un **`DataType`** peut contenir des attributs (**`+ownedAttribute`**) dans le but de modéliser des types de données structurés. Une utilisation typique des types de données en UML est de représenter

les types primitifs de langages de programmation. Par exemple, les nombres entiers (*Integer*) et les chaînes de caractères (*String*) sont souvent traités comme types de données en UML. Quant aux types énumérés (*Enumeration*), ils sont souvent représentés en UML sous forme de classe stéréotypée par ≪*enumeration*≫ dont les attributs sont les instances de **`EnumerationLiteral`**.

B.4 Packages

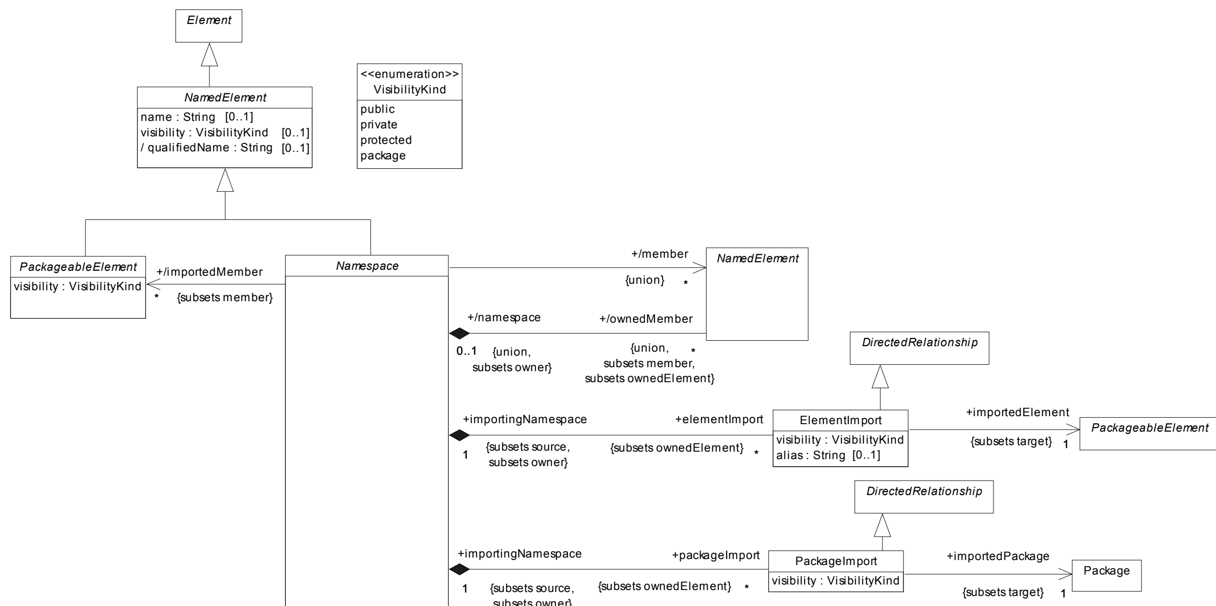


FIG. B.5 – Namespaces diagram of the Kernel package

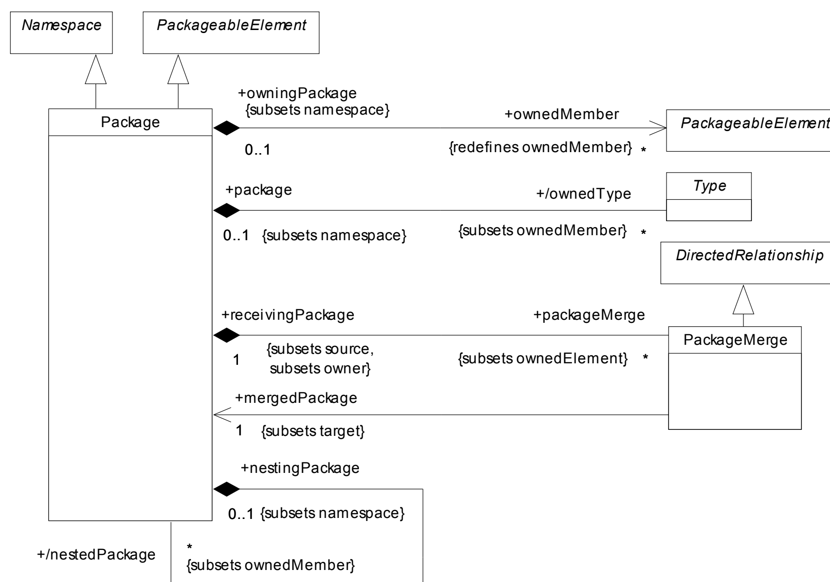


FIG. B.6 – The Packages diagram of the Kernel package

Annexe C

Exemples d'application des schémas de transformation

Dans cette annexe nous présentons quelques exemples de spécifications B extraites de certaines références bibliographiques et nous montrons une application d'instances de schémas de transformation pour en produire des diagrammes de classes.

Spécification d'un système multi-agents. La spécification B ci-dessous est extraite de :

> Mermet Bruno., « Formal Model of a Multi-agent System », *European Meeting on Cybernetics and Systems Research* : *From Agent Theory to Agent Implementation*, Vienne (Autriche), pp 653-658 (2002).

Dans cette spécification, les ensembles abstraits *Agents*, *States* et *Stimulus* représentent respectivement :

- l'ensemble des agents du système multi-agents,
- l'ensemble des états possibles de tous les agents du système,
- l'ensemble des actions ou réactions pouvant surgir lorsqu'un agent est dans un état particulier.

MACHINE
  SMA
SETS
  $Agents; States; Stimulus$
CONSTANTS
  $Nothing$
PROPERTIES
  $Nothing \in Stimulus$
VARIABLES
  $agents, state, transition, pendingMessages$
INVARIANT
  $agents \subseteq Agents \wedge$
  $state \in agents \rightarrow States \wedge$
  $transition \in agents \rightarrow ((States \times Stimulus) \leftrightarrow (States \times Stimulus)) \wedge$
  $pendingMessages \subseteq Stimulus \times (\mathbb{P}(agents))$
...

La dérivation d'un diagramme de classes à partir de cette partie structurelle de la machine *SMA* passe nécessairement par certaines réécritures. En effet, nous n'avons identifié aucun schéma de transformation pour transformer les variables *transition* et *pendingMessages*. Toutefois, les variables *agents* et *state* sont

traduites directement au moyen des schémas de transformation des structures de base, et plus particulièrement des schémas 5 et 9. Ci-dessous, nous présentons l'ensemble des étapes poursuivies pour dériver le diagramme de classes de la Fig. C.1 :

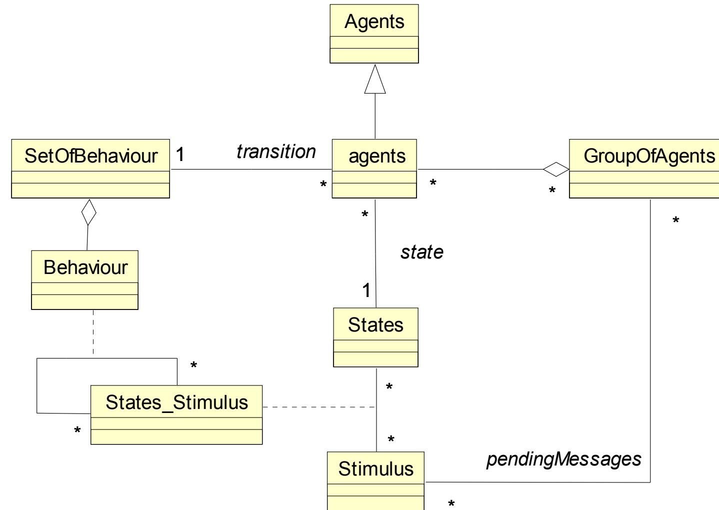


FIG. C.1 – Diagramme de classes dérivé de la machine SMA

1. Exécuter Schéma$_{3.3}$ pour transformer les ensembles abstraits *Agents*, *States*, et *Stimulus* en classes au niveau du diagramme de classes,
2. Exécuter Schéma$_{5.4}$(*Agents*, *agents*) pour transformer le sous-ensemble *agents* en une sous-classe de la classe *Agents*. Dans cette représentation, *Agents* peut être considéré comme une classe abstraite à partir de laquelle l'ensembles des agents existants est défini.
3. Réécrire l'invariant $pendingMessages \subseteq Stimulus \times \mathbb{P}(agents)$ en :

   $GroupOfAgents = \mathbb{P}(agents)$

   $pendingMessages \subseteq Stimulus \times GroupOfAgents$
4. Exécuter successivement Schéma$_{3.3}$(*GroupOfAgents*) et Schéma$_{8}$(*GroupOfAgents*, *agents*), qui permettent respectivement de créer au niveau du diagramme de classes : la classe *GroupOfAgents* et le lien d'agrégation entre *GroupOfAgents* et *agents*.
5. Exécuter Schéma$_{10}$(*pendingMessages*, *Stimulus*, *GroupOfAgents*, $\times$) pour créer l'association *pendingMessages* entre les classes *GroupOfAgents* et *Stimulus*.
6. Réécrire l'invariant de typage de la variable *transition* comme suit :

   $States_Stimulus = States \times Stimulus$

   $Behaviour = States_Stimulus \times States_Stimulus$

   $SetOfBehaviour = \mathbb{P}(Behaviour)$

   $transition \in agents \rightarrow SetfOfBehaviour$
7. Exécuter Schéma$_{10}$(*States_Stimulus*, *States*, *Stimulus*, $\times$) pour transformer *States_Stimulus* en une classe associative entre *States* et *Stimulus*.
8. Exécuter Schéma$_{10}$(*Behaviour*, *States_Stimulus*, *States_Stimulus*, $\times$) pour transformer *Behaviour* en une classe associative refléxive dont les extrémités correspondent à *States_Stimulus*.

9. Exécuter successivement Schéma$_{3.3}$(*SetOfBehaviour*) et Schéma$_{8}$(*SetOfBehaviour*, *Behaviour*), qui permettent respectivement de créer au niveau du diagramme de classes : la classe *SetOfBehaviour* et le lien d'agrégation entre *SetOfBehaviour* et *Behaviour*.
10. Exécuter Schéma$_{9.1}$(*state*, *agents*, *States*, $\rightarrow$) et Schéma$_{9.1}$(*transition*, *agents*, *SetOfBehaviour*, $\rightarrow$) pour créer les associations *state* et *transition* de multiplicités $*$ et 1.

Spécification d'un système de gestion de factures. Le diagramme de classes ci-dessous est dérivé d'une spécification B issue du B-Book (Abrial, 1996) et nommée *Invoice System* (voir B-Book, page 338). Comme indiqué au niveau du diagramme, la spécification B traitée à ce niveau modélise un ensemble de clients, de factures et de lignes de commandes portant sur des produits. Les relations de spécialisation/généralisation sont issues d'inclusions entre ensembles abstraits (désignant des ensembles d'éléments possibles) et de variables abstraites (désignant des ensembles d'éléments effectifs). Les super-classes de ce diagramme peuvent donc correspondre à des classes abstraites. Notons également que la spécification B considérée est construite de trois machines B : la machine *Client*, la machine *Product* et la machine *Invoice*. Par conséquent, le diagramme peut être construit autour de trois paquetages. Cependant, nous considérons ici un aplatissement intégrant ces trois machines B sans nous préoccuper des dépendances entre paquetages.

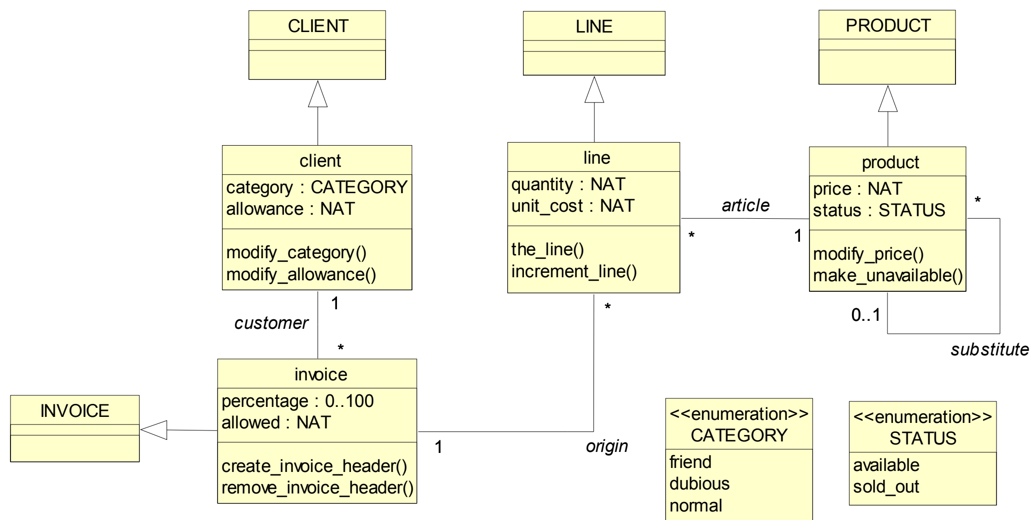


FIG. C.2 – Diagramme de classes dérivé d'une spécification de gestion de factures

Traductions de constructions B complexes. La machine B, ci-dessous, extraite de (Potet, 1998), fait partie d'un premier niveau de spécification d'un gestionnaire d'accès de personnes à des bâtiments composé de trois machines B : DONNEES (description des données nécessaires à l'énoncés des propriétés de sûreté), JOURNAL (description des usagers enregistrés dans un bâtiment) et SYSTEME (description du comportement d'un usager à un accès). La machine DONNEES que nous considérons ici introduit les notions de personnes, cartes, bâtiments, accès (en entrée et en sortie), et de cartes valides. Les hypothèses sur les données considérées par (Potet, 1998) à ce niveau sont les suivantes :

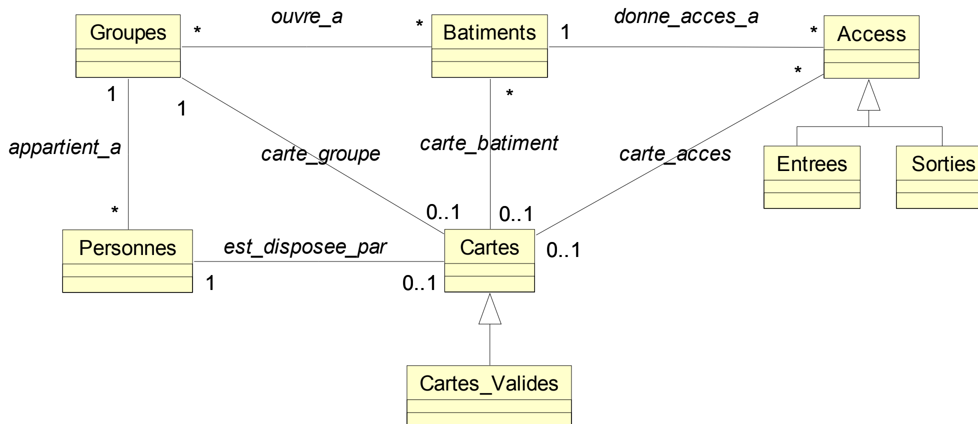
(*i*) toute personne possède une et une seule carte et toute carte est possédée par une et une seule personne,

(*ii*) toute personne appartient à un et un seul groupe mais un groupe peut être vide,

(*iii*) tout accès donne dans un bâtiment.

MACHINE
  DONNEES
SETS
  Personnes; *Cartes*; *Batiments*; *Access*; *Groupes*
CONSTANTS
  est_disposee_par, *appartient_a*, *donne_acces_a*, *ouvre_a*,
  Entrees, *Sorties*,
  Cartes_Valides,
  carte_groupe, *carte_batiment*, *carte_acces*
PROPERTIES
  $est_disposee_par \in Cartes \rightarrowtail Personnes \wedge$
  $appartient_a \in Personnes \rightarrow Groupes \wedge$
  $donne_acces_a \in Access \rightarrow Batiments \wedge$
  $ouvre_a \in Groupes \leftrightarrow Batiments \wedge$
  $Entrees \subset Access \wedge$
  $Sorties \subset Access \wedge$
  $Entrees \cup Sorties = Access \wedge$
  $Entrees \cap Sorties = \emptyset \wedge$
  $Cartes_Valides \subseteq Cartes \wedge$
  $carte_groupe = (est_disposee_par \; ; \; appartient_a) \wedge$
  $carte_batiment = (carte_groupe \; ; \; ouvre_a) \wedge$
  $carte_acces = (carte_batiment \; ; \; donne_acces_a^{-1})$
END

Notons que les invariants *Entrees* $\cap$ *Sorties* $= \emptyset$ et *Entree* $\cup$ *Sorties* $=$ *Access* ne sont pas pris en compte au niveau des transformations que nous avons proposées. En effet, lls correspondent plutôt à des contraintes sur l'usage des éléments de modélisation *Entrees*, *Sorties* et *Access*. De même pour des relations *carte_groupe*, *carte_batiment* et *carte_acces* qui sont définies par composition de relations fonctionnelles.



Ces relations sont illustrées au niveau du diagramme ci-dessus sous formes d'associations sans indiquer le fait qu'il s'agit d'associations calculées à partir d'autres associations. Des contraintes OCL peuvent donc être rajoutées pour étayer le diagramme et mettre en valeur certains traits liés à la spécification. Nous n'avons pas considéré au niveau de notre travail la dérivation de B vers OCL (nous laissons cette question aux travaux futurs), cependant, nous mettons l'accent sur le fait qu'il s'agit d'une explici-

tation d'une vue structurelle sujette à des interprétations et pouvant illustrer une vision statique de la spécification. La complexité des constructions B traitées ne doit donc pas dépasser les constructions prises en compte au niveau des schémas de transformation ou doivent être ramenées à des constructions plus simples pouvant être traduites par nos schémas. Par exemple, la traduction des relations *carte_groupe*, *carte_batiment* et *carte_acces* en associations passe par le rajout des invariants suivants :

$$carte_groupe \in Cartes \rightarrowtail Groupes$$

$$carte_batiment^{-1} \in Batiments \nrightarrow Cartes$$

$$carte_access^{-1} \in Access \nrightarrow Cartes$$

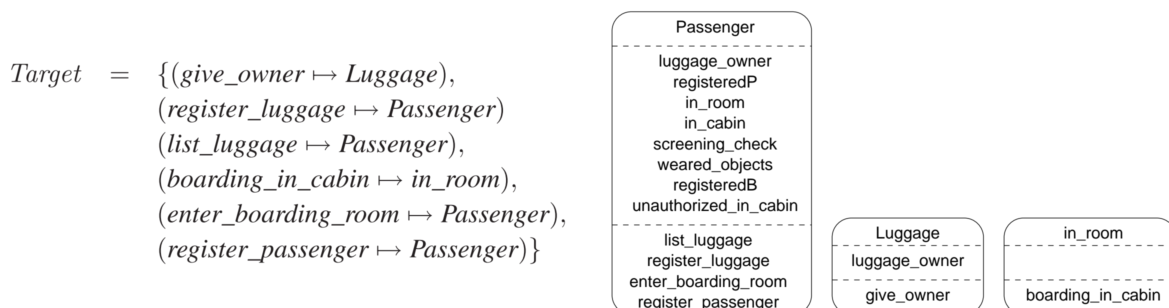
ou tout simplement par l'exécution des schémas de transformation suivants :

$$Schéma_{9.1}(carte_groupe, Cartes, Groupes, \rightarrowtail)$$

$$Schéma_{9.1}(carte_batiment, Batiments, Cartes, \nrightarrow)$$

$$Schéma_{9.1}(carte_access, Access, Cartes, \nrightarrow)$$

Autre traduction de SecureFlight. Nous considérons à ce niveau une configuration particulière de la fonction $Target$ issue de la spécification *SecureFlight* (Annexe E) ainsi que le modèle de contextes correspondant. Ensuite, nous montrons l'application d'une succession de règles de transformation du modèle de contextes en un diagramme de classes tout en mettant l'accent sur les schémas de transformation exécutées par ces règles. Notons que le diagramme de classes préliminaire issu de la machine *SecureFlight* est présenté au niveau de la Fig. 8.1 (page 143).



Le diagramme de classes construit à partir de ces données est celui de la Fig. C.3 suivante :

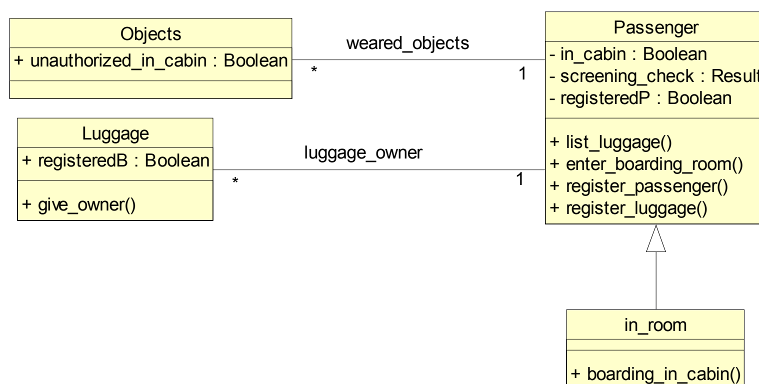


FIG. C.3 – Autre diagramme de classes dérivé de la spécification *SecureFlight*

Les étapes de dérivation de ce diagramme de classes à partir du modèle de contextes précédent, et sur la base du diagramme de structure préliminaire (Fig. 8.1) correspondent aux applications des règles suivantes :

1. **Des contextes aux classes** : Cette première étape considère deux cas dépendant du type de chaque classe candidate à l'origine d'un contexte : *Class* (pour les contextes *Passenger* et *Luggage*) et *SubClass* (pour le contexte *in_room*).

   (a) Appliquer la règle $\mathcal{R}_1$ pour transformer *Passenger* en une classe (exécution du schéma de transformation 3.3).

   (b) Appliquer la règle $\mathcal{R}_1$ pour transformer *Luggage* en une classe (exécution du schéma de transformation 3.3).

   (c) Appliquer la règle $\mathcal{R}_{2.1}$ pour transformer *in_room* en une sous-classe de la classe *Passenger* (exécution du schéma de transformation 5.4).

   Au niveau de la phase (c) de cette première étape l'application de la règle $\mathcal{R}_{2.1}$ plutôt que la règle $\mathcal{R}_{2.2}$ est justifiée par le fait que l'un des contextes du modèle de contextes provient d'une classe candidate appartenant à $Incl^+[\{in_room\}]$ (il s'agit de *Passenger*). Ainsi, à l'issue de cette étape, l'ensemble $\mathcal{K}$ des classes du diagramme de classes final est comme suit :

   $$\mathcal{K} = \{Passenger, Luggage, in_room\}$$

2. **Transformation des attributs des contextes** : Les attributs des contextes sont de natures différentes. Plusieurs règles différentes sont par conséquents appliquées.

   (a) **Cas des attributs de type SubClass** : un attributs de type *SubClass* est transformé au moyen de la règle $\mathcal{R}_{5.1}$, $\mathcal{R}_{5.2}$ ou $\mathcal{R}_{5.3}$, et ce, selon l'existence ou non d'un lien sémantique entre l'attribut et le contexte qui l'encapsule.

      i. Appliquer la règle $\mathcal{R}_{5.1}$ pour transformer *registeredP* en un attribut booléen privé de la classe *Passenger* (exécution du schéma de transformation 5.5).

      ii. Appliquer la règle $\mathcal{R}_{5.1}$ pour transformer *in_cabin* en un attribut booléen privé de la classe *Passenger* (exécution du schéma de transformation 5.5).

      iii. Appliquer la règle $\mathcal{R}_{5.2}$ pour transformer *registeredB* en un attribut booléen public de la classe *Luggage* (exécution du schéma de transformation 5.5).

      iv. Appliquer la règle $\mathcal{R}_{5.3}$ pour transformer *unauthorized_in_cabin* en un attribut booléen public d'une classe *Objects* nouvellement créée (exécution des schémas de transformation 3.3 et 5.5).

      À ce niveau, l'application de la règle $\mathcal{R}_{5.1}$ pour transformer les attributs *registeredP* et *in_cabin* en attributs booléens de la classe *Passenger* est justifiée par le fait que leur contexte correspond à leur super-classe préliminaire. Toutefois, l'attribut *in_room* n'a pas été considéré car il correspond déjà à une sous-classe de Passenger. Par ailleurs, l'application de la règle $\mathcal{R}_{5.2}$ pour transformer *registeredB* a été conditionnée par l'existence d'une classe dans l'ensemble $\mathcal{K}$ correspondant à la super-classe préliminaire de *registeredB* est distincte du contexte qui l'encapsule. Finalement, l'attribut *unauthorized_in_cabin* encapsulé par le

contexte *Passenger* ne fait pas partie des deux cas précédents, c'est pourquoi il a été transformé par la règle $\mathcal{R}_{5.3}$. Ainsi, à l'issue de cette première phase de l'étape 2, l'ensemble $\mathcal{K}$ des classes du diagramme de classes final est comme suit :

$$\mathcal{K} = \{Passenger, Luggage, in_room, Objects\}$$

(b) **Cas des attributs de type AssociativeClass** : Nous considérons dans cette seconde phase de l'étape 2, les éléments *luggage_owner*, *screening_check* et *weared_objects* avec la seule règle considérée ($\mathcal{R}_6$). Celle-ci exécute les schémas suivants :

i. $Schéma_{9.1}$ pour transformer *luggage_owner* en une association simple entre les classes *Passenger* et *Luggage* étant donné que $\{Passenger, Luggage\} \subseteq \mathcal{K}$.

ii. $Schéma_{9.1}$ pour transformer *weared_objects* en une association simple entre les classes *Passenger* et *Objects* étant donné que $\{Passenger, Objects\} \subseteq \mathcal{K}$.

iii. $Schéma_{9.2}$ pour transformer *screening_check* en un attribut privé de *Passenger* de type *Result* étant donné que $Passenger \in \mathcal{K}$ et $Result \notin \mathcal{K}$.

3. **Transformation des opérations des contextes** : Comme indiqué au niveau du chapitre 8 les opérations des contextes sont systématiquement transformées en des méthodes des classes issues des contextes qui les encapsulent.

Annexe D

Questionnaire

D.1 Introduction

Ce questionnaire s'inscrit dans le cadre de mon travail de thèse portant sur la production de vues UML à partir de spécifications B. Actuellement l'approche est outillée et permet une construction semi-automatique de vues comportementales et structurelles à partir de spécifications B.

L'objectif de ce questionnaire est d'étudier l'apport des vues graphiques UML produites à partir de spécifications. Pour ce faire, nous vous proposons des spécifications B accompagnées ou non d'un ensemble de diagrammes UML (diagrammes de classes et d'états/transitions) dont l'objectif est d'aider à la compréhension de spécifications en explicitant leurs caractéristiques structurelles et comportementales.

D.2 Questions générales

		Faible	Moyen	Bien
Q1	Évaluez votre connaissance d'UML	☐	☐	☐
Q2	Évaluez votre connaissance de B	☐	☐	☐
Q3	Évaluez votre connaissance des diagrammes d'états/transitions	☐	☐	☐
Q4	Évaluez votre connaissance des diagrammes de classes . .	☐	☐	☐

D.3 Spécifications B considérées

Nous vous proposons deux spécifications B différentes :

- Celle d'un gestionnaire de processus : SCHEDULER dont le but est d'allouer une ressource (par exemple un processeur) à un ensemble de processus. Un processus peut être dans les états suivants :
  - En attente (ou **waiting**) c'est-à-dire présent dans le système mais non candidat à l'attribution de la ressource ;
  - Prêt (ou **ready**) pour recevoir la ressource ;
  - Actif (ou **active**), c'est-à-dire possesseur de la ressource.

- Celle du contrôle d'accès (ACCESSCONTROL) permettant de gérer l'accès de personnes à un ensemble de bâtiments. L'accès aux bâtiments est réalisé selon un ensemble d'autorisations accordées aux personnes.

D.4 Le gestionnaire de processus (SCHEDULER)

```
MACHINE
  SCHEDULER
SETS
  Process
VARIABLES
  active,
  ready,
  waiting
INVARIANT
  active ⊆ Process ∧
  ready ⊆ Process ∧
  waiting ⊆ Process ∧
  card(active) ≤ 1 ∧
  (ready ∩ waiting = ∅ ) ∧
  (active ∩ waiting = ∅ ) ∧
  (active ∩ ready = ∅ ) ∧
  ((active = ∅ ) ⇒ (ready = ∅ ))
INITIALISATION
  active, ready, waiting := ∅ , ∅ , ∅

OPERATIONS

  NEW(pp) =
    PRE
      pp ∈ Process ∧
      pp ∉ (ready ∪ waiting ∪ active)
    THEN
      waiting := waiting ∪ {pp}
    END;

  READY(rr) =
    PRE
      rr ∈ Process ∧ rr ∈ waiting
    THEN
      waiting := (waiting - {rr}) ||
      IF active = ∅ THEN
        active := {rr}
      ELSE
        ready := ready ∪ {rr}
      END
    END;

  SWAP =
    PRE
      active ≠ ∅
    THEN
      waiting := waiting ∪ active ||
      IF ready = ∅ THEN
        active := ∅
      ELSE
        ANY pp WHERE
          pp ∈ Process ∧ pp ∈ ready
        THEN
          active := {pp} ||
          ready := ready - {pp}
        END
      END
    END
END
```

FIG. D.1 – Spécification B du gestionnaire de processus

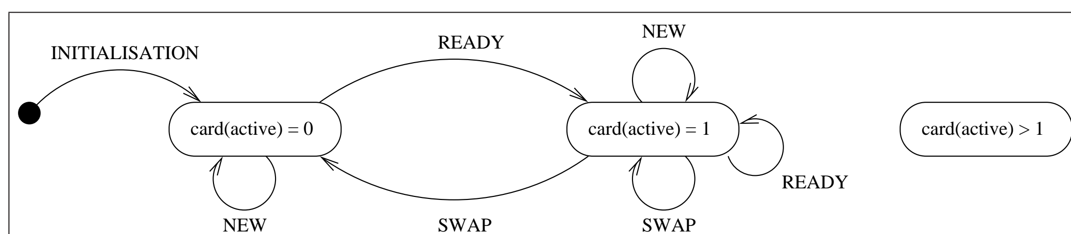


FIG. D.2 – Diagrammes d'états/transitions associé à la variable *active*

Q5 Il peut y avoir plusieurs processus actifs en même temps :

(a) Oui .... ☐
(b) Non .... ☐

Expliquer pourquoi :

....

....

Q6 L'opération *SWAP* peut être réalisée :

(a) Si aucun processus n'est actif .... ☐
(b) S'il existe au moins un processus actif .... ☐
(c) Sans aucune condition .... ☐

Expliquer pourquoi :

....

....

Q7 Pour désactiver tous les proccessus actifs il suffit d'exécuter un nombre suffisant de fois l'opération *SWAP* :

(a) Oui .... ☐
(b) Non .... ☐

Expliquer pourquoi :

....

....

Q8 L'introduction d'un nouveau processus dans le système (exécution de l'opération *NEW*) peut affecter l'ensemble des processus actifs :

(a) Oui . . . . ☐
(b) Non . . . . ☐

Expliquer pourquoi :

. . . .

. . . .

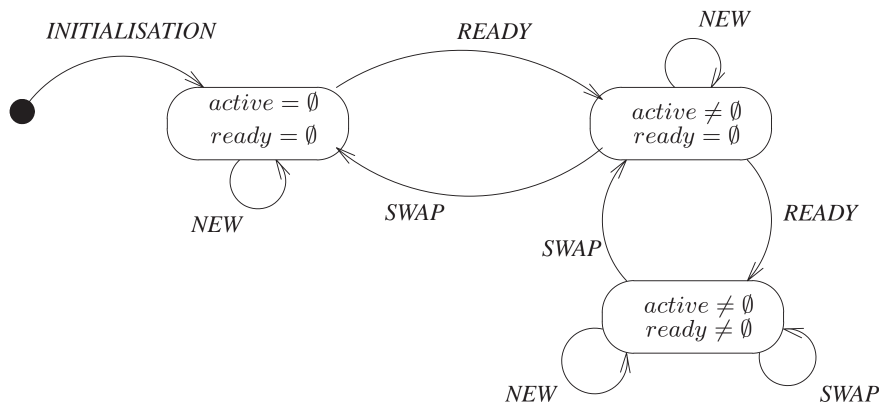


FIG. D.3 – Prise en compte d'une propriété invariante

Q9 L'état $active = \emptyset \wedge ready \neq \emptyset$ est permis par l'invariant :

(a) Oui . . . . ☐
(b) Non . . . . ☐
(c) Je ne sais pas . . . . ☐

Q10 L'état $active = \emptyset \wedge ready \neq \emptyset$ peut être atteint :

(a) Oui . . . . ☐
(b) Non . . . . ☐
(c) Je ne sais pas . . . . ☐

Expliquer pourquoi :

. . . .

. . . .

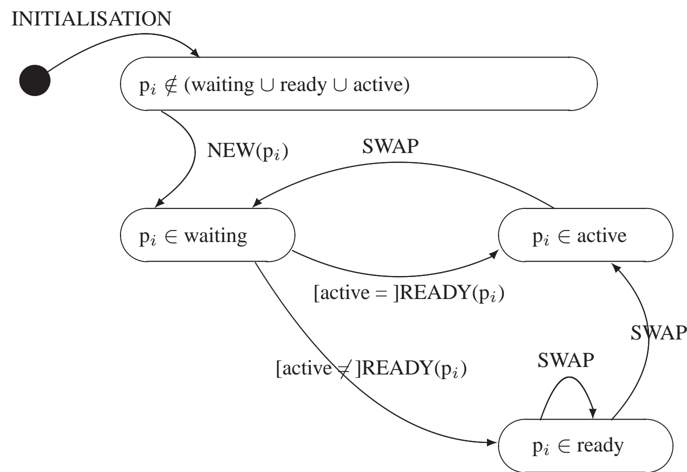


FIG. D.4 – Cycle de vie d'un processus

Q11 Un processus prêt ($p_i \in ready$) peut ne pas devenir actif :

(a) Oui .... ☐
(b) Non .... ☐

Expliquer pourquoi :

....

....

Q12 Après avoir été activé ($p_i \in active$), un processus peut bloquer l'activation d'autres processus. :

(a) Oui .... ☐
(b) Non .... ☐

Expliquer pourquoi :

....

....

Q13 Si un processus est prêt ($p_i \in ready$) alors il peut être activé :

(a) Oui .... ☐
(b) Non .... ☐

Expliquer pourquoi :

....

....

D.5 Le contrôleur d'accès

La spécification B du contrôleur d'accès présentée ci-dessous est inspirée de (Potet, 1998).

```
MACHINE
  AccessControl
SETS
  PERSONNE;
  CARTE;
  BATIMENT;
  ETAT_BAT = {ferme, ouvert}
VARIABLES
  etat_batiment,
  appartient_a,
  donne_acces_a,
  acces_a,
  VALIDE
INVARIANT
  etat_batiment ∈ BATIMENT → ETAT_BAT ∧
  appartient_a ∈ CARTE ⤔ PERSONNE ∧
  donne_acces_a ∈ CARTE ↔ BATIMENT ∧
  VALIDE ⊆ CARTE ∧
  acces_a ∈ VALIDE ⇸ BATIMENT ∧
  acces_a ⊆ donne_acces_a ∧
  etat_batiment[ran(acces_a)] ≠ {ferme} ∧
  VALIDE ⊆ dom(appartient_a) ∧
  VALIDE ⊆ dom(donne_acces_a)
```

```
fermer_batiment(bb) =
  PRE bb ∈ BATIMENT ∧ acces_a⁻¹[{bb}] = ∅
      ∧ etat_batiment(bb) = ouvert
  THEN
    etat_batiment(bb) := ferme
  END;

ouvrir_batiment(bb) =
  PRE
    bb ∈ BATIMENT ∧ etat_batiment(bb) = ferme
  THEN
    etat_batiment(bb) := ouvert
  END;
```

```
carte_personne =
  ANY pp, cc WHERE
    pp ∈ PERSONNE ∧ cc ∈ CARTE ∧
    cc ∉ dom(appartient_a) ∧
    pp ∉ ran(appartient_a)
  THEN
    appartient_a :=
            appartient_a ∪ {cc ↦ pp}
  END;

rendre_carte =
  ANY pp WHERE
    pp ∈ PERSONNE ∧ pp ∈ ran(appartient_a)
    ∧ appartient_a ⁻¹ (pp) ∉ dom(acces_a)
  THEN
    VALIDE :=
        VALIDE - {appartient_a ⁻¹ (pp)} ||
    appartient_a := appartient_a ⩥ {pp}
  END;
```

```
valider(cc) =
  PRE
    cc ∈ CARTE ∧ cc ∈ dom(appartient_a) ∧
    cc ∈ dom(donne_acces_a)
  THEN
    VALIDE := VALIDE ∪ {cc}
  END;
```

```
ajouter_acces =
  ANY cc, bb WHERE
    cc ∈ CARTE ∧ bb ∈ BATIMENT
  THEN
    donne_acces_a :=
            donne_acces_a ∪ {cc ↦ bb}
  END;

enlever_acces =
  ANY cc WHERE
    cc ∈ CARTE ∧
    cc ∈ dom(donne_acces_a) ∧
    cc ∉ dom(acces_a)
  THEN
    donne_acces_a := {cc} ⩤ donne_acces_a
    ||
    VALIDE := VALIDE - {cc}
  END;
```

```
entrer =
  ANY cc, bb WHERE
    bb ∈ BATIMENT ∧ cc ∈ CARTE ∧
    cc ∈ VALIDE ∧
    etat_batiment(bb) = ouvert ∧
    (cc ↦ bb) ∈ donne_acces_a ∧
    acces_a[{cc}]= ∅
  THEN
    acces_a := acces_a ∪ {cc ↦ bb}
  END;

sortir =
  ANY cc, bb WHERE
    bb ∈ BATIMENT ∧ cc ∈ CARTE ∧
    (cc ↦ bb) ∈ acces_a
  THEN
    acces_a := acces_a - {cc ↦ bb}
  END
```

FIG. D.5 – Spécification B du contrôleur d'accès

Les ensembles abstraits *PERSONNE*, *CARTE* et *BATIMENT* spécifient respectivement les personnes, les cartes et les bâtiments du système étudié. Pour décrire les états *fermé* et *ouvert* de chaque bâtiment nous définissons la fonction totale *etat_batiment* associant à chaque bâtiment de l'ensemble *BATIMENT* une des valeurs *ferme* ou *ouvert* définies par énumération au niveau de *ETAT_BAT*. Les cartes sont associées à des personnes (variable *appartient_a*) et permettent d'accéder à plusieurs bâtiments (variable *donne_acces_a*). La fonction partielle *acces_a* désigne l'ensemble des cartes utilisées pour un accès en cours dans un bâtiment.

D.5.1 Vue Structurelle

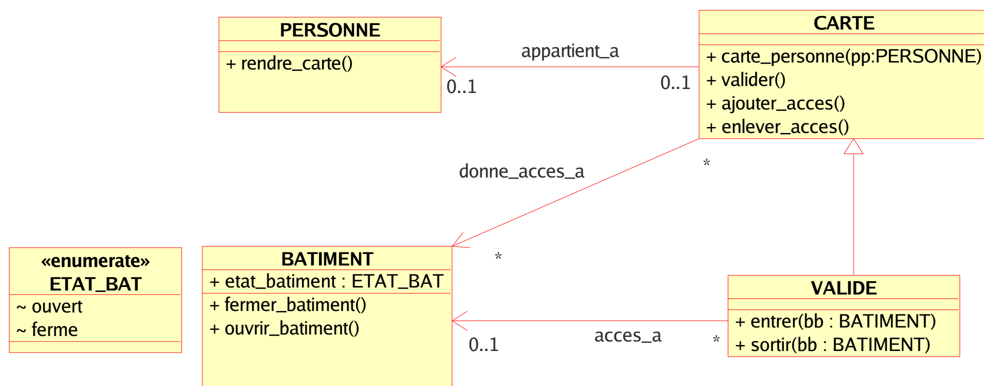


FIG. D.6 – Diagramme de classes du contrôleur d'accès

Q14 Une carte peut appartenir à plusieurs personnes :

(a) Oui . . . . ☐
(b) Non . . . . ☐

Expliquer pourquoi :

. . . .

. . . .

Q15 Uniquement les cartes valides peuvent être utilisées pour accéder à un bâtiment (variable $acces_a$) :

(a) Oui . . . . ☐
(b) Non . . . . ☐

Expliquer pourquoi :

. . . .

. . . .

Q16 Dans quels états peut être un batiment :

....

....

Q17 Une carte peut donner accès à plusieurs bâtiments :

(a) Oui .... ☐
(b) Non .... ☐

Expliquer pourquoi :

....

....

Q18 Une personne peut avoir plusieurs cartes d'accès :

(a) Oui .... ☐
(b) Non .... ☐

Expliquer pourquoi :

....

....

Q19 une carte valide appartient toujours à quelqu'un :

(a) Oui .... ☐
(b) Non .... ☐

Expliquer pourquoi :

....

....

D.5.2 Vues comportementales

Dans les diagrammes suivants, la mention ≪possibly≫ présentée avant le nom de la transition veut dire que la transition peut être exécutée à partir de l'état source mais pas toujours. Nous dirons que la transition est éventuellement déclenchable.

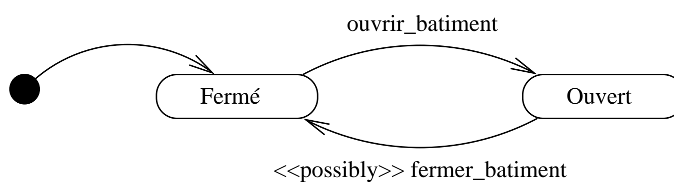


FIG. D.7 – Un diagramme d'états/transitions associé à la classe Batiment

Par exemple dans le diagramme de la Fig. D.7, la transition *fermer_batiment* peut être exécutée quand le batiment est ouvert, mais cette exécution n'est pas toujours réalisable. En effet, avant de fermer un batiment il faut s'assurer qu'il n'y a personne à l'intérieur . . . Tandis que l'exécution de la transition *ouvrir_batiment* est toujours possible quand le batiment est fermé.

Si la mention ≪possibly≫ est située après le nom de la transition alors la transition peut atteindre l'état cible mais pas toujours. Dans ce cas nous dirons que la transition est éventuellement atteignable.

Finalement, si la transition est éventuellement déclenchable et éventuellement atteignable alors elle est représentée par des pointillés.

Dans ce qui suit, nous distinguons les états suivants pour un batiment bb :
- l'état *Fermé* défini par $etat_batiment(bb) = Ferme$,
- l'état *Ouvert* défini par $etat_batiment(bb) = Ouvert$,
- l'état *Accessible* défini par $bb \in ran(donne_acces_a)$,
- l'état *Inaccessible* défini par $bb \notin ran(donne_acces_a)$,
- l'état *Vide* défini par $bb \in ran(acces_a)$, et
- l'état *Occupé* défini par $bb \notin ran(acces_a)$,

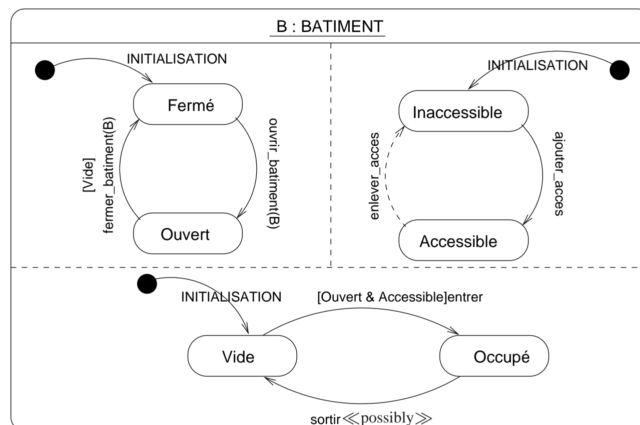


FIG. D.8 – Diagramme d'états/transitions concurrent pour à la classe “BATIMENT”

Q20 Pour fermer un bâtiment, il faut que le bâtiment soit seulement à l'état *Ouvert* :

(a) Oui . . . . ☐
(b) Non . . . . ☐

Si non, le bâtiment doit être dans quels états pour qu'il puisse être fermé :

. . . .

. . . .

Q21 Quelle opération permet de passer un bâtiment de l'état *Vide* à l'état *Occupé* :

. . . .

Dans quels cas est-ce possible :

(a) Le bâtiment est ouvert . . . . ☐
(b) Le bâtiment est Accessible . . . . ☐
(c) Le bâtiment est Accessible et Vide . . . . ☐

Expliquer pourquoi :

. . . .

. . . .

Q22 Comment peut-on faire passer un bâtiment de l'état *Accessible* à l'état *Inaccessible* :

. . . .

. . . .

Q23 Comment peut-on faire passer un bâtiment de l'état *Inaccessible* à l'état *Accessible* :

. . . .

. . . .

Nous considérons maintenant les états suivants associés à une carte cc :

- Valide ($cc \in VALIDE$)
- Invalide ($cc \notin VALIDE$)
- Active ($cc \in dom(donne_acces_a)$)
- Inactive ($cc \notin dom(donne_acces_a)$
- Out ($cc \notin dom(acces_a)$
- In ($cc \in dom(acces_a)$
- Utilisable ($cc \in dom(appartient_a))$
- Inutilisable ($cc \notin dom(appartient_a))$

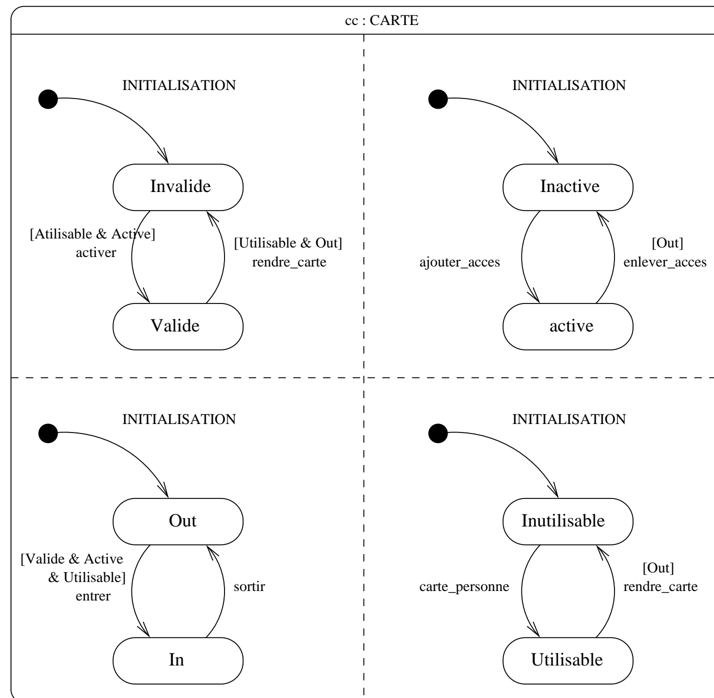


FIG. D.9 – Diagramme d'états/transitions concurrent pour à la classe "CARTE"

Q24 Supposons que nous disposons d'un seul bâtiment et d'une seule carte. Si le bâtiment est à l'état *Vide*, *Ouvert* et *Accessible*, et la carte est à l'état : *Valide*, *Active*, *Utilisable* et *Out*, alors quels sont les états de carte et de bâtiment après l'appel de l'opération *entrer*.

. . . .

. . . .

Q25 Sachant qu'à l'initialisation un bâtiment est à l'état : *Fermé, Vide* et *Inaccessible* et qu'une carte est à l'état : *Invalide, Inactive, Out* et *Inutilisable* ; donnez une séquence d'opérations permettant d'atteindre les états *Ouvert*, *Occupé* et *Accessible* de Bâtiment, et les états *Valide*, *Active*, In et *Utilisable* de carte.

. . . .

. . . .

D.6 Évaluation

Q26 Avez-vous compris sans difficulté les spécifications :

(a) Pas d'accord .... ☐
(b) Plutôt pas d'accord .... ☐
(c) Plutôt d'accord .... ☐
(d) Tout à fait d'accord.................................... ☐

Q27 Les diagrammes vous ont-ils aidés à comprendre les spécifications :

(a) Pas d'accord .... ☐
(b) Plutôt pas d'accord .... ☐
(c) Plutôt d'accord .... ☐
(d) Tout à fait d'accord.................................... ☐

Q28 Les diagrammes sont plus lisibles que les spécifications formelles :

(a) Pas d'accord .... ☐
(b) Plutôt pas d'accord .... ☐
(c) Plutôt d'accord .... ☐
(d) Tout à fait d'accord.................................... ☐

Q29 Les spécifications formelles sont plus complètes que les diagrammes :

(a) Pas d'accord .... ☐
(b) Plutôt pas d'accord .... ☐
(c) Plutôt d'accord .... ☐
(d) Tout à fait d'accord.................................... ☐

Q30 Commentaires libres :

....

....

....

....

....

Annexe E

La spécification SecureFlight

MACHINE
$SecureFlight$
SETS
$Passenger \; ; \; Object \; ; \; Luggage \; ;$
$Result = \{success, \; failure\}$
CONSTANTS
$unauthorized_in_cabin, \; screening_check,$
$weared_objects, \; register_checks, \; luggage_owner$
PROPERTIES
$unauthorized_in_cabin \subseteq Object \; \land$
$weared_objects \in Object \rightarrow Passenger \; \land$
$screening_check \in Passenger \rightarrow Result \; \land$
$register_checks \in Passenger \rightarrow Result \; \land$
$luggage_owner \in Luggage \rightarrow Passenger$
VARIABLES
$in_room, \; in_cabin, \; registeredP, \; registeredB$
INVARIANT
$registeredP \subseteq Passenger \land registeredB \subseteq Luggage \; \land$
$registeredP \subseteq register_checks^{-1} \; [\{success\}] \; \land$
$luggage_owner[registeredB] \subseteq registeredP \; \land$
$in_room \subseteq registeredP \land in_cabin \subseteq registeredP \; \land$
$(in_cabin \cap in_room) = \emptyset \; \land$
$\forall \; pp \; . \; (pp \in Passenger \land screening_check(pp) = success$
$\quad \Rightarrow weared_objects^{-1} \; [\{pp\}] \cap unauthorized_in_cabin = \emptyset \;) \; \land$
$\forall \; pp \; . \; (pp \in in_room \Rightarrow$
$\quad weared_objects^{-1} \; [\{pp\}] \cap unauthorized_in_cabin = \emptyset \;) \; \land$
$\forall \; pp \; . \; (pp \in in_cabin \Rightarrow$
$\quad weared_objects^{-1} \; [\{pp\}] \cap unauthorized_in_cabin = \emptyset \;) \; \land$
$in_room \subseteq screening_check^{-1} \; [\{success\}] \; \land$
$in_cabin \subseteq screening_check^{-1} \; [\{success\}]$
INITIALISATION
$registeredP, \; registeredB := \emptyset \; , \; \emptyset \; ||$
$in_room, \; in_cabin := \emptyset \; , \; \emptyset$

```
OPERATIONS
   enter_boarding_room =
      ANY pp WHERE
         pp ∈ Passenger ∧
         pp ∉ (in_room ∪ in_cabin) ∧
         screening_check(pp) = success ∧
         weared_objects ⁻¹ [{pp}] ∩
         unauthorized_in_cabin = ∅
      THEN
         in_room := in_room ∪ {pp}
      END;

   boarding_in_cabin =
      ANY pp WHERE
         pp ∈ in_room
      THEN
         in_cabin := in_cabin ∪ {pp} ||
         in_room := in_room - {pp}
      END;

   ll ← list_luggage(pp) =
      PRE pp ∈ Passenger THEN
         ll := luggage_owner ⁻¹ [{pp}]
      END;

   pp ← give_owner(ll) =
      PRE ll ∈ Luggage THEN
         pp := luggage_owner(ll)
      END;

   register_passenger =
      ANY pp WHERE
         pp ∈ Passenger
      THEN
         registeredP := registeredP ∪ {pp}
      END;

   register_luggage =
      ANY pp, ll WHERE
         pp ∈ registeredP ∧ ll ∈ Luggage ∧
         (ll ↦ pp) ∈ luggage_owner
      THEN
         registeredB := registeredB ∪ {ll}
      END
END
```

Glossaire

Analyse Formelle de concepts ou AFC. L'analyse formelle de concepts (AFC) constitue une approche théorique de structuration des données permettant d'identifier des concepts potentiellement intéressants au sein d'un ensemble de données, décrits par une relation binaire d'incidence.

Classe. Une classe est une construction standard d'UML utilisée pour spécifier le canevas à partir duquel les objets seront fabriqués à l'exécution. Une classe est une spécification - un objet est une instance d'une classe.

Diagramme de classes. Diagramme UML représentant un ensemble d'éléments déclaratifs (statiques) du modèle comme les classes, les types ainsi que leurs contenus et relations.

Invariant. Prédicat exprimant des propriétés portant sur les données d'un composant. On distingue deux sortes d'invariants dans le langage B : l'invariant de la clause INVARIANT, qui porte sur les données du composant et l'invariant de boucle WHILE qui porte sur les données.

Méta-modèle. Un métamodèle peut être défini comme un modèle d'un langage de modélisation. Un méta-modèle sert ainsi à exprimer les concepts communs à l'ensemble des modèles d'un même domaine.

Model Driven Architecture ou MDA. Le MDA est une méthodologie de conception de logiciel, proposée et soutenue par l'OMG. C'est une variante particulière de l'ingénierie des modèles (IdM) ou en anglais MDE (Model Driven Engineering). L'idée fondamentale est que les fonctionnalités du système à développer sont définies dans un modèle indépendant de la plateforme (Platform Independent Model, PIM), en utilisant un langage de spécifications approprié, puis traduites dans un ou plusieurs modèles spécifiques à la plateforme (Platform Specific Model, PSM) pour l'implémentation concrète du système.

Opération en B. Service offert par un module B. Les opérations constituent la partie dynamique d'un module (ou composant B).

Preuve. Activité mathématique consistant à démontrer la vérité d'Obligations de Preuve. Le développement d'un projet comporte principalement deux grandes activités : l'écriture de composants et la preuve des Obligations de Preuves associées à ces composants.

Raffinement. Le raffinement (noté M_n) d'un composant (noté M_{n-1}) est une nouvelle formulation de M_{n-1}, dans laquelle certains constituants de M_{n-1} sont remplacés par une représentation plus détaillée (les constantes abstraites et les variables abstraites, l'initialisation, les opérations).

Substitution. Notation mathématique permettant de modéliser la transformation de formules mathématiques.

Treillis. Désigne un ensemble ordonné où toute paire d'éléments a une borne supérieure et une borne inférieure.

UML ou Unified Modelling Language. Langage graphique de modélisation des données et des traitements. C'est une formalisation de la modélisation objet utilisée en génie logiciel. L'OMG travaille actuellement sur la version UML 2.1.

Variable abstraite. Donnée appartenant à un composant dont le type est quelconque et qui est raffinée au cours du raffinement du composant.

Variable concrète. Donnée appartenant à un composant et conservée au cours du raffinement qui représente soit un scalaire, soit un tableau.

Résumé

Les exigences qui s'appliquent aux composants logiciels et aux logiciels embarqués justifient l'utilisation des meilleures techniques disponibles pour garantir la qualité des spécifications et conserver cette qualité lors du développement du code. Les méthodes formelles, et parmi elles la méthode B, permettent d'atteindre ce niveau de qualité. Cependant, ces méthodes utilisent des notations et des concepts spécifiques, qui génèrent souvent une faible lisibilité et une difficulté d'intégration dans les processus de développement et de certification. Ainsi, proposer des environnements de spécification, de développement de programmes et de logiciels, combinant des méthodes formelles et des méthodes semi-formelles largement utilisées dans les projets industriels, en l'occurrence B et UML, s'avère d'une grande importance. Notre intérêt porte précisément sur la méthode B qui est une méthode formelle utilisée pour modéliser des systèmes et prouver l'exactitude de leur conception par raffinements successifs. Mais les spécifications formelles sont difficiles à lire quand elles ne sont pas accompagnées d'une documentation. Cette lisibilité est essentielle pour une bonne compréhension de la spécification, notamment dans des phases de validation ou de certification. Aujourd'hui, en B, cette documentation est fournie sous forme de texte, avec, quelquefois, des schémas explicitant certaines caractéristiques du système. L'objectif de ce travail de thèse est de mettre en relation des spécifications en B avec des diagrammes UML, qui constituent un standard de facto dans le monde industriel et dont le caractère graphique améliore la lisibilité. Nous avons axé notre processus de dérivation de diagrammes de classes à partir de spécifications B autour d'une technique d'ingénierie inverse guidée par un ensemble de correspondances structurelles et sémantiques spécifiées à un méta-niveau. Quant à la dérivation de diagrammes d'états/transitions, elle a été orientée vers une technique d'abstraction de graphes d'accessibilité construits par une exploration exhaustive du comportement de la spécification.

Mots-clefs : Méthode B, UML, Intégration de méthodes, Méta-modélisation, Ingénierie inverse, Analyse formelle de concepts.

Abstract

The complex requirements of software systems justify the use of the best existing techniques to guarantee the quality of specifications and to preserve this quality during the programming phase of a software life-cycle. Formal methods, such as B, make it possible to reach such a level of quality. However, there is a cultural gap between formal methods, with their mathematical concepts and notations, and the usual techniques of the various stakeholders in industrial projects (graphical formalisms and natural language documents). There is thus a significant risk that errors such as misinterpretation of the requirements and specification documents lead to erroneously validate the specification, and hence to produce the wrong system. So, software specification and development tools, which combine formal and graphical methods widely used in industrial projects, such as B and UML, are likely of a great importance. In this work we focus our interest on the B method which is a formal method used to model systems and prove their correctness by successive refinements. Our goal is to produce graphical UML views from existing formal B specifications in order to ease their readability and then help their external validation. In fact, such views can be useful for various stakeholders in a formal development process : they are intended to support the understanding of the formal specifications by the requirements holders and the certification authorities ; they can also be used by the B developers to get an alternate view on their work. We centered the derivation of UML class diagrams from B specifications on a reverse-engineering technique guided by a set of structural and semantic mappings specified on a meta-level. The derivation of state/transition diagrams is based on an abstraction technique applied to accessibility graphs which are built by an exhaustive unfolding of the behavior of B specifications.

Keywords : B method, UML, Method Integration, Meta-modeling, Reverse-engineering, Formal concept analysis.