



Utilisation de B pour la vérification de spécifications UML et le développement formel orienté objet

THÈSE

présentée et soutenue publiquement le 05 Mai 2006

pour l'obtention du

Doctorat de l'université Nancy 2

(spécialité informatique)

par

TRUONG Ninh Thuan

Composition du jury

- Président :** Didier Bert, CR CNRS, IMAG Grenoble
- Rapporteurs :** Véronique Viguier Donzeau-Gouge, Professeur, CNAM, Paris
Henri Habrias, Professeur, Université Nantes
- Examineurs :** Vincent Poirriez, Maître de Conférences, Université de Valenciennes
Michael Rusinowitch, DR Inria, Loria Nancy
- Directeur de thèse :** Jeanine Souquières, Professeur, Université Nancy 2

Mis en page avec la classe thloria.

REMERCIEMENTS

C'est avec un très grand plaisir que je vais remercier individuellement les personnes qui ont rendu ce travail possible. Tous ne peuvent figurer, puissent les absents accepter mes excuses.

***Jeanine Souquières**, directrice de thèse, m'a toujours fait confiance et a veillé à ce que je puisse travailler dans les meilleures conditions. Je tiens à lui exprimer ma profonde gratitude pour avoir dirigé ce travail, pour sa disponibilité et sa patience, pour l'aide qu'elle m'a apporté tout au long de ces années.*

***Véronique Viguier Donzeau-Gouge**, **Henri Habrias**, rapporteurs de thèse. Je les remercie d'avoir bien voulu accepter la charge de ce travail et d'avoir consacré le temps nécessaire pour examiner ma recherche.*

***Didier Bert**, **Vincent Poirriez**, **Michael Rusinowitch**, examinateurs. Je les remercie d'avoir accepté de participer à ce jury.*

*Pendant cette thèse, j'ai eu la chance de pouvoir collaborer avec les collègues dans le groupe DEDALE, **Jean-Pierre Jacquot**, **Francis Alexandre**, **Dieu-Donné Okalas Ossami**, **Riad Chiboub**, **Arnaud Lanoix**, **Samir Chouali**, ..., je les remercie pour le travail que nous avons fait en commun et pour nos discussions sur les sujets les plus variés.*

*Mes plus grands remerciements s'adressent à **ma famille** et à **mes amis**.*

À THAI-DUY,

TABLE DES MATIÈRES

Table des figures	1
Introduction	3
Contexte	4
Méthode formelle B	4
Notations à objets UML	5
Dérivation d'UML en B	5
Objectif du travail de recherche	5
Contribution	6
Vérification et validation de spécification UML en utilisant B	6
Prise en compte de l'objet dans le développement formel B	7
Plan du mémoire	8
I Étude des bases et état de l'art	11
1 Introduction aux méthodes formelles et à la méthode B	13
1.1 Introduction aux méthodes formelles	13
1.1.1 Définition	13
1.1.2 Utilisation	14
1.1.3 Classification	15
1.2 La méthode B	16
1.2.1 Présentation informelle	16
1.2.2 Fondements de la méthode B	18
1.2.2.1 Machine abstraite	18
1.2.2.2 Raffinement	21
1.2.2.3 Mécanismes de composition	25
1.2.3 Extensions	27
1.3 Synthèse	28
2 Construction par objets et UML	29
2.1 Construction par objets	29

2.2	UML	31
2.2.1	Notation UML - les différents diagrammes	31
2.2.1.1	Diagrammes d'objets	33
2.2.1.2	Diagrammes de classes	33
2.2.1.3	Diagrammes de collaboration	34
2.2.1.4	Diagrammes de séquence	34
2.2.1.5	Diagrammes d'état-transitions	36
2.2.2	La sémantique de UML	36
2.2.3	Introduction à OCL	41
2.2.3.1	Expressions	42
2.2.3.2	Contraintes	42
2.3	Synthèse	43
3	État de l'art	45
3.1	Dérivation d'UML en B	45
3.1.1	Dérivation de diagrammes de classes	45
3.1.2	Dérivation de diagrammes d'état-transitions	48
3.1.3	Dérivation de diagrammes de collaboration	50
3.1.4	Dérivation d'expressions OCL	52
3.1.5	Dérivation du méta-modèle UML en méthodes formelles	53
3.2	Vérification et validation formelle	53
3.2.1	Vérification formelle	54
3.2.2	Validation formelle	55
3.3	Spécification formelle orientée objet	56
3.4	Synthèse	58
II	Vérification de spécifications UML en utilisant B	61
4	Vérification de la sémantique des modèles UML	63
4.1	Dérivation d'un objet d'une méta-classe en B	64
4.2	Analyse des règles de bonne formation du méta-modèle UML	64
4.3	Procédure de dérivation du méta-modèle UML en B	66
4.4	Dérivation et vérification des diagrammes de classes	67
4.4.1	Structure générale du méta-modèle des diagrammes de classes et dérivation en B	67
4.4.2	Étude de cas : système d'impression	68
4.5	Dérivation et vérification des diagrammes de collaboration	72
4.5.1	Structure générale du méta-modèle des diagrammes de collaboration et dérivation en B	72

4.5.2	Étude de cas : système d'impression	73
4.5.2.1	Dérivation en B	73
4.5.2.2	Vérification des éléments du modèle du diagramme de collaboration	74
4.6	Dérivation du méta-modèle des diagrammes d'état-transitions en B	78
4.7	Synthèse	78
5	Vérification de diagrammes statiques UML	79
5.1	Étude de cas	80
5.2	Dérivation des diagrammes statiques UML en B	81
5.2.1	Dérivation du diagramme de classes	81
5.2.2	Dérivation des contraintes UML et OCL	81
5.2.3	Dérivation du diagramme d'objets	83
5.3	Vérification des contraintes du système par combinaison des dérivations	86
5.3.1	Vérification des contraintes simples	86
5.3.2	Vérification des contraintes combinées	88
5.4	Synthèse	89
6	Vérification des scénarios UML en utilisant B	91
6.1	Dérivation d'un scénario en B	92
6.2	Vérification de spécifications UML à l'aide des obligations de preuve de B	94
6.2.1	Obligations de preuve dans une machine abstraite	94
6.2.2	Obligations de preuve dans une implantation (ou raffinement)	94
6.3	Étude de cas	95
6.3.1	Description du système	95
6.3.2	Modélisation des scénarios du système à l'aide de diagrammes UML et de contraintes OCL	95
6.3.3	Vérification des spécifications UML et OCL des scénarios	96
6.4	Synthèse	98
III	Prise en compte de l'objet dans le développement formel B	99
7	Prise en compte de certains types d'association des approches objets pour B	101
7.1	Les relations de composition entre machines abstraites B	102
7.2	Les types d'associations des approches orientées objets	102
7.2.1	Association bidirectionnelle	102
7.2.2	Association unidirectionnelle	103
7.3	Prise en compte des différents types d'associations UML en B	103
7.3.1	Spécification	104

7.3.2	Obligations de preuve	104
7.4	Interaction avec d'autres machines abstraites dans le modèle	106
7.5	Synthèse	108
8	Validation de spécifications orientées objets en B	111
8.1	Structure de la machine de simulation	112
8.2	Obligations de preuve préservant l'exécution d'un scénario	113
8.2.1	Obligations de preuve associées à un scénario défini par une séquence d'ap- pels d'opérations	113
8.2.2	Obligations de preuve associées à un scénario incluant une conditionnelle .	114
8.3	Expression de propriétés dynamiques et obligations de preuve	115
8.3.1	Propriété de sûreté	115
8.3.2	Propriété de vivacité	115
8.3.2.1	Propriété de réponse	115
8.3.2.2	Propriété de précedence	116
8.4	Étude de cas	116
8.4.1	Présentation de l'étude de cas	116
8.4.2	Spécification UML	117
8.4.2.1	Diagramme de classes	117
8.4.2.2	Diagramme de séquences	117
8.4.2.3	Contraintes du système	118
8.4.3	Spécification B	118
8.4.3.1	Machines abstraites	118
8.4.3.2	Machine de simulation	118
8.4.4	Validation du scénario <code>Entry_Building</code>	120
8.5	Synthèse	122
9	L'outil support au développement B par objets	123
9.1	Prise en compte des nouvelles obligations de preuve	123
9.2	L'outil B4free	124
9.3	Description de l'outil Boo	125
9.4	Synthèse	126
	Conclusion	127
	Synthèse de la contribution	127
	Vérification et validation formelle des spécifications UML	127
	Prise en compte de l'objet dans le développement formel B	128
	Perspectives	129
	Environnement de développement B par objets	129
	Extension d'autres propriétés objets dans la méthode B	129

Vérification de consistance entre diagrammes d'état-transitions et diagrammes de séquences	129
D'autres perspectives	130

Bibliographie **131**

A Glossaire B **137**

A.1 Logique	137
A.2 Les ensembles	137
A.3 Les relations binaires	138
A.4 Les fonctions	139
A.5 Les substitutions généralisées	139

B La spécification du système de passage à niveau **141**

B.1 Le composant Types.mch	141
B.2 Le composant Basic.mch	141
B.3 Le composant System.mch	143
B.4 Le composant System_imp.imp	144

C La spécification du système de contrôle d'accès **145**

C.1 Le composant Card.mch	145
C.2 Le composant Controller.mch	145
C.3 Le composant Reader.mch	146
C.4 Le composant Door.mch	147

TABLE DES FIGURES

1.1	Structure générale d'une machine B	19
1.2	Un exemple de machine abstraite	21
1.3	Développement par niveaux	22
1.4	Un raffinement de la machine Exemple1	23
1.5	Structure générale d'un raffinement	24
1.6	Définition d'un événement	27
2.1	Diagramme d'objets (extrait de [Booch 98])	33
2.2	Diagramme de classes (extrait de [Meyer 01])	33
2.3	Diagramme de collaboration (extrait de [Booch 98])	34
2.4	Un exemple de diagramme de séquence UML2.0	36
2.5	Diagramme d'état-transitions (extrait de [Meyer 01])	37
2.6	L'architecture UML en couches	38
2.7	Les paquetages de haut niveau	39
2.8	Le paquetage Foundation	39
2.9	Une partie du paquetage Core - Backbone	40
2.10	Le paquetage Behavioral Elements	41
3.1	Dérivation générale d'une classe et de ses attributs	46
3.2	Dérivation d'un attribut avec multiplicité	47
3.3	Dérivation d'une association	48
3.4	Dérivation partielle d'une opération	48
3.5	Dérivation générale des états	49
3.6	Dérivation d'une transition	49
3.7	Dérivation d'un événement	50
3.8	Diagramme de collaboration	51
3.9	Organisation en couches des opérations UML de diagrammes de collaboration	52
4.1	Relation entre classe composante et classe composite	65
4.2	Spécification de l'opération ajoutée collectData	67
4.3	Structure générale du méta-modèle des diagrammes de classes et sa dérivation en B	67
4.4	Relation entre modèle et méta-modèle du diagramme de classes	68
4.5	Machine abstraite B correspondant à l'opération createComputer	69
4.6	Spécification complète de la machine abstraite CreateComputer	70
4.7	Structure des machines B dérivées à partir du diagramme de classes du système d'impression	71
4.8	Machine Types du diagramme de classes du système d'impression	71

4.9	Dérivation en B de la règle bien-formée <i>WFR1</i>	72
4.10	Structure générale du méta-modèle des diagrammes de collaboration et dérivation en B	73
4.11	Relation entre le modèle et le méta-modèle du diagramme de collaboration	74
4.12	Machine abstraite B correspondant à l'objet <i>Interaction</i>	75
4.13	Machine <i>Types</i> du diagramme de collaboration du système d'impression	76
4.14	Dérivation en B de la règle de bonne formation <i>WFR2</i>	76
4.15	Dérivation en B de la règle de bonne formation <i>WFR3</i>	77
4.16	Structure générale du méta-modèle des diagrammes d'état-transitions et sa dérivation en B	78
5.1	Diagramme de classes d'un système de gestion	80
5.2	Machine abstraite B correspondant à la classe <i>Person</i>	82
5.3	Un exemple de diagramme d'objets du système de gestion	84
5.4	Machine abstraite B de l'objet <i>pp1</i>	85
5.5	Spécification additionnelle pour la machine abstraite <i>Person</i>	87
5.6	Machine abstraite <i>Model</i>	88
5.7	Structure générale des machines B	89
6.1	Diagramme de collaboration avec les cas de dépendance circulaire	93
6.2	Structure des machines dérivées	93
6.3	Diagramme de classes du système de passage à niveau	96
6.4	Diagramme de collaboration du scénario de fermeture de la barrière	96
7.1	Exemple de relation bidirectionnelle	103
7.2	Exemple de relation unidirectionnelle	103
7.3	Spécification générale d'une machine abstraite B dérivée à partir d'une classe UML105	
7.4	Deux machines abstraites correspondant à deux classes reliées par une association bidirectionnelle	107
7.5	Interaction avec d'autres machines dans le modèle	107
7.6	Exemple de couplage entre relations bidirectionnelle et unidirectionnelle	108
7.7	Machine <i>Flight</i>	108
7.8	Machine <i>Plane</i>	108
7.9	Machine <i>Engine</i>	108
8.1	Structure de la machine de simulation	112
8.2	Diagramme de classes du système de contrôle d'accès	117
8.3	Diagramme de séquences du scénario d'entrée dans un bâtiment	118
8.4	Machine de simulation avec un scénario	119
9.1	Un exemple de la machine abstraite intermédiaire	124
9.2	Architecture de Boo	125
9.3	Interface de l'outil Boo	126
1	Diagramme d'état-transition généré à partir de ligne de vie d'un objet de classe <i>Reader</i>	129
2	Diagramme d'état-transition d'un objet de classe <i>Reader</i>	130

INTRODUCTION

Le succès actuel des objets vient de leur utilisation dans la pratique de l'industrie du logiciel, mais aussi de la théorie et des méthodes de génie logiciel. L'objet est maintenant connu dans plusieurs domaines de l'informatique: langages de programmations, bases de données, ... Les notations à objets [Abadi 96, Meyer 88] sont souvent utilisées pour la production de systèmes logiciels complexes.

UML (Unified Modelling Language) [Booch 98] est un langage *semi-formel*. Il propose des notations graphiques aisément accessibles grâce à leur intuition, pour modéliser des objets du système. Cependant ce langage présente un risque d'incohérence et d'inconsistance dans les modèles construits, il manque des outils supports efficaces pour les vérifier et valider. Pour résoudre ce problème, l'utilisation de méthodes formelles [Hinchey 95] est considérée comme l'une des solutions grâce à leur complémentarité.

D'une part, les techniques à objets ont besoin des méthodes formelles afin de fournir des bases pour la vérification et la validation des applications. Les méthodes formelles permettent de définir un cadre sémantique rigoureux pour les modèles à objets. Elles fournissent des outils de preuve et d'animation de spécifications. Les composants définis dans un tel cadre peuvent être réutilisés de manière sûre et rigoureuse lors de nouveaux développements. Grâce aux mécanismes de raffinement des méthodes formelles, il est possible de montrer l'adéquation entre une spécification de haut niveau et le code exécutable.

D'autre part, les méthodes formelles ont besoin des techniques à objets pour leur représentation intuitive et synthétique. La modélisation à objets des spécifications favorise la compréhension du modèle et la communication entre les différents acteurs du développement. Les notations semi-formelles à objets fournissent des mécanismes de structuration et de développement nécessaires pour la construction de gros systèmes. Elles enrichissent par les notions d'héritage, d'instanciation et d'agrégation, les relations entre les différents modules de la spécification. La spécification, la preuve et le raffinement se font plus aisément sur de petits modules bien structurés et organisés. Les modules peuvent être spécifiés et implantés indépendamment et par des personnes différentes favorisant ainsi non seulement la réutilisation de la spécification mais également de la preuve.

L'intégration entre les méthodes formelles et les notations à objets peut être réalisée selon plusieurs approches, chaque approche apporte des intérêts différents. Quatre types d'approches ont été répertoriées [Meyer 01], celles-ci se différencient par le degré ou le type d'intégration souhaité et le formalisme de départ:

- **Intégration par adjonction.** Il s'agit d'une intégration partielle ; dans cette approche, une partie de la description informelle du modèle à objets est remplacée par une expression formelle.

- **Intégration par extension.** Ces approches enrichissent les langages de spécifications formels par des concepts objets. Elles offrent une notation rigoureuse complète pour définir la structure et le comportement des objets.
- **Intégration par définition.** Cette approche a comme objectif de considérer la notation à objets comme une véritable notation formelle. C'est une approche pratique, dont le but est de fournir une sémantique aux concepts du langage et de développer des outils d'analyse rigoureux qui permettent de manipuler directement les modèles à objets. L'intérêt d'une telle approche est de préserver les habitudes du développeur et de lui cacher toute la partie formelle.
- **Intégration par dérivation.** Cette approche est assez simple à mettre en œuvre et agréable à utiliser. Elle consiste en une traduction des modèles à objets vers des modèles formels ou inversement. L'analyse effectuée sur la spécification formelle obtenue est rigoureuse et facilite la détection des erreurs.

Contexte

En continuant les travaux d'intégration par dérivation de UML vers B [Meyer 01, LeDang 02a], ce travail de thèse s'inscrit dans le cadre général de la vérification de spécifications UML en utilisant la méthode B.

Méthode formelle B

La *méthode formelle B* permet d'aborder les différentes étapes de la construction, depuis une spécification abstraite jusqu'au codage en utilisant des raffinements successifs. L'objectif fondamental de B est de produire des logiciels sûrs ; pour cela chaque étape du développement doit être prouvée.

Comme toutes les méthodes formelles, B bénéficie de la rigueur et de la précision d'un formalisme et d'une sémantique mathématique. Son utilisation permet une meilleure compréhension du domaine d'application et des facilités d'abstraction. La vérification de certaines propriétés du modèle construit est réalisable par l'intermédiaire de preuves.

La méthode B possède d'autres avantages que ceux classiquement évoqués pour les méthodes formelles. Dès son origine, elle a été conçue pour être pratique et applicable dans l'industrie. Elle dispose d'outils robustes, commercialisés et utilisés dans le monde académique et industriel ([Behm 99]). Son langage, bien qu'il repose sur des fondements mathématiques solides, est très proche des langages impératifs communément utilisés par les informaticiens. La méthode B couvre également une très grande partie du cycle de vie du logiciel. Elle permet d'exprimer une abstraction du système à construire puis, par des approximations successives appelées raffinements, d'obtenir un modèle plus concret. Celui-ci pourra être traduit dans un langage de programmation en utilisant des outils disponibles. Des preuves sont réalisées tout au long de la construction, elles permettent non seulement de vérifier la cohérence intrinsèque des modèles construits mais aussi la cohérence entre les modèles des différents niveaux d'abstraction.

Malgré ces efforts, B souffre d'un certain nombre de carences héritées des méthodes formelles. Les spécifications formelles sont connues pour leur difficulté d'écriture. Les preuves bien que simplifiées par les outils existants restent laborieuses, notamment pour la vérification des raffinements. Elles n'empêchent pas l'écriture d'une mauvaise spécification. Elles permettent d'identifier les oublis ou les incohérences de ce qui a été spécifié mais pas de ce qui a été demandé par le client.

Notations à objets UML

UML (Unified Modelling Language) propose des notations graphiques de modélisation par objets. Il a été conçu pour visualiser, spécifier, construire et documenter les artefacts d'un système.

L'utilisation et les avantages des notations à objets sont maintenant reconnus par une grande majorité de la communauté scientifique mais surtout industrielle. Ce type de notation est adapté à la plupart des acteurs du développement. Il offre des mécanismes de structuration très riches tout en gardant une représentation intuitive et synthétique du système à construire. L'application est structurée en composants indépendants qui favorisent la réutilisation du logiciel.

UML est en passe de devenir le standard dans le domaine des modèles à objets et a été adopté par l'OMG (Object Management Group). Ses auteurs ont intégré le meilleur des techniques à objets pour parvenir à un ensemble cohérent et bien structuré. De plus, ce langage est supporté par un nombre très important d'outils. C'est de fait une des techniques les plus utilisées pour la modélisation de systèmes informatisés.

Malgré ces avantages indéniables, l'utilisation d'UML pose certains problèmes. Ils proviennent essentiellement de l'absence d'une sémantique précise pour les notations utilisées. La sémantique est fondée sur un méta-modèle mais reste décrite en langage naturel. La construction de systèmes complexes conduit à des modèles UML ambigus et provoque de nombreux problèmes d'interprétation.

Dérivation d'UML en B

Le travail de thèse de Meyer [Meyer 01, Meyer 99] porte sur la dérivation d'une spécification B à partir des diagrammes de classes et d'état-transitions. Ce travail poursuit les travaux de Lano [Lano 96] et Nguyen [Nguyen 98] pour dériver une spécification B à partir de modèles OMT. Les schémas de dérivation proposés par Meyer permettent de formaliser en B tous les concepts structurels UML : classe, attribut, agrégation, composition, héritage et état. Grâce à ces schémas, les données (variables, ensembles, constantes) des spécifications B dérivées sont générées automatiquement. Le travail de thèse de Ledang [LeDang 02a] poursuit ce travail avec la dérivation des concepts comportementaux comme les opérations, les événements, ... Il propose une approche pour modéliser en B les opérations UML à partir d'un diagramme de classes, il a utilisé le support de raffinement de la méthode B pour modéliser la réalisation de ces opérations par la relation "*appellant-appelées*" dans les diagrammes de collaboration [Ledang 01] et par la relation "*événement-actions*" dans les diagrammes d'état-transition [LeDang 02b]. De plus, il a défini des schémas de transformation des expressions OCL, les contraintes supplémentaires au sein des modèles UML, en B [Ledang 02c]. Ce qui permet de dériver systématiquement en B non seulement les invariants de classes en OCL, les conditions de gardes au sein des diagrammes d'état-transitions mais aussi les spécifications OCL de la forme pré- et postcondition des concepts UML comportementaux comme des opérations et des événements UML.

Objectifs de notre travail de recherche

Dans les quatre approches possibles de couplage des notations à objets et des méthodes formelles, les approches de dérivation de UML en B appartiennent à celle de l'intégration par dérivation. Ses objectifs principaux sont:

- L'utilisation des notations et concepts à objets UML comme une première représentation

intuitive du système construit pour aider et documenter la spécification formelle B. Les développeurs disposent de deux vues du système spécifié, l'une en UML et l'autre en B.

- La méthode B est utilisée pour analyser et vérifier les défauts au sein des spécifications UML grâce à ses outils supports tel que l'AtelierB, B-Toolkit ou B4free.

Cependant, les approches de dérivation UML en B proposées n'abordent que le premier objectif mais elles ne permettent pas d'analyser et de vérifier des propriétés attendues dans la spécification UML. Notre thèse se concentre sur la vérification et la validation de spécifications UML en utilisant sa transformation en B. De plus, comme la méthode B est une méthode formelle intéressante pour la spécification et la vérification des systèmes, nous proposons une approche qui supporte certaines propriétés des techniques orientées objets. Cette approche permet d'utiliser B comme une technique à objets afin de modéliser et valider les modèles (correspondant à une approche d'intégration par extension).

Contribution

La contribution de cette thèse porte sur deux aspects : la vérification de spécifications UML en utilisant B et la prise en compte de l'objet dans le développement formel B.

Vérification de spécifications UML en utilisant B

Pour vérifier des spécifications UML, on doit vérifier la sémantique des diagrammes UML utilisés ainsi que la cohérence entre les différents diagrammes dans un modèle. Les approches de dérivation de UML en B proposées ne peuvent ni prouver complètement la sémantique des diagrammes UML, ni vérifier la cohérence entre les différents diagrammes.

Les règles de dérivation d'UML vers B portent sur les modèles UML, apportant une sémantique à ces modèles UML en termes de B. Ceci est insuffisant pour vérifier la sémantique d'UML, celle-ci étant en partie exprimée à l'aide des règles de bonne formation sur les méta-modèles UML. En vu de vérifier la spécification relativement aux exigences du système exprimées par des contraintes dans les diagrammes UML et les contraintes OCL, les dérivations existantes permettent de vérifier la conformité entre les pré- et postconditions par rapport à l'invariant de classes. Cependant, cela n'est pas suffisant pour vérifier les propriétés des spécifications orientées objets.

En conséquence, nous proposons une nouvelle approche de dérivation du méta-modèle UML en B afin de vérifier complètement la sémantique des diagrammes UML.

Le méta-modèle UML définit la syntaxe et la sémantique des modèles UML. Les règles de bonne formation décrivent des contraintes sur les attributs et les associations présentés dans le méta-modèle. Chaque règle est définie comme un invariant sur des instances des méta-classes. A partir de la dérivation d'un objet d'une méta-classe en B et une analyse des règles bonne formation de la sémantique UML proposée par OMG [OMG 03], nous avons défini une procédure générique pour dériver des méta-modèles UML en B avec des instantiations de :

- diagrammes de classes,
- diagrammes d'état-transitions, et
- diagrammes de collaboration.

Cette procédure se compose de :

- la transformation des règles de bonne formation du méta-modèle UML en des invariants de spécification B,
- la transformation des instances des méta-classes en des machines abstraites B,
- l'association des attributs de ces objets dans les invariants exprimant des règles de bonne formation,
- l'utilisation l'outil AtelierB pour prouver la spécification B.

Les obligations de preuve liées à la préservation d'invariant permettent de prouver la cohérence des éléments UML relativement à sa sémantique. En fait, avec B, nous montrons que des règles exprimées dans le méta-modèle UML sont vérifiées par le modèle UML (chapitre 4).

Pour l'analyse de la consistance des propriétés exprimées dans un même diagramme ou dans différents diagrammes, le travail s'est concentré sur deux problèmes :

- La vérification de diagrammes statiques en utilisant B. L'état d'un système est une instance valide si toutes les contraintes imposées par le modèle sont satisfaites par les instances. Pour vérifier cette propriété, nous prenons un diagramme de classes comme modèle, des contraintes OCL associées à ce modèle et des instances du modèle (comme par exemple des diagrammes objets) pour les transformer en B. Les contraintes d'association dans le diagramme de classes et les contraintes exprimées par OCL fournissent des invariants à la spécification B. Nous avons transformé les diagrammes de classes, les diagrammes objets et les contraintes OCL en machines abstraites B, puis intégré ces machines abstraites à une spécification B complète permettant d'analyser la consistance entre les diagrammes statiques UML et les contraintes OCL (chapitre 5).
- La vérification de diagrammes dynamiques en utilisant B. Nous avons abordé le couplage UML et B à l'aide d'un scénario présenté par un diagramme de classes et des diagrammes de collaboration. Le scénario est transformé en B avec une amélioration de la dérivation des diagrammes de collaboration proposés par Ledang [LeDang 02a], visant à résoudre les limites des transformations existantes. Nous avons également démontré certaines propriétés des spécifications UML/OCL analysées à partir de la preuve de la méthode B, à savoir la consistance entre les pré- et postconditions des opérations séquentielles et des opérations décomposées (chapitre 6).

Prise en compte de l'objet dans le développement formel B

La méthode B est non seulement un langage utilisé pour vérifier des spécifications *semi-formelle* à l'aide de ses outils performants mais elle est également un langage de modélisation intéressant utilisant les notations mathématiques. En nous basant sur la dérivation d'UML vers B, nous proposons d'utiliser B comme un langage orienté objet pour modéliser des systèmes.

Dans les approches orientées objets utilisant B, les machines abstraites jouent un rôle équivalent à celui des classes dans les notations orientées objets. Cependant, certaines relations entre classes n'ont pas d'équivalent à celles entre machines abstraites B. Ceci correspond au paradigme "d'un écrivain/plusieurs lecteurs" et aux contraintes structurelles de B. C'est le cas par exemple, des associations bidirectionnelles. Pour faciliter l'utilisation de B dans la modélisation orientée objets, nous proposons une approche permettant de prendre en compte ces types de relation entre classes des approches objets pour les machines abstraites B. Les obligations de preuve pro-

posées permettent de prouver la préservation des invariants des classes dans le modèle (**chapitre 7**).

De plus, la validation formelle de la spécification permet aux développeurs d'avoir au plus tôt l'assurance de la complétude et de la cohérence des fonctionnalités du logiciel décrites dans ses spécifications. Les diagrammes de séquences d'UML expriment des scénarios dans les approches orientées objets, dans lesquels l'aspect de communication est prédominant, base pour la description du test [Pickin 04]. Pour assurer la correction de spécifications orientées objets utilisant des notations B présentées ci-dessus, nous proposons une approche permettant de valider ces spécifications par les interactions des opérations dans un scénario. Nous commençons à partir des spécifications UML sous la forme de diagrammes de classes et de diagrammes de séquences qui expriment des scénarios modélisant le comportement du système (**chapitre 8**).

Le diagramme de classes est dérivé en machines abstraites B. Une machine de simulation est dérivée automatiquement à partir de diagrammes de séquences, renforcée par les contraintes dynamiques du système.

A la fin du processus de construction, nous avons des spécifications orientées objet en B à notre disposition. Les spécifications B sont données au prouveur de B permettant de valider :

- la consistance de l'exécution séquentielle entre les opérations dans un scénario exprimé par des diagrammes de séquences, i.e la préservation des invariants des machines abstraites et les préconditions des opérations,
- la préservation des propriétés dynamiques lors de l'exécution de scénarios.

Pour prendre en compte les obligations de preuve proposées dans les approches du développement formel B par objets présentées ci-dessus, nous avons implanté un outil, Boo, permettant de spécifier les machines abstraites et la machine de simulation (**chapitre 9**). Cet outil permet de générer les nouvelles obligations de preuve qui sont ensuite prouvées par les outils supports de B existants.

Plan du mémoire

Ce mémoire se compose de trois parties: (i) la première partie présente les constituants de base de cette thèse, à savoir une introduction aux notations B, UML, OCL qui sont concernées dans la dérivation d'UML en B. L'état de l'art des approches de transformation d'UML en B est également présenté et les approches de l'intégration par extension des méthodes formelles avec les objets sont introduites dans cette partie; ii) la seconde partie présente nos contributions concernant la vérification et la validation de spécifications UML à partir de leur transformation en B; iii) la troisième partie présente notre approche d'extension de B avec les objets permettant de spécifier et valider des modèles orientés objets en utilisant les notations B. Voici un bref survol des chapitres de ce document.

Le **chapitre 1** présente une vue générale des méthodes formelles et de la méthode B.

Le **chapitre 2** présente la construction par objets et les notations UML utilisées dans les chapitres suivants.

Le **chapitre 3** fait un état de l'art sur la vérification et validation formelle et résume les approches de dérivation de UML en B. Une vue générale des approches d'extension des méthodes formelles par objets est donnée.

Le **chapitre 4** présente la dérivation du méta-modèle UML vers une spécification formelle B. Il décrit la dérivation du méta-modèle des diagrammes de classes, des diagrammes de collaborations et des diagrammes d'état-transitions pour vérifier la sémantique des spécifications UML.

Le **chapitre 5** introduit une approche de validation des contraintes UML et OCL. En héritant de la dérivation des diagrammes de classes, nous proposons une dérivation de diagrammes objets et les intégrons dans une spécification complète.

Le **chapitre 6** améliore la transformation de diagrammes de collaborations en B proposés dans les travaux de Ledang [LeDang 02a] pour vérifier les propriétés des opérations composées/décomposées et des opérations séquentielles.

Le **chapitre 7** présente notre approche de prise en compte de certains types d'association entre classes des approches orientées objets pour les machines abstraites B.

Le **chapitre 8** présente une proposition permettant de valider des scénarios de comportement des systèmes exprimés en UML en intégrant des contraintes dynamiques.

Le **chapitre 9** présente notre outil Boo qui permet de générer automatiquement les obligations de preuve proposées dans les chapitres précédents.

PREMIÈRE PARTIE :

Étude des bases et état de l'art

Cette partie a pour objectif de présenter le contexte de cette thèse. Nous introduisons les caractéristiques des méthodes formelles et plus précisément la méthode B. Nous présentons les principes de la construction par objets et plus précisément un sous-ensemble des notations UML et OCL. Nous présentons un état de l'art des travaux sur la vérification et la validation formelles des spécifications UML, des travaux de dérivation d'UML en B pour la vérification des spécifications UML en utilisant B. Enfin, nous introduisons des approches d'extension des méthodes formelles par objets pour les comparer avec notre contribution de développement B par objets.

Introduction aux méthodes formelles et à la méthode B

SOMMAIRE

1.1	Introduction aux méthodes formelles	13
1.1.1	Définition	13
1.1.2	Utilisation	14
1.1.3	Classification	15
1.2	La méthode B	16
1.2.1	Présentation informelle	16
1.2.2	Fondements de la méthode B	18
1.2.3	Extensions	27
1.3	Synthèse	28

Les méthodes formelles contribuent à la construction de spécifications dont l'interprétation tient moins à une intuition humaine qu'à l'utilisation de méthodes liées aux mathématiques. L'utilisation des méthodes formelles au sein du développement de logiciels permet de concevoir des produits sûrs.

L'objectif, dans ce chapitre, est d'introduire d'une manière générale les méthodes formelles pour décrire plus précisément la méthode B. Nous commençons par définir ce qu'est une méthode formelle, nous discutons de son utilisation et présentons les différentes classes de méthodes existantes. Nous abordons ensuite la méthode B dont nous détaillons les idées et concepts de base nécessaires pour la compréhension de la suite de cette thèse.

1.1 Introduction aux méthodes formelles

Les méthodes formelles ont été initiées d'une part, par les travaux du Programming Research Group de l'Université d'Oxford qui consistaient à exploiter les mathématiques pour la spécification des systèmes informatiques [Hayes 92], et d'autre part, par les études visant à raisonner plus abstraitement sur les langages de programmation [Meyer 92].

1.1.1 Définition

Une méthode est dite formelle si elle est fondée sur un langage avec une syntaxe et une sémantique précises, construit sur des bases théoriques. Ces bases sont généralement mathématiques ; des

raisonnements sur la spécification sont alors possibles : syntaxe et sémantique sont accompagnées de règles de déduction qui permettent de démontrer des propriétés d'une spécification.

1.1.2 Utilisation

Spécification. La spécification est reconnue comme une étape cruciale dans le développement du logiciel. Des travaux ont prouvé que les erreurs de spécification sont malheureusement fréquentes et les plus coûteuses à corriger [Boehm 82]. Ce problème peut en partie être résolu par l'utilisation de spécifications formelles [Choppy 88], dont l'utilité des mathématiques et du formalisme dans les spécifications a été montrée dans [Abrial 84, Meyer 85].

Dans le cycle de vie d'un logiciel, les méthodes formelles peuvent être utilisées de plusieurs manières et dans différentes étapes du développement. La définition des spécifications de l'application est réalisée dans un langage rigoureux pour éviter des ambiguïtés au niveau de son interprétation.

Développement. Une fois qu'une spécification a été développée, elle peut être utilisée comme référence pendant le développement du système concret (mise au point des algorithmes, réalisation du logiciel). Par exemple :

- Si la spécification formelle est dotée d'une sémantique opérationnelle, le comportement observé du système concret peut être comparé avec le comportement de la spécification (qui elle-même doit être exécutable ou simulable). Une telle spécification peut faire l'objet d'une traduction automatique vers un langage de programmation.
- Si la spécification formelle est dotée d'une sémantique axiomatique, les préconditions et postconditions de la spécification peuvent devenir des assertions dans le code exécutable. Ces assertions peuvent être utilisées pour vérifier le fonctionnement correct du système pendant son exécution (ou simulation), ou mieux encore des méthodes statiques (preuve de théorèmes, model-checking) peuvent être utilisées pour vérifier que ces assertions seront satisfaites pour toute exécution du système.

Vérification et validation. La vérification des spécifications est effectuée en utilisant des techniques de preuves de propriétés ou des techniques d'évaluation (model checking) qui testent toutes les exécutions possibles en vérifiant que la propriété à satisfaire reste vraie. Différentes techniques d'affinage permettent de transformer par étapes successives la spécification, en général très abstraite, vers une implantation. Les étapes d'affinage sont prouvées et garantissent l'adéquation entre les différentes spécifications.

La validation des spécifications peut être obtenue par prototypage/interprétation des spécifications ou par génération automatique de jeux de tests.

Preuves formelles. On peut distinguer deux grandes catégories d'outils permettant la preuve de propriétés sur des modèles formels:

- La preuve automatique de théorèmes, qui consiste à laisser l'ordinateur prouver les propriétés automatiquement, étant donnés une description du système, un ensemble d'axiomes et un ensemble de règles d'inférences. Cependant la recherche de preuves est connue pour être un problème non décidable en général en logique classique, c'est-à-dire que l'on sait qu'il n'existe (et n'existera jamais) aucun algorithme permettant de décider en temps fini

si une propriété est vraie ou fausse. Il existe des cas où le problème est décidable (fragment gardé de la logique du premier ordre par exemple) et où des algorithmes peuvent être appliqués. Cependant, même dans ces cas, le temps et les ressources nécessaires pour que les propriétés soient vérifiées peut dépasser des temps acceptables. Dans ce cas il existe des outils interactifs qui permettent à l'utilisateur de guider la preuve. La preuve de théorèmes, par rapport au model-checking, a l'avantage d'être indépendante de la taille de l'espace des états, et peut s'appliquer sur des modèles avec un très grand nombre d'états, ou même sur des modèles dont le nombre d'états n'est pas déterminé (modèles génériques).

- Le model checking, qui consiste à vérifier des propriétés par une énumération exhaustive et astucieuse (selon les algorithmes) des états accessibles. L'efficacité de cette technique dépend en général de la taille de l'espace des états accessibles et trouve ses limites dans les ressources de l'ordinateur pour manipuler l'ensemble des états accessibles. Des techniques d'abstraction, éventuellement guidées par l'utilisateur, peuvent être utilisées pour améliorer l'efficacité des algorithmes.

1.1.3 Classification

On a classé les méthodes formelles en quatre catégories [Meyer 01] : algébrique, ensembliste, logique et dynamique. Bien que présentées de manière indépendante, celles-ci sont souvent combinées afin de former des méthodes hybrides.

Approches algébriques. Dans l'approche algébrique, plutôt que de définir un type de donnée par la construction d'un ensemble de valeurs, on s'intéresse aux opérations sur ces valeurs et aux propriétés de ces opérations. Une spécification algébrique est un système formel dont le langage est donné par la signature et le système d'inférence est basé sur les axiomes et la déduction équationnelle. Les techniques de spécifications algébriques sont fondées sur la notion de type abstrait de données ou type abstrait algébrique. Décrire algébriquement des types de données consiste à définir trois ensembles : les sortes, la signature des opérations et les axiomes. Les sortes (types) sont des noms servant à représenter des ensembles de valeurs. La signature d'une spécification définit pour chaque opération les sortes de ses paramètres et la sorte résultat. Une expression construite à l'aide des opérations et de variables qui respectent la signature est appelée un terme. Parmi les opérations, certaines sont appelées générateurs, ce sont celles qui serviront à construire les valeurs d'une sorte. Les axiomes décrivent les propriétés des opérations sous forme d'équivalences entre termes. L'application des axiomes à des termes permet d'obtenir d'autres expressions d'équivalence.

Parmi les langages de spécification algébrique connus, nous pouvons citer OBJ [Goguen 96], LARCH [Guttag 93], LPG [Bert 95] ou SACSO [Finance 90]. Une opération de standardisation des outils et des langages de spécifications algébriques a été lancée par le projet ESPRIT COMPASS en coopération avec le groupe de travail IFIP 1.3 sous le nom de projet CoFI¹. Elle a abouti à la description d'un langage algébrique standard nommé CASL [CoF 99].

Approches ensemblistes. Une spécification ensembliste correspond à un système formel dont la syntaxe est donnée par les types et les opérations du modèle abstrait et dont la sémantique est donnée par la théorie sous-jacente au modèle abstrait (théorie des ensembles, logique du

1. <http://www.brics.dk/Projects/CoFI/>

premier ordre, théorie des types). Par opposition aux méthodes algébriques, elles utilisent des types abstraits prédéfinis pour modéliser l'état du système à construire, chaque opération est spécifiée indépendamment en décrivant son effet sur l'état du système.

Parmi les langages de spécification ensembliste connus, citons VDM [Jones 90], Z [Spivey 92] et B [Abrial 96b]. La section qui suit détaille plus précisément une de ces approches : la méthode B.

Approches logiques. Ces approches sont essentiellement basées sur la théorie des types et les logiques d'ordre supérieur. On s'intéresse à la démonstration de programmes en appliquant les théories de la démonstration automatique ou semi-automatique de théorèmes. On met l'accent sur la définition constructive des types. Des exemples connus de systèmes basés sur ces approches sont PVS [Shankar 93b], Isabelle/HOL [Paulson 94] ou Coq [Huet 99].

Approches dynamiques. Souvent liée au domaine des protocoles, la notion de base de ces approches est le processus. Plutôt que de se focaliser sur la structure, on s'intéresse aux interactions entre les différents processus. Ce type de spécification a été développé dans les systèmes de transitions (automates, réseau de Petri), dans les algèbres de processus comme CSP [Hoare 85] ou enfin, dans les logiques temporelles [Arnold 92].

Parmi les approches de spécifications hybrides, nous pouvons citer LOTOS [van Eijk 89]. C'est un langage algébrique : les expressions de contrôle sont caractérisées par une algèbre dont les termes sont des processus ; les données sont caractérisées par une algèbre de types abstraits dont les termes sont des expressions fonctionnelles. L'algèbre des processus de LOTOS vient de CCS [Milner 89], enrichie par certains concepts de CSP et l'algèbre des types abstraits repose sur le langage ACT ONE [Ehrig 85].

Afin de compléter cette étude sur les méthodes formelles, nous recommandons la lecture de synthèse proposé par van Lamsweerde [Lamsweerde 00].

1.2 La méthode B

La méthode B [Abrial 96b] a été introduite par J.-R. Abrial vers le milieu des années 80; le même auteur est à l'origine de la notation Z [Spivey 92]. Cette méthode vise la prise en charge de tous les niveaux du développement d'un logiciel, d'une spécification abstraite à la génération d'un code exécutable.

1.2.1 Présentation informelle

Introduction. La méthode B a été conçue pour être pratique et applicable dans l'industrie. L'objectif initial de cette méthode est de fournir aux industriels une technique mais surtout des outils capables d'aider la construction de logiciels de la façon la plus sûre possible. Contrairement aux techniques classiques de développement basées sur l'exécution, B est fondée sur la preuve. Des modèles sont créés de façon graduelle par approximations successives ; au fur et à mesure de cette construction, des preuves sont réalisées sur ces modèles. La sûreté du système construit dépend de la satisfaisabilité des preuves générées.

La méthode B est utilisée non seulement pour spécifier des systèmes mais également pour les simuler et vérifier.

Modèle B. L'utilisateur de B construit non pas des programmes mais des modèles mathématiques. Ces modèles doivent exprimer les propriétés auxquelles le futur système doit obéir. Un modèle B est similaire à un module (ou encore un objet), il est représenté sous la forme d'une machine abstraite. De façon générale, il se scinde en deux parties :

- une partie statique déclare des variables qui définissent l'état du système modélisé. Ces déclarations sont complétées par des conditions (invariants) qui spécifient les propriétés que l'état du modèle doit toujours satisfaire ;
- une partie dynamique définit les opérations qui décrivent l'évolution des machines abstraites (notamment son état) en utilisant un pseudo-code proche de la programmation classique.

Raffinement. La réalisation d'un système complexe ne peut s'accomplir en une seule fois. La méthode B propose une construction par approximations successives. Ces différentes approximations sont liées entre elles par une relation de raffinement. La notion de raffinement est fondamentale en B. D'une part, on va pouvoir l'utiliser pour introduire les détails de conception du cahier des charges qui n'étaient pas pris en compte auparavant. D'autre part, le raffinement est également utilisé pour concrétiser les modèles, ceci afin de s'orienter vers une implantation.

Substitution. Dans un modèle B, une substitution est un moyen de représenter une transition sur l'état du modèle B. Il existe deux substitutions élémentaires, l'une **skip** pour conserver l'état courant et l'autre pour le modifier : la substitution simple $x := E$ a pour effet d'affecter à la variable x la valeur de l'expression E . Les substitutions complexes sont construites à partir de ces deux substitutions élémentaires et des opérateurs de composition tels que le séquençement (";"), le parallélisme ("||"), le conditionnel ("if ... then ... end"), la boucle ("while ... do ... invariant ... variant"), etc.

Les substitutions généralisées ne sont pas nécessairement déterministes. Une substitution déterministe spécifie une seule transition, tandis qu'une substitution non-déterministe spécifie un ensemble de transitions potentielles. Une seule transition de cet ensemble sera effectivement réalisée lors de l'implantation, et peu importe laquelle. Le non-déterminisme est appréciable lorsqu'il s'agit de spécifier des machines abstraites ou des raffinements. Naturellement, lorsque l'on spécifie une implantation, toutes les substitutions doivent être exécutables et par conséquent déterministes.

Preuve. Les preuves réalisées au cours du développement B s'assurent que le système construit possède bien les propriétés qu'on attendait de lui. Il s'agit de preuves de conservation d'invariants et de correction du raffinement. Dans les premières preuves, on va vérifier que les propriétés décrites dans les invariants sont conservées après exécution des opérations. Les secondes assurent que les propriétés précédemment prouvées dans l'abstraction restent valables dans le raffinement. Il est important de noter que les preuves à satisfaire sont définies de façon rigoureuse dans la méthode, le praticien n'a pas à les écrire. Un outil, un générateur d'obligations de preuve, analyse le ou les modèles concernés et génère les différentes conditions qu'il est nécessaire de prouver pour assurer la cohérence de l'ensemble.

Outils et utilisations. Un des grands avantages de B est de disposer d'ateliers logiciels performants [STE 98, B-C 96] pour sa mise en œuvre dans des projets industriels et d'un outil libre

utilisable en milieu académique [Clearsy]. Ces ateliers possèdent chacun un ensemble d'outils pour :

- l'analyse lexicale et syntaxique des modèles ;
- la vérification de types ;
- la génération des obligations de preuve ;
- la preuve automatique ou interactive des obligations de preuve ;
- la traduction des modèles les plus concrets vers des langages de programmation ;
- la gestion et la documentation des projets.

Grâce à ses outils, la méthode B a rapidement été utilisée dans le monde industriel. C'est principalement dans le domaine des transports ou plus généralement dans les secteurs où la sécurité joue un rôle vital que B est utilisée. L'illustration la plus pertinente de son utilisation a été le projet de métro sans conducteur METEOR [Behm 99] réalisé par Matra Transport International pour le compte de la RATP. B a été utilisée pour s'assurer que les parties du logiciel concernant la sécurité des voyageurs étaient totalement sûres.

1.2.2 Fondements de la méthode B

Dans la méthode B, la spécification, le raffinement et l'implantation sont représentées par un langage unique, le formalisme des machines abstraites. Dans cette partie, nous présentons les primitives de composition fournies ainsi que le concept de raffinement. Nous finissons par évoquer les extensions permettant de traiter les aspects dynamiques.

La syntaxe des notions de base de B est présentée dans un glossaire du chapitre A des annexes du document.

1.2.2.1 Machine abstraite

Les spécifications B sont structurées en machines abstraites. Une machine est constituée, d'une part, de la définition d'un état, et d'autre part, de la description des transformateurs de cet état. Elle encapsule des données, un invariant que ces données doivent respecter et définit des opérations. Les données et l'invariant désignent l'état du système. Le comportement du système est modélisé par les opérations de modification ou de transformation de l'état. Le mécanisme d'encapsulation est un moyen pour protéger l'état d'une machine. Par conséquent, il faut également introduire des opérations de consultation. Ces opérations permettent aux utilisateurs extérieurs de consulter l'état d'une machine.

Structure Générale. La structure générale d'une machine B est présentée dans la Figure 1.1. Celle-ci est structurée en termes de clauses. Le rôle des différentes clauses est le suivant :

- la clause **MACHINE** spécifie le nom de la machine abstraite M suivi éventuellement par une liste de paramètres (X, x) ; ces paramètres sont des ensembles abstraits (X) ou des scalaires (x) ;
- la clause **CONSTRAINTS** définit un prédicat C qui type les scalaires des paramètres et spécifie des contraintes sur les scalaires et ensembles des paramètres ;
- la clause **SETS** déclare une liste d'ensembles abstraits (S) et définit des ensembles énumérés (T) . Les ensembles abstraits sont utilisés pour désigner des objets dont on ne veut pas définir la structure au niveau d'une abstraction. Au contraire, les ensembles énumérés servent à décrire les éléments (a, b) d'un ensemble par énumération ;

```

M.mch
MACHINE  $M(X,x)$  /** Nom de la machine */

CONSTRAINTS /** définition du type et des propriétés des paramètres formels */
   $C$ 

SETS /** déclaration d'ensembles abstraits et définition d'ensembles énumérés */
   $S$  ;
   $T = \{a,b\}$ 

CONSTANTS /** déclaration des constantes */
   $c$ 

PROPERTIES /** définition du type et des propriétés des constantes et ensembles */
   $P$ 

VARIABLES /** déclaration des variables */
   $v$ 

INVARIANT /** définition du type et des propriétés des variables */
   $I$ 

ASSERTIONS /** définition de propriétés des variables */
   $A$ 

INITIALISATION /** initialisation des variables */
   $U$ 

OPERATIONS /** déclaration et définition des opérations */
   $u \leftarrow O(w) \hat{=}$ 
  pre
     $Q$ 
  then
     $V$ 
  end

DEFINITIONS /** déclaration et définition d'alias utilisables dans le texte des autres clauses
  */
   $D(z) \hat{=} X$ 

END

```

Figure 1.1 – Structure générale d'une machine B

- la clause **CONSTANTS** déclare une liste de constantes (c) ;
- la clause **PROPERTIES** définit un prédicat P qui type les constantes et spécifie des conditions sur les constantes et ensembles de la machine abstraite ;
- la clause **VARIABLES** déclare une liste de variables (v) ;
- la clause **INVARIANT** définit un prédicat I qui type et spécifie des contraintes sur les variables ;
- la clause **ASSERTIONS** définit des propriétés déductibles de l'invariant ;
- la clause **INITIALISATION** spécifie une substitution U qui fixe la valeur initiale de chaque variable ;
- la clause **OPERATIONS** définit une liste d'opérations. Ces opérations peuvent être paramétrées en entrée (w) et en sortie (u) par une liste de scalaires. Lorsque l'opération est paramétrée en entrée, une précondition Q est utilisée pour typer les paramètres. La substitution V définit le comportement de l'opération vis à vis de l'état de la machine ;
- la clause **DEFINITIONS** définit des alias (D) éventuellement paramétrés (z) sous la forme d'une expression X pouvant être utilisée dans le corps de toutes les autres clauses.

Les constituants des différentes clauses peuvent être utilisés dans d'autres clauses de la même machine. Le Tableau 1.1 montre les règles de visibilité de ces constituants d'une même machine abstraite :

<i>Constituants</i>	<i>Clauses</i>			
	CONSTRAINTS	PROPERTIES	INVARIANT	OPERATIONS
Paramètres	✓		✓	✓
Ensembles		✓	✓	✓
Constantes		✓	✓	✓
Variables			✓	✓
Opérations				✓

Tableau 1.1 – Règle de visibilité des constituants d'une machine abstraite

La méthode impose certaines contraintes au niveau des machines abstraites :

- le séquençement des substitutions n'est pas autorisé, seule la substitution parallèle doit être utilisée ;
- les substitutions de boucle ne sont pas admises.

Exemple. La machine *Exemple1* (extraite de [Abrial 96a]) donnée en Figure 1.2 définit une variable interne y représentant un ensemble d'entiers ($y \in \mathbb{F}(\mathbb{N}_1)$), y est membre de l'ensemble des parties finies (\mathbb{F}) de l'ensemble des entiers non nuls \mathbb{N}_1 . Cet ensemble est initialisé par un ensemble vide ($y := \emptyset$).

Les seules manipulations autorisées sur cet ensemble sont définies par les deux opérations *enter* et *maximum* qui permettent respectivement d'ajouter une valeur lue à l'ensemble y ($y := y \cup \{n\}$) et de récupérer la valeur maximale actuellement présente dans l'ensemble y ($m := MAX(y)$).

Obligations de preuve. La notion de preuve fait partie intégrante de la méthode B. A chaque étape du développement, la méthode fournit des obligations de preuve, à savoir des propriétés qui doivent être satisfaites pour que le système construit soit correct. Au niveau des machines

```

Exemple1.mch
MACHINE Exemple1

VARIABLES y

INVARIANT  $y \in \mathbb{F}(\mathbb{N}_1)$ 

INITIALISATION  $y := \emptyset$ 

OPERATIONS

 $enter(n) = \text{pre } n \in \mathbb{N}_1 \text{ then } y := y \cup \{n\} \text{ end};$ 

 $m \leftarrow \text{maximum} = \text{pre } y \neq \emptyset \text{ then } m := \text{MAX}(y) \text{ end}$ 

END

```

Figure 1.2 – *Un exemple de machine abstraite*

abstraites, la preuve consiste à vérifier la cohérence du modèle mathématique. Cette cohérence est vérifiée par rapport à l'invariant de la machine. Il s'agit de vérifier l'établissement de l'invariant par l'initialisation et la conservation de celui-ci par les différentes opérations. Ces règles sont les suivantes :

- l'initialisation U établit l'invariant I si le prédicat $C \wedge P \Rightarrow [U]I$ est satisfait ;
- si l'invariant I est établi avant substitution, alors toute opération O de la forme $Q \mid V$ doit le garder établi en vérifiant que le prédicat $C \wedge P \wedge I \wedge Q \Rightarrow [V]I$ est satisfait.

1.2.2.2 Raffinement

Le raffinement en B est le mécanisme de transformation graduelle des spécifications vers des implantations traduisibles en langage de programmation. C'est un moyen de lever le non-déterminisme et de préciser des choix de réalisation.

Un raffinement R d'une machine M est un nouveau composant plus déterministe que M , il modifie éventuellement l'état ou les opérations de M , mais, vu de l'extérieur, il a un comportement compatible avec celui de M . On distingue différents types de raffinement :

- **le raffinement de données** : il consiste en la transformation des variables d'états en variables plus concrètes (plus proches des types de données classiques) ou à l'adjonction de nouvelles variables afin d'introduire des détails de réalisation ;
- **le raffinement procédural** : il concerne le changement du corps des opérations d'une machine par réduction du non-déterminisme, affaiblissement des préconditions ou ajout de détails de conception notamment en utilisant des structures de contrôle plus concrètes.

La méthode B distingue un ultime niveau de raffinement, l'implantation, considéré comme le niveau le plus concret et directement traduisible en langage de programmation. A ce niveau, toutes les variables doivent être définies en termes de structures de données programmables, les

ensembles et les constantes doivent être valués, des structures de contrôle comme les itérations peuvent être utilisées. Il ne doit y avoir ni non-déterminisme, ni précondition, ni parallélisme.

Développement B par raffinements successifs. Les étapes de raffinement concernent le passage d'une machine abstraite à un raffinement, d'un raffinement à un autre raffinement ou encore d'un raffinement à une implantation. Un raffinement se distingue d'une machine abstraite par les points suivants :

- selon le type de raffinement, la clause **MACHINE** est remplacée par la clause **REFINEMENT** (raffinement intermédiaire) ou **IMPLEMENTATION** (raffinement ultime) ;
- il possède une clause **REFINES** qui spécifie le nom de la machine ou du raffinement raffiné ;
- il définit les mêmes opérations (identiques au niveau de la signature) que celles du composant qu'il raffine ;
- il ne spécifie pas explicitement de nouveaux paramètres de composant car ceux-ci sont implicitement donnés par le composant raffiné ;
- l'invariant contient des prédicats de liaison entre les nouvelles variables et les variables abstraites du niveau précédent ; il est appelé **invariant de liaison** ou **invariant de collage**.

Un composant ne peut être raffiné que par un seul autre composant et aucun paramètre n'est admis dans la clause **REFINES**. L'implantation en B est un raffinement spécial et unique contraint par de fortes restrictions :

- l'implantation n'a pas d'état propre ;
- les opérations sont implantées par importation d'opérations d'autres machines ; les opérations de l'implantation ne peuvent pas modifier ou référencer les variables des composants importés, seules les opérations importées y sont autorisées ;
- les paramètres des composants importés sont instanciés dans la clause **IMPORTS** (voir section 1.2.2.3) ;
- les substitutions non-déterministes et la composition parallèle sont interdites.

Ces contraintes ont pour objectif d'assurer que l'implantation est concrète et uniquement dépendante de la spécification d'autres machines et non de leur implantation. Un exemple de développement classique B par raffinements successifs est donné Figure 1.3. On appelle, en B, ce genre de développement un *développement par niveaux*.

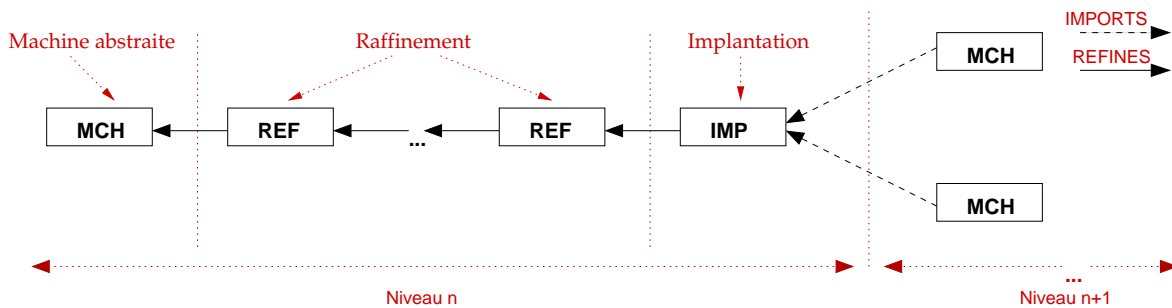


Figure 1.3 – Développement par niveaux

Un raffinement est transitif et monotone. Il est équivalent d'utiliser une machine abstraite ou son raffinement et un raffinement peut utiliser d'autres machines abstraites. Néanmoins en B, l'interface reste toujours la machine abstraite.

```

Exemple2.ref
MACHINE Exemple2

REFINES Exemple1

VARIABLES  $z$ 

INVARIANT  $z \in \text{NAT}$ 

INITIALISATION  $z := 0$ 

OPERATIONS

 $enter(n) = \text{pre } n \in \mathbb{N}_1 \text{ then } z := \text{MAX}(\{z, n\}) \text{ end};$ 

 $m \leftarrow maximum = \text{pre } z \neq 0 \text{ then } m := z \text{ end}$ 

END

```

Figure 1.4 – Un raffinement de la machine Exemple1

Exemple. La Figure 1.4 présente un raffinement de la machine Exemple1 décrite dans la Figure 1.2. Cette machine encapsule une variable z qui n'est plus un ensemble mais un entier initialisé à la valeur nulle. Les manipulations autorisées sur cet entier ont les mêmes signatures que les opérations décrites dans la machine Exemple1. On retrouve les opérations `enter` qui modifie la valeur de z ($z := \text{MAX}(z, n)$) et `maximum` qui fournit la valeur actuelle de z ($m := z$).

Par le raffinement, nous avons vu que les opérations initialement définies à l'aide de manipulation sur un ensemble (voir la machine Exemple1), peuvent être exprimées différemment par des opérations manipulant une seule valeur entière. Les relations entre la machine abstraite et son raffinement s'expriment par les obligations de preuves.

Obligations de preuve. La méthode B supporte le mécanisme de raffinement, ce qui signifie qu'un processus de preuve lui est associé. Il consiste à montrer que le modèle raffiné est cohérent avec le modèle abstrait.

Le raffinement d'une machine abstraite est généralisé par le raffinement de ses opérations. De manière informelle, une substitution T raffine une substitution S si le comportement de T correspond au comportement attendu de S à la condition qu'elles agissent sur le même état. La substitution T est le raffinement de S où S est l'abstraction de T .

La Figure 1.5 montre la structure générale d'un raffinement R affinant la machine abstraite M . Afin d'accomplir les preuves d'un raffinement R d'un composant M , il faut vérifier que :

- l'initialisation U_r raffine l'initialisation U de la machine abstraite. Il s'agit de montrer que l'initialisation du raffinement ne fait rien qui soit contraire à ce que fait l'initialisation de l'abstraction. En d'autres termes, il ne doit pas exister de cas où U établisse le contraire de l'invariant de raffinement I_r , ce qui s'exprime par l'obligation de preuve suivante : $C \wedge P \wedge P_r \Rightarrow [U_r] \neg [U] \neg I_r$;

```

R.ref
REFINEMENT  $R(X,x)$   /** Nom du raffinement */

REFINES  $M$   /** Nom du composant raffiné */

SETS
     $S_r$  ;
     $T_r = \{a_r, b_r\}$ 

CONSTANTS
     $c_r$ 

PROPERTIES
     $P_r$ 

VARIABLES
     $v_r$ 

INVARIANT
     $I_r$ 

INITIALISATION
     $U_r$ 

OPERATIONS
     $u \leftarrow O(w) \hat{=}$ 
    pre
         $Q_r$ 
    then
         $V_r$ 
    end

DEFINITIONS
     $D_r(z_r) \hat{=} X_r$ 

END

```

Figure 1.5 – Structure générale d'un raffinement

- chaque opération O du raffinement raffine bien celle du composant raffiné. Il s'agit de montrer que l'exécution d'une substitution raffinée qui établit l'invariant correspond à une exécution de la substitution du niveau abstrait. En d'autres termes, il faut montrer l'obligation de preuve suivante : $C \wedge P \wedge P_r \wedge I \wedge I_r \wedge Q \Rightarrow Q_r \wedge [V_r] \neg [V] \neg I_r$.

1.2.2.3 Mécanismes de composition

Afin de permettre la construction de spécifications de grande taille et faciliter la modularité, des mécanismes de composition (**INCLUDES**, **EXTENDS**, **IMPORTS**) et de partage (**SEES**, **USES**) ont été définis. Ces mécanismes apparaissent dans les spécifications en termes de clauses additionnelles. Lorsque ces clauses sont introduites, les obligations de preuve présentées ci-avant sont affinées pour intégrer les données récupérées par composition ou partage.

Clauses de composition. La clause **INCLUDES** permet de construire de manière modulaire des machines abstraites ou des raffinements. Lorsqu'une machine est incluse :

- son contenu (excepté ses paramètres et ses opérations) est implicitement ajouté à celui de la machine incluante ;
- les opérations de la machine incluante peuvent utiliser (appeler) les opérations de la machine incluse, les variables de la machine incluse sont uniquement modifiables à l'intérieur de la machine incluante par appel des opérations de la machine incluse.

Constituants de M_2	Clauses de M_1			
	INCLUDES	PROPERTIES	INVARIANT	OPERATIONS
Paramètres				
Ensembles		✓	✓	✓
Constantes		✓	✓	✓
Variables			✓	lecture
Opérations				✓

avec M_1 qui inclut M_2 , "lecture" signifie que les variables sont consultables mais pas modifiables.

Tableau 1.2 – Règles de visibilité de la clause INCLUDES

Les règles de visibilité de cette clause sont présentées dans le Tableau 1.2. Elle est transitive, c'est à dire que si M_1 inclut M_2 et M_2 inclut M_3 alors M_1 inclut implicitement M_3 . D'autre part, une machine ne peut être incluse qu'une seule fois dans l'ensemble d'un projet. Toutefois, l'inclusion d'une même machine par plusieurs machines est possible en utilisant un renommage, les instances ainsi considérées sont différentes. Les opérations de la machine incluse ne sont pas considérées comme des opérations de la machine incluante, elles n'appartiennent pas à l'interface de cette machine. En utilisant, la clause **PROMOTES** on peut promouvoir ces opérations comme des opérations de la machine incluse. L'utilisation de la clause **EXTENDS** à la place de la clause **INCLUDES** est équivalente à une inclusion et à la promotion de toutes les opérations de la machine incluse.

Au niveau de la visibilité interne des constituants d'une machine abstraite (Tableau 1.1), il est à noter que les paramètres, ensembles et constantes d'une machine abstraite peuvent être utilisés comme paramètres de la clause **INCLUDES**.

La clause **IMPORTS** est utilisée uniquement dans une implantation. Elle est analogue à la clause **INCLUDES** utilisée dans la spécification abstraite et les raffinements intermédiaires.

Clause SEES. Elle permet de référencer dans un composant une autre machine abstraite, afin de consulter ses constituants sans les modifier. Si M_1 est une machine abstraite qui voit la machine M_2 , le Tableau 1.3 précise pour chaque constituant de M_2 ses utilisations possibles dans les clauses de M_1 . Cette clause est en général utilisée pour partager la définition d'ensembles ou de constantes à l'intérieur d'un même projet B.

Constituants de M_2	Clauses de M_1			
	INCLUDES	PROPERTIES	INVARIANT	OPERATIONS
Paramètres				
Ensembles	✓	✓	✓	✓
Constantes	✓	✓	✓	✓
Variables				lecture
Opérations				

avec M_1 qui voit M_2 , "lecture" signifie que les variables sont consultables mais pas modifiables.

Tableau 1.3 – Règles de visibilité de la clause SEES

La clause **SEES** n'est pas transitive : si M_1 voit M_2 et M_2 voit M_3 , alors M_1 ne voit pas implicitement M_3 , il faut le spécifier explicitement. Cette clause peut être utilisée dans les machines abstraites, les raffinements et les implantations. Dans le cas d'un raffinement ou d'une implantation, les règles de visibilité évoluent légèrement : les opérations de consultation (ne modifiant pas les variables) de la machine vue sont visibles dans les opérations du raffinement ou de l'implantation.

Clause USES. Elle permet à une ou plusieurs machines d'utiliser les constituants d'une autre machine à l'exception de ses opérations. En général, cette clause est utilisée lorsqu'un composant inclut plusieurs machines pour partager les données d'une des machines incluses à l'intérieur des autres machines incluses. Seule la clause **USES** permet à une machine d'utiliser les noms des paramètres formels d'une autre machine ou encore les variables de celle-ci dans plusieurs autres machines. Nous montrons dans le Tableau 1.4 les utilisations possibles des constituants d'une machine M_2 utilisée à l'intérieur des clauses de la machine utilisatrice M_1 .

Constituants de M_2	Clauses de M_1			
	INCLUDES	PROPERTIES	INVARIANT	OPERATIONS
Paramètres			✓	✓
Ensembles		✓	✓	✓
Constantes		✓	✓	✓
Variables			✓	lecture
Opérations				

avec M_1 qui utilise M_2 , "lecture" signifie que les variables sont consultables mais pas modifiables.

Tableau 1.4 – Règle de visibilité de la clause USES

Comme la clause précédente, la clause **USES** n'est pas transitive. Elle est utilisée uniquement au niveau abstrait et ne peut être ni raffinée, ni importée, ni vue.



Figure 1.6 – Définition d'un événement

1.2.3 Extensions

A l'origine, B n'a pas été défini pour traiter les systèmes distribués ; une extension [Abrial 96a] a été proposée afin de spécifier et concevoir des systèmes où la distribution des exécutions prend une place importante. Cette extension a pour but de couvrir les phases amont du développement, ceci pour définir une simulation mathématique du système global. Par la suite, nous verrons que la prise en compte de ce nouvel objectif ne nécessite pas un changement profond de B, mais seulement une modification au niveau des termes employés et de l'interprétation des opérations.

Modèle global versus modèle local. L'extension de B pour la prise en compte des systèmes distribués tient essentiellement à la sémantique donnée aux machines abstraites et à leurs opérations. Jusqu'à présent, nous avons considéré une machine abstraite comme le modèle d'un futur logiciel informatique, J.-R. Abrial parle de *modèle local* [Abrial 00]. Les opérations représentent des services que ce logiciel fournit à ses utilisateurs. En général, chaque opération est préconditionnée par une contrainte qui statue sur une condition initiale nécessaire pour une bonne utilisation de ce service, sans quoi, le bon fonctionnement de l'ensemble n'est pas garanti. Ces services sont raffinés successivement au fil de la construction, les données des machines et le contenu de ces services peuvent changer mais restent identiques dans leurs formes extérieures (même signature). Lorsqu'on s'intéresse à la modélisation complète d'un ensemble dynamique dans lequel de nombreux agents sont actifs et coopèrent, les machines abstraites sont des objets plus importants que des logiciels informatiques. On parle alors de *modèle global* qui représente la simulation d'une certaine réalité observable. Les machines abstraites sont maintenant des systèmes abstraits qui représentent des composants informatiques, des composants matériels ou des éléments de communication entre les deux. Les opérations ne représentent plus des services mais des événements.

Événement. Commençons par établir une distinction entre une opération et un événement : une opération est appelée par un "agent" alors qu'un événement apparaît dynamiquement et spontanément lorsqu'une condition de déclenchement est validée. Dans un tel modèle, un seul événement au plus peut se produire à un instant donné et son exécution est supposée être instantanée.

La Figure 1.6 présente une définition d'un événement à l'aide d'une opération B. Chaque opération est gardée, et non plus préconditionnée, par une contrainte G qui représente la condition de déclenchement de l'événement. Lorsque cette contrainte n'est pas valide, l'événement est suspendu. Il est à nouveau activé et son action W exécutée lorsque sa garde G redevient vraie.

L'utilisation d'une garde à la place d'une précondition se justifie par le fait qu'une opération gardée s'exécute toujours quand sa garde est valide, et qu'il n'est pas nécessaire de prouver à

l'avance que celle-ci est toujours valide. A l'opposé, une précondition doit toujours être vraie. C'est ce qui doit être prouvé lorsque l'opération est appelée c'est-à-dire avant son exécution.

Propriétés dynamiques. En complément des propriétés dites statiques (invariants), il est possible d'exprimer des propriétés sur la dynamique du système. Il s'agit généralement de propriétés temporelles [Shankar 93a] qui expriment de quelle manière un système est autorisé à évoluer. Elles apparaissent dans la littérature sous différentes formes et sous différents noms : interblocage, sûreté, vivacité, etc.

L'extension de la méthode B présentée dans [Abrial 98] pour exprimer de telles propriétés propose la prise en compte :

- d'invariants dynamiques qui expriment des contraintes sur les transitions, c'est à dire entre deux états successifs ;
- de modalités qui expriment des propriétés sur des séquences de transitions.

Ces propriétés sont respectivement exprimées à l'intérieur de deux nouvelles clauses **DYNAMICS** et **MODALITIES**. Afin d'exprimer des propriétés de la logique temporelle, le langage B est également étendu par des opérateurs (connecteurs) de cette logique. D'un point de vue vérification, l'approche retenue consiste toujours à prouver des obligations de preuve. Celles-ci ont été spécialement définies pour les clauses nouvellement introduites.

Une approche d'intégration et de vérification de propriétés dynamiques a été proposée dans [Julliand 98]. Cette étude propose d'étendre le langage de spécification B par de la logique temporelle linéaire et d'effectuer la vérification de propriétés dynamiques du système par "model checking".

Une autre approche d'intégration de spécification, preuve et raffinement de propriétés de vivacité sous hypothèse d'équité faible dans le B événementiel a été proposée dans [Barradas 05]. Cette approche divise les propriétés de vivacité en deux classes : propriétés de base et propriétés générales. Les obligations de preuve fondées sur le calcul des plus faibles préconditions sont données pour prouver les propriétés de base. Les propriétés de vivacité générales sont prouvées en utilisant les définitions et théorèmes de la logique UNITY.

1.3 Synthèse

- Une méthode formelle est basée sur un langage avec une syntaxe et une sémantique précises construit sur des fondements mathématiques.
- Les méthodes formelles sont utilisées pour spécifier des systèmes, les spécifications obtenues sont vérifiées par des techniques de preuve ou des techniques d'évaluation.
- B est une méthode formelle pour le développement de logiciels prouvés.
- Le développement B consiste à construire des modèles mathématiques exprimant les propriétés auxquelles le futur logiciel doit obéir. Ces modèles sont construits graduellement (raffinés) jusqu'à l'obtention d'un modèle concret traduisible en un programme exécutable.
- Au cours du développement des preuves sont réalisées, celles-ci s'assurent que le système construit possède et vérifie bien les propriétés attendues. Ces preuves consistent non seulement à prouver les différents modèles construits mais également à vérifier leur cohérence au cours des raffinements.
- B est une méthode formelle dotée d'outils performants et industriels pour sa mise en œuvre.

Construction par objets et UML

SOMMAIRE

2.1	Construction par objets	29
2.2	UML	31
2.2.1	Notation UML - les différents diagrammes	31
2.2.2	La sémantique de UML	36
2.2.3	Introduction à OCL	41
2.3	Synthèse	43

Dans ce chapitre, nous introduisons les concepts fondamentaux de la construction par objets. Nous analysons les intérêts et les problèmes soulevés par l'utilisation de cette technologie. Nous présentons ensuite la notation UML de modélisation par objets, sa sémantique et le langage OCL.

2.1 Construction par objets

Le développement orienté objet est une technique de modélisation qui consiste à voir un système comme un ensemble d'objets interagissant les uns avec les autres. Dans la pratique industrielle, l'intérêt pour les objets n'a cessé de croître et de nombreux langages de programmation intégrant cette approche ont vu le jour. Son utilisation s'est généralisée à de nombreuses phases du cycle de développement d'un logiciel, depuis l'expression des besoins jusqu'au codage. Cet intérêt s'explique par le fait que la technologie objet possède un certain nombre d'atouts, parmi lesquels, on trouve :

- la compréhension : les modèles construits sont plus faciles à comprendre car on peut directement les mettre en correspondance avec la réalité, la différence sémantique entre réalité et système est faible;
- la réutilisation et la maintenance : les systèmes construits sont génériques et modulaires, la construction est basée sur la réutilisation de composants existants. Les modifications que l'on apporte ont tendance à être locales avec des impacts très restreints, ce qui favorise la maintenance des systèmes.

Le développement par objets vise à maîtriser la construction de systèmes en les décomposant en sous-systèmes plus ou moins indépendants. Cette technologie est basée sur la définition de composants réutilisables et adaptables.

Les termes et les concepts suivants sont caractéristiques de la technique orientée objet [Meyer 90, Graham 97, Oussalah 97, Tkach 98] :

Objet. Un objet est généralement compris comme une abstraction d'un élément du monde réel ; c'est une traduction informatique d'une entité physique ou d'un concept du monde réel. Un objet se caractérise par une structure et un comportement. La structure d'un objet est composée d'un ensemble de données, attributs ou champs. Le comportement d'un objet est défini par un ensemble de procédures et de fonctions connues sous le nom de méthodes ; ces méthodes sont les seules à pouvoir manipuler la structure de l'objet. On associe souvent à un objet les concepts d'état et d'identité. L'état d'un objet est défini par la valeur de ses attributs à un instant donné. C'est un moyen de mémoriser les effets des méthodes sur celui-ci. L'identité d'un objet constitue le moyen de l'identifier par rapport aux autres objets du système. Un objet est une entité unique et permanente, il doit être créé et peut être détruit.

Classe. Une classe est la description d'une famille d'objets ayant la même structure et le même comportement. La structure d'une classe est décrite par un ensemble de champs appartenant chacun à un type donné et le comportement est décrit par un ensemble de méthodes. Une classe est le patron dans lequel les objets peuvent être créés en appelant certaines méthodes spécifiques (constructeur ou destructeur). Les objets créés sont également appelés instances, la création d'un objet à partir d'une classe étant appelé instantiation de classe.

Encapsulation. L'encapsulation peut être définie comme une forme de masquage de l'information. Les structures de données et les détails d'implantation des opérations sont cachés aux autres objets du système. La seule façon d'accéder à ou de modifier l'état d'un objet est de lui envoyer un message qui déclenche l'exécution de l'une de ses méthodes. Un message est un signal envoyé d'un objet à un autre, qui demande à l'objet récepteur d'appliquer une de ses méthodes, il est similaire à un appel de fonction.

Généricité. La généricité est la possibilité de définir des modules (classes) paramétrés. Les classes peuvent avoir des paramètres génériques formels qui représentent des types. Les classes génériques servent à décrire des structures de données générales et favorisent la réutilisation.

Héritage. L'héritage est un mécanisme de partage et de factorisation des connaissances. Il est souvent considéré comme une relation "est-un" : si la classe *A* hérite de la classe *B*, il est possible de dire que *A* "est-un" *B*. Il permet la réutilisation du comportement d'une classe dans la définition de nouvelles classes. Une sous-classe hérite de la structure de données et des méthodes de sa classe parente (super-classe). La structure et les méthodes de la super-classe peuvent être complétées, modifiées ou tout simplement masquées (gardées telles qu'elles ont été définies dans la super-classe). L'héritage est un mécanisme de spécialisation/généralisation entre classes.

Composition. La composition permet la création d'objets à partir de l'agrégation d'autres objets. L'état d'un composé est défini par la concaténation des états de ses composants.

Polymorphisme. La possibilité de recourir à la même expression pour dénoter différentes opérations est désignée par le terme polymorphisme. Il réfère à la capacité de cacher différentes

significations ou implantations derrière une interface unique. Ainsi, par exemple, le “+” renvoie aussi bien à l’addition entière qu’à l’addition réelle ou qu’à la concaténation de deux chaînes de caractères. Avec ce concept, le même message peut être interprété différemment par des objets de différentes classes et donc produire des effets différents mais appropriés selon le contexte courant. Le polymorphisme est souvent implanté par liaison dynamique. La liaison dynamique désigne la capacité de déterminer la classe d’un objet à l’exécution et donc de déterminer l’implantation adéquate.

Modularité. Un *module* est un élément de petite taille (en général un ou quelques sous-programmes) qui sert, par assemblage à la construction de logiciels. Un module doit être cohérent et autonome. Un ensemble de modules (un logiciel ou un assemblage de modules) doit être bien organisé en architecture robuste.

2.2 UML

UML (Unified Modelling Language) est un langage standard de modélisation par objets résultant d’un effort d’unification des méthodes Booch [Booch 94], OMT [Rumbaugh 94] et OOSE [Jacobson 92] au sein de la société Rational². Aujourd’hui, la dernière version disponible est UML 2.0 [OMG 04]. Dans cette thèse, nous travaillons pour une bonne partie avec la version UML 1.x et les résultats de notre proposition ne sont pas influencés la nouvelle version. En ce qui concerne l’expression de scénarios modélisant le comportement de systèmes, voir **chapitre 8**, nous utiliserons les diagrammes de séquence d’UML 2.0, plus riches du point de vue de l’expression que ceux de UML 1.x.

UML a été conçu pour *visualiser, spécifier, construire et documenter* les artefacts d’un système. Il est fondé sur une notation visuelle qui mélange représentation graphique et textuelle. Même si ce n’est pas un langage de programmation, certains de ses modèles peuvent être directement traduits dans différents langages de programmation.

La description d’UML se compose de :

- ses notations : les notations graphiques pour visualiser la représentation de la sémantique UML,
- sa sémantique : un méta-modèle qui spécifie la syntaxe abstraite et la sémantique des concepts de modélisation UML.

2.2.1 Notation UML - les différents diagrammes

Comme son nom l’indique, UML est fait pour construire des modèles. Pour ce faire, on dispose d’un certain nombre d’outils de représentation : les diagrammes. Un diagramme est une représentation graphique d’un ensemble d’éléments qui constituent un système. Il ne correspond pas à un modèle mais à une représentation de quelques éléments de celui-ci. Il se présente la plupart du temps par un graphe connexe où les sommets sont des éléments et les arcs des relations. Les diagrammes ont pour objectif de visualiser un système sous différents angles, un diagramme seul ne représente en général qu’une vue partielle du système construit. UML 1.x dispose de neuf

2. <http://www.rational.com/uml/>

diagrammes, chacun d'eux représente une vision spécifique du système :

- **Diagrammes de classes** : ils correspondent à une vue statique structurelle du système. Ils représentent un ensemble de classes, d'interfaces, de collaborations et leurs relations. Ce sont les diagrammes les plus fréquents dans une modélisation par objets.
- **Diagrammes d'objets** : ils représentent une vue statique des instances des éléments qui apparaissent dans les diagrammes de classes. Ils sont composés d'un ensemble d'objets et de leurs relations.
- **Diagramme de cas d'utilisation** : ils permettent de modéliser les fonctionnalités des applications et sont représentés par un ensemble de cas d'utilisation, d'acteurs et leurs relations. Ils sont particulièrement importants dans l'organisation et la modélisation des comportements d'un système.
- **Diagrammes de séquence et diagrammes de collaboration** : ce sont des **diagrammes d'interaction**, c'est à dire qu'ils présentent les correspondances entre les différents objets en particulier les messages échangés. Les diagrammes de séquence mettent l'accent sur le classement chronologique des messages (ce sont des scénarios) alors que les diagrammes de collaboration mettent l'accent sur l'organisation structurelle des objets qui envoient et reçoivent des messages.
- **Diagrammes d'état-transitions** : ils correspondent à une vue dynamique d'un système. Ils sont représentés par des automates à états finis composés de transitions, d'évènements et d'activités. Un automate à états finis est un comportement qui spécifie les séquences d'états qu'un objet ou une interaction traverse durant sa vie en réponse à des événements, avec ses réponses et ses activités. Ils sont particulièrement adaptés pour la modélisation du comportement d'une classe, d'une interface ou d'une collaboration. Grâce à un ordonnancement par les événements, ceux-ci sont utiles pour la modélisation de systèmes réactifs.
- **Diagrammes d'activités** : ce sont des diagrammes d'état-transitions décrivant la succession des activités au sein d'un système, les états sont des activités représentant la réalisation d'opérations et les transitions sont déclenchées par la complétion des opérations. Ils sont particulièrement adaptés pour la modélisation des fonctions d'un système et pour décrire le flot de contrôle entre les objets.
- **Diagrammes de composants** : ils présentent l'organisation et les dépendances au sein d'un ensemble de composants et fournissent une vue statique d'implantation d'un système. Ces diagrammes sont fortement liés aux diagrammes de classes, un composant correspond généralement à une ou plusieurs classes, interfaces ou collaborations.
- **Diagrammes de déploiement** : ils représentent la configuration des nœuds de processus en phase d'exécution ainsi que les composants qui y résident. Ils fournissent la vue statique de déploiement d'une architecture. Ils sont liés aux diagrammes de composants car un nœud renferme souvent un ou plusieurs composants.

UML 2.0 présente de nouveaux diagrammes tels les diagrammes de structure composite (composite structure diagram), diagrammes de temps (timing diagram), ... avec lesquels nous n'avons pas travaillé dans cette thèse. Dans les paragraphes suivants, nous allons détailler les concepts autour des diagrammes importants concernant notre travail de vérification et de validation de la spécification UML à partir de sa dérivation en B.

2.2.1.1 Diagrammes d'objets

Un diagramme d'objets est l'image d'un système à un instant donné. Pour cette raison, il est souvent appelé "snapshot". Il peut être utilisé, par exemple, pour illustrer les structures de données complexes ou pour montrer le comportement par une séquence de snapshots durant l'exécution de système (Figure 2.1). Tous les snapshots sont des exemples d'un système, mais ils ne sont pas les définitions du système. La définition de la structure et du comportement d'un système se trouve dans les vues de définition et la construction des vues de définition est l'objectif de la modélisation et de la conception.

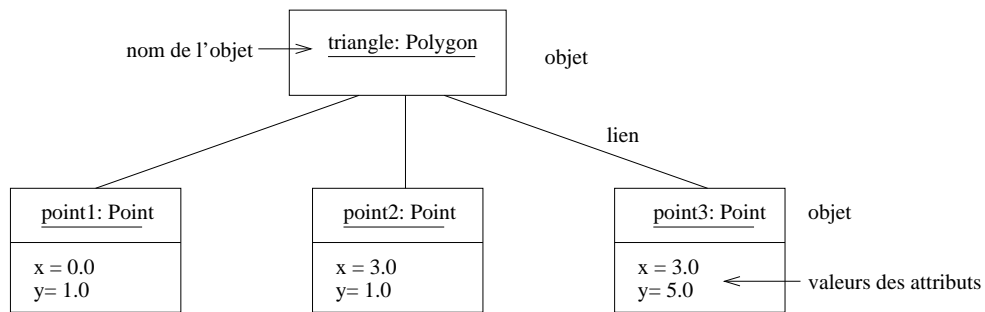


Figure 2.1 – Diagramme d'objets (extrait de [Booch 98])

2.2.1.2 Diagrammes de classes

Les classes sont présentées graphiquement dans les diagrammes de classes; ces diagrammes forment les principaux constituants de la vue logique de l'architecture du système. Un diagramme de classes montre des aspects statiques du modèle et fait abstraction des aspects dynamiques ou temporels. Il déclare cependant des éléments comportementaux comme des opérations, mais la dynamique de celles-ci est exprimée dans d'autres diagrammes : les diagrammes de collaborations, de séquence ou d'activité.

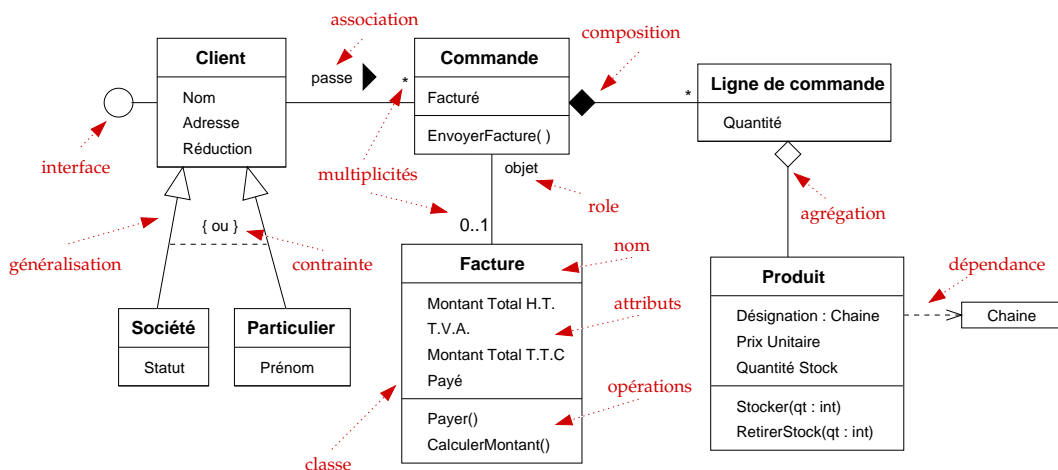


Figure 2.2 – Diagramme de classes (extrait de [Meyer 01])

Les diagrammes de classes (Figure 2.3) contiennent, entre autres, les éléments suivants : des classes ; des interfaces ; des relations de dépendance, de généralisation et d'association. Les associations représentent des relations structurelles entre classes d'objets. La plupart des associations

sont binaires, c'est à dire qu'elles connectent deux classes. Les associations n-aires peuvent généralement se présenter en promouvant l'association au rang de classe et en ajoutant une contrainte qui exprime que les multiples branches de l'association s'instancient toutes simultanément. Il y a cinq types d'associations : classe d'association, agrégation (agrégation de base, agrégation de composition), associations réflexives, associations bidirectionnelles et associations unidirectionnelles.

Comme tous les autres diagrammes, ils peuvent contenir des notes, des contraintes, des stéréotypes et des étiquettes.

2.2.1.3 Diagrammes de collaboration

Les diagrammes de collaboration montrent des interactions entre objets en insistant particulièrement sur la structure spatiale statique qui permet la mise en collaboration d'un groupe d'objets. Les diagrammes de collaboration expriment à la fois le contexte d'un groupe d'objets (au travers des objets et des liens) et l'interaction entre ces objets (par les envois de messages).

Dans [Booch 98] sont cités deux utilisations possibles des diagrammes de collaboration : la réalisation d'un cas d'utilisation ou l'implantation d'une opération.

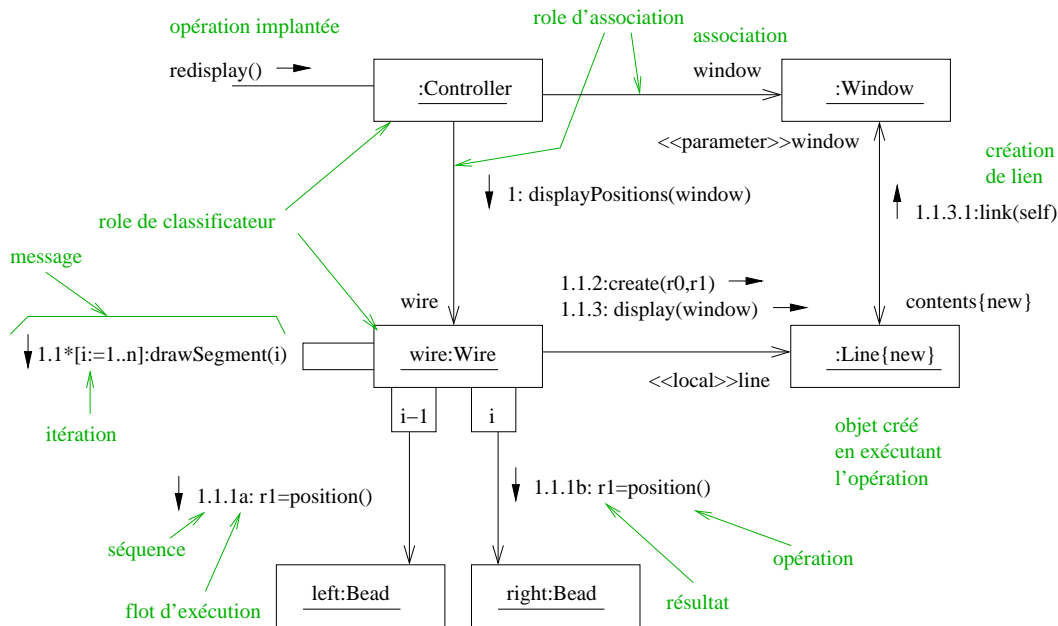


Figure 2.3 – Diagramme de collaboration (extrait de [Booch 98])

Dans notre travail, nous utilisons les diagrammes de collaboration pour représenter la réalisation des opérations dans un scénario et vérifier ses propriétés.

2.2.1.4 Diagrammes de séquence

Les diagrammes de séquence permettent de représenter des collaboration entre objets selon un point de vue temporel; on y met l'accent sur la chronologie des envois de messages. Les diagrammes de séquence contiennent la même information que des diagrammes de collaboration, mais ils soulignent l'ordre des messages au lieu des relations entre les objets. L'ordre d'envoi d'un message est déterminé par sa position sur l'axe vertical du diagramme ; le temps s'écoule "de haut en bas" de cet axe.

Les diagrammes de séquence définis en UML 1.x souffraient cependant d'un gros inconvénient. La quantité de diagrammes à réaliser pouvait atteindre un nombre conséquent dès lors que l'on souhaitait décrire avec un peu de détail les différentes branches comportementales d'une fonctionnalité. Nous donc présentons et utilisons dans cette thèse les diagrammes de séquence d'UML 2.0 [OMG 04]. Rappelons que les diagrammes de séquences d'UML 1.x présentent des messages asynchrones et des messages synchrones entre objets. Les diagrammes de séquences UML 2.0 ajoutent les notations appelées les *fragments combinés* (*combined fragment* ou *inline frame*). Un fragment combiné représente des articulations d'interactions. Il est défini par un **opérateur** et des **opérandes**. L'opérateur conditionne la signification du fragment combiné. Il existe dix opérateurs définis dans la notation UML 2.0. Les fragments combinés permettent de décrire des diagrammes de séquence de manière compacte. Les fragments combinés principaux sont :

Alternative. L'opérateur **alt** désigne un choix, une alternative. Il représente deux comportements possibles, c'est en quelque sorte l'équivalent du IF...THEN...ELSE. Donc, une seule des deux branches sera réalisée dans un scénario donné. La condition d'exécution d'une des deux branches (l'équivalent du IF) peut être explicite ou implicite. L'utilisation de l'opérateur **else** permet d'indiquer que la branche est exécutée si la condition du **alt** est fausse.

Option. L'opérateur **opt** désigne un fragment combiné optionnel comme son nom l'indique : c'est-à-dire qu'il représente un comportement qui peut se produire ... ou pas. Un fragment optionnel est équivalent à un fragment **alt** qui ne posséderait pas d'opérande **else** (qui n'aurait qu'une seule branche). Un fragment optionnel est donc une sorte de IF...THEN.

Break. L'opérateur **break** est utilisé dans les fragments combinés qui représentent des scénarios d'exception. Les interactions de ce fragment seront exécutées à la place des interactions suivantes. Il y a donc une notion d'interruption du flot "normal" des interactions.

Critical. L'opérateur **critical** désigne une section critique. Une section critique permet d'indiquer que les interactions décrites dans cet opérateur ne peuvent pas être interrompues par d'autres interactions décrites dans le diagramme. On dit que l'opérateur impose un traitement atomique des interactions qu'il contient.

Loop. L'opérateur **loop** (boucle) est utilisé pour décrire un ensemble d'interactions qui s'exécutent en boucle. En général, une contrainte appelée garde indique le nombre de répétitions (minimum et maximum) ou bien une condition booléenne à respecter.

Combinaison des opérateurs. Les fragments combinés et leurs opérateurs peuvent être combinés/mixés en vue de décrire des comportements complexes.

La Figure 2.4³ montre un exemple de combinaison de fragments : le diagramme de séquence indique que lorsque l'utilisateur se trompe trois fois de code, la carte est gardée et le distributeur se remet en mode d'attente d'une carte.

3. Cette figure est extraite de <http://www-128.ibm.com/developerworks/rational/library/3101.html>

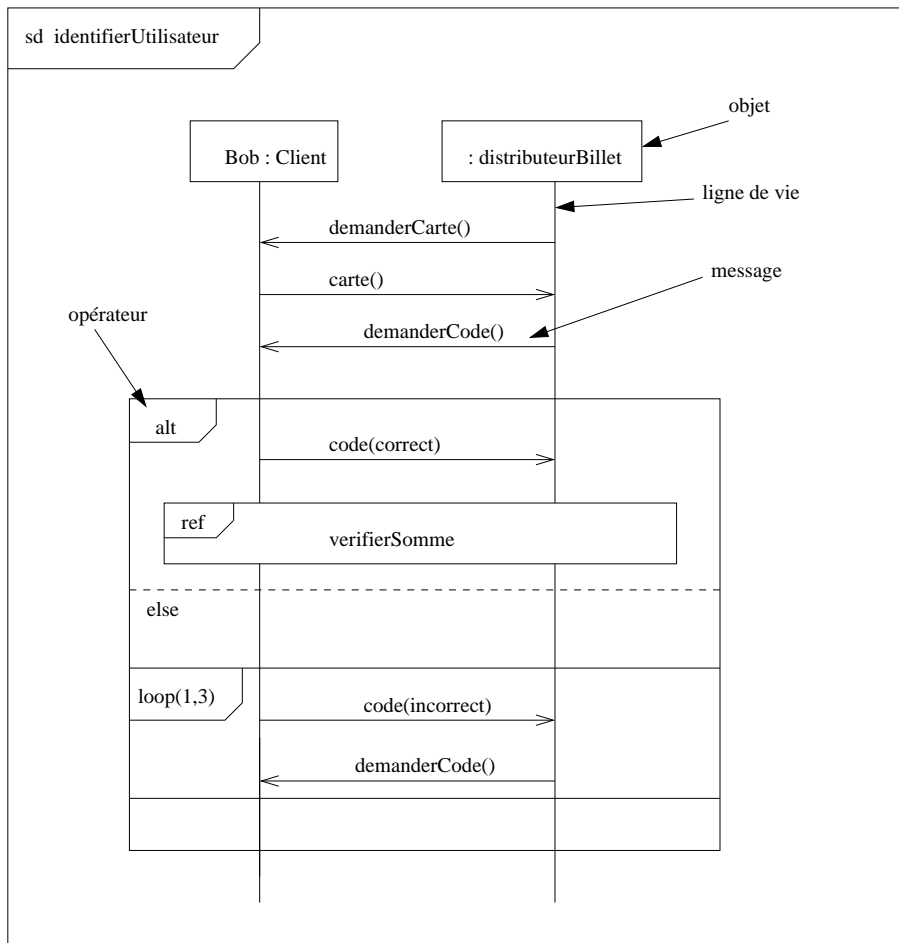


Figure 2.4 – Un exemple de diagramme de séquence UML2.0

2.2.1.5 Diagrammes d'état-transitions

Les diagrammes d'état-transitions sont utilisés pour saisir les transformations du système à travers le temps. Au moment de l'exécution, chaque objet possédant des attributs non constants pourra avoir potentiellement un certain nombre d'états. En règle générale, les état-transitions sont associés à des classes particulières (un ou plusieurs diagrammes d'état-transitions peuvent être introduits en vue de décrire complètement les états potentiels d'une classe).

Les diagrammes d'état-transitions UML se concentrent sur le comportement d'un objet lors de la réception des événements. Une utilisation correcte des état-transitions peut aider à révéler les variables d'instance nécessaires à la conservation des états d'un objet ainsi que les préconditions s'appliquant à une transition (mise à jour des variables d'instance) vers un autre état.

Les diagrammes d'état-transitions UML (Figure 2.5) sont utilisés pour modéliser les aspects dynamiques d'un système. La sémantique et la notation des diagrammes d'état-transitions UML sont celles des state-charts de Harel [Harel 87] avec les adaptations au contexte objet.

2.2.2 La sémantique de UML

La caractéristique essentielle qui différencie la notation UML des autres notations réside dans sa base formelle appelée *méta-modèle* : la syntaxe de la notation est décrite très précisément en utilisant la notation elle-même. C'est en quelque sorte l'équivalent de la notation BNF des langages

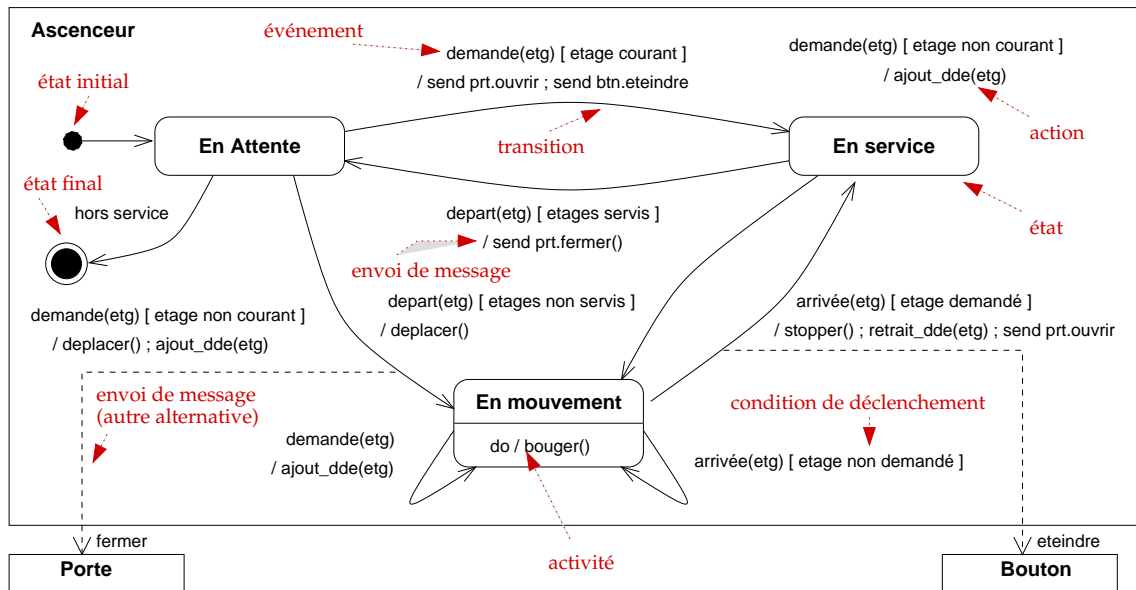


Figure 2.5 – Diagramme d'état-transitions (extrait de [Meyer 01])

de programmation; ce méta-modèle permet de définir l'arbre de syntaxe abstraite des modèles. Ce concept de méta-modèle est fondamental. Non seulement il aide à réduire les ambiguïtés et les incohérences de la notation, mais il constitue un précieux atout pour les concepteurs, dans le cadre d'une automatisation du processus de développement logiciel.

Le méta-modèle UML définit sa sémantique complète pour représenter les modèles objets en utilisant UML. Il est défini par méta-circularité, en utilisant un sous-ensemble des notations UML et sa sémantique.

Le méta-modèle est défini à l'aide de quatre couches d'architecture de méta-modélisation UML [OMG 03]. Cette architecture est une infrastructure prouvée pour définir la sémantique précise requise par des modèles complexes. Les quatre couches sont :

- **Méta-méta-modèle (M_3):** La couche de méta-méta-modèle forme la base pour l'architecture de méta-modélisation UML. La responsabilité principale de cette couche est de définir un langage pour spécifier un méta-modèle. Un méta-méta-modèle définit un modèle pour une couche d'abstraction plus haute qu'un méta-modèle.
- **Méta-modèle (M_2):** Un méta-modèle est une instance d'un méta-méta-modèle. La responsabilité principale de la couche du méta-modèle est de définir un langage pour spécifier des modèles. Les méta-modèles sont typiquement plus élaborés que le méta-méta-modèle qui les décrit, particulièrement quand ils définissent la sémantique dynamique.
- **Modèle (M_1):** Un modèle est une instance d'un méta-modèle. Sa responsabilité principale est de définir un langage qui décrit les domaines de l'information.
- **Objets utilisateur (M_0):** Les objets utilisateur sont des instances d'un modèle. La responsabilité principale des objets utilisateur est de décrire un domaine de l'information spécifique.

Prenons l'exemple d'un système de gestion de stocks présenté Figure 2.6⁴. Les objets de la couche M_0 sont les identifiants des clients et les marchandises commandées avec, pour chacune d'elle,

4. Cette figure est extraite dans une présentation de John Hogg, IBM

son nom et sa quantité, par exemple la marchandise *sawdust* avec une quantité de 2 tonnes.

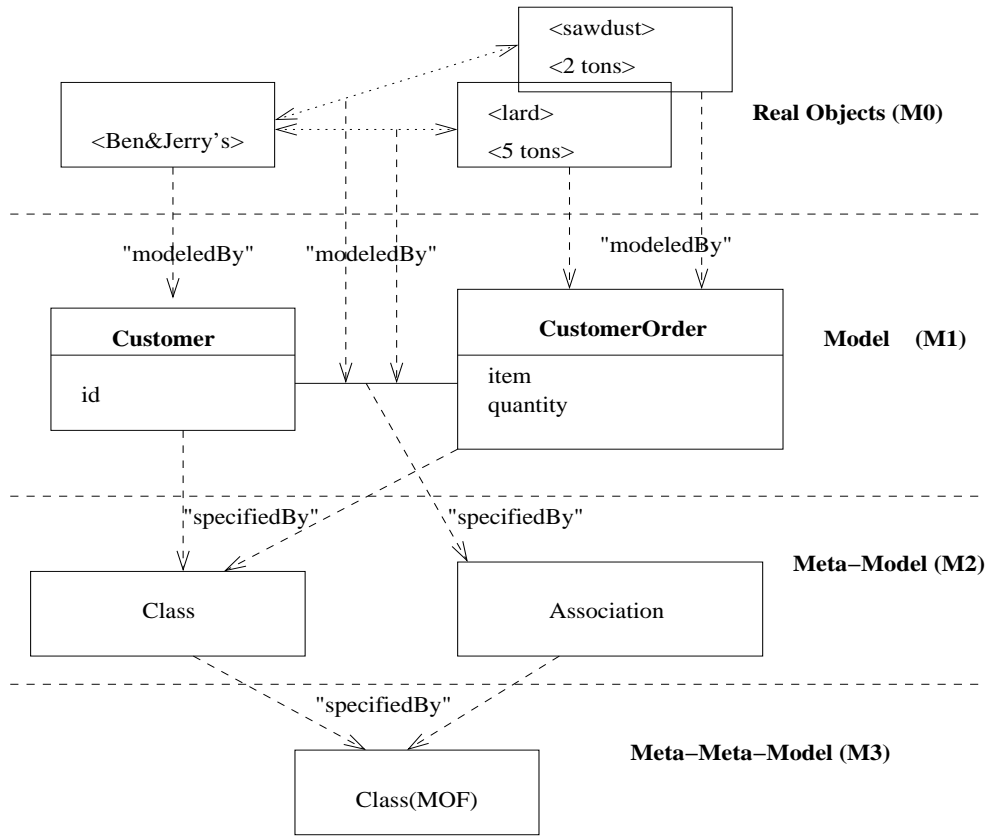


Figure 2.6 – L'architecture UML en couches

Le modèle M_1 de ces objets permet de spécifier tous les objets qui sont présentés dans la couche M_0 , en généralisant leurs propriétés et les relations entre ces objets. Dans l'exemple, une commande est définie par une classe possédant deux *attributs* (*item*, *quantity*). Ce modèle M_1 définit la *relation* - appartient à - entre un client et les marchandises commandées.

De manière similaire, les concepts de la couche M_1 sont définis dans la couche M_2 (méta-modèle), par exemple, les éléments d'association dans la couche M_1 sont spécifiés par une méta-classe Association dans la couche M_2 , les classes (**Customer**, **CustomerOrder**) sont spécifiées par la méta-classe **Class**, ... Enfin, les concepts de la couche M_2 sont définis dans la couche M_3 .

Le méta-modèle est défini de manière semi-formelle en utilisant trois vues, qui aident à la compréhension de la sémantique d'UML :

- **Syntaxe abstraite** : Les diagrammes de classes UML sont utilisés pour présenter le méta-modèle UML, ses concepts (méta-classes), ses relations et ses contraintes. Le diagramme présente également quelques règles bien-formées, concernant principalement les exigences de multiplicités des associations.
- **Règles bien-formées** (well-formedness rules) : Un ensemble des règles et de contraintes sur les modèles valides est défini comme l'invariant d'une instance de la méta-classe. Cet invariant doit être satisfait à la construction. Les règles spécifient des contraintes sur des attributs et des associations définies dans le méta-modèle. Elles sont exprimées en anglais et en OCL (voir section 2.2.3).
- **Sémantique** : La sémantique du modèle est décrite en anglais.

Structure des paquetages. La complexité du méta-modèle UML est gérée par l'organisation en paquetages logiques. Le méta-modèle est décomposé en paquetages de haut niveau présentés dans la Figure 2.7.

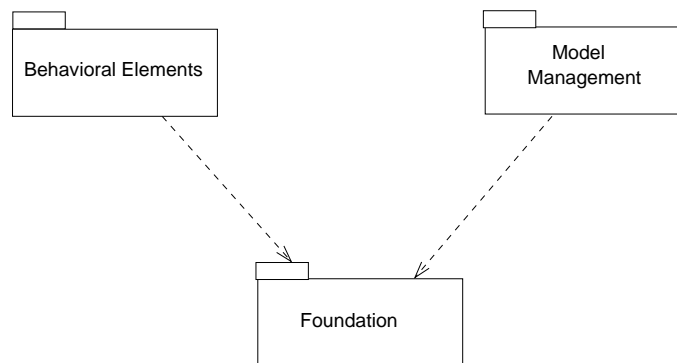


Figure 2.7 – Les paquetages de haut niveau

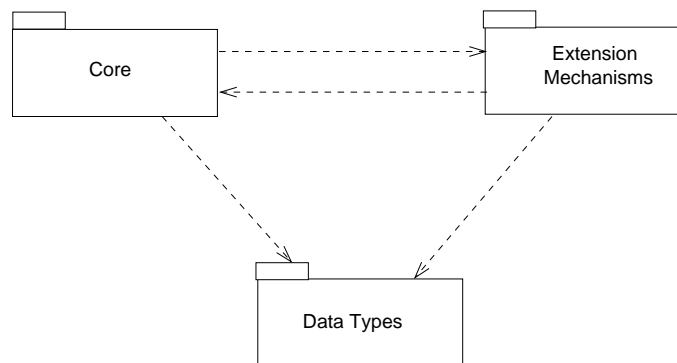


Figure 2.8 – Le paquetage Foundation

Le paquetage Foundation présenté Figure 2.8 spécifie la structure statique des modèles. Il est décomposé en sous-paquetages : Core, Extension Mechanisms et Data Types. Le paquetage Core spécifie les concepts de base requis pour un méta-modèle élémentaire et définit un *backbone* architecture pour attacher le langage de construction additionnel comme des méta-classes, des méta-associations et des méta-attributs. Le paquetage auxiliaire Elements définit les constructions additionnelles qui étendent Core pour supporter les concepts avancés comme des dépendances, des structures physiques, ... Le paquetage Extension Mechanisms spécifie comment des éléments du modèle sont étendus avec les nouvelles sémantiques utilisant des stéréotypes, des contraintes, des tags. Le paquetage Data Types définit les structures de données de base pour le langage.

La Figure 2.9 présente un exemple avec une partie du paquetage Core-Backbone.

Le paquetage Behavioral Elements présenté Figure 2.10 spécifie le comportement dynamique des modèles. Il est décomposé en sous-paquetages : Common Behavior, Collaborations, Use Cases, State Machines, Activity Graphs et Actions. Common Behavior spécifie les concepts noyaux requis pour les éléments comportementaux. Le paquetage Collaborations spécifie un contexte comportemental en utilisant des éléments du modèle pour accomplir une tâche particulière. Le paquetage Use Cases spécifie le comportement en utilisant des acteurs et des cas d'utilisation. Le paquetage State Machines définit le comportement en utilisant des systèmes de transitions d'états finis. Le paquetage

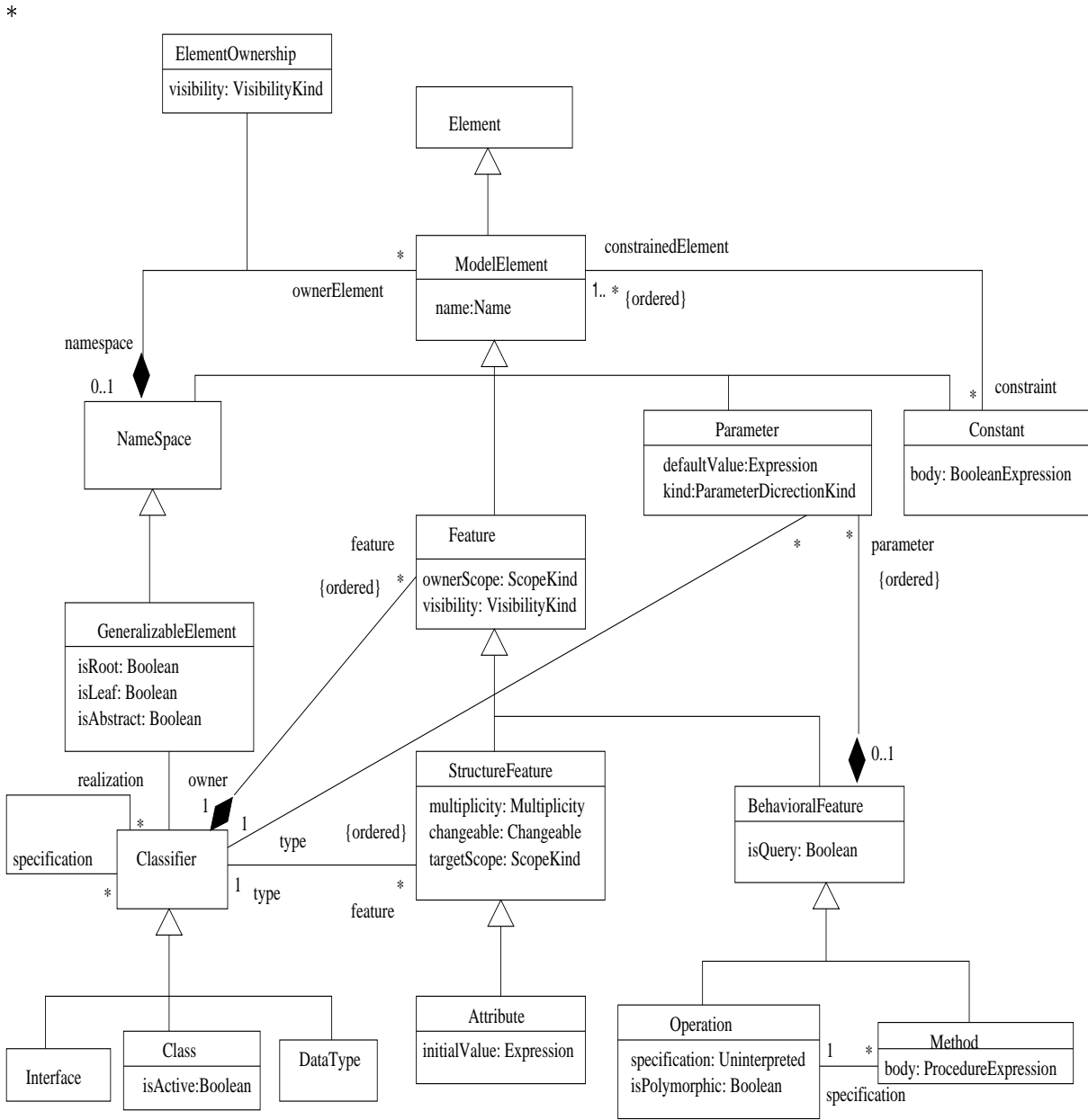


Figure 2.9 – Une partie du paquetage Core - Backbone

Activity Graphs définit un cas spécial de la machine d'état qui est utilisé pour modéliser les processus d'une ou plusieurs classes. La paquetage Actions définit le comportement en utilisant un modèle détaillé de calcul.

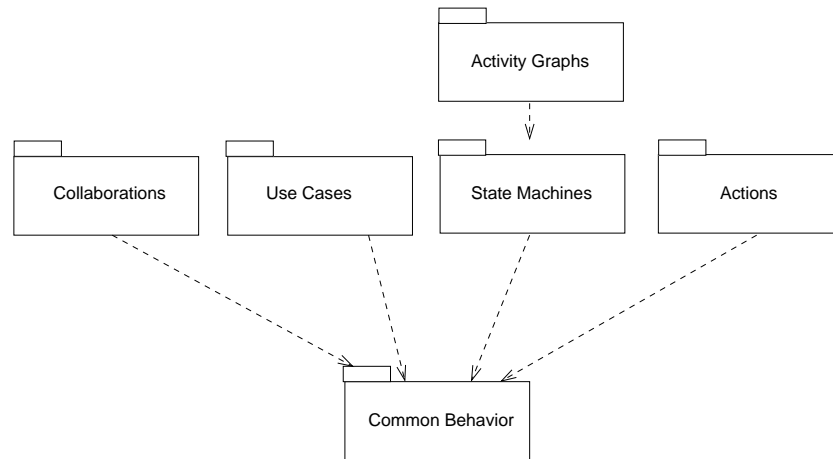


Figure 2.10 – *Le paquetage Behavioral Elements*

Dans la suite de notre travail, nous prenons en compte la structure du paquetage Core pour vérifier la sémantique des éléments des diagrammes de classes, la structure des paquets Collaborations et State Machines pour vérifier la sémantique des éléments des diagrammes de collaboration et des diagrammes d'état-transitions.

2.2.3 Introduction à OCL

Ce paragraphe présente OCL (Object Constraint Language), considéré comme le langage "formel" officiel au sein de modèles UML. Les auteurs d'OCL [Warmer 99] proposent d'utiliser les expressions OCL pour :

- spécifier les invariants de classes et les types dans les diagrammes de classes;
- spécifier le "type invariant" pour les stéréotypes;
- décrire les pré- et postconditions des opérations déclarées dans les diagrammes de classes;
- décrire les gardes au sein des diagrammes d'état-transitions;
- spécifier les chemins de navigation dans un diagramme de classes;
- spécifier les contraintes des opérations OCL définies par l'utilisateur (depuis UML version 1.4).

Le point fort d'OCL est son orientation objet. La syntaxe d'OCL est similaire à celle des langages de programmation orientée objet. Le langage fournit des variables et des opérations pour construire les expressions de contraintes. Les caractéristiques fréquemment utilisées sont les accès aux propriétés d'objets, y compris les navigations au sein des réseaux d'objets, les requêtes sur les objets. Le langage OCL permet de définir des types, y compris les types de base (*Integer*, *Boolean*, ...) et les types de collections (*Set*, *Sequence*, *Bag*).

2.2.3.1 Expressions

Une expression OCL est une application d'une opération OCL sur les constantes, des variables et des expressions OCL de telle sorte que la conformité des types OCL soit respectée. Les constantes et les variables sont considérées comme des expressions OCL les plus simples. Les expressions telles que les invariants écrits dans le contexte d'une classe peuvent faire référence à une instance de la classe en utilisant le mot clé `self`. Les expressions plus compliquées peuvent être construites au moyen des appels d'opérations de la construction `if then else endif`. Le premier argument d'un appel d'opération dénote la valeur ou l'objet cible à laquelle/auquel l'opération est appliquée. Ceci s'effectue en utilisant la notation où l'expression cible est suivi par un point ".", le nom de l'opération et éventuellement les arguments supplémentaires entre parenthèses. L'emboîtement des appels d'opérations est possible. Les attributs de classes peuvent être accédés de la même manière. Si la cible d'une opération est une valeur de collection, une flèche "->" est utilisée au lieu d'un point⁵.

2.2.3.2 Contraintes

Le contexte d'une expression OCL est soit un classificateur (une classe ou un type), soit une opération d'un classificateur. Si le contexte spécifie un classificateur, l'expression de contrainte correspond à un invariant que toutes les instances du classificateur doivent respecter. Si le contexte est une opération d'un classificateur, l'expression de contrainte spécifie la pré- et postcondition de l'opération. Dans les deux cas, il est possible de référencer une instance du classificateur en question en utilisant le mot clé `self`. Il est également possible de faire référence à l'ensemble des instances effectives de la classe `Class` jouant le rôle du classificateur courant en utilisant l'expression `Class.allInstance`.

Invariant. Les invariants sont des conditions qui portent sur toutes les instances de classificateurs. Ce sont souvent des conditions supplémentaires au sein d'une classe, d'un type ou sur l'association entre les classes ou sur l'association entre les classes que l'on ne peut pas⁶ spécifier en utilisant les notations graphiques UML. Un invariant est accompagné d'un contexte spécifiant le classificateur auquel l'invariant est appliqué. Supposons que la classe `Customer` dans le modèle présenté dans la couche M_1 de la Figure 2.6 a un attribut `age` de type `Integer`. La condition :

```
Context Customer
    inv: self.age > 0
```

exige que la valeur de `age` soit toujours positive pour toutes les instances de la classe `Customer`. Si l'invariant porte sur une association entre les classes, une des classes participant à l'association est utilisée comme contexte.

Pré- et Postcondition. La spécification en OCL sous la forme de pré- et postcondition d'une opération a besoin d'un contexte de type `Classificateur::oper(...): typeRetour`, qui est composé du classificateur et de la signature de l'opération. La précondition décrit la condition que l'on doit respecter avant l'exécution de l'opération. La postcondition décrit les effets produits par l'exécution de l'opération. Supposons que la classe `Customer` dans le modèle présenté dans la couche

5. Il y a cependant une exception pour les opérations d'accès aux propriétés d'objets, à savoir le point "." est utilisé quel que soit la cible, qu'il s'agisse d'un objet ou d'une collection d'objets.

6. Quelquefois, on ne veut pas utiliser la notation graphique afin de simplifier la représentation.

M_1 de la Figure 2.6 a un opération `addItem`. La condition :

```
Context Customer::addItem(item : String)
pre: not self.orderItems ->includes(item)
post: self.orderItems ->includes(item)
```

exige qu'avant l'exécution de l'opération `addItem(item)`, `item` n'est pas apparu dans la liste des articles commandés (`orderItems`) et il doit être apparu après l'exécution de cette opération.

2.3 Synthèse

- La construction à base d'objets est une technique performante pour la modélisation de systèmes. C'est une approche de développement très utilisée en milieu industriel.
- Les modèles à objets sont faciles à comprendre car ils sont directement interprétables dans le monde réel; la maintenance est facilitée et les composants décrits sont fortement réutilisables.
- UML est un langage de modélisation unifié basé sur les concepts à objets. Ce n'est pas une méthode mais un ensemble de notations graphiques conçu pour visualiser, spécifier, construire et documenter les artefacts d'un système.
- UML dispose de neuf diagrammes⁷ afin de visualiser un système sous différents angles. Un diagramme est une vue partielle du système construit, c'est une représentation graphique d'un ensemble d'éléments.
- Le méta-modèle est défini comme une couche de l'architecture de méta-modélisation UML, constituée de quatre couches: méta-méta-modèle, méta-modèle, modèle et objets utilisateur.
- Le méta-modèle est défini de manière semi-formelle en utilisant trois vues: syntaxe abstraite, règles bien-formées et sémantique.
- OCL est un langage formel qui permet de décrire des expressions et des contraintes au sein des modèles orientés objets.

7. UML 2.0 dispose de treize diagrammes

SOMMAIRE

3.1	Dérivation d'UML en B	45
3.1.1	Dérivation de diagrammes de classes	45
3.1.2	Dérivation de diagrammes d'état-transitions	48
3.1.3	Dérivation de diagrammes de collaboration	50
3.1.4	Dérivation d'expressions OCL	52
3.1.5	Dérivation du méta-modèle UML en méthodes formelles	53
3.2	Vérification et validation formelle	53
3.2.1	Vérification formelle	54
3.2.2	Validation formelle	55
3.3	Spécification formelle orientée objet	56
3.4	Synthèse	58

Ce chapitre présente l'état de l'art de nos travaux. Nous commençons par un rappel sur les dérivations de diagrammes UML en B. Ces schémas de dérivation portent sur les diagrammes de classes et d'état-transition, présentés dans la thèse de Meyer [Meyer 01], les événements de diagrammes d'état-transitions, les diagrammes de collaboration et les expressions OCL en B, présentés dans la thèse de Ledang [LeDang 02a]. Ce rappel est nécessaire pour la compréhension de nos travaux de vérification et la validation de spécifications UML en utilisant B. Un état de l'art sur la vérification et validation formelle est introduit. Nous faisons également une synthèse des travaux sur l'extension par objet des méthodes formelles.

3.1 Dérivation d'UML en B

Comme dit auparavant, notre travail concerne la vérification et la validation formelle de spécifications UML en utilisant B. Ce travail peut être vu comme une prolongation des travaux de dérivation d'UML en B. Nous résumons l'état de l'art des approches de dérivation d'UML en B.

3.1.1 Dérivation de diagrammes de classes

Les diagrammes de classes sont les diagrammes les plus courants dans la modélisation des systèmes orientés objets. On les utilise pour modéliser la vue de conception statique d'un système. Les éléments structurels - classe, attribut, association - sont formalisés en B comme des données et les opérations sont dérivées comme des opérations B.

Dérivation 3.1 (Classe)

Une classe *Class* donne lieu à *CLASS* et *class* :

- *CLASS* est une constante B pour modéliser l'ensemble des objets possibles de *Class* : *CLASS* est définie comme un sous-ensemble de l'ensemble de tous les objets possibles *OBJECTS*, lui-même défini comme un ensemble abstrait ;
- *class* est une variable B pour modéliser l'ensemble des objets instanciés de *Class* : *class* est définie comme un sous-ensemble de *CLASS*; la multiplicité (si elle existe) de *Class* se traduit par un prédicat d'invariant additionnel pour contraindre la cardinalité de *class* ;
- la substitution d'initialisation pour *class* doit prendre en compte les contraintes de multiplicité de *Class*. Dans le cas de l'absence de telles contraintes, *class* est initialisée par l'ensemble vide.

Dans un but de modularité, chaque couple *CLASS*, *class* conduit à la construction d'une machine abstraite *Class* ; *OBJECTS*, quant à lui, est déclaré dans une machine abstraite spéciale *Types* dans laquelle sont modélisés également les types qui apparaissent dans les diagrammes de classes mais qui ne correspondent pas aux types prédéfinis dans le langage B; ce sont les types des attributs ou des paramètres d'opérations. *Types* est ainsi vue (la clause **SEES**) par les machines abstraites dérivées des classes. Dans la Figure 3.1, les clauses SETS, CONSTANTS, PROPERTIES sont déclarées dans la machine *Types*.

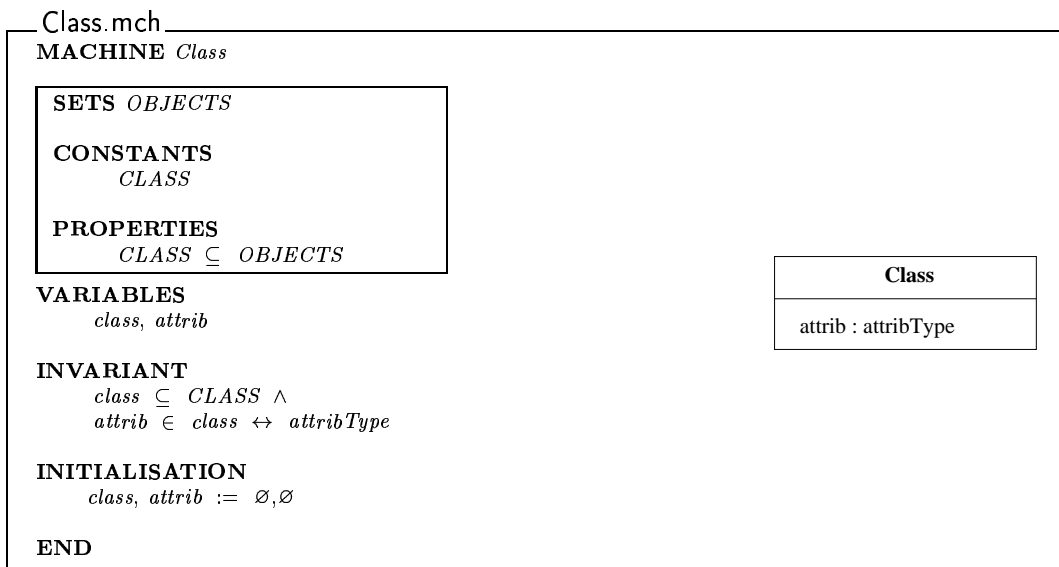


Figure 3.1 – Dérivation générale d'une classe et de ses attributs

Dérivation 3.2 (Attribut)

Un attribut *attrib* est dérivé formellement en B par la déclaration d'une nouvelle variable *attrib* dans la machine abstraite associée à la classe de l'attribut. Cette variable est définie dans l'invariant par une relation entre l'ensemble des objets instanciés de la classe et son type *attribType*, elle est initialisée par l'ensemble vide (Figure 3.1).

Comme dans le cas d'une classe, il est également possible de préciser en UML la multiplicité de chaque attribut. Cette possibilité permet d'affiner la définition formelle d'un attribut en utilisant,

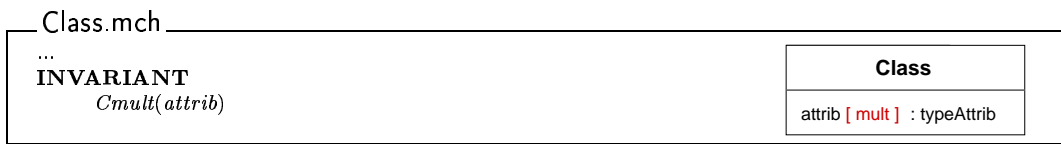


Figure 3.2 – Dérivation d'un attribut avec multiplicité

par exemple, une fonction à la place d'une relation.

Dérivation 3.3 (Multiplicité d'attribut)

Une multiplicité *mult* d'un attribut *attrib* précise la dérivation formelle de celui-ci (cf. Dérivation 3.2). La définition de la variable *attrib* est remplacée par un nouveau prédicat $Cmult(attrib)$ (Figure 3.2). Le Tableau 3.1 montre pour chaque type de multiplicité le prédicat qui doit être stipulé. Ces prédicats sont reliés par une conjonction lorsqu'une liste de multiplicités est spécifiée.

mult	$Cmult(attrib)$
* ou 0..*	$attrib \in class \leftrightarrow typeAttrib$
0..1	$attrib \in class \rightarrow typeAttrib$
1	$attrib \in class \rightarrow typeAttrib$
1..*	$attrib \in class \leftrightarrow typeAttrib \wedge \text{DOM}(attrib) = class$
n	$attrib \in class \leftrightarrow typeAttrib \wedge \forall x.(x \in class \Rightarrow \text{CARD}(attrib[\{x\}]) = n)$
0..n	$attrib \in class \leftrightarrow typeAttrib \wedge \forall x.(x \in class \Rightarrow \text{CARD}(attrib[\{x\}]) \leq n)$
1..n	$attrib \in class \leftrightarrow typeAttrib \wedge \text{DOM}(attrib) = class \wedge \forall x.(x \in class \Rightarrow \text{CARD}(attrib[\{x\}]) \leq n)$
n..*	$attrib \in class \leftrightarrow typeAttrib \wedge \forall x.(x \in class \Rightarrow \text{CARD}(attrib[\{x\}]) \geq n)$
n..m	$attrib \in class \leftrightarrow typeAttrib \wedge \forall x.(x \in class \Rightarrow n \leq \text{CARD}(attrib[\{x\}]) \leq m)$

où *n* et *m* sont des entiers positifs tels que $n \geq 2 \wedge n \leq m$.

Tableau 3.1 – Affinement de la dérivation d'un attribut par sa multiplicité

Dérivation 3.4 (Association)

Une association *assos* entre la classe *Class1* et la classe *Class2* est dérivée formellement en B par la déclaration d'une nouvelle variable *assos* (Figure 3.3). Cette variable est définie dans un invariant par une relation entre l'ensemble des objets instanciés *class1* et l'ensemble des objets instanciés *class2*, elle est initialisée par l'ensemble vide.

Remarque 3.1 Bien que cela ne soit pas nécessaire dans la notation UML, on impose pour la dérivation formelle d'une association qu'un nom et une direction soient spécifiés.

Dérivation 3.5 (Opération)

Une opération *oper* d'une classe *Class* est dérivée formellement en B dans la machine *Class* par la construction partielle d'une opération *oper* (Figure 3.4). Le paramètre en sortie d'opération est

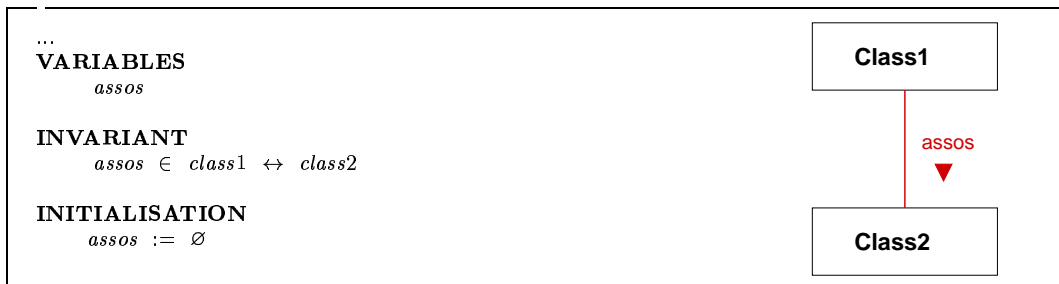


Figure 3.3 – Dérivation d'une association

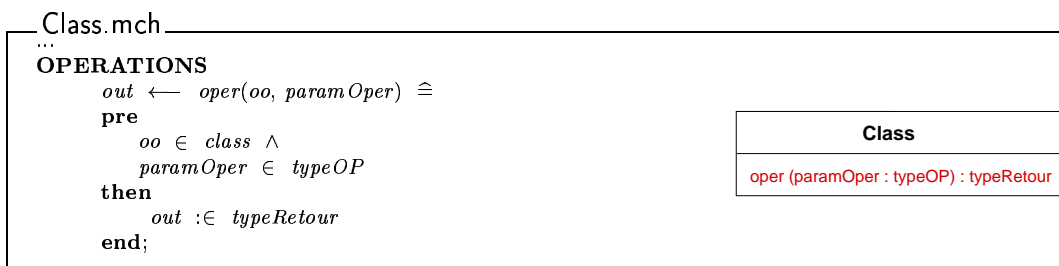


Figure 3.4 – Dérivation partielle d'une opération

spécifié par une variable *out* définie dans le corps de l'opération comme un élément quelconque du type du résultat (*typeRetour*). Le squelette de l'opération est paramétré en entrée par

- l'instance cible : une variable *oo* déclarée dans les paramètres de l'opération B précise l'instance qui va recevoir le service, *oo* est définie dans la précondition de l'opération comme un élément de l'ensemble des objets instanciés *class* ;
- les paramètres de l'opération : pour chaque paramètre *paramOper* de la signature, une variable du même nom est déclarée en paramètre de l'opération B, celle-ci est définie par une précondition d'appartenance à *typeOP*.

3.1.2 Dérivation de diagrammes d'état-transitions

On utilise les automates à états finis pour modéliser le comportement de n'importe quel élément de modélisation. Les diagrammes d'état-transitions se concentrent sur le comportement d'un objet ordonné par les événements, ce qui est particulièrement utile lorsqu'on modélise des systèmes réactifs.

UML fournit une représentation graphique qui permet de mettre en évidence les éléments importants dans la vie de cet objet : états, transitions, événements et actions. Nous présentons maintenant les différentes propositions existantes pour l'intégration des diagrammes d'état-transitions. Les dérivations décrites ci-dessous interviennent après la traduction des diagrammes de classes. Dans notre thèse, nous ne travaillons que sur le méta-modèle des diagrammes d'état-transitions et n'utilisons pas les dérivations sur les modèles des diagrammes d'état-transitions. Cependant, pour avoir une vue générale de l'état de l'art des approches de dérivation d'UML en B, nous résumons quelques schémas de dérivation de diagrammes d'état-transitions.

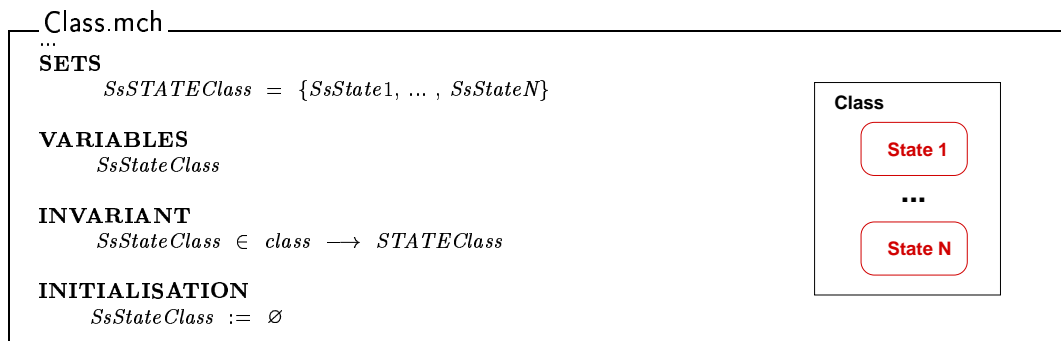


Figure 3.5 – Dérivation générale des états



Figure 3.6 – Dérivation d'une transition

Dérivation 3.6 (Etats)

Les états d'un diagramme d'état-transitions d'une classe **Class** sont dérivés formellement en B (Figure 3.5) par la définition dans la machine *Class* :

- d'un ensemble énuméré *STATEClass*, l'ensemble des états possibles des objets de la classe **Class** ;
- d'une variable *stateClass* définie par une fonction totale entre l'ensemble *class* et l'ensemble *STATEClass* et initialisée par l'ensemble vide ; elle définit l'état courant de chaque objet de la classe.

Si les états sont directement liés aux valeurs des attributs et que cette relation peut être exprimée par un prédicat, celui-ci doit être rajouté à l'invariant de la machine.

Dérivation 3.7 (Transition)

Une transition est dérivée formellement en B par une opération dont le rôle est de réaliser le changement d'état. Soit un diagramme d'état-transitions définissant le comportement des objets d'une classe **Class** et décrivant une transition entre un état source **StateS** et un état cible **StateC**. La dérivation formelle de la transition ci-avant génère dans la machine abstraite *Class* une nouvelle opération *transition_{S_C}* (Figure 3.6). Cette opération est paramétrée par l'objet *oo* recevant la transition, elle vérifie dans ses préconditions que l'état courant de celui-ci est **StateS** et opère dans le corps de l'opération au changement d'état en associant à *oo* via la fonction *stateClass* le nouvel état **StateC**.

Dans le contexte des diagrammes d'état-transitions, une transition est en général déclenchée par un événement qui spécifie l'apparition d'une occurrence à un instant donné.

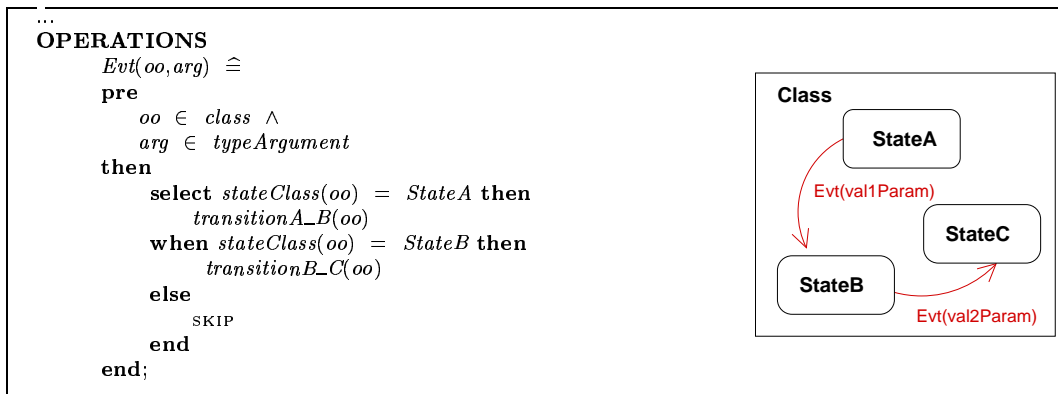


Figure 3.7 – Dérivation d'un événement

Dérivation 3.8 (Action)

La dérivation courante des actions traite seulement le cas où une action est un appel d'opération. Dans ce cas, la modélisation B d'une action est similaire à la modélisation B d'une opération de classe (Dérivation 3.5)

Dérivation 3.9 (Événement déclencheur)

Un événement déclencheur Evt est dérivé formellement en B par une opération *Evt* (Figure 3.7). Cette opération est paramétrée par l'objet cible et les éventuels arguments. On crée pour chaque transition décorée par cet événement une sélection (un cas d'une substitution de sélection). La garde de cette sélection est exprimée par un prédicat qui vérifie que l'objet est bien dans l'état source de la transition, son corps fait appel à l'opération de changement d'état associée à la transition (Dérivation 3.7).

En général, les transitions du diagramme d'état-transitions sont accompagnées de conditions de garde et/ou de listes d'actions. Les premières permettent d'affiner et de préciser les conditions de déclenchement, les dernières précisent les traitements à effectuer lors de la transition.

Dans sa thèse, Ledang [LeDang 02a] a proposé une contribution à la dérivation de diagramme d'état-transitions. L'opération B abstraite d'un événement est raffinée en appelant les opérations B pour les transitions et les actions éventuellement déclenchées par l'événement.

3.1.3 Dérivation de diagrammes de collaboration

En intégrant des diagrammes de classes et des diagrammes de collaboration, le travail de thèse de Ledang [LeDang 02a] a donné une approche pour modéliser en B les opérations UML. Cette approche procède en deux temps: (i) modéliser chaque opération UML par une opération B abstraite; (ii) modéliser la réalisation de certaines opérations UML (qui sont dites *non-basiques* pour les distinguer des opérations UML de base considérées comme simples et qui n'ont pas de réalisation) par des opérations B d'implantation. Dans UML, on décrit la réalisation d'une opération en utilisant les diagrammes d'interaction ou les diagrammes d'activité (*diagrammes de réalisation*). Dans cette dérivation, on fait seulement référence aux diagrammes de collaboration, cependant il est tout à fait possible d'étendre la présentation pour les diagrammes d'activité et de séquences.

Les opérations communiquent entre objets dans le diagramme de collaboration par les messages. Chaque diagramme de collaboration a un message origine qui se décompose en plusieurs messages, ces messages eux-mêmes continuent à se décomposer comme le message origine. Chaque message contient un numéro et un appel à une opération. Le numéro indique l'ordre d'exécution et l'appel d'opération pour réaliser ce message. Pour la démonstration de la procédure de dérivation, nous donnons un exemple général de diagramme de collaboration (voir figure Figure 3.8). L'opération *op1* (contenue dans le message 1) qui appartient à *Class1*, appelle les opérations *op1.1* (*Class2*) et *op1.2* (*Class3*). L'opération *op1.1* appelle les opérations *op1.1.1* (*Class4*), *op1.1.2* (*Class2*) et *op1.1.3* (*Class5*).

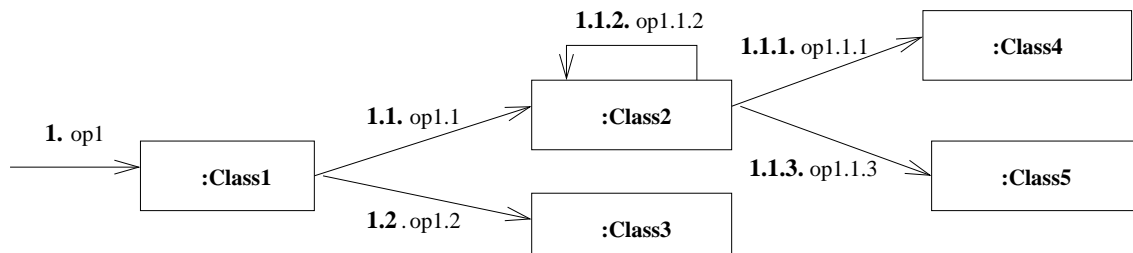


Figure 3.8 – Diagramme de collaboration

Dans la spécification B correspondante, l'opération *op1* se décompose en deux opérations *op1.1* et *op1.2* et l'opération *op1.1* se décompose en trois opérations *op1.1.1*, *op1.1.2* et *op1.1.3*, ...

Définition 3.1 (Dépendance appelante-appelée) Une paire **appelante-appelée** relie une opération UML ayant une réalisation avec une des opérations UML qui participe à sa réalisation. La relation appelante-appelée au sein de l'ensemble des opérations UML définit toutes les paires appelante-appelées entre les opérations UML en question.

La procédure de dérivation permet de générer automatiquement l'architecture de la spécification B dérivée. Elle utilise deux procédures **division** et **duplication** pour organiser les opérations UML en couches de telle sorte que chaque couche d'opérations UML donne lieu à une machine abstraite. L'idée intuitive de la procédure de division est de :

- **Créer la couche la plus haute** : toutes les opérations UML qui n'ont aucune opération appelante mais ont au moins une opération appelée forment la couche la plus haute.
- **Créer la couche la plus basse** : toutes les opérations UML qui n'ont aucune opération appelée forment la couche la plus basse.
- **Créer la(es) couche(s) intermédiaire(s)** : à partir de la couche la plus haute, on cherche les opérations UML qui sont des opérations appelées par au moins une opération dans la couche la plus haute. Si aucune opération n'est trouvée (c'est-à-dire on a rencontré la couche la plus basse) alors on s'arrête, autrement les opérations UML trouvées forment la première couche intermédiaire. On répète cette démarche mais cette fois-ci on cherche les opérations UML qui sont des opérations appelées par les opérations dans la couche la plus haute ou bien dans les couches intermédiaires formées précédemment jusqu'à ce qu'on rencontre la couche la plus basse.

Remarque 3.2 Les relations appelante-appelée contenant des paires récursives ou des dépendances circulaires entre les opérations ne sont pas prises en compte afin d'éviter la boucle infinie de la

division. Notre contribution, chapitre 6, vise à résoudre ces limites et apporte une solution à la vérification des pré- et postconditions des opérations.

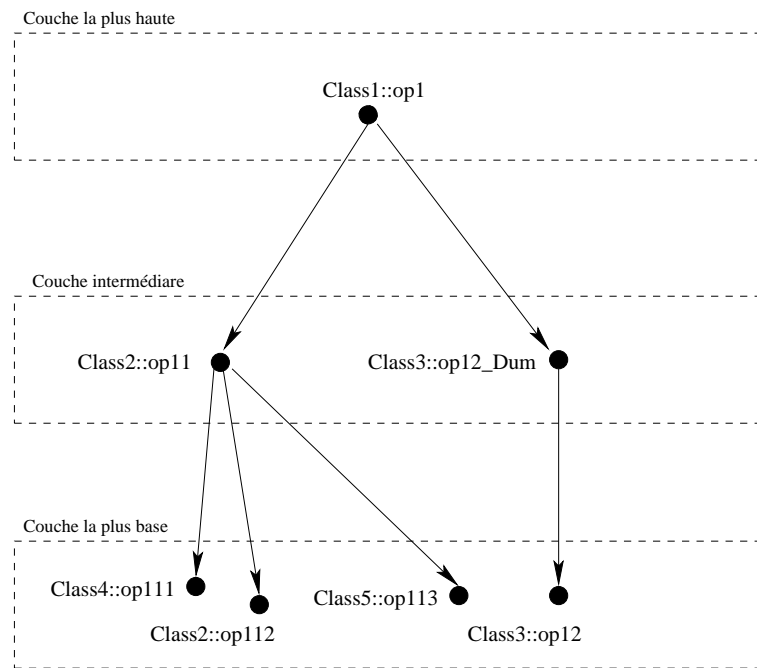


Figure 3.9 – Organisation en couches des opérations UML de diagrammes de collaboration

A partir des couches d'opérations UML établies par la procédure de division, on crée une machine abstraite pour chaque couche. Les machines B correspondantes sont créées : *SystemMachine* pour les opérations UML dans la couche la plus haute, *IntermediateMachine(s)* pour les opérations dans les couches intermédiaire(s) et *BasicMachine* pour les opérations dans la couche la plus basse. Il y a un problème, lorsque les opérations modélisées dans *SystemMachine* ont des opérations appelées modélisées soit dans une machine *IntermediateMachine* soit dans *BasicMachine*. Les machines *IntermediateMachine* et *BasicMachine* sont importées dans l'implantation de *SystemMachine*. Ceci est interdit en B car *BasicMachine* est aussi importée dans l'implantation de *IntermediateMachine*.

Pour remédier à une telle situation souvent rencontrée dans les couches d'opérations UML après l'application de la procédure de dérivation, certaines opérations UML doivent être dupliquées dans les couches supérieures afin d'éliminer des flèches qui relient des opérations UML qui ne sont pas dans deux couches consécutives.

3.1.4 Dérivation d'expressions OCL

Dans le cadre de la transformation d'UML en B, les travaux de thèse de Ledang [LeDang 02a] et Marcano [Marcano 01] ont transformé les expressions OCL en B, ce qui permet de transformer automatiquement en B non seulement la spécification OCL des opérations UML mais aussi les invariants supplémentaires des concepts structurels UML et les conditions de garde au sein de diagrammes d'état-transitions. Les schémas de dérivation d'OCL vers B sont définis pour les concepts relatifs aux expressions OCL :

- les types OCL et les opérations sur les types ;
- les postconditions d'une opération UML.

Il est logique de modéliser une expression OCL en une expression B parce que OCL et B sont basés sur la logique du premier ordre. Cependant, les types et les opérateurs de OCL et B sont différents donc dans les travaux de dérivation de OCL en B, on énumère seulement les notations équivalentes entre les deux formalismes. Certaines méthodes de dérivation ont été proposées mais ce ne sont pas des schémas globaux pour dériver OCL vers B. Intuitivement, une expression OCL pour un invariant de classes est dérivée vers un invariant d'une machine B. Par exemple :

OCL	B
Context Customer inv: self.age > 0	$\forall xx.(xx \in CUSTOMER \Rightarrow customer_age(xx) > 0)$

Les pré- et postconditions d'une opération OCL sont dérivées en une opération dans une machine abstraite B :

Context Customer::addItem(item : String) pre: not self.orderItems ->includes(item) post: self.orderItems ->includes(item)	<pre> addItem(item) = pre item ∉ orderItems then orderItems := orderItems ∪ item end </pre>
--	---

3.1.5 Dérivation du méta-modèle UML en méthodes formelles

Dans le cadre des systèmes d'information, Laleau *et al* [Laleau 00] expriment la sémantique des modèles de classes en termes de méta-modèles UML avec une extension en utilisant les notations de B, notamment les invariants.

La dérivation du méta-modèle UML vers des méthodes formelles a été considérée, comme par exemple, vers Object-Z [Kim 00] ou vers ASM (Abstract State Machine) [Cavarrà 04]. Ces approches permettent d'exprimer la structure du méta-modèle sous la forme de diagrammes de classes et de la sémantique UML par les notations des méthodes formelles. Cependant, il est à noter que toutes ces approches ne permettent pas de vérifier la sémantique des éléments du modèle UML parce qu'elles transforment les méta-classes mais ne pas considèrent les instances des méta-classes.

3.2 Vérification et validation formelle

La validation correspond au processus d'évaluation du logiciel à la fin de son développement pour s'assurer qu'il ne défaille pas et qu'il répond aux spécifications des besoins. On considère qu'il y a défaillance d'un système quand le comportement observé est différent du comportement spécifié. Cette validation s'effectue à travers l'utilisation de techniques de tests. La validation doit permettre de répondre à la question suivante: Construisons-nous le bon produit?

La vérification revient à s'assurer que le logiciel (ou une partie de celui-ci) arrivé à une fin de phase du cycle de développement remplit bien les conditions requises pour passer à la phase suivante. Les techniques de revues du logiciel sont certainement les plus utilisées. On peut citer également les preuves de programme qui permettent de vérifier la conformité d'un logiciel par rapport à ses spécifications formelles. La vérification doit permettre de répondre à la question suivante: Construisons-nous correctement le produit?

Les méthodes formelles consistent à intégrer une étape de vérification formelle dans les démarches de conception de logiciels. Elles reposent sur la formulation rigoureuse des besoins (spécification des besoins) sous forme de propriétés attendues du système et sur la modélisation des fonctionnements du système développé pour satisfaire ces besoins. La formulation doit être rigoureuse, concise et précise tout en étant accessible à l'utilisateur et manipulable par un environnement. La question cruciale des méthodes formelles est de concilier les deux objectifs contradictoires : expressivité des langages et faible complexité de la vérification. C'est pourquoi il existe de nombreuses approches ciblées sur des classes d'applications pour limiter l'expressivité des langages et outiller la méthode (voir section 1.1.3).

Les méthodes de preuve sont très puissantes au sens où elles traitent des systèmes à nombre d'états infini, mais elles sont indécidables dans le cas général, et par conséquent difficiles à automatiser. Par contre les méthodes de model checking sont entièrement automatiques mais limitées aux systèmes ayant un nombre fini d'états et dans ce cadre posent des problèmes en temps de traitement et en espace mémoire liés à l'explosion combinatoire du nombre d'états.

Les méthodes formelles ne suffisent pas pour garantir que le système développé satisfait l'utilisateur. D'une part les propriétés formalisées ne couvrent pas nécessairement la totalité des besoins, d'autre part il n'est pas exclu d'avoir mal interprété les besoins à la fois dans l'expression des propriétés et dans la modélisation du système. Par conséquent la vérification formelle doit être complétée par la validation à partir de jeux de tests. La validation formelle de la spécification permet aux développeurs d'avoir au plus tôt l'assurance de la complétude et de la cohérence des fonctionnalités du logiciel décrites dans ses spécifications.

3.2.1 Vérification formelle

Contrairement à la vérification traditionnelle basée sur l'expérimentation, la vérification formelle est basée sur la démonstration logique ou mathématique, elle est effectuée par un prouveur qui fait un raisonnement formel. Nous pouvons utiliser la méthode de vérification formelle pour s'assurer que des propriétés spécifiques sont bien respectées par le système construit ou pour s'assurer de la consistance des spécifications semi-formelles.

La méthode B est fondée sur la preuve, cela signifie que la vérification formelle de spécifications B est immédiate par les obligations de preuve générées. Dans nos travaux de thèse, nous utilisons la méthode B pour vérifier des spécifications semi-formelles UML.

Un grand nombre de méthodes formelles ont été proposées pour détecter les erreurs dans les modèles UML. Nous pouvons citer les approches de vérification formelle de spécifications UML suivantes :

Model Checking. L'approche [Schafer 01] vérifie le comportement de la spécification UML par deux types de diagrammes : diagrammes d'état-transitions et diagrammes de collaboration. Le diagramme d'état-transitions est compilé dans un modèle PROMELA et le diagramme de collaboration est interprété par un ensemble d'automates de Büchi. Le modèle checker SPIN est utilisé pour vérifier le modèle par rapport aux automates. L'approche [Gallardo 02] propose d'intégrer UML et le model-checking afin de vérifier les erreurs de la spécification dans les premières versions du modèle. Cette approche permet d'analyser les propriétés générales des diagrammes d'état-transitions et de vérifier ces diagrammes par rapport aux diagrammes de séquence (*desirable* et *non-desirable*).

Réseaux de Petri. On considère également les réseaux de Petri pour la simulation et la validation de la spécification UML. L'approche [Paludetto 05] aborde la formalisation et la validation des besoins à travers les cas d'utilisation associés aux diagrammes de collaboration et de séquence. Les diagrammes de séquence sont traduits en réseaux de Petri, et l'assemblage des différents modèles associés à chaque diagramme de séquence permet de construire les réseaux de Petri de comportement des objets.

Esterel. L'approche [Dion 01] propose d'intégrer Esterel dans la spécification UML d'une application temps réel (UML/RT). Cette approche permet de vérifier que le système n'aboutit jamais aux états inattendus, elle vérifie également les propriétés d'atteignabilité. L'approche a été appliquée dans l'industrie (le projet Fuel Control chez Dassault Aviation et le projet Trusted Third Party chez Thomson CSF).

ASM. Un outil utilise ASM (Abstract State Machines) [Shen 01]. Il permet de vérifier les aspects statiques et certaines propriétés dynamiques. Cet outil utilise SMV (Symbolic Model Verifier) comme un model checker pour la vérification de modèle UML et il se focalise sur les diagrammes d'état-transitions.

Méthode B. Il existe quelques approches proposées pour vérifier la spécification UML en utilisant la méthode B. K. Lano *et al* [Lano 04] définissent une dérivation des diagrammes de classes UML et des contraintes OCL vers B pour vérifier la consistance des modèles UML et la correction des propriétés attendues de ces modèles, à savoir les multiplicités spécifiées dans les associations des diagrammes de classes. Marcano *et al* [Marcano 02] propose une approche assez similaire à celle de K. Lano, qui permet de transformer automatiquement les diagrammes de classes et les contraintes OCL en B grâce à l'outil UML2B [Hazem 04]. Cette approche permet de vérifier l'incohérence des invariants, l'incomplétude des préconditions et l'inconsistance des postconditions. La différence de notre approche par rapport à ces approches est représentée par la procédure de dérivation d'UML vers B et les propriétés à vérifier.

3.2.2 Validation formelle

Notre objectif est de valider des spécifications UML en utilisant les outils de B. Ce travail est effectué en prenant les jeux de tests pour valider la spécification construite.

Concernant les approches de validation des spécifications, Kemmerer [Kemmerer 85] propose une approche où les spécifications sont exécutables avec des jeux de tests. Kneuper [Kneuper 89] présente l'animation de spécifications en utilisant l'exécution symbolique comme une méthode de validation.

Concernant les approches de test à partir de la méthode B, nous pouvons citer les approches suivantes :

L'outil BZ-Testing-Tool [Legeard 02, Ambert 02] est issu de travaux menés depuis 1995 au LIFC, à Besançon. Il est basé sur l'exécution symbolique d'un modèle qui permet de parcourir le graphe d'atteignabilité et ainsi de calculer les tests comme des séquences d'activation de stimuli sur le système sous tests. Il supporte trois notations, la notation B (niveau machine abstraite), les statecharts (Statemate) et les diagrammes de classes et d'état-transitions avec des expressions OCL d'UML. Chacune de ces notations est traduite dans le format intermédiaire de l'outil BZ-TT.

Ce format fait lui-même l'objet d'une traduction en un système de contraintes pour permettre l'évaluation symbolique, base de l'animation et de la génération de tests. La génération automatique des tests fonctionnels s'appuie sur une modélisation des spécifications fonctionnelles du système sous tests. Elle est pilotée par l'ingénieur validation sur la base de critères de couverture du modèle. La stratégie de génération suit les étapes suivantes :

- Partitionnement du modèle pour calculer les différents comportements.
- Pour chaque comportement, calcul des valeurs limites afin d'obtenir une instance du comportement sur les valeurs extrêmes des variables du modèle.
- Génération du cas de test par recherche d'atteignabilité de chaque instance de comportement aux limites à partir de l'état initial du système.

La technologie BZ-TT est industrialisée au sein de la société Leirios Technologies avec de l'outil Leirios Test Generator⁸.

ProTest [Satpathy 04] est un environnement de test automatique pour des spécifications B. Il est basé sur ProB, un modèle checker et un outil d'animation pour B [Leuschel 03]. Il génère des jeux de tests à partir de spécifications B par l'analyse de la partition des états dans l'invariant et des préconditions des opérations. Il anime simultanément les spécifications et simule leur exécution en respectant des jeux de tests et en assignant des verdicts si l'exécution a réussi le test. Cet environnement impose quelques restrictions sur les arguments et les résultats. ProTest permet d'animer et de vérifier une seule machine B à la fois.

Dans notre proposition présentée dans le chapitre 8, B est utilisé pour spécifier des modèles orientés objets dont les opérations interagissent entre elles pour atteindre un but. Les opérations dans les machines abstraites sont modélisées de manière indépendante, c'est-à-dire que les développeurs modélisent les opérations mais ils ne s'assurent pas que les enchaînements sont possibles et qu'elles satisfont des contraintes du système. Cela pose de problèmes de validation des spécifications orientées objets en B. Nous proposons de tester l'exécution d'une séquence d'opérations, exprimée par des diagrammes de séquences UML, y compris des contraintes dynamiques du système. Ces points ne sont pas pris en compte dans les approches BZ-TT et ProTest.

L'utilisation de scénarios exprimés par des diagrammes de séquences pour tester les spécifications orientées objets est considérée dans [Pickin 04]. Cette approche utilise *TeLa*⁹ où le test peut être décrit en utilisant le scénario de *TeLa one-tier*, associant des conditions avec les messages des diagrammes de séquences, ou le scénario *TeLa two-tier*, combinant des diagrammes d'activités UML avec des diagrammes de séquences. Notre proposition est inspirée de cette approche pour tester l'exécution des opérations spécifiées dans des machines abstraites B.

3.3 Spécification formelle orientée objet

La spécification est une étape du développement d'un logiciel qui consiste à décrire ce que le logiciel doit faire. Plusieurs types de spécifications peuvent être définies :

- les spécifications informelles,
- les spécifications semi-formelles,
- les spécifications formelles

8. <http://www.leirios.com>

9. Test Description Language

Un nombre d'approches de spécification formelle orientées objets sont proposées dans la littérature, elles proposent l'extension à objets de langages formels, permettant de modéliser les propriétés des systèmes objets avec les notations de ces méthodes :

- Object-Z, VDM++, Z++, OOZE, MooZ, ... pour les approches fondées sur la logique du premier ordre et la théorie des ensembles;
- HOSA, TROLL, ... pour les approches algébriques;
- CLOWN, CO, OPN, ... pour les approches basées sur les réseaux de Pétri et les réseaux de haut niveau;
- TRIO+, OO-LTL, ... pour les approches basées sur la logique temporelle.

Nous détaillons suivante les extensions à objets de certains langages formels fondés sur la logique du premier ordre et la théorie des ensembles (comme la méthode B) :

Object-Z. Object-Z est une extension du langage de spécification Z aux concepts à objets [Casais 93, Duke 90, Smith 92a, Cusack 91]. L'objectif initial du langage est de structurer des spécifications Z pour les notations orientées objets. Pour cela, un nouveau schéma encapsule différents schémas Z dans une classe. Un schéma de classe permet de regrouper un ensemble cohérent de schémas d'état ou d'opération dans une même structure syntaxique. Object-Z ajoute des opérateurs de schéma à Z: le choix indéterministe $S \parallel T$, la conjonction $S \bullet T$ (accessibilité des déclarations) et l'opérateur parallèle $S \parallel\parallel T$, qui correspond en fait à une sorte de sérialisation car les paramètres de même nom sont unifiés.

Une interprétation comportementale est possible au travers de traces, appelées invariant historique. Un historique est une séquence d'événements que subit un objet. Ces événements correspondent à des appels d'opérations. L'invariant historique pourrait être donnée en termes de prédicats Z [Smith 92b].

La sémantique d'Object-Z est une extension de celle de Z. En Z, chaque expression est typée et chaque type détermine un ensemble support comme dans les algèbres. Object-Z rajoute le type classe. L'état est une fonction de l'espace des variables vers les ensembles de valeurs.

Cusack distingue l'héritage incrémental de l'héritage de sous-type [Cusack 91]. Ce dernier peut être défini par un modèle basé sur la théorie des ensembles tandis que le premier correspond à un raffinement de schéma (ajout de schémas d'opérations, assouplissement de préconditions, renforcement de postconditions, extension des variables avec renforcement de l'invariant et du schéma d'initialisation). Smith propose une sémantique abstraite en termes d'équivalence observationnelle [Smith 92a]. Il sépare l'aspect structurel et l'aspect temporel des classes dont les preuves sont implicitement de nature différente.

VDM++. VDM++ est une extension du langage VDM aux spécifications modulaires par classes et au parallélisme [Durr 92]. Une spécification VDM++ est la donnée de types, d'un ensemble de classes et d'un espace de travail optionnel. Bien que dans VDM définisse une notion de module, la seule structure possible en VDM++ est la classe. Les modules VDM, par contre, sont utilisés pour décrire la sémantique des classes. Sommairement, la classe définit l'état des objets par un ensemble de variables d'instance, le comportement par un ensemble de méthodes et le comportement dynamique par des traces.

VDM++ distingue l'héritage simple de structure de l'héritage multiple du comportement. La structure d'un objet définit entièrement son type. Une sous-classe est obtenue par extension de

type (ajout de variables d'instance), avec compatibilité ascendante. Ce n'est pas un sous-type au sens habituel du terme, mais une sous-classe.

VDM++ définit un espace de travail, appelé “*workspace metaphor*”, tel que la spécification du système est la description de l'ensemble des classes et celle de l'espace de travail. L'espace de travail correspond aux éléments dynamiques du système, comme dans le langage Smalltalk. Ce concept permet une conception et une implantation dans un style interactif et itératif. Pour unifier les concepts du langage, l'espace de travail est une classe. Une autre originalité de VDM++ est le traitement de la concurrence et du temps-réel. Ce dernier est traité par des fonctions de manipulation de temps continu et discret (*now*, *duration*, ...). Pour la concurrence, les auteurs proposent une panoplie de notations, regroupées dans deux parties: le comportement dynamique pour les objets actifs et le contrôle de la synchronisation entre les méthodes.

OOZE. OOZE (Object-Oriented Z Environment) [Alencar 91] est un langage destiné à couvrir plusieurs activités du développement. Ce langage supporte la description des besoins, la spécification, les programmes interprétés et compilés. Parmi toutes les extensions de Z, OOZE est un peu particulière. Elle a une sémantique algébrique basée sur OBJ [Goguen 96] et FOOPS [Rapanotti 92] (et non sur Z) mais sa syntaxe reste celle de Z. Le langage est très riche, il couvre une large partie des concepts à objets notamment les méta-classes. Par contre, les aspects dynamiques ne sont pas pris en compte.

Z++. Z++ [Lano 91] est certainement le plus complet des langages de spécification et de conception à objets basé sur Z. Sa syntaxe s'inspire des machines abstraites de B. Le langage fournit des mécanismes de modularisation de la spécification. Il ne prend pas en compte la notion d'identité d'objet ou de manipulation directe des instances d'une classe. Une sémantique algébrique sert de support au raisonnement et à la preuve des spécifications et une sémantique basée sur les machines abstraites est utilisée pour le raffinement et le prototypage.

La description ci-dessus n'est pas complète mais reflète de façon significative les travaux menés dans le domaine de l'extension des méthodes formelles par des concepts à objets. Une description détaillée et une analyse comparative peuvent être consultées dans [André 95]. Dans notre travail de thèse, nous proposons une approche de spécification formelle orientée objets en prenant en compte certains types d'association pour les machines abstraites B.

3.4 Synthèse

Dans ce chapitre, nous avons présenté l'état de l'art de nos travaux d'utilisation de B pour la validation et la vérification de spécifications UML et le développement formel orienté objets.

Nos travaux constituent une suite aux travaux de thèse de Meyer [Meyer 01] et Ledang [LeDang 02a] qui étudient la dérivation des notations de base de UML vers B. Un résumé des schémas de dérivation de diagrammes de classes, de diagrammes d'état-transitions, de diagrammes de collaboration UML et d'expressions OCL en B est présenté.

La vérification formelle est basée sur la démonstration logique ou mathématique. Plusieurs approches formelles ont été proposées pour la vérification de spécifications UML à l'aide de leur prouveur, tels que le Model Checking, Esterel, les réseaux de Petri, la méthode B, etc.

La validation formelle concerne essentiellement le test de spécifications, permettant aux développeurs d'avoir au plus tôt l'assurance de la complétude et de la cohérence des fonctionnalités dans les spécifications.

Les approches de spécification formelle orientée objets permettent de modéliser les propriétés objets des systèmes avec les notations des méthodes formelles. Les approches connues dans la littérature sont : Object-Z, Z++, VDM++, etc.

DEUXIÈME PARTIE :

Vérification de spécifications UML en utilisant B

Les spécifications UML utilisent les notations graphiques pour modéliser les systèmes. Cependant, les développeurs manquent d'outils pour vérifier et valider ces spécifications. Une solution pour atteindre un niveau de confiance plus élevé pour les systèmes logiciels est d'utiliser des techniques de validation et de vérification formelles pendant le processus de développement.

La dérivation de spécifications UML vers B est considérée comme une approche appropriée pour utiliser conjointement UML et la méthode formelle B dans un développement unifié et rigoureux de logiciels. On peut utiliser la spécification UML/OCL pour modéliser un système et il est possible d'utiliser des outils supports puissants de B comme AtelierB pour analyser les spécifications B dérivées afin d'identifier les défauts au sein des spécifications UML/OCL.

A partir des schémas de dérivation d'UML vers B proposés dans la section 3.1, nous intégrons et étendons ces dérivations afin de vérifier la spécification UML. Nous proposons la :

Vérification de la sémantique de spécifications UML. Nous effectuons la dérivation de la structure du méta-modèle UML et de règles bien-formées qui expriment la sémantique UML à l'aide d'expressions OCL, vers des machines abstraites B. Cette dérivation a pour but de vérifier les éléments du modèle UML qui doivent satisfaire la sémantique UML.

Vérification des contraintes dans la spécification UML. Nous intégrons la dérivation de diagrammes de classes et de diagrammes objets pour vérifier des contraintes sur les diagrammes de classes et les expressions OCL. Nous intégrons également la dérivation de diagrammes de classes et de diagrammes de collaboration pour vérifier les messages composés/décomposés et les messages séquentiels.

Vérification de la sémantique des modèles UML

SOMMAIRE

4.1	Dérivation d'un objet d'une méta-classe en B	64
4.2	Analyse des règles de bonne formation du méta-modèle UML	64
4.3	Procédure de dérivation du méta-modèle UML en B	66
4.4	Dérivation et vérification des diagrammes de classes	67
4.4.1	Structure générale du méta-modèle des diagrammes de classes et dérivation en B	67
4.4.2	Étude de cas : système d'impression	68
4.5	Dérivation et vérification des diagrammes de collaboration	72
4.5.1	Structure générale du méta-modèle des diagrammes de collaboration et dérivation en B	72
4.5.2	Étude de cas : système d'impression	73
4.6	Dérivation du méta-modèle des diagrammes d'état-transitions en B	78
4.7	Synthèse	78

Le méta-modèle UML définit la syntaxe et la sémantique pour les notations des modèles UML : il est formé d'une syntaxe abstraite présentée par un ensemble de diagrammes de classes et de la sémantique statique donnée en OCL qui garantit que tous les éléments de spécification UML sont statiquement bien formés.

L'idée principale de ce chapitre est la transformation de méta-modèles en B pour vérifier la sémantique des éléments du modèle UML. Pour ce faire, nous proposons de

- transformer les règles de bonne formation du méta-modèle UML en invariants B,
- transformer des objets (instances) des méta-classes en B,
- associer des attributs de ces objets aux invariants transformés à partir de règles de bonne formation,
- utiliser les outils supports de B (AtelierB) pour prouver la spécification B obtenue.

La présentation de ce chapitre est la suivante. D'abord, nous présentons la dérivation d'un objet d'une méta-classe en B. Puis nous analysons des règles de bonne formation du méta-modèle pour donner une procédure de dérivation du méta-modèle UML en B. Nous détaillons la dérivation du méta-modèle des diagrammes de classes, des diagrammes de collaboration et des diagrammes d'état-transitions. Pour chaque type de diagramme, nous donnons une étude de cas et montrons

comment ses éléments sont prouvés à l'aide des outils de preuve. La présentation de ce chapitre est l'intégration et l'extension de trois papiers [Truong 04a, Truong 05b, Truong 06b].

4.1 Dérivation d'un objet d'une méta-classe en B

Plusieurs approches de dérivation des diagrammes de classes UML en B ont été proposées. Dans ces approches, un attribut d'une classe UML est transformé en une variable d'une machine abstraite B, son type est une relation (ou fonction) entre un ensemble d'identifiants d'objets et le type de l'attribut (voir dérivation 3.2).

Grâce à cette dérivation, on peut exprimer facilement les attributs des classes par la construction de relations binaires de la théorie des ensembles. Cette transformation utilise l'ensemble d'identifiants d'objets qui est une variable de la machine abstraite. Toute création d'objet est ajoutée dans cette variable.

La syntaxe abstraite UML présentée dans le méta-modèle, (voir section 2.2.2), correspond à un ensemble de paquetages MOF (Meta Object Facility) qui utilise un sous-ensemble de UML pour décrire les objets manipulés par les outils de conception. XMI (XML Metadata Interchange) indique comment les modèles MOF peuvent être traduits en XML. Le but de ces standards est de permettre à des ateliers logiciels (Rational Rose, ArgoUML, ...) d'explorer et d'échanger les définitions des structures de données, leurs propriétés, les relations les unissant, etc. Pour simplifier, nous travaillons avec la structure XMI et les valeurs des attributs de la spécification XMI.

Les identifiants d'objets des méta-classes sont déterminés par ceux de la spécification XMI générée par les outils d'édition UML. Nous pouvons définir simplement le type des variables dérivées grâce au prédicat suivant:

$$attr_i \in CLASS \rightarrow TYPE(attr_i)$$

où \rightarrow dénote une fonction partielle, $CLASS$ est l'ensemble des identifiants d'objets et $TYPE(attr_i)$ un ensemble des valeurs des attributs des objets.

4.2 Analyse des règles de bonne formation du méta-modèle UML

Comme présenté dans la section 2.2.2, les règles de bonne formation spécifient des contraintes sur les attributs et les associations définies dans le méta-modèle. Chaque règle est définie comme un invariant des instances d'une méta-classe. Un exemple de règle de bonne formation des instances de la méta-classe `Parameter` est exprimé par [OMG 03] :

WFR1. All Parameters should have a unique name

```
self.parameter -> forAll(p1,p2 | p1.name = p2.name implies p1 = p2)
```

Les règles de bonne formation sont transformées en invariants d'une machine B. Ces invariants sont quantifiés existentiellement et/ou universellement. Pour vérifier ces prédicats, on doit, en principe, donner toutes les valeurs potentielles des variables participant dans le prédicat pour que le prouveur examine chaque valeur.

Dans la méthode B, les outils de preuve génèrent automatiquement des obligations de preuve pour prouver la spécification, en vérifiant la préservation des invariants par les substitutions

des opérations et l'initialisation. Cela signifie que les valeurs des variables satisfont toujours les invariants des machines abstraites après l'exécution des substitutions. Cela implique aussi que les variables participant dans les invariants transformés à partir des règles de bonne formation sont des variables d'une machine abstraite B et qu'elles doivent contenir toutes les valeurs des attributs des objets après substitution.

Selon la dérivation des attributs des objets (section 4.1), tous les attributs des objets instanciés à partir d'un attribut d'une méta-classe sont du même type. Celui-ci est une relation entre un ensemble des identifiants d'objets et un ensemble des valeurs des attributs des objets. Ceci nous amène à créer une nouvelle variable *attr* typée comme celle des objets, cette variable permettant de rassembler toutes les variables des machines abstraites des objets :

$$\begin{aligned} attr &\in CLASS \rightarrow TYPE(attr) \\ attr &:= attr_1 \cup attr_2 \cup \dots \cup attr_n \end{aligned}$$

La valeur de la variable *attr*, est un ensemble de couples d'identifiants d'objets correspondant aux valeurs des attributs des objets :

$$attr = \{object_1 \mapsto value_1, object_2 \mapsto value_2, \dots, object_n \mapsto value_n\}$$

Nous devons créer une nouvelle machine abstraite de même structure que celle des objets. Dans cette machine, nous ajoutons une opération, appelée *collectData*, permettant d'intégrer les valeurs des objets à ses variables (les nouvelles variables).

Remarque 4.1 Les règles de bonne formation s'appliquent seulement à un objet composite. Par exemple, la règle *WFR1* est correcte avec les paramètres d'une opération mais elle ne peut pas être appliquée aux paramètres d'autres opérations. Dans ce contexte, cette opération est appelée un objet composite, ses paramètres sont appelés des objets composants. Pour réduire le nombre de machines dans le système, nous proposons de joindre la machine qui contient les nouvelles variables et l'opération *collectData* des objets composants dans la machine de son objet composite.

Pour faciliter la compréhension de la procédure de dérivation du méta-modèle UML en B, nous introduisons deux définitions :

Définition 4.1 Une classe composite est une classe qui correspond à "l'ensemble" dans une relation de composition; un objet composite est une instance de la classe composite.

Définition 4.2 Une classe composant est une classe qui correspond à "la part" dans une relation de composition; un objet composant est une instance de la classe composante (Figure 4.1).

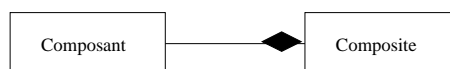


Figure 4.1 – Relation entre classe composante et classe composite

4.3 Procédure de dérivation du méta-modèle UML en B

Le modèle UML est composé de plusieurs diagrammes (section 2.2.1). Nous nous concentrons sur la vérification de la sémantique des diagrammes les plus utilisés : diagramme de classes, diagramme de collaboration et diagramme d'état-transitions.

La structure du méta-modèle des diagrammes de classes et des diagrammes de comportements (diagrammes de collaboration et diagrammes d'état-transitions) est similaire. Les diagrammes de classes sont utilisés pour décrire les propriétés statiques (attributs et associations) de modèles UML. Cependant, les attributs des objets du paquetage *Behavioural Elements* (il se compose des paquetages *Collaboration* et *State Machines*) peuvent être valués par un ensemble d'éléments, tandis que ceux du paquetage *Core* ont seulement une valeur. Les attributs sont transformés en variables B, le type des variables est soit une fonction partielle soit une relation entre l'ensemble des identifiants d'objets et le type des attributs.

Une autre différence entre les paquetages *Behavioural Elements* et *Core* concerne les règles de bonne formation. Dans le paquetage *Core*, les règles de bonne formation sont simples, chaque règle exprime des contraintes pour un seul attribut. Les règles de bonne formation du paquetage *Behavioural Elements* sont plus complexes, avec plusieurs attributs intervenant dans une règle. Pour vérifier la correction de chaque règle transformée en un invariant B, les variables qui participent à l'invariant doivent être valuées simultanément.

A partir de ces remarques, nous proposons une procédure de dérivation générale afin de vérifier des éléments du modèle UML.

Soit $attr_i$ ($i = 1..m$) une variable additionnelle d'une machine à objets composites, utilisé pour rassembler les valeurs des objets,

- $attr_{ij}$ ($i = 1..m, j = 1..n$) une variable d'une machine à objets composants,
- m est le nombre d'attributs de l'objet composant,
- n est le nombre d'objets composants d'un objet composite.

Procédure de dérivation.

- Chaque objet d'une méta-classe (un élément du modèle UML) est dérivé en une machine abstraite B, les attributs de l'objet sont dérivés en variables de la machine abstraite. Le type des variables est exprimé dans la clause INVARIANT par une fonction partielle de l'ensemble des identifiants d'objets vers le type des attributs :

$$attr_{ij} \in CLASS \rightarrow TYPE(attr_{ij})$$

- La valeur des variables est initialisée dans la clause INITIALISATION par un ensemble d'identifiants d'objets correspondant à la valeur de l'attribut :

$$attr_{ij} := \{object_j \mapsto value_{ij}\}$$

- Les machines à objets composites contiennent non seulement des variables transformées à partir des attributs de ces objets, mais également des variables additionnelles pour rassembler les valeurs des variables dans les machines à objets composants. Ces variables sont typées comme celles des objets composants :

$$attr_i \in CLASS \rightarrow TYPE(attr_i)$$

- Une opération supplémentaire est ajoutée dans la clause OPERATIONS des machines à objets composites pour rassembler les valeurs des variables des machines à objets composants (voir Figure 4.2).

```

?
collectData =
pre
   $\bigwedge attr_{ij} = value_{ij}$ 
then
   $attr_1 := attr_{11} \cup attr_{12} \cup \dots \cup attr_{1n} \parallel$ 
   $attr_2 := attr_{21} \cup attr_{22} \cup \dots \cup attr_{2n} \parallel$ 
  ...
   $attr_m := attr_{m1} \cup attr_{m2} \cup \dots \cup attr_{mn}$ 
end

```

Figure 4.2 – Spécification de l'opération ajoutée collectData

Nous pouvons rassembler les variables des machines à objets composants dans les variables additionnelles parce que les types de ces variables sont les mêmes.

- Les règles de bonne formation des classes composantes du méta-modèle sont dérivées en des invariants des machines à objets composites.

4.4 Dérivation et vérification des diagrammes de classes

En appliquant la procédure de dérivation présentée dans la section 4.3, nous introduisons la structure du méta-modèle des diagrammes de classes et sa dérivation en B, puis nous illustrons cette dérivation et la vérification de la sémantique de diagrammes de classes par une étude de cas.

4.4.1 Structure générale du méta-modèle des diagrammes de classes et dérivation en B

La structure générale du méta-modèle UML des diagrammes de classes est résumée par la spécification XMI présentée en partie gauche de la Figure 4.3. La structure des différentes machines abstraites B dérivées suit la même structure comme présenté dans la partie droite de la Figure 4.3.

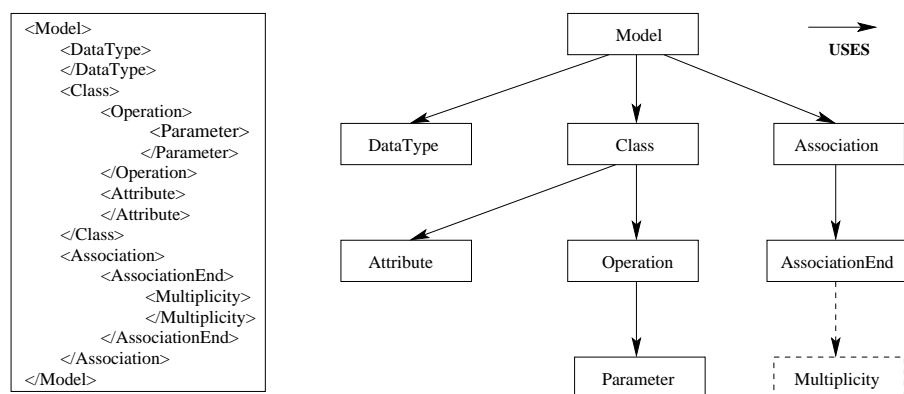


Figure 4.3 – Structure générale du méta-modèle des diagrammes de classes et sa dérivation en B

Remarque 4.2 Les machines abstraites à objets de la méta-classe Multiplicity sont combinées avec les machines à objets de la méta-classe AssociationEnd pour devenir un type de machine : les machines à objets de la méta-classe AssociationEnd. Les attributs des objets de la méta-classe Multiplicity

sont dérivés en variables de machines abstraites à objets de la méta-classe `AssociationEnd`. Cette dérivation a pour but de rassembler les valeurs des attributs d'objets et de les associer dans les règles de bonne formation des objets de la méta-classe `Association`.

Cette dérivation de la structure des machines abstraites permet de maintenir la structure de machines abstraites correspondant aux méta-classes dans le méta-modèle. Elle est claire et simple. Elle permet d'utiliser les règles de bonne formation du méta-modèle comme des invariants dans les machines abstraites et d'exploiter le prouveur B pour prouver des éléments des modèles UML.

4.4.2 Étude de cas : système d'impression

Pour illustrer notre approche de la dérivation et la vérification de la sémantique des éléments du modèle UML, nous présentons la spécification d'un système d'impression utilisé pour imprimer des fichiers à partir d'un ordinateur.

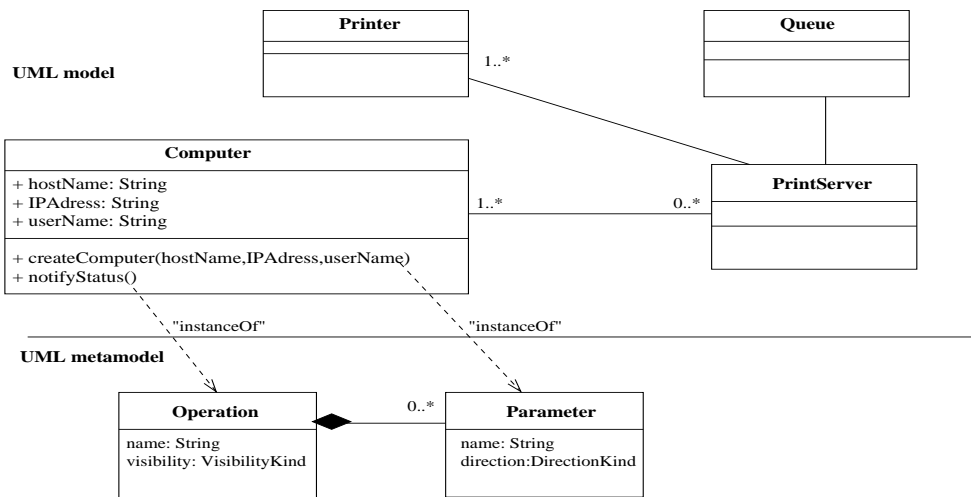


Figure 4.4 – Relation entre modèle et méta-modèle du diagramme de classes

Le fonctionnement de ce système est présenté comme suit : lorsqu'un utilisateur demande l'impression d'un fichier, cette commande est transférée au serveur d'impression. Si l'imprimante est occupée, le fichier à imprimer est stocké dans une file d'attente, sinon il est imprimé; le serveur d'impression notifie l'état des processus à l'ordinateur. La Figure 4.4 présente le diagramme de classes de ce système et sa relation avec certaines méta-classes du paquetage `Core` du méta-modèle. Nous décrivons dans ce diagramme les éléments et les propriétés nécessaires pour illustrer la dérivation. Chaque élément du diagramme de classes du modèle UML est une instance d'une méta-classe du méta-modèle.

Dérivation en B. Considérons la dérivation en B d'un objet de la méta-classe `Operation`, l'opération `createComputer` appartenant à la classe `Computer` du modèle UML du système d'impression. Un exemple de sa spécification XMI générée par les outils d'édition UML est donné par :

```

<UML:Operation xmi.id="xmi.011">
  <UML:ModelElement.name> createComputer </UML:ModelElement.name>
  <UML:ModelElement.visibility xmi.value="public"/>
  <UML:ModelElement.isSpecification xmi.value="false"/>
  <UML:BehavioralFeature.isQuery xmi.value="false"/>

```

```

CreateComputer.mch
MACHINE CreateComputer

SEES Types

VARIABLES
  createComputer_name,
  createComputer_visibility,
  ...

INVARIANT
  createComputer_name ∈ OPERATION → OPERATION_NAME ∧
  createComputer_visibility ∈ OPERATION → VISIBILITYKIND ∧
  ...

INITIALISATION
  createComputer_name := {O11 ↦ createComputer} ||
  createComputer_visibility := {O11 ↦ public} ||
  ...

END

```

Figure 4.5 – Machine abstraite B correspondant à l'opération *createComputer*

```

<UML:Operation.isRoot xmi.value="false"/>
<UML:Operation.isLeaf xmi.value="false"/>
<UML:Operation.isAbstract xmi.value="false"/>
<UML:Feature.owner>
  Here defines the parameters
</UML:Feature.owner>
</UML:Operation>

```

Le résultat de la dérivation des attributs de l'objet *createComputer* du modèle UML vers une machine abstraite B, en appliquant la procédure de dérivation présentée dans la section 4.3, est donné dans la Figure 4.5.

En continuant l'illustration de la procédure de dérivation, les machines d'objets composites contiennent non seulement des variables transformées à partir des attributs de ces objets, mais aussi des variables additionnelles permettant de rassembler les valeurs des variables dans les machines correspondant aux objets composants.

Rappelons que chaque paramètre de l'opération *createComputer* est dérivé en une machine abstraite B (*CreateComputer_HostName*, *CreateComputer_IPAdress*, *CreateComputer_UserName*); la structure de ces machines, selon la procédure de dérivation, est similaire à celle de la machine *CreateComputer*. Dans le méta-modèle UML, la méta-classe *Parameter* est une composante de la méta-classe *Operation*. La machine abstraite correspondant à l'opération *createComputer* contient donc des variables additionnelles permettant de rassembler les valeurs des variables des machines de ses paramètres. La spécification complète de la machine abstraite *CreateComputer* est présentée Figure 4.6.

Nous détaillons maintenant les relations entre les machines abstraites B dérivées à partir du diagramme de classes présenté Figure 4.4.

```

CreateComputer.mch
MACHINE CreateComputer

SEES Types

USES
    CreateComputer_HostName, CreateComputer_IPAdress, CreateComputer_UserName

VARIABLES
    parameter_name,
    parameter_direction,
    ...

INVARIANT
    parameter_name ∈ PARAMETER → PARAMETER_NAME ∧
    parameter_direction ∈ PARAMETER → DIRECTIONKIND ∧
    ...

INITIALISATION
    parameter_name := ∅ ||
    parameter_direction := ∅ ||
    ...

OPERATIONS
    collectData =
        pre
            hostName_name = {P1 ↦ hostName} ∧
            hostName_direction = {P1 ↦ in} ∧
                /* à partir de la machine CreateComputer_HostName */
            ipAdress_name = {P2 ↦ ipAdress} ∧
            ipAdress_direction = {P2 ↦ in} ∧
                /* à partir de la machine CreateComputer_IPAdress */
            userName_name = {P3 ↦ userName} ∧
            userName_direction = {P3 ↦ in} ∧ ...
                /* à partir de la machine CreateComputer_UserName */
        then
            parameter_name :=
                hostName_name ∪ ipAdress_name ∪ userName_name ||
            parameter_direction :=
                hostName_direction ∪ ipAdress_direction ∪ userName_direction ||
            ...
        end

END

```

Figure 4.6 – Spécification complète de la machine abstraite CreateComputer

La machine *Model* utilise (USES) les machines des objets (*Computer_PrintServer*, *PrintServer_Printer*, ...) de la méta-classe Association¹⁰ et les machines des objets de la méta-classe Class (i.e. *Computer*, *PrintServer*, ...) (voir Figure 4.7). Les machines des objets de la méta-classe Association (i.e. *Computer_PrintServer*, ...) utilisent les machines des objets de la méta-classe AssociationEnd (i.e. *Computer_PrintServer_computer*, *Computer_PrintServer_printserver*, ...).

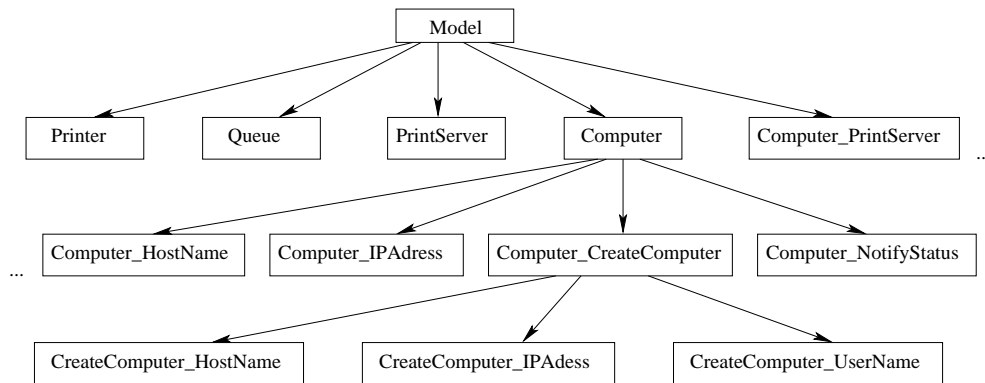


Figure 4.7 – Structure des machines *B* dérivées à partir du diagramme de classes du système d'impression

Les machines des objets de la méta-classe Class (i.e. *Computer*, ...) utilisent les machines des objets de la méta-classe Attribute (i.e. *Computer_HostName*, *Computer_IPAdress*, *Computer_UserName*) et les machines des objets de la méta-classe Operation (i.e. *Computer_CreateComputer*, *Computer_NotifyStatus*) (leurs noms sont préfixés par le nom de la méta-classe). Les machines des objets de la méta-classe Operation (i.e. *Computer_CreateComputer*, ...) utilisent les machines des objets de la méta-classe Parameter (i.e. *CreateComputer_HostName*, *CreateComputer_IPAdress*, *CreateComputer_UserName*) (leurs noms sont préfixés par le nom de l'opération afin de les distinguer des machines des objets de la méta-classe Attribute).

```

Types.mch
-----
MACHINE Types
SETS
  CLASS = {C1, C2, C3, C4}; /* xmi.id=C1, ...*/
  OPERATION = {O11, O12}; /* xmi.id=O11, ...*/
  PARAMETER = {P1,P2,P3};
  PARAMETER_NAME = {hostName,ipAdress,userName}
  VISIBILITY_KIND = {public,private,protected};
  DIRECTION_KIND = {in,out,inout};
  ...
END

```

Figure 4.8 – Machine *Types* du diagramme de classes du système d'impression

Toutes les machines dans le système voient (clause SEES) la machine *Types* (Figure 4.8) qui définit les types de données du système, c'est-à-dire des ensembles dont les éléments sont extraits à partir de la spécification XMI du méta-modèle du diagramme de classes.

10. Pour les associations n'ayant pas de nom, nous donnons le nom composé des noms des deux classes connectées par l'association

Vérification des éléments du modèle de diagramme de classes. Nous utilisons la règle de bonne formation *WFR1* présentée dans la section 4.2 pour illustrer la vérification des éléments du diagramme de classes. Ce prédicat OCL est dérivé en un invariant B, présenté Figure 4.9. Les règles de bonne formation de la méta-classe *Parameter* sont incluses dans la machine abstraite des objets de la méta-classe *Operation*. L'attribut *name* est renommé en *parameter_name* pour avoir la même notation que la variable additionnelle de la machine.

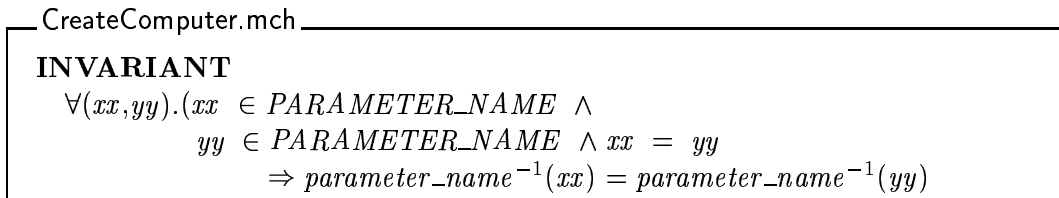


Figure 4.9 – Dérivation en B de la règle bien-formée *WFR1*

Les ensembles *PARAMETER* et *PARAMETER_NAME* sont définis comme des ensembles énumérés dans la machine *Types* du diagramme de classes (voir Figure 4.8).

Nous rappelons que l'obligation de preuve qui garantit la préservation de l'invariant pour une opération B d'une machine abstraite est de la forme : $I \wedge P \Rightarrow [S]I$, dans laquelle *P* est la précondition de l'opération, *S* est sa substitution et *I* l'invariant de la machine abstraite.

Appliquons cette obligation de preuve à l'invariant de la Figure 4.9 et à l'opération *collectData* de la machine *CreateComputer* (Figure 4.6). L'obligation de preuve concrète générée par le prouveur B est :

$$\forall(xx,yy).(xx \in \{hostName, ipAdress, userName\} \wedge yy \in \{hostName, ipAdress, userName\} \wedge xx = yy \Rightarrow (\{P1 \mapsto hostName, P2 \mapsto ipAdress, P3 \mapsto userName\}^{-1}(xx) = \{P1 \mapsto hostName, P2 \mapsto ipAdress, P3 \mapsto userName\}^{-1}(yy)))$$

Il est simple de prouver que le résultat de ce prédicat est *true*.

De façon similaire, nous transformons les autres règles de bonne formation du paquetage *Core* en B pour vérifier la sémantique des éléments du diagramme de classes du modèle UML.

4.5 Dérivation et vérification des diagrammes de collaboration

Le méta-modèle des diagrammes de collaboration appelé le paquetage *Collaboration* est un sous-paquetage de *Behavioural Elements*. Il spécifie les concepts nécessaires pour exprimer comment les différents éléments d'un modèle interagissent.

4.5.1 Structure générale du méta-modèle des diagrammes de collaboration et dérivation en B

En appliquant la procédure de dérivation présentée dans la section 4.3, nous dérivons méta-modèle du diagramme de collaboration en B afin de vérifier ses éléments sur son modèle. Les attributs

de chaque objet dans le paquetage **Collaboration** sont transformés en variables dans les machines abstraites, leur type est déterminé comme dans la procédure générale de dérivation. Remarquons que le type des variables transformées à partir des attributs des objets qui peuvent contenir un ensemble d'éléments est défini par une relation de l'ensemble des identifiants d'objets vers l'ensemble des valeurs des attributs d'objets : $attr_i \in CLASS \leftrightarrow TYPE(attr_i)$.

La structure générale des machines abstraites **B** transformées à partir des objets du méta-modèle UML du diagramme de collaboration est présentée dans la **Figure 4.10**. Les machines correspondant à des objets composites utilisent les machines correspondant à des objets composants (dans la spécification XMI, les classes composites sont exprimées par les parents, ses classes composantes par ses enfants).

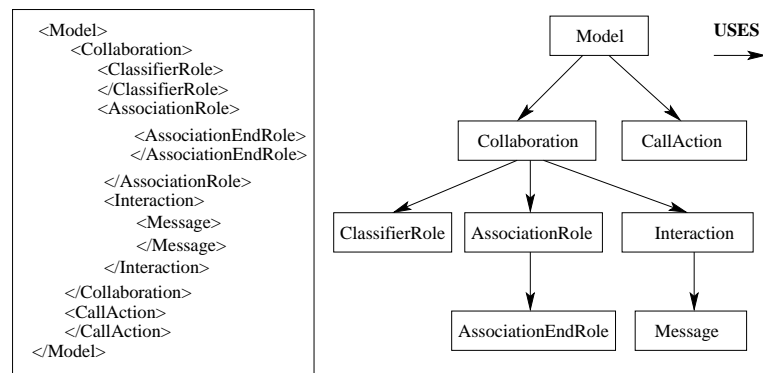


Figure 4.10 – Structure générale du méta-modèle des diagrammes de collaboration et dérivation en *B*

La partie gauche de la figure donne une description XMI résumant les diagrammes de collaboration du modèle UML. La partie droite de la figure représente la structure générale des machines abstraites **B** correspondantes : les machines des objets de la méta-classe **Model** utilise les machines des objets de la méta-classe **Collaboration** et de la méta-classe **CallAction**; les machines des objets de la méta-classe **Collaboration** utilisent les machines des objets des méta-classes **ClassifierRole**, méta-classe **AssociationRole** et méta-classe **Interaction**; etc.

4.5.2 Étude de cas : système d'impression

La **Figure 4.11** introduit le diagramme de collaboration du système d'impression et ses relations avec le méta-modèle des diagrammes de collaboration. Comme dans le cas des diagrammes de classes, chaque élément du diagramme de collaboration du modèle UML est une instance d'une méta-classe du méta-modèle correspondant. Par exemple, le message 1.3.notifyStatus du diagramme de collaboration du système d'impression est une instance de la classe **Message** du méta-modèle.

4.5.2.1 Dérivation en **B**

Pour illustrer la dérivation en **B** du méta-modèle UML des diagrammes de collaboration du système d'impression (**Figure 4.11**), nous introduisons les machines abstraites des objets des méta-classes **Message** et **Interaction**.

Quatre instances sont identifiées dans le modèle UML pour la méta-classe **Message** du méta-modèle : les messages 1, 1.1, 1.2 et 1.3. Selon la procédure de dérivation, chaque instance de la

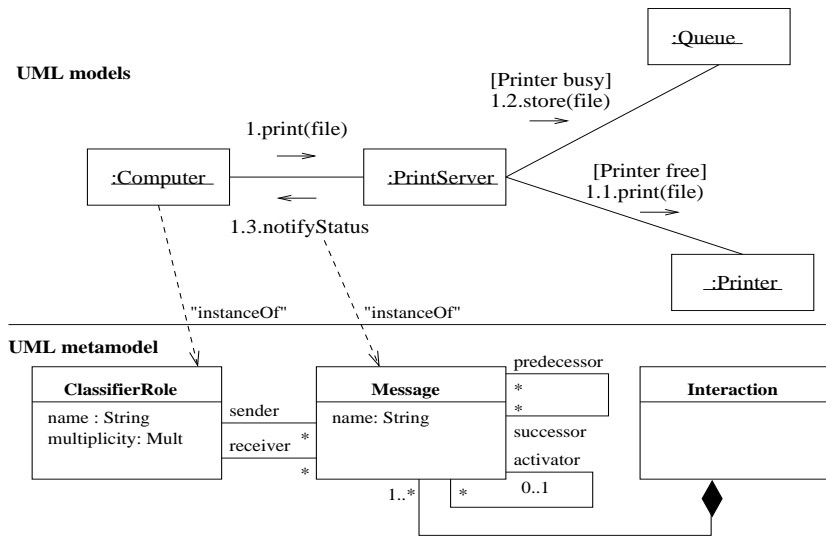


Figure 4.11 – Relation entre le modèle et le méta-modèle du diagramme de collaboration

méta-classe *Message* est dérivée en une machine abstraite *B*, nommée respectivement *Message1*, *Message11*, *Message12* et *Message13*. La méta-classe *Interaction* de cette étude de cas a seulement une instance transformée en une machine abstraite *B* présentée dans la Figure 4.12. La machine abstraite *Interaction* contient non seulement les variables transformées à partir des attributs de l'objet de la méta-classe *Interaction* (préfixé par *interaction*), mais elle contient aussi des variables additionnelles pour rassembler les valeurs des variables des machines des objets de la méta-classe *Message* (préfixée par *message*). Le rassemblement des variables est réalisé par l'opération *collectData* de la machine *Interaction*.

Toutes les machines dans le système voient la machine *Types* (Figure 4.13) qui définit les ensembles utilisés dans le diagramme de collaboration du système d'impression (Figure 4.11).

4.5.2.2 Vérification des éléments du modèle du diagramme de collaboration

Nous exprimons la dérivation des règles de bonne formation de la méta-classe *Message* et la vérification des éléments du modèle UML du paquetage *Collaboration* qui doivent satisfaire ces règles.

WFR2. The predecessors and the activator must be contained in the same interaction.

```
self.predecessor -> forAll(p | p.interaction = self.interaction) and
self.activator -> forAll( a | a.interaction = self.interaction)
```

Ce prédicat OCL est dérivé en un invariant *B* présenté dans la Figure 4.14.

Avec l'étude de cas du système d'impression, la valeur de l'ensemble *MESSAGE* est déterminée dans la machine *Types* (Figure 4.13):

$$MESSAGE = \{mess1, mess11, mess12, mess13\}$$

Les valeurs des variables participant dans la règle de bonne formation *WFR2* sont établies par l'opération *collectData* de la machine *Interaction* (Figure 4.12):

```
message_predecessor = {mess12 ↦ mess11, mess13 ↦ mess11, mess13 ↦ mess12};
message_activator = {mess11 ↦ mess1, mess12 ↦ mess1, mess13 ↦ mess1};
```

```

Interaction.mch
MACHINE Interaction

SEES Types

USES Message1, Message11, Message12, Message13

VARIABLES
interaction_name, interaction_context,
...
message_name, message_interaction, message_sender,
message_activator, message_predecessor,
...
INVARIANT
interaction_name ∈ INTERACTION → INTERACTION_NAME ∧
interaction_context ∈ INTERACTION → COLLABORATION ∧
...
message_name ∈ MESSAGE → MESSAGE_NAME ∧
message_interaction ∈ MESSAGE → INTERACTION ∧
message_sender ∈ MESSAGE → CLASSIFIER_ROLE ∧
message_activator ∈ MESSAGE → MESSAGES ∧
message_predecessor ∈ MESSAGE ↔ MESSAGE ∧
...
INITIALISATION
interaction_name := { inte1 ↦ interaction1 } ||
interaction_context := { inte1 ↦ coll1 } ||
...
message_name := ∅ || message_interaction := ∅ || message_sender := ∅ ||
message_activator := ∅ || message_predecessor := ∅ ||
...

OPERATIONS
  collectData =
    pre
      message1_predecessor = ∅ ∧
      message11_predecessor = ∅ ∧
      message12_predecessor = { mess12 ↦ mess11 } ∧
      message13_predecessor = { mess13 ↦ mess11, mess13 ↦ mess12 } ∧
      ...
    then
      message_predecessor := message1_predecessor ∪
      message11_predecessor ∪ message12_predecessor ∪ message13_predecessor ||
      ...
    end

END

```

Figure 4.12 – Machine abstraite *B* correspondant à l'objet *Interaction*

```

Types.mch
...
SETS
MESSAGE = {mess1,mess11,mess12,mess13};
MESSAGE_NAME = {print,restore,notifyStatus};
CLASSIFIER_ROLE = {class1,class2,class3,class4};
CLASSIFIER_ROLE_NAME = {Computer,PrintServer,Queue,Printer};
...
END
    
```

Figure 4.13 – Machine Types du diagramme de collaboration du système d'impression

```

Interaction.mch
INVARIANT
∀ pp.(pp ∈ MESSAGE ∧ message_predecessor[{pp}] ≠ ∅
    ⇒ message_interaction[message_predecessor[{pp}]] =
        message_interaction[{pp}]) ∧ (WFR2a)
∀ aa.(aa ∈ MESSAGE ∧ message_activator[{aa}] ≠ ∅
    ⇒ message_interaction[message_activator[{aa}]] =
        message_activator[{aa}]) (WFR2b)
    
```

Figure 4.14 – Dérivation en B de la règle de bonne formation WFR2

$message_interaction = \{mess1 \mapsto inte1, mess11 \mapsto inte1, mess12 \mapsto inte1, mess13 \mapsto inte1\}$

A partir de ces valeurs, nous pouvons illustrer la preuve des éléments du diagramme de collaboration du système d'impression par rapport à la règle de bonne formation comme suit :

Preuve de WFR2a.

$pp = mess1$ alors $message_predecessor[\{mess1\}] = \emptyset$;

$pp = mess11$ alors $message_predecessor[\{mess11\}] = \emptyset$;

Dans les deux cas ci-dessus, l'hypothèse de *WFR2a* n'est pas satisfaite.

$pp = mess12$ alors $message_predecessor[\{mess12\}] = \{mess11\}$

$\Rightarrow message_interaction[\{mess11\}] = \{inte1\}$

Donc $message_interaction[message_predecessor[\{mess12\}]] = \{inte1\}$

Comme $message_interaction[\{mess12\}] = \{inte1\}$

on peut en réduire *WFR2a* = true

$pp = mess13$ alors $message_predecessor[\{mess13\}] = \{mess11, mess12\}$

Remarque que: $ran(u \triangleleft r) = r[u]$ avec $u \subseteq s \wedge r \in s \leftrightarrow t$

$\Rightarrow message_interaction[\{mess11, mess12\}]$

$= ran(\{mess11, mess12\} \triangleleft message_interaction)$

$$= \text{ran}(\{ \text{mess11} \mapsto \text{inte1}, \text{mess12} \mapsto \text{inte1} \}) = \{ \text{inte1} \}$$

et $\text{message_interaction}[\{ \text{mess13} \}] = \{ \text{inte1} \} \Rightarrow \text{WFR2a} = \text{true}$.
Cela implique que $\text{WFR2a} = \text{true}$ pour chaque valeur de pp .

Preuve de WFR2b.

$aa = \text{mess1}$ alors $\text{message_activator}[\{ \text{mess1} \}] = \emptyset$
 $aa = \text{mess11}$ ou $aa = \text{mess12}$ ou $aa = \text{mess13}$
alors $\text{message_activator}[\{ aa \}] = \{ \text{mess1} \}$
 $\Rightarrow \text{message_interaction}[\{ \text{mess1} \}] = \{ \text{inte1} \}$
et $\text{message_interaction}[\{ aa \}] = \{ \text{inte1} \} \Rightarrow \text{WFR2b} = \text{true}$ pour chaque valeur de aa .

Par conséquent, nous déduisons que $\text{WFR2} = \text{WFR2a} \wedge \text{WFR2b} = \text{true}$. Autrement dit, nous avons prouvé que les messages prédécesseurs et les messages activateurs dans le diagramme de collaboration du système d'impression sont contenus dans une même interaction avec le message courant.

WFR3. The predecessors must have the same activator as the Message
`self.allPredecessors -> forAll(p | p.activator = self.activator)`

Ce prédicat OCL est dérivé en invariant B présenté dans la Figure 4.15.

Interaction.mch

INVARIANT
 $\forall xx.(xx \in \text{MESSAGE} \wedge \text{message_predecessor}[\{ xx \}] \neq \emptyset \Rightarrow$
 $\text{message_activator}[\text{message_predecessor}[\{ xx \}]] = \text{message_activator}[\{ xx \}]) \quad (\text{WFR3})$

Figure 4.15 – Dérivation en B de la règle de bonne formation WFR3

Preuve de WFR3.

$xx = \text{mess1}$, $\text{message_predecessor}[\{ \text{mess1} \}] = \emptyset$
 $xx = \text{mess11}$, $\text{message_predecessor}[\{ \text{mess11} \}] = \emptyset$
 $xx = \text{mess12}$, $\text{message_predecessor}[\{ \text{mess12} \}] = \{ \text{mess11} \}$
Cela implique que $\text{message_activator}[\{ \text{mess11} \}] = \{ \text{mess1} \}$
et $\text{message_activator}[\{ \text{mess12} \}] = \{ \text{mess1} \}$
 $\Rightarrow \text{WFR3} = \text{true}$

$xx = \text{mess13}$, $\text{message_predecessor}[\{ \text{mess13} \}] = \{ \text{mess11}, \text{mess12} \}$
alors $\text{message_activator}[\{ \text{mess11}, \text{mess12} \}]$
 $= \text{ran}(\{ \text{mess11}, \text{mess12} \} \triangleleft \text{message_activator})$
 $= \text{ran}(\{ \text{mess11} \mapsto \text{mess1}, \text{mess12} \mapsto \text{mess1} \}) = \{ \text{mess1} \}$
et $\text{message_activator}[\{ \text{mess13} \}] = \{ \text{mess1} \}$
Nous donc obtenons que $\text{WFR3} = \text{true}$.

En conséquence, nous avons prouvé que le résultat de ce prédicat est vrai pour tous les messages dans le diagramme de collaboration du système d'impression présenté.

La vérification des règles de bonne formation peut être implantée dans AtelierB dont le prouveur permet de démontrer automatiquement et interactivement des théorèmes.

4.6 Dérivation du méta-modèle des diagrammes d'état-transitions en B

Le méta-modèle des diagrammes d'état-transitions est appelé le paquetage *State Machines*. Il définit un ensemble de concepts utilisés pour spécifier le comportement d'un système par un automate à états finis. La procédure de dérivation du paquetage *State Machines* en B est similaire à celle du paquetage *Collaboration*. En nous basant sur la structure du paquetage *State Machines* (partie gauche de la Figure 4.16), nous décomposons les machines abstraites B comme présenté dans la partie droite de la Figure 4.16.

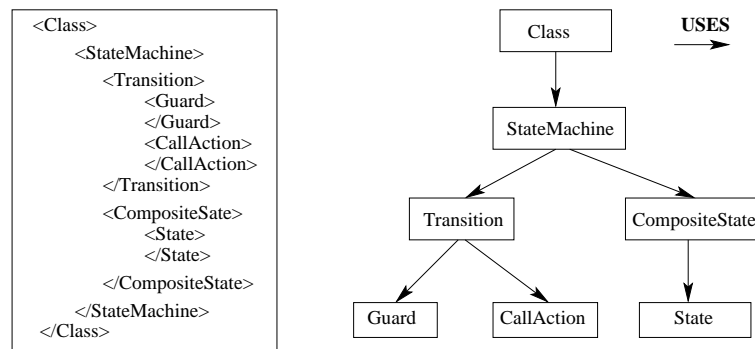


Figure 4.16 – Structure générale du méta-modèle des diagrammes d'état-transitions et sa dérivation en B

Les machines des objets de la méta-classe **Class** utilisent les machines des objets de la méta-classe **StateMachine**; les machines des objets de la classe **StateMachine** utilisent les machines des objets de la méta-classe **Transition** et les machines des objets de la méta-classe **CompositeState**, etc.

La vérification de la sémantique des diagrammes d'état-transitions est similaire à celle du diagramme de collaboration.

4.7 Synthèse

Dans ce chapitre, nous avons proposé une nouvelle approche pour la dérivation du méta-modèle UML en B afin de vérifier la sémantique d'UML. Cette sémantique est exprimée par les règles de bonne formation qui sont transformées en invariants B. Nous avons transformé la structure des méta-classes en des machines abstraites B. Les éléments du modèle de la spécification UML sont intégrés aux invariants de la machine abstraite par les substitutions des opérations supplémentaires. Les obligations de preuve de préservation des invariants permettent de prouver la correction des éléments d'UML par rapport à leur sémantique. Les paquetages *Core*, *Collaboration* et *State Machines* sont considérés pour vérifier la sémantique des éléments des diagrammes de classes, des diagrammes de collaboration et des diagrammes d'état-transitions. Notre approche de dérivation et de vérification est illustrée par une étude de cas, le système d'impression.

Vérification de diagrammes statiques UML

SOMMAIRE

5.1	Étude de cas	80
5.2	Dérivation des diagrammes statiques UML en B	81
5.2.1	Dérivation du diagramme de classes	81
5.2.2	Dérivation des contraintes UML et OCL	81
5.2.3	Dérivation du diagramme d'objets	83
5.3	Vérification des contraintes du système par combinaison des dérivations	86
5.3.1	Vérification des contraintes simples	86
5.3.2	Vérification des contraintes combinées	88
5.4	Synthèse	89

Le but d'un modèle est de décrire les états possibles d'un système et son comportement. L'état d'un système comporte des objets, des valeurs et des liens. Chaque objet est décrit par un descripteur de classe. La donnée d'un objet comporte une valeur pour chaque attribut dans son descripteur de classe. La valeur doit être conforme au type de l'attribut. La donnée d'un lien comporte un tuple contenant une liste d'instances. Les instances et les liens doivent obéir à toutes les contraintes sur les descripteurs desquels ils sont des instances (y compris des contraintes explicites et des contraintes intégrées telles que la multiplicité).

L'état d'un système est une instance valide de ce système si toute instance se compose des instances de certains éléments dans le modèle de système et si toutes les contraintes imposées par le modèle sont satisfaites par ces instances.

L'idée principale de ce chapitre est de vérifier si un état d'un modèle UML est une instance valide. Les descripteurs du système que nous prenons sont décrits par un diagramme de classes, des contraintes OCL associées et des instances du modèle décrit par des diagrammes objets. Les contraintes du modèle sont vérifiées si elles satisfont les données des diagrammes objets.

Nous utilisons la technique présentée dans le chapitre 4 pour décrire cette approche, dans laquelle, les objets d'un diagramme objets jouent le rôle des objets d'une méta-classe, les variables des machines correspondant à des classes sont utilisées pour rassembler les valeurs des machines des objets. Les règles de bonne formation sont remplacées par les contraintes UML et OCL.

La structure de ce chapitre est comme suit. D'abord, nous présentons une étude de cas pour illustrer notre approche. Ensuite, nous introduisons la dérivation des diagrammes statiques en B à partir de la dérivation des diagrammes de classes et des contraintes OCL vers B proposées dans les thèses [Meyer 01, LeDang 02a] et en ajoutant la dérivation des diagrammes d'objets en

B. Nous intégrons ces dérivations pour valider les contraintes UML et OCL. La présentation de ce chapitre est une extension du papier [Truong 05a].

5.1 Étude de cas

Nous illustrons notre approche par une étude de cas, un système qui gère des informations relatives aux banques et compagnies auxquelles des personnes adhèrent. Ce système peut être décrit à l'aide de trois classes : **Person**, **Bank** et **Company** présentées dans le diagramme de classes de la Figure 5.1. Chaque personne dispose d'au plus un compte bancaire. Toute personne peut être employée ou le manager dans une compagnie. Le système doit satisfaire les exigences suivantes :

1. L'âge de chaque personne est positif
context Person
 inv: *self.age* > 0
2. Chaque compagnie a au moins cinq employés
context Company
 inv: *self.numberOfEmployee* > 5
3. L'âge des managers est supérieur à 40 ans
context Company
 inv: *self.manager.age* > 40
4. Si une personne est un manager ou un employé d'une compagnie, elle n'est pas un chômeur
context Company
 inv: *self.manager.isUnemployed* = false
 inv: *self.employee.isUnemployed* = false

Les contraintes (1) et (2) sont des contraintes simples parce qu'elles portent sur des attributs d'une même classe. Les contraintes (3) et (4) sont des contraintes combinées car elles relient les propriétés de plusieurs classes.

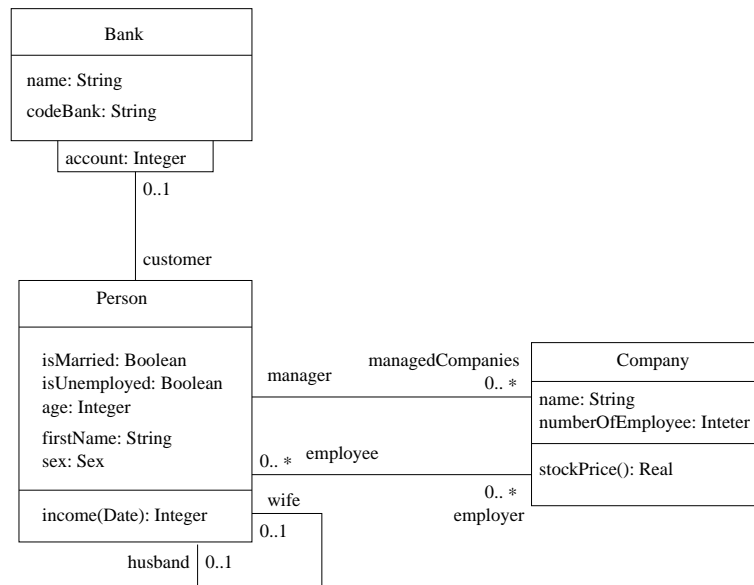


Figure 5.1 – Diagramme de classes d'un système de gestion

tel-00080852, version 1 - 21 Jun 2006

5.2 Dérivation des diagrammes statiques UML en B

La dérivation des spécifications UML en B a été présentée dans la section 3.1. Cependant, cette dérivation ne peut pas être utilisée pour vérifier les contraintes du modèle UML. En utilisant la technique de dérivation des méta-classes du méta-modèle en B, présentée dans la section 4.3, nous effectuons la dérivation vers B du diagramme de classes, du diagramme d'objets et des contraintes sur le modèle UML vers B.

5.2.1 Dérivation du diagramme de classes

Une classe décrit un ensemble d'objets ayant une structure et un comportement similaires. Chaque classe est dérivée en une machine abstraite B.

Dérivation 5.1 (Attribut)

Un attribut *attrib* est dérivé formellement en B par la déclaration d'une nouvelle variable *attrib* dans la machine abstraite associée à la classe de l'attribut. Cette variable est définie dans l'invariant par une fonction partielle entre l'ensemble des objets de la classe et son type *typeAttrib* :

$$attr_c \in CLASS \rightarrow typeAttr_c$$

où *CLASS* est l'ensemble des identifiants de l'objet de la classe, *typeAttr_c* est l'ensemble des valeurs des attributs. Cette dérivation est différente de la dérivation 3.2 (section 3.1) car *typeAttr_c* n'est pas le type de l'attribut dérivé en B. Les variables dérivées à partir des attributs sont initialisées par l'ensemble vide.

Dérivation 5.2 (AssociationEnd)

Les approches précédentes [Meyer 01, LeDang 02a] sont basées sur la dérivation des associations en B (dérivation 3.4). Notre approche prend en compte la dérivation des extrémités d'association (associationEnd) qui connectent l'association avec les classes. Par exemple, dans la Figure 5.1, les extrémités d'association sont: *employee*, *manager*, ...

Un extrémité d'association de la classe source *CLASS_s* à la classe cible *CLASS_t* est dérivée formellement en B comme une variable. Cette variable est définie dans l'invariant par une relation entre l'ensemble des identifiants d'objets de la classe source et ceux de la classe cible :

$$assoEnd \in CLASS_s \leftrightarrow CLASS_t$$

Les variables dérivées à partir des extrémités d'association sont également initialisées à l'ensemble vide.

La spécification présentée Figure 5.2 illustre la dérivation en B de la classe *Person* du diagramme de classes de la Figure 5.1. Les attributs (*isMarried*, *age*, ...) sont dérivés en variables (*person_isMarried*, *person_age*, ...), les extrémités d'association (*employer*, ...) sont dérivées en variables (*person_employer*, ...).

5.2.2 Dérivation des contraintes UML et OCL

Pour vérifier la spécification UML, nous devons vérifier deux catégories de contraintes : les contraintes d'association UML et les contraintes OCL.

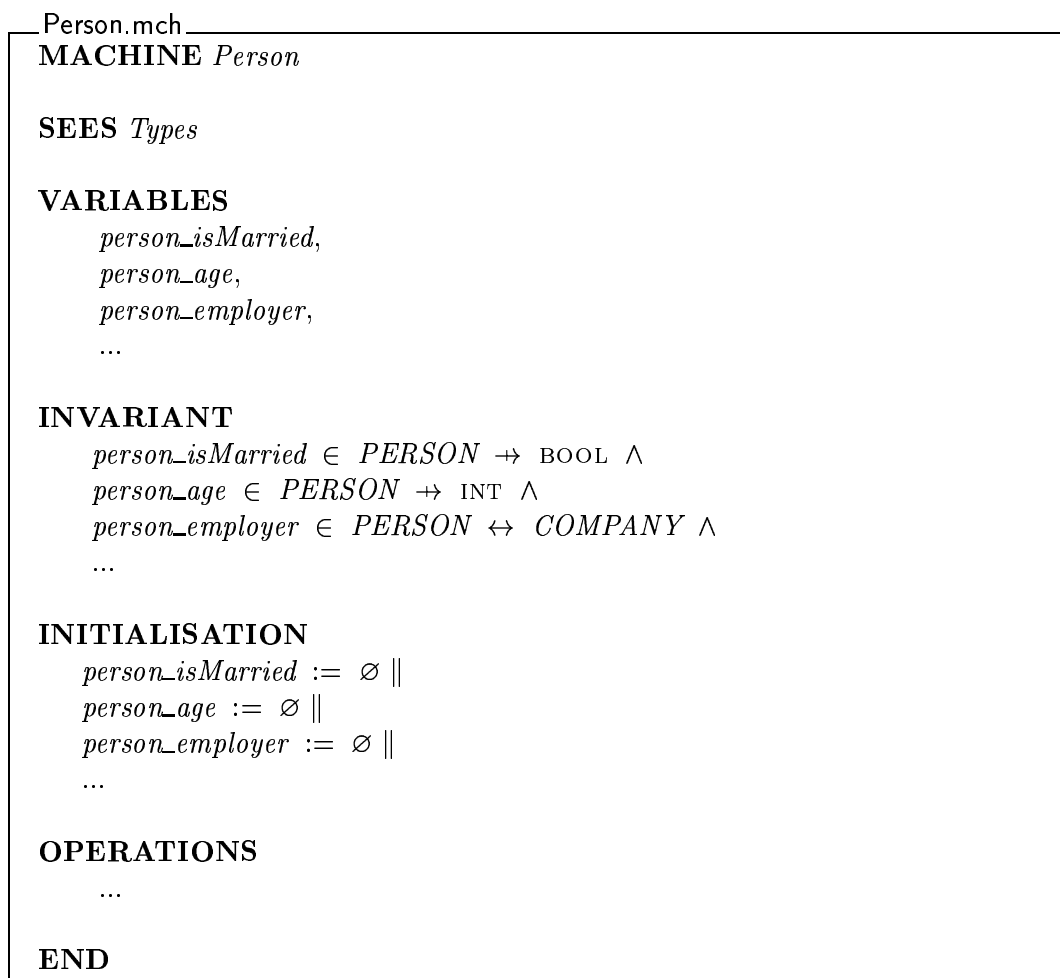


Figure 5.2 – *Machine abstraite B correspondant à la classe Person*

Contraintes d'association. Les contraintes d'association sont attachées aux associations des diagrammes de classes, plus précisément, elles sont liées aux extrémités d'association. Chaque extrémité d'association se compose du nom (rôle), de la visibilité et de la multiplicité qui correspond au nombre d'instances de la classe reliée à une autre classe. Les contraintes de multiplicité exprimées sont : 1, 0..1, n..m, *, 0..*, 1..*.

Les variables des machines abstraites B dérivées à partir des extrémités d'association sont utilisées pour exprimer ces contraintes, celles-ci sont présentées dans le Tableau 5.1.

Multiplicité d'UML	Invariant de B
n	$\text{CARD}(\text{assoEnd}) = n$
n..*	$\text{CARD}(\text{assoEnd}) \geq n$
n..m	$n \leq \text{CARD}(\text{assoEnd}) \leq m$

où $n \geq 0$, $m \geq 1$ et $n \leq m$.

Tableau 5.1 – Dérivation des contraintes d'association vers B

Contraintes OCL. Les contraintes OCL permettent de relier les attributs d'une classe ou de plusieurs classes.

Une contrainte OCL est un invariant d'une classe; elle doit être satisfaite pour tous les objets de la classe. Dans l'étude de cas, nous avons exprimé les exigences du système en OCL (section 5.1). Selon la définition de la dérivation des expressions OCL en B (section 3.1.4), ces exigences sont dérivées en B comme suit :

1. **context** Person

inv: *self.age* > 0

$$\forall xx.(xx \in \text{Person} \Rightarrow \text{person_age}(xx) > 0)$$

2. **context** Company

inv: *self.numberOfEmployee* > 5

$$\forall xx.(xx \in \text{Company} \Rightarrow \text{company_numberOfEmployee}(xx) > 5)$$

3. **context** Company

inv: *self.manager.age* > 40

$$\begin{aligned} \forall xx.(xx \in \text{COMPANY} \wedge \text{company_manager}[\{xx\}] \neq \emptyset \\ \Rightarrow \text{person_age}[\text{company_manager}[\{xx\}]] > 40) \end{aligned}$$

4. **context** Company

inv: *self.manager.isUnemployed* = false

$$\begin{aligned} \forall xx.(xx \in \text{COMPANY} \wedge \text{company_manager}[\{xx\}] \neq \emptyset \\ \Rightarrow \text{person_isUnemployed}[\text{company_manager}[\{xx\}]] = \text{FALSE}) \end{aligned}$$

inv: *self.employee.isUnemployed* = false

$$\begin{aligned} \forall xx.(xx \in \text{COMPANY} \wedge \text{company_employee}[\{xx\}] \neq \emptyset \\ \Rightarrow \text{person_isUnemployed}[\text{company_employee}[\{xx\}]] = \text{FALSE}) \end{aligned}$$

5.2.3 Dérivation du diagramme d'objets

Un diagramme d'objets est une instance d'un diagramme de classes, il a été décrit dans la section 2.2.1.1; il se compose des objets avec les valeurs des attributs. La Figure 5.3 illustre un exemple d'un diagramme d'objets correspondant au diagramme de classes présenté dans la Figure 5.1. La dérivation des diagrammes d'objets est introduite comme suit :

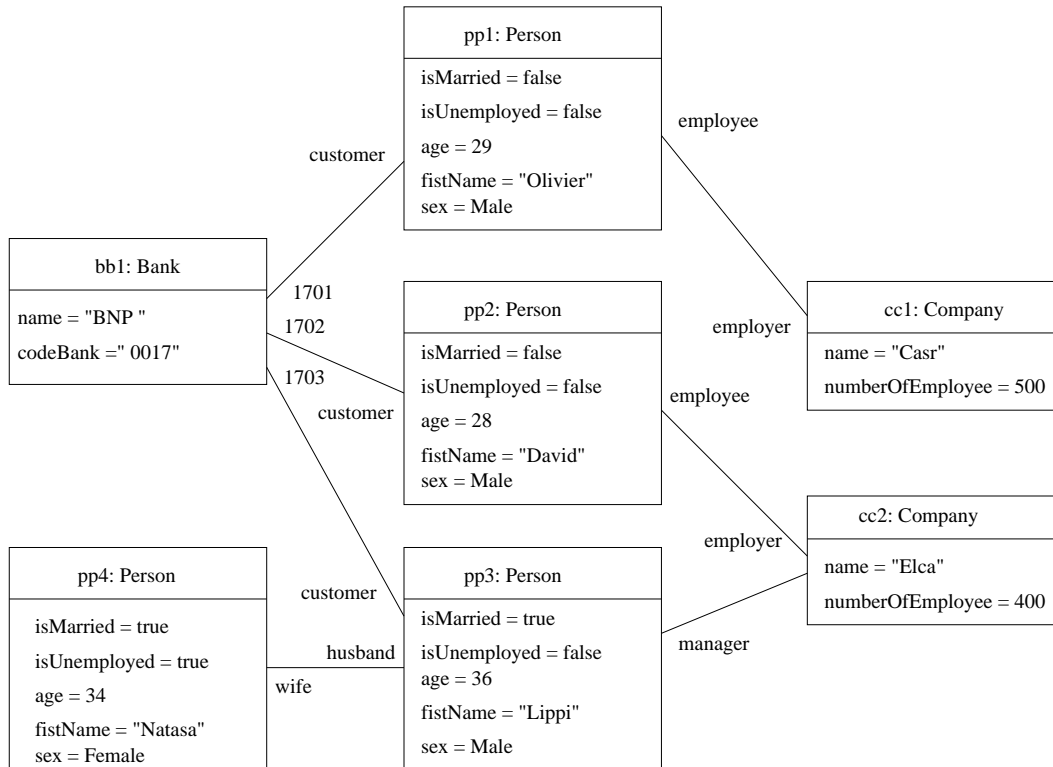


Figure 5.3 – Un exemple de diagramme d'objets du système de gestion

- Chaque objet est dérivé en une machine abstraite B, le nom de la machine est l'identifiant de l'objet.
- Les attributs des objets sont dérivés vers des variables dans la machine abstraite B. Le type des variables est une fonction partielle de l'ensemble des identifiants des objets vers le type des attributs:

$$attr_o \in CLASS \leftrightarrow typeAttr_o$$

Il est impératif que le type des variables des objets soit identique à celui des variables de leur classe. Cette définition permet de rassembler les valeurs des variables des machines des objets avec les variables des machines de leur classe.

- Chaque variable est initialisée dans la clause INITIALISATION par un couple identifiant de l'objet, la valeur des attributs des objets ($id \mapsto value$).
- Comme les attributs, chaque extrémité d'association est dérivée en une variable (dans le cas où l'extrémité d'association n'a pas de nom, nous utiliserons le nom de la classe cible). Le type des variables dérivées à partir des extrémités d'association est une relation entre la classe source et la classe cible ($assoEnd \in CLASS_s \leftrightarrow CLASS_t$). Sa valeur est initialisée par un couple identifiant de l'objet source, identifiant de l'objet cible ($idO_s \mapsto idO_t$).

En appliquant ces définitions de dérivation à l'objet pp1 de la classe Person, de la Figure 5.3, nous obtenons la machine B présentée Figure 5.4.

Remarque 5.1 Afin d'éviter le conflit de noms dans la dérivation B, nous préfixons le nom des variables dérivées à partir des attributs et des extrémités d'association des objets par le nom de

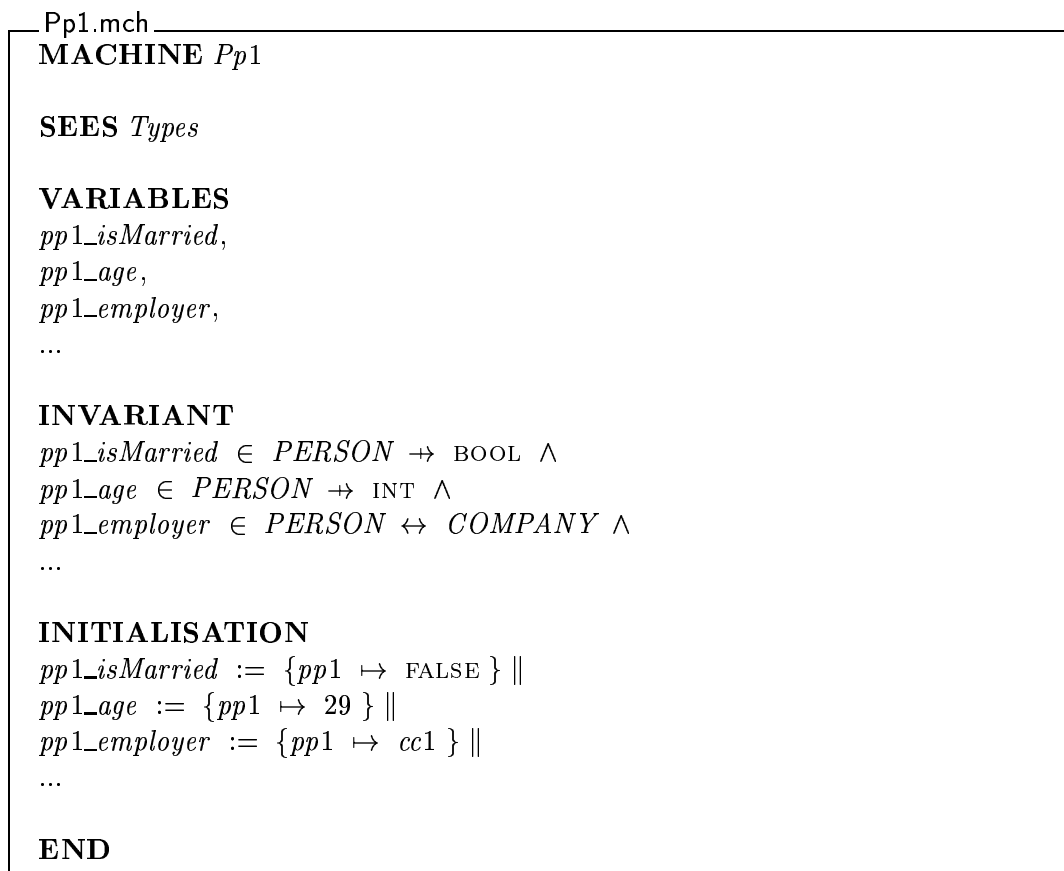


Figure 5.4 – Machine abstraite B de l'objet pp1

ces objets, nous préfixons le nom des variables dérivées à partir des attributs et des extrémités d'association des classes par le nom de ces classes. Par exemple, l'attribut `age` de l'objet `pp1` est dérivé en la variable `pp1_age` (voir Figure 5.4), l'attribut `age` de la classe `Person` est dérivé en la variable `person_age` (Figure 5.2).

5.3 Vérification des contraintes du système par combinaison des dérivations

La méthode B fournit des mécanismes de structuration qui permettent à des machines d'être exprimées comme des combinaisons de machines auxiliaires. Dans la section 5.2, nous avons présenté la dérivation des diagrammes de classes, des diagrammes d'objets et des contraintes OCL vers B. Dans cette section, nous utilisons les relations de composition entre machines abstraites B pour obtenir un système et utilisons l'outil de preuve (AtelierB) pour vérifier les contraintes UML et OCL.

5.3.1 Vérification des contraintes simples

Pour importer les variables des machines des objets dans les machines de leur classe, nous utilisons la clause `INCLUDES`. Les variables des machines abstraites des classes sont utilisées pour exprimer les invariants dérivés à partir des contraintes d'association et des contraintes exprimées en OCL. Les invariants sont des prédicats qui contiennent des quantificateurs universels ou existentiels pour restreindre les propriétés de tous les objets. Cependant, lorsqu'on prouve ces invariants avec les prouveurs de la méthode B, on ne peut pas obtenir le résultat attendu parce que les valeurs des objets ne sont pas affectées aux variables dans les obligations de preuve.

Pour incorporer les valeurs des propriétés des objets aux variables des invariants, nous ajoutons une opération supplémentaire `collectData` dans les machines de classes; sa forme est présentée dans la Figure 4.2, dans laquelle les variables additionnelles ($attr_1, attr_2, \dots, attr_n$) sont des variables de la machine de classe.

Selon cette proposition, nous construisons l'opération `collectData` de la machine `Person`, introduite dans la Figure 5.5.

La preuve des contraintes UML et OCL est effectuée par le prouveur des outils supports de B (AtelierB), ce processus est exécuté d'une manière similaire à la preuve des règles de bonne formation présentée dans le chapitre 4. Nous prenons un exemple avec l'invariant suivant :

$$\forall xx.(xx \in Person \Rightarrow person_age(xx) > 0)$$

L'ensemble `Person` est défini dans la machine `Types` par :

$$Person = \{pp1, pp2, pp3, pp4\}$$

A partir de l'opération `person_collectData` dans la spécification de la machine de classe `Person` (Figure 5.5), nous déduisons que la valeur de la variable dans l'invariant est:

$$person_age = \{pp1 \mapsto 29, pp2 \mapsto 28, pp3 \mapsto 36, pp4 \mapsto 34\}$$

En remplaçant l'ensemble `Person` et la variable `person_age` par leur valeur dans l'invariant, nous pouvons prouver facilement le prédicat.

D'une façon similaire, l'outil support de B permet de vérifier l'adéquation des valeurs des objets avec d'autres contraintes OCL du système.

```

Person.mch
MACHINE Person
...
EXTENDS Pp1, Pp2, Pp3, Pp4
...
OPERATIONS
  person_collectData =
  pre
    pp1_isMarried = { pp1 ↦ FALSE } ∧
    pp1_age = { pp1 ↦ 29 } ∧
    pp1_employer = { pp1 ↦ cc1 } ∧
    pp2_isMarried = { pp2 ↦ FALSE } ∧
    pp2_age = { pp2 ↦ 28 } ∧
    pp2_employer = { pp2 ↦ cc2 } ∧
    pp3_isMarried = { pp3 ↦ TRUE } ∧
    pp3_age = { pp3 ↦ 36 } ∧
    pp3_employer = { pp3 ↦ cc2 } ∧
    pp4_isMarried = { pp4 ↦ TRUE } ∧
    pp4_age = { pp4 ↦ 34 } ∧
    pp4_employer = ∅ ∧
    ...
  then
    person_isMarried := pp1_isMarried ∪ pp2_isMarried ∪
                        pp3_isMarried ∪ pp4_isMarried ||
    person_age := pp1_age ∪ pp2_age ∪ pp3_age ∪ pp4_age ||
    person_employer := pp1_employer ∪ pp2_employer ∪
                       pp3_employer ∪ pp4_employer ||
    ...
  end
END

```

Figure 5.5 – Spécification additionnelle pour la machine abstraite *Person*

```

Model.mch
MACHINE Model

SEES Types

INCLUDES Person, Company, Bank

INVARIANT
 $\forall xx.(xx \in COMPANY \wedge company\_manager[\{xx\}] \neq \emptyset$ 
 $\Rightarrow person\_age(company\_manager(xx)) > 40) \wedge$ 
 $\forall yy.(yy \in COMPANY \wedge company\_manager[\{yy\}] \neq \emptyset$ 
 $\Rightarrow person\_isUnemployed(company\_manager(yy)) = FALSE)$ 
...
OPERATIONS
  model_collectData =
    pre
    ...
    then
      company_collectData || person_collectData || bank_collectData
    end

END

```

Figure 5.6 – Machine abstraite Model

5.3.2 Vérification des contraintes combinées

Nous considérons ici les contraintes combinées OCL dans lesquelles les attributs d'une classe sont appelés par des objets d'autres classes à travers les extrémités d'association. Les exemples de ces contraintes sont illustrés par les contraintes (3) et (4).

Comme les contraintes simples, afin que des variables participant aux invariants dérivés à partir des contraintes combinées contiennent leurs valeurs lorsqu'on génère des obligations de preuve dans l'outil support, nous proposons de créer une nouvelle machine appelée *Model* qui a pour but d'incorporer les données des machines de classes dans un diagramme de classes.

Les contraintes combinées OCL sont dérivées en des invariants B. Une opération supplémentaire appelée *model_collectData* est ajoutée dans la clause OPERATIONS pour intégrer les données des objets aux invariants de la machine *Model*. Cette opération est formée en appelant les opérations, qui rassemblent les valeurs des objets, des machines de classes. Dans l'étude de cas présentée, ces opérations sont: *person_collectData*, *bank_collectData* et *company_collectData*. La machine *Model* du système de gestion des informations personnelles est introduite dans la Figure 5.6.

Structure générale des machines. La Figure 5.7 résume la structure générale d'une spécification B obtenue en combinant les dérivations séparées des diagrammes statiques d'UML afin de vérifier les contraintes UML et OCL. La machine *Model* inclut (la clause INCLUDES) les machines abstraites de classes, une machine abstraite de classe inclut les machines abstraites des objets. Toutes les machines dans le système voient (la clause SEES) la machine *Types* qui définit tous les

types du système.

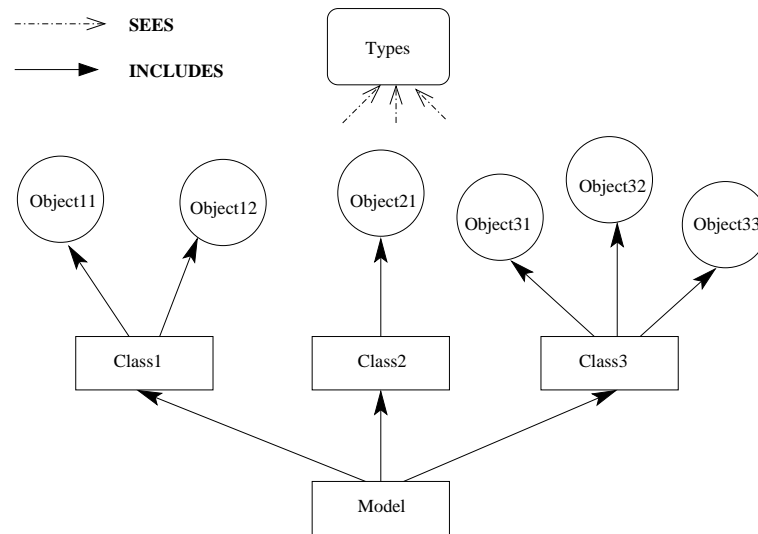


Figure 5.7 – Structure générale des machines B

Remarque 5.2 Pour importer les données des machines des objets dans les machines de leur classe, nous utilisons la clause `INCLUDES` qui est *transitive*. Ainsi les valeurs des variables des machines abstraites des objets sont automatiquement importées dans les machines abstraites de classes.

5.4 Synthèse

Dans ce chapitre, nous avons présenté une approche de dérivation de diagrammes statiques en B pour vérifier si un diagramme d'objets est une instance correcte d'un modèle UML. Nous utilisons la technique de vérification de la sémantique des éléments d'un modèle UML présentée dans le chapitre précédent, dans laquelle, les objets dans le diagramme d'objets jouent le rôle des objets des méta-classes. Les classes sont transformées en machines abstraites B avec prise en compte des valeurs des objets. Les valeurs des objets sont vérifiées par rapport aux contraintes simples dans les machines des classes et aux contraintes combinées OCL dans la machine `Model` qui inclue toutes les machines des classes.

Vérification des scénarios UML en utilisant B

SOMMAIRE

6.1	Dérivation d'un scénario en B	92
6.2	Vérification de spécifications UML à l'aide des obligations de preuve de B	94
6.2.1	Obligations de preuve dans une machine abstraite	94
6.2.2	Obligations de preuve dans une implantation (ou raffinement)	94
6.3	Étude de cas	95
6.3.1	Description du système	95
6.3.2	Modélisation des scénarios du système à l'aide de diagrammes UML et de contraintes OCL	95
6.3.3	Vérification des spécifications UML et OCL des scénarios	96
6.4	Synthèse	98

Les scénarios sont des processus importants dans le développement de logiciels, ils sont associés à des vues statiques et à des vues dynamiques dans la spécification UML. Un scénario est une instance d'un cas d'utilisation. Il décrit un exemple d'interaction possible entre le système et les acteurs. Dans ce chapitre, nous présentons la vérification des diagrammes dynamiques présentés sous la forme d'un scénario. Un scénario peut être exprimé en UML par un diagramme de classes, un ensemble d'expressions OCL et un diagramme d'interaction (diagramme de collaboration ou diagramme de séquences). Les scénarios exprimés à l'aide de diagrammes de séquences seront étudiés dans le chapitre 8. La transformation de diagrammes de collaboration en B a été proposée par Ledang (voir section 3.1.3), avec quelques limites. Nous proposons une solution à ces limites et nous abordons le problème de la vérification des propriétés des scénarios. Les propriétés vérifiées sont :

- les invariants de classes,
- les contraintes de pré- et postconditions des opérations par rapport aux invariants dans une classe et une implantation (via les diagrammes d'interaction),
- les contraintes de pré- et postconditions des opérations séquentielles,
- les contraintes de pré- et postconditions des opérations appelées par rapport aux contraintes de pré- et postconditions d'opérations appelantes (les propriétés de décomposition).

L'idée intuitive de transformation un scénario exprimé en UML/OCL en B est de disposer des opérations appelantes dans une machine et des opérations appelées dans une autre machine abstraite. La machine qui contient les opérations appelantes est raffinée, chaque opération raffinée

se définissent à partir de la décomposition introduite dans le diagramme de collaboration. La vérification des deux premières propriétés est effectuée par les obligations de preuve dans une machine abstraite, la vérification des deux dernières propriétés est effectuée par les obligations de preuve des machines de raffinement et d'implantation.

La structure de ce chapitre est la suivante. D'abord, nous décrivons la dérivation d'un scénario UML en B. L'application de cette dérivation pour la validation de spécifications UML à l'aide des obligations de preuve de B est ensuite présentée. Une étude de cas, un système de passage à niveau illustre notre approche. La présentation de ce chapitre est une extension du papier [Truong 04b].

6.1 Dérivation d'un scénario en B

Un scénario peut être décrit comme le déroulement d'une instance d'un diagramme de cas d'utilisation, il est raffiné à l'aide de diagrammes de collaboration. Le scénario raffiné est complété par un diagramme de classes qui spécifie sa structure statique. Les diagrammes de classes spécifient les attributs et la signature des opérations tandis que les diagrammes de collaboration modélisent la décomposition et l'ordre d'exécution de ces opérations. Au sein de chaque classe, des contraintes OCL portant sur toutes les instances de classificateurs peuvent être ajoutées. Chaque opération est spécifiée par des pré- et postconditions définissant un contrat que l'implantation doit satisfaire. Dans l'exécution d'un scénario, nous devons nous assurer que *les préconditions de l'opération de l'objet récepteur sont satisfaites par les postconditions de l'opération de l'objet expéditeur*.

La dérivation d'un scénario en B est en partie présentée dans le travail de dérivation des diagrammes de réalisation en B par Ledang (voir section 3.1.3). Dans ces travaux, les relations appelante-appelées contenant des paires récursives et des dépendances circulaires entre les opérations ne sont pas prises en compte afin d'éviter la boucle infinie de la procédure de division (voir remarque 3.2). C'est le cas du message 1.1.1.1 présenté dans la Figure 6.1. Si l'opération `op1.1.1` du message 1.1.1 appelle l'opération `op1` du message 1.1.1.1, une dépendance circulaire est établie entre les messages $1.1 \rightarrow 1.1.1 \rightarrow 1.1.1.1 \rightarrow 1.1$ et donc on ne peut pas distribuer les opérations attachées à ces messages en différentes couches proposées par l'approche de dérivation. Ou bien, si le message 1.2 (`Class3`) appelle l'opération `op1.1` qui appartient à la `Class2`, on ne sait pas à quelle couche l'opération `op1.1` appartient, parce que avec le message 1.1, elle appartient à la couche 2 et avec le message 1.2.1, elle appartient à la couche 3. Ce type de dépendance entre messages apparaît souvent dans les diagrammes de collaboration UML et la programmation orientée objet. De plus, cette dérivation ne permet pas la vérification des propriétés dans la spécification UML et OCL. Nous proposons une amélioration à cette dérivation pour résoudre ces limites.

Procédure de dérivation.

- Chaque classe du modèle UML est dérivée en une machine abstraite B.
- Deux machines abstraites *System.mch* et *Basic.mch* sont créées et les données de chaque machine sont toutes les données des machines dérivées à partir des classes dans le scénario.
- Les opérations qui appellent une ou plusieurs opérations dans le scénario (les opérations décomposées) appartiennent à la machine *System.mch* (les opérations `op1`, `op1.1`, `op1.1.1`, etc).

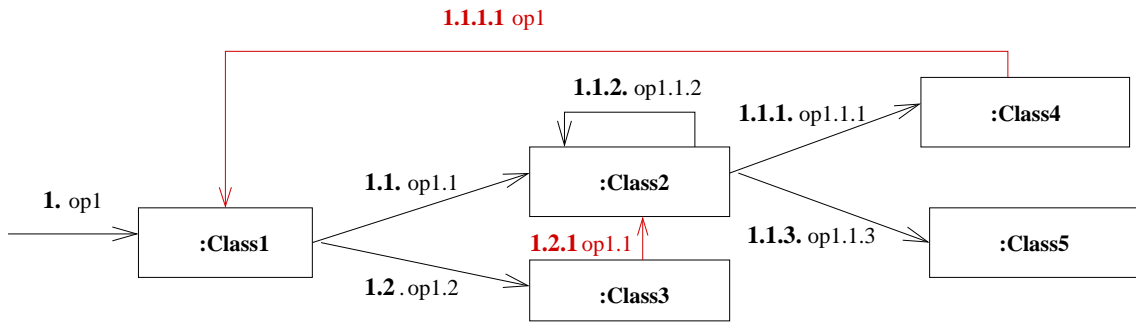


Figure 6.1 – Diagramme de collaboration avec les cas de dépendance circulaire

- Les opérations appelées dans le scénario (les opérations composées) appartiennent à la machine *Basic.mch* (op1.1, op1.2, op1.1.1, etc).
- L'implantation *System.imp* raffine (REFINE) la machine abstraite *System.mch* et importe (IMPORT) la machine *Basic.mch* (voir figure 6.2).
- Le contenu d'une opération de machine abstraite B est dérivé à partir de la spécification OCL de l'opération UML correspondante (voir section 3.1.4).
- Le contenu des opérations dans la machine d'implantation B est dérivé à partir des messages dans le scénario.
- L'invariant de chaque machine *Basic.mch* et *System.mch* est associé aux invariants des classes dans le modèle UML et aux invariants OCL de ces classes.

Toutes les machines introduites dans la dérivation (Figure 6.2) voient (la clause SEES) la machine *Types.mch*.

Remarque 6.1 Une spécification B ne peut pas contenir plusieurs opérations ayant le même nom. C'est le cas des opérations *op1*, *op1.1*, *op1.1.1* qui apparaissent à la fois dans la machine *System* et dans la machine *Basic*. Nous proposons de renommer toutes les opérations de la machine *System.mch* (exemple: *op1_sys*, *op1.1_sys*, *op1.1.1_sys*).

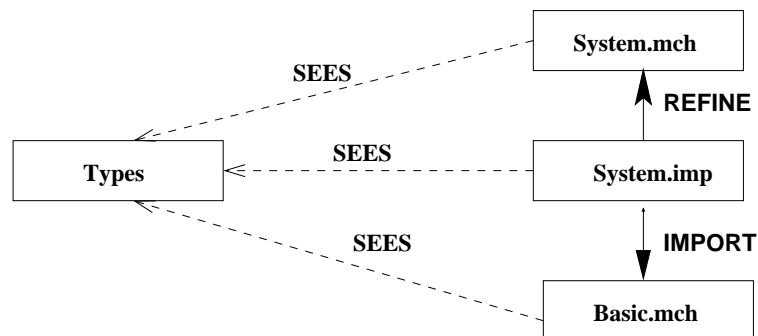


Figure 6.2 – Structure des machines dérivées

Il est nécessaire de mettre les opérations des différentes classes dans la machine *System* et *Basic*, ces opérations dans une classe utilisent les données des autres classes. Mais chaque classe est transformée en une machine abstraite et ces machines abstraites ne peuvent pas accéder aux propriétés des autres machines, il faut donc regrouper les données et les opérations dans une même machine.

6.2 Vérification de spécifications UML à l'aide des obligations de preuve de B

L'approche proposée est intéressante car elle permet de décomposer la vérification en s'intéressant aux différents scénarios du système de manière indépendante plutôt qu'au système vu de manière globale. Dans cette section nous considérons ce que le prouveur de B peut prouver et vérifier dans un scénario à l'aide de l'analyse des obligations de preuve générées.

6.2.1 Obligations de preuve dans une machine abstraite

Une machine abstraite B et ses obligations de preuve s'expriment comme présenté dans la section 1.2.2.1. Les opérations d'une classe UML sont dérivées formellement en des opérations d'une machine abstraite B, les invariants OCL sont transformés en invariants B des machines abstraites. Ces obligations de preuve garantissent la correction des invariants par rapports aux opérations des classes UML. L'utilisation de ces obligations de preuve pour prouver les spécifications UML est abordée dans les travaux de [Marcano 02, Hazem 04].

6.2.2 Obligations de preuve dans une implantation (ou raffinement)

Le mécanisme de raffinement de la méthode B a été présenté dans la section 1.2.2.2. Chaque opération de l'implantation est une nouvelle version (concrète) d'une opération précédemment spécifiée. Les entêtes des deux opérations sont identiques, seule la substitution généralisée définissant l'effet de l'opération est modifiée. *Une opération d'une implantation est correcte lorsqu'elle préserve l'invariant sans contredire l'opération qui raffine.*

Comme pour les opérations des raffinements, le but à prouver est basé sur l'invariant de liaison de l'implantation en conjonction avec un prédicat d'égalité entre les variables de sortie de l'opération de l'implantation et les variables de sortie renommées de l'opération abstraite.

Pour une opération de contenu T de la machine de raffinement (ou implantation) qui raffine l'opération **pre P then S end**, les obligations de preuve s'écrivent :

$$P \wedge I \wedge J \Rightarrow [T] \neg [S] \neg (J)$$

où I est l'invariant de la machine abstraite, J est l'invariant de la machine d'implantation, *u* est la liste des paramètres résultats.

Les obligations de preuve de l'implantation sont utilisées pour prouver l'intégrité entre les pré- et postconditions des opérations séquentielles et entre les opérations décomposées et composées (appelantes/appelées). Une machine d'implantation qui importe certaines machines abstraites va appeler les opérations de ces machines. Les opérations appelées vont s'exécuter séquentiellement. Nous utilisons les axiomes suivants [Abrial 96b] pour décrire le séquençement entre deux opérations :

$$\begin{aligned} (P \mid S); T &\Leftrightarrow P \mid (S; T); (1) \\ S; (P \mid T) &\Leftrightarrow [S]P \mid (S; T); (2) \end{aligned}$$

A partir de (1) et (2), la séquence entre deux opérations s'exprime :

$$op1(); op2() = (S \mid P); (T \mid Q) \Leftrightarrow S \mid (P; (T \mid Q)) \Leftrightarrow S \mid ([P]T \mid (P; Q))$$

Cela signifie que le prouveur va prouver la consistance de toutes les pré- et postconditions des opérations appelées pendant l'exécution du système. *La postcondition de la première opération doit vérifier la précondition de l'opération suivante ([P]T).*

Les obligations de preuves générées par l'implantation d'une opération vérifient la pré- et la postcondition des opérations décomposées (T) qui correspondent à la pré- et postcondition d'une opération raffinée (P, S) par :

$$P \wedge I \wedge J \Rightarrow [T] \neg [S] \neg (J)$$

Soit u la liste des variables résultats des postconditions de l'opération décomposée (T), u' la liste des variables résultats des postconditions des opérations raffinées (P, S), E, E' sont des expressions. L'obligation de preuve de l'implantation, avec un invariant de liaison vide se réécrit :

$$[u' := E'] \neg [u := E] \neg (u' = u) \sim [u' := E'](u' = E) \sim E = E'$$

Les obligations de preuve de l'implantation, dans le cas où l'invariant de liaison est vide, sont correctes si et seulement si $E = E'$, c'est-à-dire si les valeurs des variables de la machine abstraite sont équivalentes à celles de la machine d'implantation. Appliquons cette obligation de preuve à la procédure de dérivation présentée dans la section 6.1. Nous prouvons que *les valeurs des variables de l'opération appelante sont identiques aux valeurs des variables dans l'exécution de ses opérations appelées.*

6.3 Étude de cas

Dans cette section, nous donnons un exemple, un système de passage à niveau, pour illustrer la preuve de propriétés dans les scénarios. Nous modélisons les scénarios du système par les diagrammes UML et puis nous les dérivons en B. La consistance entre les opérations séquentielles et les opérations décomposées est prouvée par les obligations de preuve du mécanisme de raffinement de B.

6.3.1 Description du système

Un système de passage à niveau simplifié (contrôle de trains) se compose d'un contrôleur, d'un feu de signalisation et d'une barrière. A l'état normal, le feu est vert, la barrière est ouverte. Lorsque le train arrive, le contrôleur donne une commande pour arrêter la circulation, le feu passe à l'orange puis au rouge et la barrière se ferme. Quand le train est passé, le contrôleur donne une commande pour remettre en route la circulation. La barrière s'ouvre et le feu passe au vert.

6.3.2 Modélisation des scénarios du système à l'aide de diagrammes UML et de contraintes OCL

La Figure 6.3 présente le diagramme de classes du modèle UML de ce système. La Figure 6.4 présente le diagramme de collaboration du scénario correspondant à la fermeture de la barrière. L'opération `Control::close` (l'opération `close` appartient à la classe `Control`) se décompose en trois opérations : `Light::lightYellow`, `Light::lightRed` et `Barrier::close`. Soit les opérations OCL de ce scénario :

Context `Control::close():void`
pre:

Context `Light::lightYellow():void`
pre:

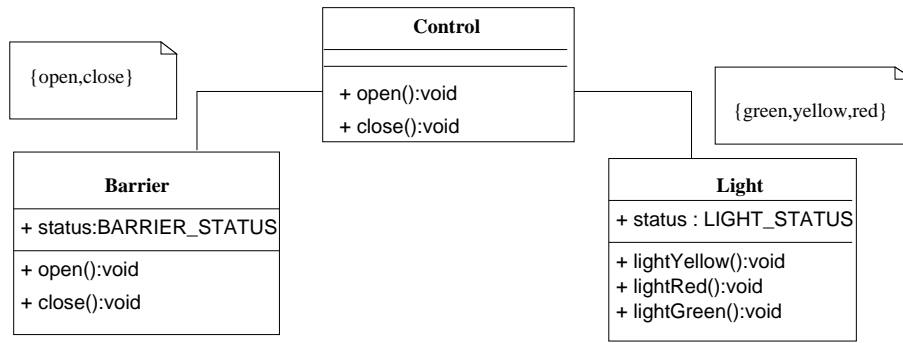


Figure 6.3 – Diagramme de classes du système de passage à niveau

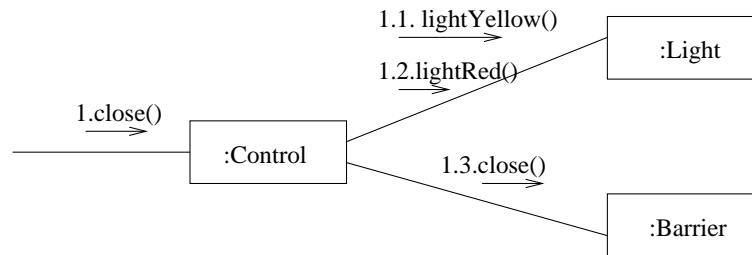


Figure 6.4 – Diagramme de collaboration du scénario de fermeture de la barrière

<i>light.status = green</i>	<i>self.status = green</i>
post:	post:
<i>light.status = red</i> and	<i>self.status = yellow</i>
<i>barrier.status = close</i>	

Context Light::lightRed():void

pre:
self.status = green
 post:
self.status = red

Context Barrier::close():void

pre:
self.status = open
 post:
self.status = close

On va analyser la correction des pré- et postconditions des opérations OCL du scénario présentées dans la Figure 6.4.

6.3.3 Vérification des spécifications UML et OCL des scénarios

Le diagramme de classes UML et les opérations OCL sont dérivées en B selon la procédure de dérivation présentée dans la section 6.1, avec renommage des propriétés UML (opération, attribut, ...) en les préfixant par le nom de la classe. Par exemple, l'attribut *status* de la classe *Light* devient la variable *light.status*. La spécification B de ce système est donnée en Annexe, chapitre C. L'Atelier B donne les résultats de la preuve de toutes les machines. Elles sont correctes, sauf pour la machine d'implantation *System_imp.imp*, qui présente deux "unproved". Le prouveur interactif nous donne l'obligation de preuve correspondant au premier "unproved" (extrait de l'outil AtelierB) :

co: control\$1 &

```

light_status$(controlLight$(co)) = light_green &
"Check preconditions of called operation, or While loop
construction, or Assert predicates" &
=>
(light_status$(co) <+ {controlLight$(co) | -> light_yellow})
(controlLight$(co)) = light_green

```

Cette erreur est introduite par l'exécution séquentielle entre les opérations composées dans la spécification. L'opération *lightLightYellow* est suivie de l'opération *lightLightRed*. La précondition de l'opération *lightLightRed* est *self.status = green* et la postcondition de l'opération *lightLightYellow* est *self.status = yellow*, elles sont incohérentes. Donc, la spécification OCL de l'opération *lightRed* de la classe *Light* doit être revue :

```

Context Light::lightRed():void
pre:
    self.status = yellow
post:
    self.status = red

```

Le prouveur interactif nous donne l'obligation de preuve correspondant au deuxième "unproved" comme suit :

```

co: control$(co) &
light_status$(controlLight$(co)) = light_green &
"Check preconditions of called operation, or While loop
construction, or Assert predicates" &
=>
barrier_status$(controlBarrier$(co)) = barrier_open

```

L'obligation de preuve appartient aux formules impliquées de la forme $P \Rightarrow Q$, avec

$$P = (\text{light_status}(\text{controlLight}(co)) = \text{light_green}),$$

$$Q = (\text{barrier_status}(\text{controlBarrier}(co)) = \text{barrier_open}).$$

On ne peut pas utiliser les hypothèses courantes pour prouver le but Q à partir des machines importées dans la spécification B, le but Q correspond à la précondition de l'opération *barrierClose* dans la spécification OCL (*barrier_status = open*). Nous voyons qu'il n'y a pas de relation entre les éléments de ces deux formules. Pour prouver cette obligation de preuve, on doit ajouter donc une hypothèse dans les hypothèses dérivées, c'est l'hypothèse de but prouvé. Dans la spécification OCL, on doit rajouter l'hypothèse (*barrier.status = open*) à la précondition de l'opération *Control::close*. Donc la spécification OCL de l'opération *close* de la classe *Control* est modifiée comme suit :

```

Context Control::close():void
pre:
    light.status = green and
    barrier.status = open
post:
    light.status = red and
    barrier.status = close

```

Cet exemple montre l'application du prouveur de la méthode B pour analyser les pré- et post-conditions des opérations décomposées et celles des opérations séquentielles.

6.4 Synthèse

Dans ce chapitre, nous avons présenté une façon de vérifier un scénario de spécification UML/OCL en le dérivant en B. Le scénario est exprimé par un diagramme de classes, un diagramme de collaboration et des expressions OCL. La dérivation de diagrammes de classes en B permet de construire la structure des machines B; la dérivation des expressions OCL en B fournit le contenu des opérations (les transformations des expressions OCL en B peuvent être remplacées par les transformations de diagrammes d'état-transitions en B) et la dérivation des diagrammes de collaboration en B établit l'interaction entre les opérations dans la spécification B. Avec cette technique de dérivation, nous pouvons construire un système B contenant moins de composants et résolvant les limites de la transformation de diagrammes de réalisation en B proposée par Ledang [LeDang 02a]. La structure de la spécification B et les obligations de preuves générées par l'outil de preuves permettent de valider et de vérifier les propriétés dans la spécification UML.

Nous avons utilisé l'approche de dérivation et de validation sur un seul scénario UML. Cette approche peut être généralisée à plusieurs scénarios de diagrammes de collaboration en l'appliquant simultanément à chacun des scénarios. La vérification s'effectuant entre les pré- et postconditions d'opérations séquentielles, la vérification globale est donc immédiate.

TROISIÈME PARTIE :

Prise en compte de l'objet dans le développement formel B

Dans la partie précédente, nous avons présenté des approches de vérification de spécifications UML en utilisant B grâce à ses outils de preuve. B est un langage de modélisation intéressante utilisant les notations mathématiques dans lequel les machines abstraites jouent le rôle de classes dans les notations orientées objets. Les machines abstraites peuvent servir à instancier des objets donc les notations B peuvent être utilisées comme des notations orientées objets (object-based notations) [Malioukov 98]. Les clauses d'assemblage en B correspondent au paradigme "un écrivain/plusieurs lecteurs", ce qui ne permet pas de prendre en compte les différents types de relations des approches objets. Dans cette partie, nous proposons de prendre en compte les types d'association entre classes des approches objets pour les machines abstraites B.

Les diagrammes de séquences d'UML exprimant un scénario dans les approches orientées objets, dans lesquels l'aspect de communication est prédominant, sont une base pour la validation de spécifications. Nous proposons d'utiliser B et ses outils de preuve pour valider les spécifications. Un outil support permettant de spécifier des machines abstraites et de prouver des obligations de preuve a été également implanté.

Prise en compte de certains types d'association des approches objets pour B

SOMMAIRE

7.1	Les relations de composition entre machines abstraites B	102
7.2	Les types d'associations des approches orientées objets	102
7.2.1	Association bidirectionnelle	102
7.2.2	Association unidirectionnelle	103
7.3	Prise en compte des différents types d'associations UML en B	103
7.3.1	Spécification	104
7.3.2	Obligations de preuve	104
7.4	Interaction avec d'autres machines abstraites dans le modèle	106
7.5	Synthèse	108

La méthode B fournit un support pour la modularité, la réutilisation de modules et la décomposition de la preuve. Rappelons que la preuve d'une machine abstraite B assure la préservation de son invariant par les substitutions de l'initialisation et des opérations. Dans la dérivation systématique d'UML en B, on utilise une machine abstraite B pour décrire une classe. Cependant, les approches objets visent à simuler les interactions entre objets et les types d'association entre classes (voir section 2.2.1.2) sont fournis afin de modéliser ces interactions. A l'heure actuelle, les travaux sur la transformation d'UML en B ne prennent pas en compte l'expression de certains types d'association tels que les associations binaires parce que B n'autorise pas l'accès réciproque entre deux machines. Dans ce chapitre, nous proposons de prendre en compte l'expression en B de ces types d'association. L'idée principale est de :

- rassembler les machines reliées entre elles par un de ces types d'association dans un groupe,
- définir des obligations de preuve, permettant d'assurer la préservation des invariants des machines dans le groupe par des substitutions de leurs initialisations et de leurs opérations.
- utiliser les obligations de preuve des clauses de relation de B entre une machine B et le groupe.

La structure de ce chapitre est la suivante. D'abord, nous analysons les relations entre machines abstraites B et les relations entre classes dans les approches objets par une présentation des mécanismes de composition entre les machines abstraites et des types d'associations des approches orientées objets considérés. Ensuite, nous rassemblons les machines abstraites dérivées à partir des classes reliées par certains types d'association que B ne les supporte pas et proposons

des obligations de preuve pour ces machines. Enfin, nous considérons le cas de réutilisation des obligations de preuve des clauses de relation entre machines B pour les autres machines dans le modèle.

7.1 Les relations de composition entre machines abstraites B

Un des points forts de B est la composition de spécifications et de leur développement via des mécanismes d'assemblage. Le principe consiste à composer en même temps les preuves d'invariants et les preuves de raffinements. Ceci favorise la modularité et minimise l'activité de preuve. La modularisation des preuves repose sur un partitionnement des variables en suivant le principe "un écrivain/plusieurs lecteurs". Plusieurs mécanismes d'assemblage sont proposés avec des restrictions justifiant la correction de l'approche sous-jacente à la méthode B [Potet 03], ces restrictions étant inhérentes aux langages basés sur la notion d'état. Les mécanismes d'assemblage entre machines donnent certains accès aux objets et aux traitements d'une autre machine. Ils garantissent que les invariants des machines prouvées restent préservés pour la nouvelle spécification. Une présentation détaillée de ces clauses est donnée dans la section 1.2.2.3; on peut les résumer comme suit :

- la clause INCLUDES permet d'appeler toute opération et d'énoncer des invariants sur les variables de la machine incluse,
- la clause SEES permet uniquement l'appel d'opérations de consultation et les invariants ne peuvent pas porter sur les variables vues,
- la clause USES permet d'accéder en lecture seulement à des variables d'une autre machine.

Il en résulte que les clauses d'assemblage proposées par la méthode B correspondent à des relations unidirectionnelles.

7.2 Les types d'associations des approches orientées objets

Dans un diagramme de classes, très peu de classes sont décrites de manière isolée et la plupart collabore entre elles de différentes manières, via les dépendances ou relations d'utilisation, les généralisations et les associations décrivant des relations structurelles qui indiquent les interactions entre les objets des classes concernées. Comme décrit dans la section 2.2.1.2, UML propose cinq types d'associations. Nous nous focalisons sur l'expression en B des associations bidirectionnelles et les cas particulière des associations unidirectionnelles. Les autres types d'associations, tels que la composition et l'agrégation, peuvent être exprimées par les clauses de composition B.

7.2.1 Association bidirectionnelle

Une association bidirectionnelle définit les rôles réciproques de deux classes mises en relation. Elle est indiquée dans la littérature à l'aide d'un seul trait plein. Figure 7.1 montre qu'un objet de la classe `Flight` est associé à un objet de la classe `Plane` par une association bidirectionnelle. La classe `Flight` connaît cette association, elle joue le rôle *assignedFlight*. Un objet de la classe `Plane` connaît son association avec la classe `Flight`, dont le nom de rôle est *assignedPlane*.

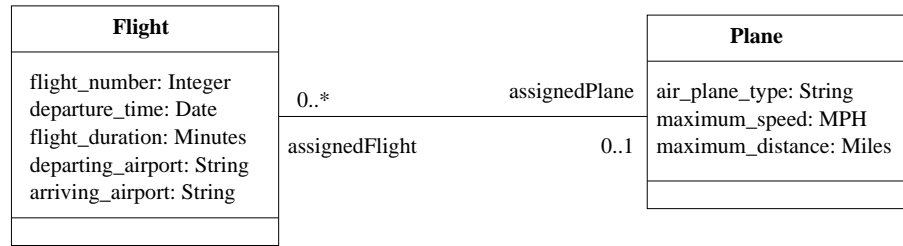


Figure 7.1 – Exemple de relation bidirectionnelle

7.2.2 Association unidirectionnelle

Dans une association unidirectionnelle, seule une des classes liée a accès aux l'objet de l'autre classe. Autrement dit, c'est une association qui possède un seul sens de lecture, elle n'autorise que la navigation d'un objet d'une classe vers un objet d'une autre classe. Dans le diagramme présenté Figure 7.2, un objet de la classe *Student* est associé à un objet de la classes *Course*, la flèche indiquant le sens de la navigation. L'association définit le rôle *assistTo* joué par un objet de la classe *Course* pour un objet de la classe *Student*.

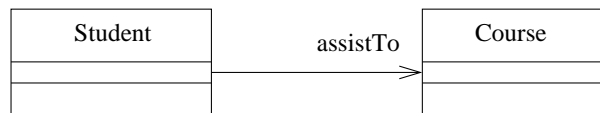


Figure 7.2 – Exemple de relation unidirectionnelle

Les relations entre machines B représentent une partie des relations entre les classes UML. Certaines relations entre classes UML, que nous appellerons *relations spéciales* dans la suite, ne peuvent pas être spécifiées par les relations entre machines abstraites B. Il s'agit :

- des associations bidirectionnelles ou des associations n-aires,
- deux associations unidirectionnelles de sens opposés,
- des relations entre classes UML constituant un cycle : $A \rightarrow B, B \rightarrow C, C \rightarrow A$, etc.

Pour résoudre ce problème, les approches de transformation d'UML vers B proposent une machine de données (*Types.mch*) contenant des données communes, cette machine étant vue (SEES) par toutes les autres machines dans le système. Cependant, cette machine ne peut pas partager des variables et des opérations. Certaines approches ont été proposées introduisant des machines qui partagent des variables et des opérations communes [Buchi 99]. Celles-ci vont l'encontre des aspects de modularité des spécifications orientées objets (voir section 2.1).

7.3 Prise en compte des différents types d'associations UML en B

Les cas de deux machines abstraites générées à partir de deux classes UML reliées par une association binaire ou par deux associations unidirectionnelles de sens opposés sont traités de manière similaire. Nous parlerons dans la suite de ces cas sous le nom *relations bidirectionnelles*.

Exprimer des relations bidirectionnelles entre deux machines abstraites M et N , signifie que la machine M peut accéder aux données de la machine N et réciproquement. Pour ce faire, nous proposons :

- d'utiliser la structure des machines abstraites dérivées de classes UML comme présenté dans la Figure 3.1,
- de supprimer les clauses d'assemblage entre les machines abstraites reliées par une relation bidirectionnelle dans les spécifications B correspondantes,
- d'établir les nouvelles obligations de preuve permettant d'exprimer la réciprocité entre les machines concernées.

7.3.1 Spécification

La forme générale des machines B en relation bidirectionnelle est présentée figure 7.3. Elle correspond à une machine abstraite B dans laquelle les clauses d'assemblage ont été supprimées. Dans une telle machine,

- la clause SETS est utilisée pour déclarer l'ensemble *OBJECTS* et les autres ensembles de la classe,
- la clause CONSTANTS déclare les constantes B correspondant à l'ensemble des objets possibles de la classe,
- la clause PROPERTIES définit les propriétés de ses constantes,
- la clause VARIABLES déclare les attributs de la classe. Le type de chaque attribut est donné dans la clause INVARIANT,
- la clause INVARIANT décrit l'invariant global qui doit être préservé par l'initialisation et par chacune des opérations,
- la clause INITIALISATION initialise les variables avec l'ensemble vide.

Il est à noter que dans notre travail, nous utilisons la structure d'une machine abstraite B avec une utilisation différente de ses constituants. Nous autorisons la symétrie des clauses de composition de B (USES, SEES, etc.). Une machine M en relation bidirectionnelle avec une machine N peut utiliser et modifier les données de cette machine et réciproquement. Ce processus est généralisable à plusieurs machines, correspondant au cas des relations n-aire.

7.3.2 Obligations de preuve

L'expression des relations bidirectionnelles entre machines consiste à :

- étendre la forme de partage autorisée entre les machines B concernées par ces relations,
- lier les invariants des variables de ces machines.

Ceci doit être accompagné de l'adjonction de nouvelles obligations de preuve permettant de garantir les invariants de ces machines. Une relation bidirectionnelle entre deux machines M et N est vue comme une inclusion dans les deux sens.

Obligations de preuve d'une machine abstraite. Soit la machine abstraite *Class* avec ses constituants présentée figure 7.3. Les obligations de preuve de cette machine sont relatives à son invariant comme celles des machines B présentées dans la section 1.2.2.1. Elles s'expriment de la manière suivante :

- en ce qui concerne l'initialisation : $T \wedge Prop \Rightarrow [U]I$

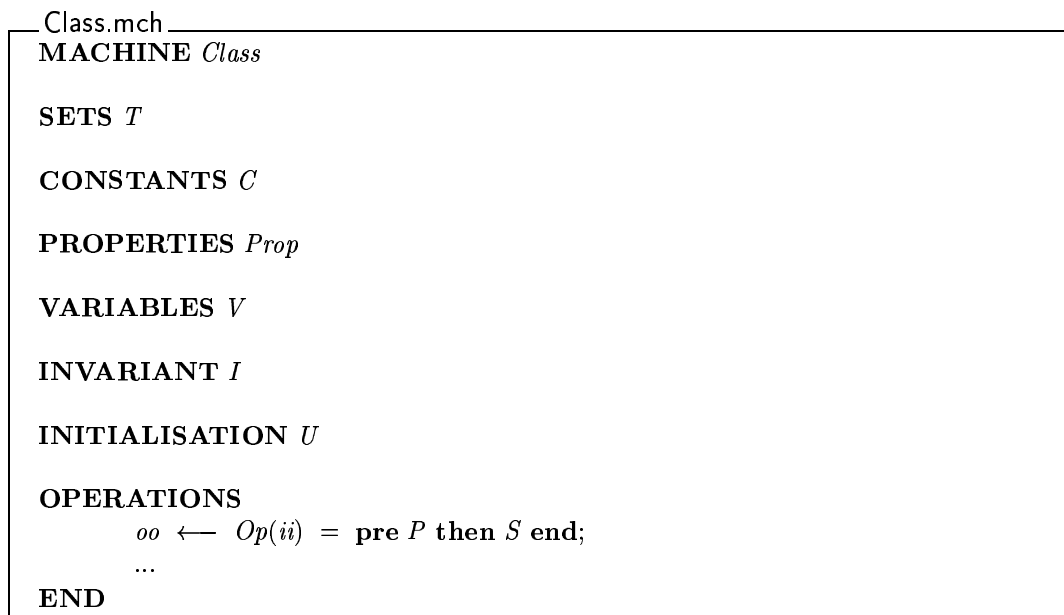


Figure 7.3 – Spécification générale d'une machine abstraite B dérivée à partir d'une classe UML

- pour chaque opération : $T \wedge Prop \wedge I \wedge P \Rightarrow [S]I$

T correspond aux contraintes sur les ensembles de la machine abstraite.

Obligations de preuve de la clause INCLUDES. Les obligations de preuve correspondant à l'inclusion d'une machine M dans la machine N s'expriment de la manière suivante :

- en ce qui concerne l'initialisation :

$$T_M \wedge T_N \wedge Prop_M \wedge Prop_N \Rightarrow [U_M; U_N]I_N$$

Dans notre approche orientée objets, chaque machine correspond à une classe : les variables de la machine M sont disjointes de celles de la machine N . Il en résulte que U_M est indépendante de U_N et de I_N . Par conséquent, la formule précédente peut être simplifiée en :

$$T_M \wedge T_N \wedge Prop_M \wedge Prop_N \Rightarrow [U_N]I_N$$

- pour chaque opération de la machine N :

$$T_M \wedge T_N \wedge Prop_M \wedge Prop_N \wedge I_M \wedge I_N \wedge P_N \Rightarrow [S_N]I_N$$

Obligations de preuve correspondant à une relation bidirectionnelle entre deux machines M et N . Comme dit précédemment, une relation réciproque entre deux machines M et N signifie que la machine M inclut la machine N et la machine N inclut la machine M . En conséquence, les opérations et les initialisations des machines M et N doivent préserver :

- l'invariant de la machine M et
- l'invariant de la machine N .

Donc la *conjonction des invariants* des machines M et N , $I_M \wedge I_N$, doit être préservée. Les obligations de preuve correspondant à une relation bidirectionnelle entre deux machines M et N

s'expriment de la manière suivante :

- en ce qui concerne l'initialisation :

$$T_M \wedge T_N \wedge Prop_M \wedge Prop_N \Rightarrow [U_M]I_M \quad (\text{car } U_M \text{ est indépendante de } I_N)$$

$$T_M \wedge T_N \wedge Prop_M \wedge Prop_N \Rightarrow [U_N]I_N$$

- pour chaque opération des machines M et N :

$$T_M \wedge T_N \wedge Prop_M \wedge Prop_N \wedge I_M \wedge I_N \wedge P_M \Rightarrow [S_M](I_N \wedge I_M)$$

$$T_M \wedge T_N \wedge Prop_M \wedge Prop_N \wedge I_M \wedge I_N \wedge P_N \Rightarrow [S_N](I_N \wedge I_M)$$

Généralisation des obligations de preuve à une relation n -aire. La relation n -aire que nous abordons ici correspond au cas où n machines abstraites qui représentent les classes reliées par une association n -aire ou par des relations circulaire.

Supposons que ce groupe est composé de n machines abstraites, chacune d'elle étant indicée par i ($i = 1..n$). Soit T la conjonction des contraintes sur les ensembles, $Prop$ la conjonction des propriétés spécifiées dans la clause PROPERTIES et I la conjonction des invariants des n machines considérées :

$$T = \bigwedge T_i, Prop = \bigwedge Prop_i, I = \bigwedge I_i$$

Les obligations de preuve dans ce groupe sont définies comme suit :

- l'initialisation U_i de chaque machine abstraite i préserve son invariant I_i :

$$T \wedge Prop \Rightarrow [U_i]I_i$$

- chaque opération OP_{ik} de la machine abstraite i préserve la conjonction des invariants I :

$$T \wedge Prop \wedge I \wedge P_{ik} \Rightarrow [S_{ik}]I$$

Pour reconnaître les machines abstraites dans le modèle B dérivées à partir de classes reliées par des relations spéciales, nous proposons d'ajouter une clause "**GROUP** *NameOfGroup*" dans la structure de ces machines abstraites. Les machines dans un groupe portent le même nom de groupe.

La Figure 7.4 représente une partie de la spécification des deux machines abstraites Flight.mch et Plane.mch qui correspondent aux classes de la Figure 7.1.

Remarque 7.1

- Les données des machines abstraites sont disjointes donc nous ne considérons pas les problèmes de substitutions parallèles d'une même donnée dans les obligations de preuve.
- Les machines abstraites définies à partir de relations spéciales ne sont pas prouvées séparément mais elles sont prouvées ensemble selon les obligations de preuve proposées.

7.4 Interaction avec d'autres machines abstraites dans le modèle

Dans la section 7.3, nous avons présenté la prise en compte dans B de relations spéciales dans un groupe de classes UML. Dans la réalité, un système est spécifié par un modèle orienté objets avec plusieurs classes. Nous pouvons utiliser les obligations de preuve proposées dans le cas d'une relation n -aire pour prouver la préservation des invariants de toutes les classes (machines) dans un modèle. Cependant, la charge de preuve dans le moteur de preuve est grande car le nombre

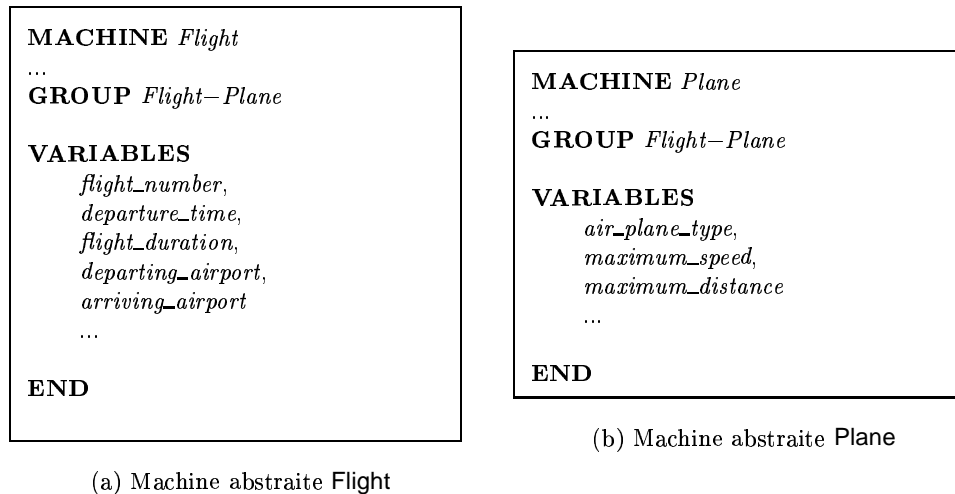


Figure 7.4 – Deux machines abstraites correspondant à deux classes reliées par une association bidirectionnelle

des obligations de preuve générées est grande. Pour diminuer la charge de preuve, nous proposons de réutiliser les relations de composition entre machines abstraites B avec les associations unidirectionnelles.

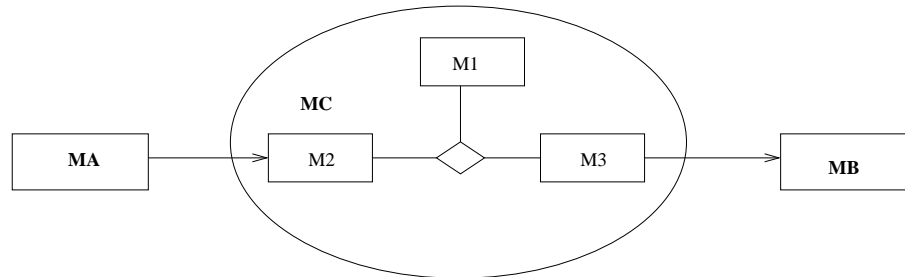


Figure 7.5 – Interaction avec d'autres machines dans le modèle

Pour ce faire, nous considérons les spécifications des machines abstraites dans ce groupe comme la spécification d'une machine abstraite B, cela est la base pour établir les obligations de preuve. Supposons que MC est le nom du groupe des machines abstraites M_i dans ce groupe.

$$T_{MC} = \bigwedge T_i, Prop_{MC} = \bigwedge Prop_i, I_{MC} = \bigwedge I_i$$

- Si une classe MA utilise les données d'autre(s) classe(s) dans le groupe MC (Figure 7.5), la machine MA inclut (INCLUDES) d'autre(s) machine(s) dans ce groupe. Les obligations de preuve sont établies :

- pour l'initialisation de la machine MA (U_{MA}) :

$$T_{MA} \wedge T_{MC} \wedge Prop_{MA} \wedge Prop_{MC} \Rightarrow [U_{MA}]I_{MA}$$

- pour chaque opération de la machine MA , $OP_{MA} = P_{MA} \mid S_{MA}$:

$$T_{MA} \wedge T_{MC} \wedge Prop_{MA} \wedge Prop_{MC} \wedge I_{MA} \wedge I_{MC} \wedge P_{MA} \Rightarrow [S_{MA}]I_{MA}$$

- Si une classe M_i dans le groupe MC utilise les données de la classe MB située à l'extérieur du graphe MC (Figure 7.5), la machine M_i inclut machine MB . Les obligations de preuve de la machine M_i sont établies :

- pour l'initialisation de chaque machine M_i (U_{M_i}) dans le groupe MC :

$$T_{MC} \wedge T_{MB} \wedge Prop_{MC} \wedge Prop_{MB} \Rightarrow [U_{M_i}]I_{M_i}$$

- pour chaque opération $OP_i = P_{M_i} \mid S_{M_i}$ de chaque machine M_i dans le groupe MC :

$$T_{MC} \wedge T_{MB} \wedge Prop_{MC} \wedge Prop_{MB} \wedge I_{MC} \wedge I_{MB} \wedge P_{M_i} \Rightarrow [S_{M_i}]I_{MC}$$

Remarquons que dans ce cas, les obligations de preuve sont similaires à celles des machines dans un groupe constitué de relations spéciales avec adjonction des données de la machine MB dans les hypothèses des obligations de preuve.

La Figure 7.6 illustre un exemple du couplage entre les relations bidirectionnelles et les relations unidirectionnelles dans les modèles orientés objets. Dans cet exemple, les deux classes **Flight** et **Plane** sont reliées par une association bidirectionnelle; la classe **Plane** est un composite de la classe **Engine**.

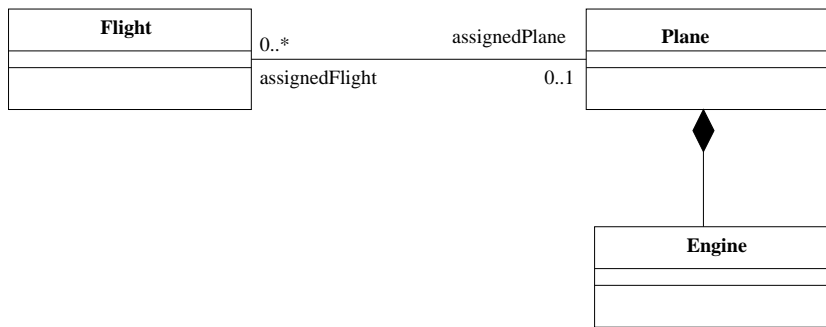


Figure 7.6 – Exemple de couplage entre relations bidirectionnelle et unidirectionnelle

Ces trois classes sont dérivées en trois machines abstraites B. Les machines **Flight** et **Plane** sont groupées dans le groupe **Flight_Plane**, la machine **Plane** inclut la machine **Engine** (voir Figure 7.7, Figure 7.8 et Figure 7.9).

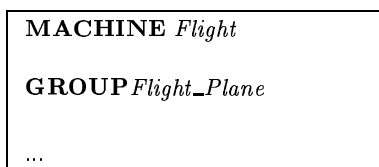


Figure 7.7 – Machine Flight

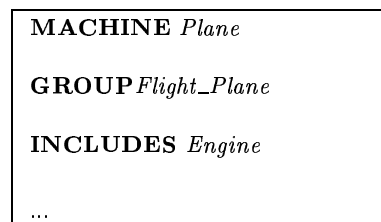


Figure 7.8 – Machine Plane

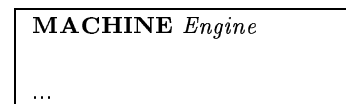


Figure 7.9 – Machine Engine

7.5 Synthèse

Les relations entre classes jouent un rôle important dans la modélisation orientée objet. Pour faciliter l'utilisation de B dans la modélisation orientée objet, ce chapitre propose une approche

permettant la prise en compte de certaines relations entre classes pour les machines abstraites. Nous rassemblons les machines abstraites générées à partir des classes UML reliées par les types d'association spéciales et étendons la forme de partage autorisée entre ces machines. Pour cela, de nouvelles obligations de preuve ont été définies, permettant de garantir les invariants des machines concernées. Avec les autres classes (machines) situées à l'extérieur, reliées avec un ou plusieurs classes dans le groupe par les associations unidirectionnelles, nous utilisons les obligations de preuve des relations de composition B pour prouver la préservation des invariants par les substitutions de chaque machine dans le groupe ainsi que la machine à l'extérieur.

Validation de spécifications orientées objets en B

SOMMAIRE

8.1	Structure de la machine de simulation	112
8.2	Obligations de preuve préservant l'exécution d'un scénario	113
8.2.1	Obligations de preuve associées à un scénario défini par une séquence d'appels d'opérations	113
8.2.2	Obligations de preuve associées à un scénario incluant une conditionnelle	114
8.3	Expression de propriétés dynamiques et obligations de preuve	115
8.3.1	Propriété de sûreté	115
8.3.2	Propriété de vivacité	115
8.4	Étude de cas	116
8.4.1	Présentation de l'étude de cas	116
8.4.2	Spécification UML	117
8.4.3	Spécification B	118
8.4.4	Validation du scénario Entry_Building	120
8.5	Synthèse	122

Dans ce chapitre, nous proposons une approche pour la validation de spécifications orientées objets en utilisant les notations B. L'idée fondamentale est de simuler les scénarios exprimés par des diagrammes de séquences en utilisant le prouveur B.

Un scénario peut être modélisé par des diagrammes de collaboration (chapitre 6) ou des diagrammes de séquences. Nous ne nous intéressons pas ici à la décomposition des messages qui est souvent exprimée dans les diagrammes de collaboration. En raison de cela, nous choisissons les diagrammes de séquences pour décrire les scénarios. Une autre raison est que les diagrammes de séquences, dans lesquels l'aspect de communication est prédominant, sont une base intéressante pour la description de tests [Pickin 04].

Le point de départ est une spécification UML constituée d'un diagramme de classes et de diagrammes de séquences qui expriment des scénarios modélisant le comportement du système. Ces diagrammes sont transformés en spécifications B permettant de tester :

- l'exécution d'une séquence d'opérations dans un scénario exprimé par des diagrammes de séquence,
- l'exécution de scénarios avec les propriétés dynamiques du système.

Pour atteindre ce but, nous proposons de :

- dériver une spécification B à partir d'un diagramme de classes UML,
- introduire une nouvelle machine B, appelée *machine de simulation*, afin de spécifier des scénarios de test comme une séquence d'appels des opérations,
- intégrer les contraintes dynamiques du système dans cette nouvelle machine.
- utiliser le prouveur B (AtelierB) pour prouver les obligations de preuve proposées

La structure de ce chapitre est comme suit. Nous commençons par la présentation de la structure de la machine de simulation. Puis, nous proposons les obligations de preuve associées pour préserver l'exécution des opérations dans un scénario et pour des contraintes dynamiques. L'exemple du système de contrôle d'accès illustre cette approche. La présentation de ce chapitre est l'extension du papier [Truong 06a].

8.1 Structure de la machine de simulation

L'idée fondamentale est de simuler les diagrammes de séquence UML 2.0 qui décrivent l'interaction entre les messages. Ces diagrammes peuvent inclure des gardes : quand on modélise des interactions d'objets, des conditions doivent être vérifiées pour qu'un message soit envoyé à l'objet. Les gardes ne sont pas suffisantes pour exprimer toutes les écoulements conditionnels des messages. Nous prenons en compte les fragments combinés proposés par UML 2.0, pour grouper des ensembles de messages, à savoir l'alternative et le choix d'option. Les alternatives sont employées pour indiquer un choix mutuellement exclusif entre deux ou plusieurs séquences de messages, modélisant la clause "*if then else*". Le fragment combiné choix d'option est utilisé pour modéliser l'exécution d'une séquence avec certaines conditions. Il correspond à la modélisation du "*if then*".

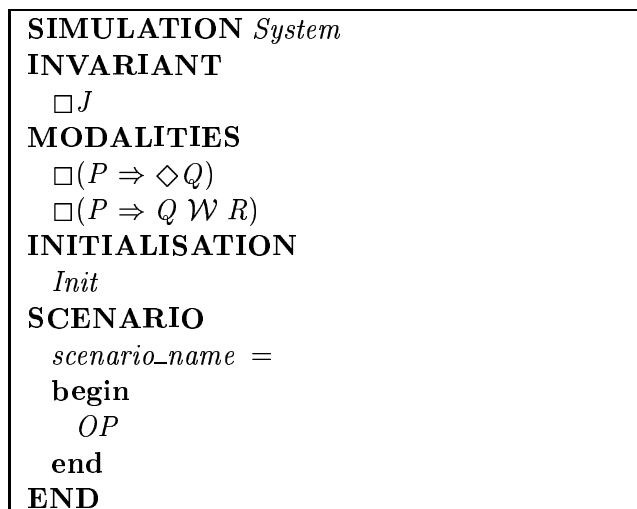


Figure 8.1 – *Structure de la machine de simulation*

Des scénarios décrits par des diagrammes de séquence d'UML 2.0 sont transformés en une machine de simulation, comme présenté Figure 8.1. Chaque message dans un système orienté objet

est spécifié par une opération dans une machine abstraite B. Cette machine se compose de quatre clauses :

- la clause INVARIANT exprime les propriétés de sûreté du système,
- la clause MODALITIES exprime les propriétés dynamiques, présentée dans la section 8.3,
- la clause INITIALISATION permet d'initialiser des états du système pour le scénario donné,
- la clause SCENARIO contient la définition d'un scénario. Il correspond à la transformation des diagrammes de séquence d'UML 2.0, composé d'une séquence des appels d'opération, y compris des notations de garde, d'alternative et choix d'option.

8.2 Obligations de preuve préservant l'exécution d'un scénario

Cette section propose des obligations de preuve permettant d'assurer la correction du scénario modélisé. Nous commençons à établir des obligations de preuve pour les opérations séquentielles. Nous pouvons réutiliser ces obligations de preuve dans le cas de la séquence des opérations contenant des notations de garde, d'alternative et choix d'option.

8.2.1 Obligations de preuve associées à un scénario défini par une séquence d'appels d'opérations

Supposons que la simulation est spécifiée par une séquence de n appels d'opérations, chacune est indexée par i , chaque opération est composée d'une précondition et un corps de substitution :

$$\begin{aligned} OP &= OP_1; OP_2; \dots; OP_n \\ OP_i &= P_i \mid S_i \end{aligned}$$

donc

$$OP = [P_1 \mid S_1]; [P_2 \mid S_2]; \dots; [P_n \mid S_n]$$

Afin d'effectuer la simulation, nous remplaçons les paramètres formels dans les opérations appelées par leurs valeurs effectives. La définition de l'opération OP_i est exprimée par :

$$r_i \longleftarrow OP_i(\text{para}_1, \text{para}_2, \dots, \text{para}_m)$$

et son appel est de la forme :

$$v_i \longleftarrow OP_i(\text{value}_1, \text{value}_2, \dots, \text{value}_m)$$

Pour chaque opération appelée OP_i , en utilisant la sémantique de la substitution, nous devons montrer que les valeurs effectives de ses paramètres satisfont sa précondition P_i :

$$P_{iv} = [\text{para}_1, \text{para}_2, \text{para}_m := \text{value}_1, \text{value}_2, \dots, \text{value}_m] P_i$$

Après avoir remplacé chaque paramètre dans le corps de l'opération par sa valeur, nous obtenons :

$$S_{iv} = [r_i, \text{para}_1, \text{para}_2, \dots, \text{para}_m := v_i, \text{value}_1, \text{value}_2, \dots, \text{value}_m] S_i$$

Soit $[S_v^i]$ l'exécution des substitutions dans le corps des i premières opérations du scénario après remplacement de chaque paramètre de chaque opération par sa valeur. En prenant en compte l'initialisation de la machine de simulation, nous obtenons :

$$[S_v^i] = [Init][S_{1v}][S_{2v}] \dots [S_{iv}]$$

Un scénario de la machine de simulation appelle n opérations définies dans des machines abstraites. Soit A la conjonction des contraintes des ensembles, $Prop$ la conjonction des propriétés et I la conjonction des invariants de toutes les k machines abstraites du système :

$$A = \bigwedge A_j, Prop = \bigwedge Prop_j, I = \bigwedge I_j \quad \text{avec } j \in [1..k]$$

Le scénario exprimé par une séquence d'appels d'opérations est validé si les obligations de preuve suivantes sont vérifiées :

- Pour $i = 0 .. n-1$ ($i = 0$ correspond à l'initialisation).

Les obligations de preuve garantissent que l'exécution du système établit l'invariant I de toutes les machines abstraites définissant le système. Pour chaque opération appelée OP_i , nous devons vérifier que la précondition de l'appel de la $(i+1)^{ime}$ opération OP_{i+1} est satisfaite par la postcondition obtenue par l'exécution des premiers i appels d'opérations du scénario :

$$A \wedge Prop \wedge I \wedge P_{iv} \Rightarrow [S_v^i](P_{(i+1)v} \wedge I) \quad (1)$$

- Pour $i = n$.

Les obligations de preuve garantissent que l'exécution du scénario préserve l'invariant I de toutes les machines abstraites :

$$A \wedge Prop \wedge I \wedge P_{nv} \Rightarrow [S_v^n]I \quad (2)$$

8.2.2 Obligations de preuve associées à un scénario incluant une conditionnelle

Considérons le cas d'un scénario spécifié par une séquence d'appels d'opérations incluant une conditionnelle. La forme générale de la conditionnelle est exprimée par :

if P then Q else R end

Cette construction est présentée dans B pour l'interaction entre le choix et la garde. En fournissant une règle de précondition la plus faible pour cette construction, il y a deux cas à considérer. Afin de s'assurer que T soit vrai après son exécution, si P est vrai alors Q doit établir T , et si P est faux, alors R doit établir T . Ceci résulte dans la règle suivante:

$$[\text{if } P \text{ then } Q \text{ else } R \text{ end}]T = (P \Rightarrow [Q]T) \sqcap (\neg P \Rightarrow [R]T)$$

Considérons le cas où Q et R correspondent à une liste d'opérations :

$$\begin{aligned} Q &= OP_{q1}; \dots; OP_{qq} \\ R &= OP_{r1}; \dots; OP_{rr} \end{aligned}$$

Le scénario comprenant la clause IF est exprimé par :

$$\begin{aligned} &OP_1; OP_2; \dots; OP_i; \\ &\text{if } P \text{ then } OP_{q1}; \dots; OP_{qq} \text{ else } OP_{r1}; \dots; OP_{rr} \text{ end;} \\ &OP_{i+1}; \dots; OP_n \end{aligned}$$

Deux cas doivent être considérés :

- si $[S_1][S_2] \dots [S_i](P) = true$, le chemin d'exécution du scénario est :

$$OP_1; OP_2; \dots; OP_i; OP_{q1}; \dots; OP_{qq}; OP_{i+1}; \dots; OP_n$$

- si $[S_1][S_2] \dots [S_i](P) = false$, le chemin d'exécution du scénario est :

$$OP_1; OP_2; \dots OP_i; OP_{r_1}; \dots; OP_{r_r}; OP_{i+1}; \dots; OP_n$$

Pour chaque cas de la décomposition du scénario, nous obtenons une séquence d'appels des opérations nous ramenant aux obligations de preuve d'un scénario défini par une séquence d'appels d'opération, tel que défini dans la section 8.2.1.

Remarque 8.1 La forme générale de la condition étudiée ci-dessus correspond à la construction alt de UML 2.0. Deux autres constructions, la garde et l'option *opt* correspondent à une forme simplifiée exprimée par “if P then Q ” :

- dans le cas de la garde, Q correspond à un message simple,
- dans le cas de l'option, Q correspond à une séquence de messages.

8.3 Expression de propriétés dynamiques et obligations de preuve

Nous présentons les propriétés dynamiques qui doivent être satisfaites par la validation d'un scénario. Ces propriétés sont exprimées dans la machine de simulation par :

- les propriétés de sûreté dans la clause INVARIANT et
- les propriétés de vivacité dans la clause MODALITIES. Nous introduisons les propriétés de réponse et de précedence [Manna 91], qui ont été introduites dans les notations orientées objets [Dietrich 00, Distefano 00, Sykes 01].

8.3.1 Propriété de sûreté

Une *propriété de sûreté* se rapporte à une formule P et exige que P soit un invariant. Dans la logique temporelle, une telle propriété est exprimée par $\square P$.

L'obligation de preuve suivante garantit que la validation du scénario établit l'invariant J de la machine de simulation.

$$\forall i.(i \in [0..n] \Rightarrow A \wedge Prop \wedge I \wedge P_{iv} \Rightarrow [S_v^i]J) \quad (3)$$

où n est le nombre d'opérations d'exécution dans le scénario.

8.3.2 Propriété de vivacité

Nous considérons deux types de propriété de vivacité, la propriété de réponse et la propriété de précedence [Dietrich 00].

8.3.2.1 Propriété de réponse

Une *propriété de réponse* se rapporte à deux formules P et Q et exige que tous les P -state (les états qui satisfont P) surgissant dans une exécution soient fatalement suivis par un Q -state. Dans la logique temporelle, ceci s'écrit $\square(P \Rightarrow \diamond Q)$.

Pour vérifier cette propriété, les deux obligations de preuve suivantes doivent être établies :

$$\exists i.(i \in [0..n-1] \wedge A \wedge Prop \wedge I \wedge P_{iv} \Rightarrow [S_v^i]P) \quad (4)$$

Cette première obligation de preuve (4) vérifie si P peut être établi par l'exécution du scénario. Si cette obligation de preuve est satisfaite, cela signifie qu'il existe une opération OP_i ($i \in [1..n-1]$) dans le scénario qui aboutit à l'état s_i où la propriété P est établie. Nous devons alors vérifier la deuxième obligation de preuve (5) :

$$\exists j.(j \in [i+1..n] \wedge A \wedge Prop \wedge I \wedge P_{jv} \Rightarrow [S_v^j]Q) \quad (5)$$

Cette obligation de preuve est vérifiée si le prédicat Q est satisfait par un état s_j qui suit l'état s_i pendant l'exécution ($j > i$).

8.3.2.2 Propriété de précédence

Une *propriété de précédence* se rapporte à trois formules P , Q et R . Elle exige que tous les états qui satisfont P (P -state) soient suivis par une séquence dans laquelle Q est satisfaite et que cette séquence se termine soit par les états satisfaisant R , soit par les états satisfaisant Q .

Dans la logique temporelle, cette propriété est exprimée par $\Box(P \Rightarrow Q \mathcal{W} R)$.

Nous devons d'abord vérifier si le prédicat P est établi par l'exécution du scénario :

$$\exists i.(i \in [0..n-1] \wedge A \wedge Prop \wedge I \wedge P_{iv} \Rightarrow [S_v^i]P) \quad (6)$$

Si le prédicat P est satisfait par l'état s_i , ($i < n$), on vérifie s'il existe un appel d'opération OP_j ($j \in [i+1..n]$) dans le scénario qui conduit à un état s_j où R est établi :

$$\exists j.(j \in [i+1..n] \wedge A \wedge Prop \wedge I \wedge P_{jv} \Rightarrow [S_v^j]R) \quad (7)$$

- Si le prédicat R n'est pas établi, nous devons prouver que chaque appel d'opération OP_j ($j \in [i+1..n]$) dans l'exécution établit le prédicat Q :

$$\forall j.(j \in [i+1..n] \Rightarrow A \wedge Prop \wedge I \wedge P_{jv} \Rightarrow [S_v^j]Q) \quad (8)$$

- Si le prédicat R est établi par l'état s_j , nous devons prouver que le prédicat Q n'est pas établi pour cet état :

$$A \wedge Prop \wedge I \wedge P_{jv} \Rightarrow [S_v^j](\neg Q) \quad (9)$$

8.4 Étude de cas

Nous illustrons notre approche par une étude de cas simplifiée, un système de contrôle d'accès qui gère les entrées/sorties des personnes dans un bâtiment.

8.4.1 Présentation de l'étude de cas

Le contrôle a lieu sur la base de l'autorisation d'accès par des personnes identifiées à un bâtiment donné. Chaque personne reçoit une carte magnétique avec un code d'identification unique, gravé sur la carte elle-même. Un lecteur de cartes est installé à l'entrée (et à la sortie) du bâtiment. Une personne souhaitant entrer dans le bâtiment suit une procédure systématique composée de la séquence d'opérations suivante. Lorsqu'elle met sa carte dans le lecteur et entre son code, deux cas se présentent :

- si elle est autorisée, son entrée est acceptée :
 - la porte s'ouvre,

- la carte est éjectée par le lecteur,
- la personne prend sa carte,
- la personne entre dans le bâtiment et
- la porte se ferme;
- si elle n'est pas autorisée, son entrée est refusée :
 - la porte reste fermée,
 - la carte est éjectée par le lecteur et
 - la personne prend sa carte.

8.4.2 Spécification UML

Nous présentons d'abord un diagramme de classes d'UML pour structurer ce système. Ensuite nous spécifions le comportement attendu du système proposé à l'aide d'un diagramme de séquences. Puis nous précisons les propriétés du système.

8.4.2.1 Diagramme de classes

Les cartes sont les seules informations connues du contrôleur. Par conséquent, nous décidons d'assimiler personne et carte dans la modélisation, en introduisant la classe `Card`.

L'opération `insertCard` dans la classe `Reader` inclut l'insertion de la carte dans le lecteur et l'entrée du code par la personne.

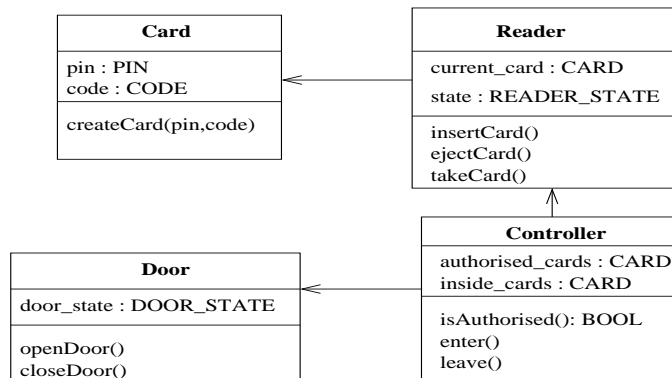


Figure 8.2 – Diagramme de classes du système de contrôle d'accès

Les autorisations sont représentées dans la classe `Controller` par un ensemble appelé `authorised_cards`. La situation dynamique des personnes dans le bâtiment est représentée par l'ensemble `inside_cards`.

8.4.2.2 Diagramme de séquences

La Figure 8.3 exprime le scénario d'entrée d'une personne dans un bâtiment présenté dans la section 8.4.1 par un diagramme de séquences d'UML 2.0 (voir la section 2.2.1.4).

Dans le diagramme de séquences, le fragment combiné `alt` correspond aux deux cas :

- la personne (i.e. la carte) est autorisée à entrer,
- la personne n'est pas autorisée à entrer.

Remarque 8.2 Une personne est autorisée à entrer si elle est connue par le système et qu'elle n'est pas dans le bâtiment.

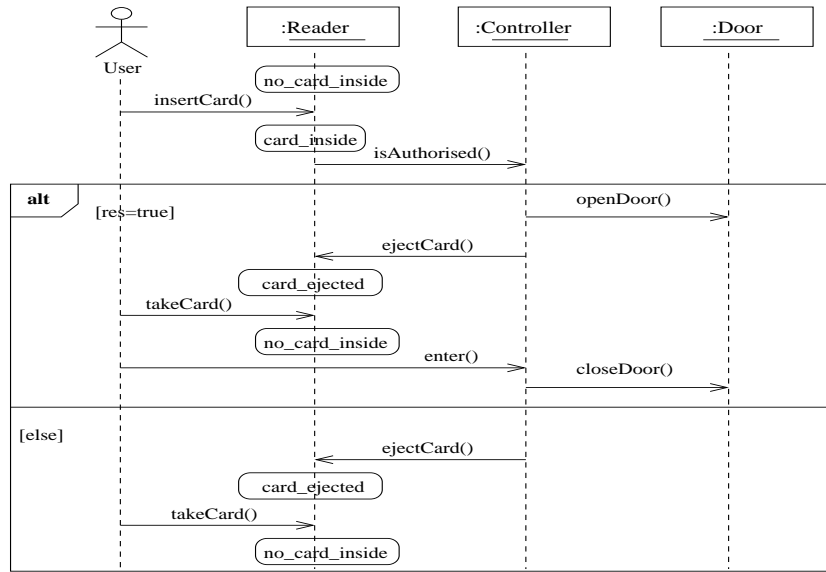


Figure 8.3 – Diagramme de séquences du scénario d’entrée dans un bâtiment

8.4.2.3 Contraintes du système

Une propriété implicite concerne l’impossibilité pour qu’une même personne soit dans le bâtiment et souhaite entrer dans ce bâtiment. Cette propriété de sûreté peut être exprimée comme suit :

- à n’importe quel moment, une personne autorisée à entrer dans le bâtiment est soit à l’intérieur du bâtiment, soit à l’extérieur.

Le système doit satisfaire les contraintes dynamiques suivantes :

- si une personne entre une carte, cette carte sera éjectée,
- la porte est maintenue fermée jusqu’à ce qu’une personne soit autorisée à entrer.

8.4.3 Spécification B

A partir de la spécification *semi-formelle* du système en UML, nous produisons une spécification *formelle* B.

8.4.3.1 Machines abstraites

Chaque classe dans le diagramme de classes est spécifiée par une machine abstraite B. Le nom des opérations dans les classes est transformé automatiquement en utilisant les règles de transformation d’UML en B. Le contenu de ces opérations est complété par les développeurs. Nous devons tester si le contenu de ces opérations est correct. Les machines abstraites *Controller*, *Reader*, *Card* et *Door* correspondant respectivement aux classes *Controller*, *Reader*, *Card* et *Door* sont présentées dans le Chapitre C de l’annexe.

8.4.3.2 Machine de simulation

La Figure 8.4 présente la machine de simulation de ce système avec un scénario correspondant à l’entrée dans le bâtiment décrit dans le diagramme de séquence Figure 8.3. Les contraintes du système sont exprimées par les clauses INVARIANT et MODALITIES, y compris les propriétés

 Access-Control.mch

SIMULATION *Access_Control*
INVARIANT

/** A n'importe quel moment, une personne autorisée à entrer dans le bâtiment est soit à l'intérieur du bâtiment, soit à l'extérieur */

 $\forall xx.(xx \in \text{authorised_cards})$
 $\square(xx \in \text{inside_cards} \vee xx \in \text{authorised_cards} - \text{inside_cards})$
MODALITIES

/** Si une personne entre une carte, cette carte sera éjectée */

 $\square(\text{state} = \text{card_inside} \Rightarrow \diamond \text{state} = \text{card_ejected})$

/** La porte est maintenue fermée jusqu'à ce qu'une personne soit autorisée à entrer */

 $\exists xx.(xx \in \text{cards}) \square(\text{state} = \text{no_card_inside} \wedge \text{door_state} = \text{close} \Rightarrow$
 $\text{door_state} = \text{close} \mathcal{W}$
 $(xx \in \text{authorised_cards} - \text{inside_cards} \wedge \text{door_state} = \text{open}))$
INITIALISATION
 $\text{cards} := \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c, 4 \mapsto d\} \parallel$
 $\text{authorised_cards} := \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\} \parallel$
 $\text{inside_cards} := \{3 \mapsto c\} \parallel$
 $\text{door_state} := \text{close} \parallel$
 $\text{state} := \text{no_card_inside} \parallel$
 $\text{current_card} := \emptyset$
SCENARIOS
Entry_Building =

begin
var *ca, auth* **in**
 $ca \leftarrow \text{insertCard}(\text{pin}, \text{code});$
 $auth \leftarrow \text{isAuthorised}(ca);$
if *auth* **then**
 $\text{openDoor};$
 $\text{ejectCard};$
 $\text{takeCard};$
 $\text{enter}(ca);$
 closeDoor
else
 $\text{ejectCard};$
 takeCard
end
end
end
END

Figure 8.4 – Machine de simulation avec un scénario

de sûreté et les propriétés de vivacité présentées dans la section 8.4.2. L'initialisation donne un point de départ pour tester ce scénario.

8.4.4 Validation du scénario Entry_Building

Le scénario proposé Figure 8.4 ne modifie pas l'ensemble des cartes existantes ni l'ensemble des cartes autorisées. Étant donné une carte, i.e. un pin et un code, il vérifie si la carte est autorisée ou non à entrer dans le bâtiment. Pour simuler chacune des situations possibles, nous devons présenter trois cas de tests.

(i.) Cas où la personne est autorisée à entrer dans le bâtiment.

Opération i	Précondition de l'opération	$state$ variable	$door_state$ variable	$inside_cards$ variable	$current_card$ variable
0 (<i>Init</i>)	–	no_card_inside	$close$	$\{3 \mapsto c\}$	\emptyset
1 (<i>insertCard</i>)	$state = no_card_inside$	$card_inside$	$close$	$\{3 \mapsto c\}$	$\{1 \mapsto a\}$
2 (<i>isAuthorised</i>)	$ca \in cards$	$card_inside$	$close$	$\{3 \mapsto c\}$	$\{1 \mapsto a\}$
authorised = true					
3 (<i>openDoor</i>)	$door_state = close$	$card_inside$	$open$	$\{3 \mapsto c\}$	$\{1 \mapsto a\}$
4 (<i>ejectCard</i>)	$state = card_inside$	$card_ejected$	$open$	$\{3 \mapsto c\}$	\emptyset
5 (<i>takeCard</i>)	$state = card_ejected$	no_card_inside	$open$	$\{3 \mapsto c\}$	\emptyset
6 (<i>enter</i>)	$ca \in authorised_cards - inside_cards$	no_card_inside	$open$	$\{3 \mapsto c, 1 \mapsto a\}$	\emptyset
7 (<i>closeDoor</i>)	$door_state = open$	no_card_inside	$close$	$\{3 \mapsto c, 1 \mapsto a\}$	\emptyset

Tableau 8.1 – Test du scénario Entry_Building avec une carte autorisée

Le Tableau 8.1 présente l'évolution des différentes variables du système en exécutant chaque étape du scénario *Entry_Building* pour une carte donnée, $pin = 1$, $code = a$. La colonne *Précondition de l'opération* rappelle, pour chaque opération du scénario, sa précondition définie dans les machines abstraites B. Nous pouvons voir qu'elles sont satisfaites pour chaque d'appel d'opération.

L'exécution de ce scénario est validée si les obligations de preuve présentées dans la section 8.2 sont prouvées. Voyons l'obligation de preuve (1). Par exemple, nous pouvons voir que après l'exécution des substitutions dans le corps des quatre premières opérations du scénario, $[S_v^4]$, la valeur de la variable $state$ de l'état du lecteur de cartes est $card_ejected$. Cet état satisfait la précondition de l'opération suivante du scénario, à savoir *takeCard* correspondant à $i = 5$.

Preuve de l'invariant : propriété de sûreté.

*/** A n'importe quel moment, une personne autorisée à entrer dans le bâtiment est soit à l'intérieur du bâtiment, soit à l'extérieur */*

$\forall xx.(xx \in authorised_cards) \square (xx \in inside_cards \vee xx \in authorised_cards - inside_cards)$

L'obligation de preuve (3) est satisfaite par l'exécution du scénario.

Preuve des modalités : propriétés de vivacité.

- La propriété de réponse est exprimée par :

*/** Si une personne entre une carte, cette carte sera éjectée */*
 $\square(state = card_inside \Rightarrow \diamond state = card_ejected)$

Dans l'introduction des propriétés de réponse présentée Figure 8.1, P correspond à $(state = card_inside)$ et Q à $(state = card_ejected)$.

L'obligation de preuve (4) est établie pour $i = 1 : [S_v^1]P = true$.

L'obligation de preuve (5) est établie pour $j = 4 : [S_v^4]Q = true$.

- La propriété de précédence est exprimée par :

*/** La porte est maintenue fermée jusqu'à ce qu'une personne soit autorisée à entrer */*
 $\exists xx.(xx \in cards) \square(state = no_card_inside \wedge door_state = close \Rightarrow door_state = close \mathcal{W}$
 $(xx \in authorised_cards - inside_cards \wedge door_state = open))$

Dans l'introduction des propriétés de précédence présentée Figure 8.1,

P correspond à $(state = no_card_inside \wedge door_state = close)$,

Q à $(door_state = close)$ et

R à $(xx \in authorised_cards - inside_cards \wedge door_state = open)$.

L'obligation de preuve (6) montre que P est établie pour $i = 0$, par l'initialisation.

L'obligation de preuve (7) montre que R est établie pour $j = 3$, avec l'appel de l'opération *OpenDoor*.

L'obligation de preuve (9) est prouvée pour $j = 3 : [S_v^3](\neg Q)$.

L'obligation de preuve (8) est prouvée pour $j = 1..2 : \forall j.(j \in [1..2] \Rightarrow [S_v^j]Q)$

(ii.) Cas où la personne n'est pas autorisée à entrer dans le bâtiment.

Le Tableau 8.2 présente l'évolution des différentes variables du système en exécutant chaque étape du scénario *Entry_Building* pour la carte non autorisée : $pin = 1$, $code = f$.

Opération i	Précondition de l'opération	$state$ variable	$door_state$ variable	$inside_cards$ variable	$current_card$ variable
0 (<i>Init</i>)	–	no_card_inside	$close$	$\{3 \mapsto c\}$	\emptyset
1 (<i>insertCard</i>)	$state = no_card_inside$	$card_inside$	$close$	$\{3 \mapsto c\}$	$\{1 \mapsto f\}$
2 (<i>isAuthorised</i>)	$ca \in cards$	$card_inside$	$close$	$\{3 \mapsto c\}$	$\{1 \mapsto f\}$
authorised = false					
3 (<i>ejectCard</i>)	$state = card_inside$	$card_ejected$	$close$	$\{3 \mapsto c\}$	\emptyset
4 (<i>takeCard</i>)	$state = card_ejected$	no_card_inside	$close$	$\{3 \mapsto c\}$	\emptyset

Tableau 8.2 – Test du scénario *Entry_Building* avec une carte non autorisée

(iii.) Cas où la personne est déjà à l'intérieur du bâtiment.

Le Tableau 8.3 présente l'évolution des différentes variables du système en exécutant chaque étape du scénario *Entry_Building* pour une carte à l'intérieur du bâtiment : $pin = 3$, $code = c$.

A la fin de l'exécution du scénario par chacun des trois cas de test, nous pouvons voir que :

- le scénario est prouvé et

Opération i	Précondition de l'opération	$state$ variable	$door_state$ variable	$inside_cards$ variable	$current_card$ variable
0 (<i>Init</i>)	–	no_card_inside	$close$	$\{3 \mapsto c\}$	\emptyset
1 (<i>insertCard</i>)	$state = no_card_inside$	$card_inside$	$close$	$\{3 \mapsto c\}$	$\{3 \mapsto c\}$
2 (<i>isAuthorised</i>)	$ca \in cards$	$card_inside$	$close$	$\{3 \mapsto c\}$	$\{3 \mapsto c\}$
authorised = false					
3 (<i>ejectCard</i>)	$state = card_inside$	$card_ejected$	$close$	$\{3 \mapsto c\}$	\emptyset
4 (<i>takeCard</i>)	$state = card_ejected$	no_card_inside	$close$	$\{3 \mapsto c\}$	\emptyset

Tableau 8.3 – Test du scénario *Entry_Building avec une carte à l'intérieur*

- les propriétés de sûreté et de vivacité du système sont satisfaites.

Le résultat dans les trois tables peut être prouvé par rapport à des propriétés du système par des outils de preuve qui déchargent des obligations de preuve proposées dans la section 8.2 et la section 8.3.

8.5 Synthèse

Nous avons présenté une approche pour la validation de spécifications orientées objets en utilisant la notation B. Nous commençons à partir des spécifications UML exprimées sous la forme d'un diagramme de classes et d'un ensemble de diagrammes de séquences exprimant des scénarios du comportement du système. Le diagramme de classes est alors dérivé automatiquement en une spécification B. Cette spécification est complétée par la définition des opérations (messages dans les diagrammes de séquences correspondant aux opérations dans le diagramme de classes) et par une nouvelle machine, appelée la machine de simulation. Cette machine contient la dérivation du diagramme de séquences augmenté par les propriétés dynamiques du système.

La validation des scénarios et la satisfaction des propriétés est effectuée par des prouveurs de théorème de B. Afin d'utiliser ce prouveur, nous avons défini les obligations de preuve pour la machine de simulation. Nous devons choisir des jeux de test pour la validation et le processus s'effectue en deux étapes :

- d'abord, nous prouvons un scénario du comportement d'un système défini à l'aide d'un diagramme de séquences,
- lorsque l'exécution de ce scénario a été prouvée, nous devons prouver que le scénario satisfait les propriétés dynamiques du système.

L'outil support au développement B par objets

SOMMAIRE

9.1	Prise en compte des nouvelles obligations de preuve	123
9.2	L'outil B4free	124
9.3	Description de l'outil Boo	125
9.4	Synthèse	126

Pour prendre en compte les obligations de preuve proposées dans les approches présentées dans les chapitre 7 et 8, nous avons implanté un outil, nommé Boo, permettant de spécifier les machines abstraites et la machine de simulation. Les nouvelles obligations de preuve générées sont ensuite prouvées par les outils supports de B existants.

La présentation de ce chapitre est la suivante. Nous commençons par l'introduction du principe de prise en compte des nouvelles obligations de preuve dans les outils supports de B. Puis, nous présentons l'outil B4free, le prouveur utilisé pour prouver les obligations de preuve. Ensuite nous décrivons notre outil Boo.

9.1 Prise en compte des nouvelles obligations de preuve

Notre proposition consiste à introduire en B les obligations de preuve pour prendre en compte des propriétés de réciprocité des approches objets et pour tester les spécifications orientées objets. La méthode B présente des outils supports dont la mission principale est :

- l'analyse lexicale et syntaxique des modèles ;
- la génération des obligations de preuve et
- la preuve automatique ou interactive des prédicats générés à partir des obligations de preuve.

Pour prendre en compte de nouvelles obligations de preuve, nous utilisons la clause ASSERTIONS disponible dans une machine abstraite B (voir Figure 1.1). Cette clause est constituée d'une liste de prédicats qui sont des conséquences logiques des autres axiomes et de l'invariant déclaré dans la machine. Ces prédicats peuvent être vus comme des lemmes utiles pour les preuves ou la compréhension de la spécification. Ces assertions devront être vérifiées par le prouveur.

A partir d'une spécification B à prouver, nous construisons une nouvelle machine abstraite, nommée *Proof*, dans laquelle sa clause ASSERTIONS contiendra toutes les obligations de preuve générées. Pour prouver cette machine, les prédicats dans la clause ASSERTIONS sont importés dans le moteur de preuve des outils supports.

```

Proof.mch
-----
MACHINE Proof

ASSERTIONS
   $\forall(y,n).(y \in \mathbb{F}(\mathbb{N}_1) \wedge n \in \mathbb{N}_1 \Rightarrow [y := y \cup \{n\}](y \in \mathbb{F}(\mathbb{N}_1)))$ 
END

```

Figure 9.1 – Un exemple de la machine abstraite intermédiaire

Pour illustrer ce principe dans la machine *Proof*, nous prenons une machine B simple présentée dans la Figure 1.2 (*Exemple1.mch*). Nous allons illustrer la génération des obligations de preuve de préservation des invariants par les substitutions dans les opérations.

Les obligations de preuve qui préservent les invariants par des opérations sont exprimées sous la forme : $I \wedge P \Rightarrow S[I]$

L'invariant de la machine : $y \in \mathbb{F}(\mathbb{N}_1)$

Prenons l'opération *enter* définie par : *enter*(*n*) = **pre** *n* ∈ \mathbb{N}_1 **then** $y := y \cup \{n\}$ **end**

Nous devons donc prouver : $y \in \mathbb{F}(\mathbb{N}_1) \wedge n \in \mathbb{N}_1 \Rightarrow [y := y \cup \{n\}](y \in \mathbb{F}(\mathbb{N}_1))$

La machine abstraite intermédiaire est proposée dans la Figure 9.1. C'est une machine abstraite B qui peut être prouvée par les outils supports de B.

9.2 L'outil B4free

La méthode B possède plusieurs outils supports comme par exemple AtelierB, B-Toolkit, B4free. Nous avons choisi le prouveur de B4free pour le connecter avec notre interface, parce que B4free est un outil gratuit, disponible pour les utilisateurs académiques. B4free est utilisé en mode console. D. Cansell *et al* [Cansell 03] ont développé une interface pour faciliter l'utilisation de cet outil et pour prouver leur contributions concernant la preuve sur des ensembles.

Les commandes de B4free peuvent être classées en trois catégories.

1. Les commandes générales :

```

(cd ) change_directory      (h  ) help
(lrf ) load_res_file        (q  ) quit
(v   ) version_print

```

2. Les commandes de niveau projet :

```

(arc ) archive              (crp ) create_project
(op  ) open_project        (rp  ) remove_project
(res ) restore              (spl ) show_projects_list

```

3. Les commandes de niveau machine (disponible après l'ouverture d'un projet) :

```

(af  ) add_file             (b   ) browse
(clp ) close_project        (ic  ) infos_component
(po  ) pogenerate          (pr  ) prove
(rc  ) remove_component     (sml ) show_machines_list

```

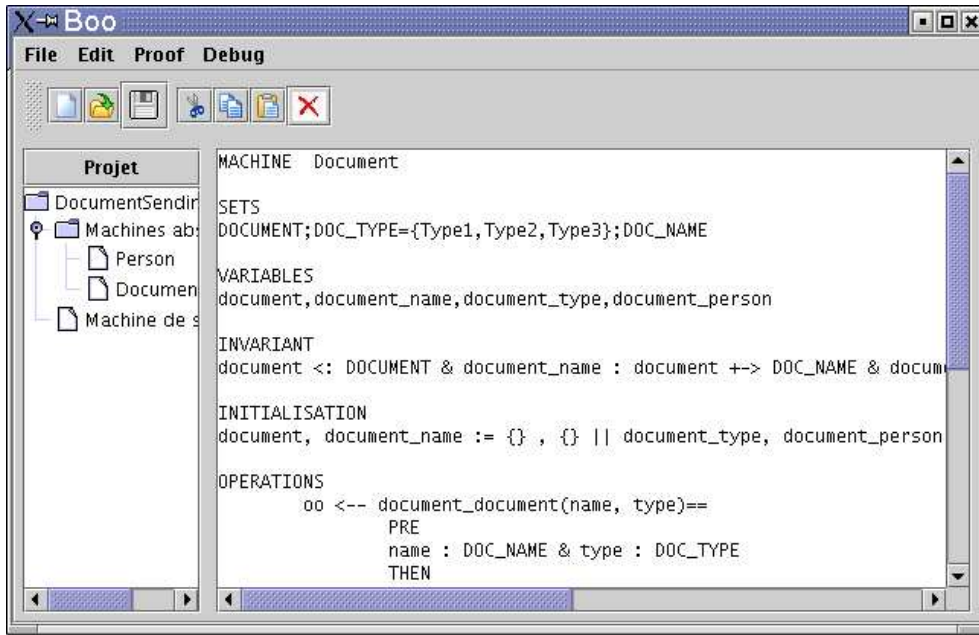



Figure 9.3 – Interface de l'outil Boo

9.4 Synthèse

Dans ce chapitre, nous avons présenté un outil pour prendre en compte les obligations de preuve de nos travaux de développement formel B par objets. L'objectif de cet outil est :

- d'éditer les machines abstraites et la machine de simulation,
- de générer les obligations de preuve et
- d'utiliser le prouveur de B4free pour effectuer les preuves.

L'implantation de cet outil n'a pas encore fini. L'interface de l'outil et sa connexion avec le prouveur de B4free ont été implantés, avec la génération des obligations de preuve définies dans le chapitre 7.

CONCLUSION

Le couplage des approches orientées objets avec la méthode B apporte une amélioration à l'activité de spécification et de développement. La méthode B fournit des notations de spécification et des outils supports puissants permettant de modéliser et de vérifier des modèles. Les approches objets fournissent des mécanismes de structuration et de développement intéressants pour le développement de gros systèmes. L'apport de notre travail de thèse contribue aux activités de couplage entre ces deux formalismes.

Synthèse de la contribution

La contribution de cette thèse porte sur l'utilisation de B pour la vérification formelle de spécifications UML et le développement formel orienté objets.

Vérification formelle de spécifications UML

Notre contribution sur la vérification de spécifications UML en utilisant B couvre les aspects de spécifications UML. Nos propositions permettent la vérification de la sémantique des éléments du modèle UML ainsi que des propriétés sur les diagrammes statiques et sur les diagrammes dynamiques.

Vérification de la sémantique des éléments du modèle UML. Nous proposons une approche de transformation de méta-modèles UML en B permettant de vérifier la sémantique des éléments du modèle UML. Les propriétés des éléments du modèle sont modélisées par des méta-classes et sa sémantique est définie par les règles de bonne formation dans le méta-modèle UML. En nous basant sur la structure des méta-classes, chaque objet des méta-classes est transformé en une machine abstraite B, les règles de bonne formation sont transformées en invariants des machines abstraites d'objets composites qui utilisent les machines abstraites d'objets composants. Nous ajoutons de nouvelles opérations dans les machines abstraites permettant d'associer ces attributs aux contraintes de règles de bonne formation. La vérification est effectuée à l'aide des obligations de preuve préservant l'invariant des machines abstraites. Ces obligations de preuve sont générées automatiquement et prouvées par l'AtelierB. En ce qui concerne ce travail, nous avons transformé le méta-modèle UML des diagrammes de classes en B pour vérifier la sémantique des éléments des diagrammes de classes, le méta-modèle UML des diagrammes de collaboration et des diagrammes d'état-transitions en B pour vérifier la sémantique des éléments des diagrammes correspondants (chapitre 4).

Vérification des propriétés des diagrammes statiques. La technique de transformation de méta-modèle UML en B permettant de vérifier des éléments du modèle UML est appliquée pour analyser des propriétés des diagrammes statiques UML. Des diagrammes de classes, des contraintes

OCL associées au modèle et des instances de ces diagrammes (diagrammes d'objets) sont transformées en B. Les contraintes d'association dans les diagrammes de classes et les contraintes exprimées en OCL donnent des invariants de la spécification B. Cet ensemble a été transformé et intégré dans une spécification B complète. L'analyse de cette spécification B permet de vérifier la consistance entre les diagrammes statiques UML et les contraintes OCL (chapitre 5).

Vérification des propriétés des diagrammes dynamiques. Nous avons abordé le couplage UML et B à l'aide d'un scénario présenté par un diagramme de classes et des diagrammes de collaboration (chapitre 6). Ce scénario est transformé en B en apportant une solution aux limites des dérivations de diagrammes de collaboration en B proposées par Ledang [LeDang 02a]. Nous avons également démontré des propriétés des spécifications UML/OCL analysées à partir de la preuve de la méthode B, à savoir la consistance entre les pré- et postconditions des opérations séquentielles et des opérations décomposées.

Prise en compte de l'objet dans le développement formel B

Il s'agit d'utilisation B avec une orientation objet. Notre apport sur ce point concerne la prise en compte de certains types d'association dans les approches orientées objets pour les relations entre machines abstraites et la validation de spécifications orientées objets en B.

Prise en compte de certains types d'association des approches objets pour B. Pour faciliter l'utilisation de B dans la spécification orientée objets, nous proposons de prendre en compte certains types d'association entre classes UML pour les machines abstraites B. Nous rassemblons les machines abstraites transformées à partir des classes UML reliées par les types d'association qui ne sont pas autorisés en B dans un groupe, nous étendons la forme de partage autorisée entre ces machines. Pour cela, de nouvelles obligations de preuve ont été définies, permettant de garantir les invariants des machines concernées. Avec les autres machines dans le système ayant les relations unidirectionnelles avec les machines dans le groupe rassemblé, nous utilisons les obligations de preuve des clauses de composition entre les machines abstraites B pour prouver la préservation des invariants (chapitre 7).

Validation de spécifications orientées objets en B. A partir d'un modèle objets spécifié en UML par des diagrammes de classes qui structurent le système et des diagrammes de séquences qui expriment des scénarios de l'exécution, nous dérivons les diagrammes de classes en machines abstraites B, cette spécification est complétée par les définitions des opérations. Nous proposons une machine de simulation permettant de spécifier l'exécution des messages dans les diagrammes de séquences. La machine de simulation peut être renforcée par les contraintes de sûreté et les contraintes de vivacité. La validation de l'exécution des opérations dans un scénario et la satisfaction des propriétés dynamiques est effectuée par la démonstration des obligations de preuve proposées (chapitre 8).

Implantation. Nous avons implanté un outil, nommé Boo, permettant de spécifier les machines abstraites et la machine de simulation (chapitre 9). Cet outil permet de générer les nouvelles obligations de preuve qui sont ensuite prouvées par les outils supports de B existants.

Perspectives

L'accomplissement de l'implantation de l'outil Boo, la prise en compte d'autres propriétés objets dans la méthode B et la vérification de consistance entre diagrammes d'état-transitions et diagrammes de séquences en B sont des perspectives à court terme de notre travail.

Environnement de développement B par objets

L'implantation de l'outil support Boo est en cours afin de fournir un environnement d'aide à la spécification, la vérification et la validation des modèles orientés objets en utilisant les notations B.

Prise en compte d'autres propriétés objets pour les notations B

En utilisant les notations B pour décrire des modèles orientés objets, nous avons pris en compte certaines propriétés objets pour les machines abstraites de B. D'autres propriétés n'ont pas été considérées, telles que l'héritage qui joue un rôle important dans la spécification et la programmation orientée objets. Certaines méthodes formelles ont été prises en compte l'héritage dans la spécification. Nous allons étudier pour proposer un mécanisme pertinent de l'héritage entre machines abstraites B.

Vérification de la consistance entre diagrammes d'état-transitions et diagrammes de séquences

Les diagrammes de séquences UML 2.0 permettent d'exprimer les états sur la ligne de vie des objets. Pour chaque objet présent dans un scénario, ils proposent un graphe de transition concrète entre certains états de cet objet. Celui-ci peut être généré sous la forme d'un diagramme d'état-transitions. Illustrons ceci avec l'exemple du diagramme de séquences (Figure 8.3) du système de contrôle d'accès présenté dans la section 8.4. Les transitions entre états sur la ligne de vie d'un objet de classe Reader est générées dans la Figure 1.

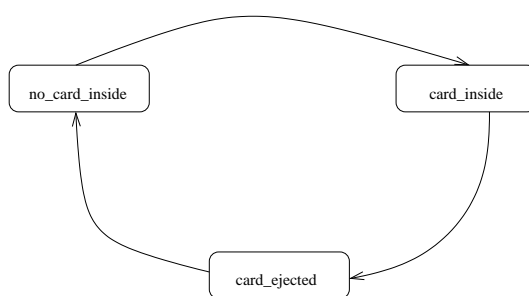


Figure 1 – Diagramme d'état-transition généré à partir de ligne de vie d'un objet de classe Reader

Les diagrammes d'état-transitions sont une représentation graphique d'une machine d'état fini, expriment toutes les transitions possibles entre états dans un objet. Le diagramme d'état-transition des objets de la classe Reader est présenté Figure 2.

En utilisant le raffinement B, nous pouvons vérifier que les transitions entre états sur les lignes de vie des diagrammes de séquences (Figure 1) sont un chemin correct des transitions de diagrammes d'état-transitions (Figure 2).

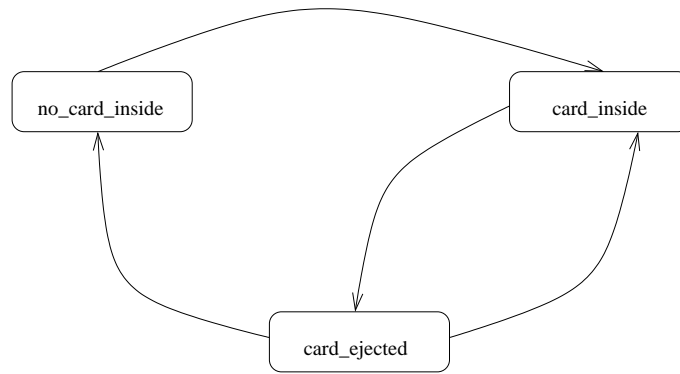


Figure 2 – Diagramme d'état-transition d'un objet de classe Reader

D'autres perspectives

Les perspectives à long terme de notre travail concerne les deux points suivants :

- le couplage de nos résultats avec les opérateurs de développement proposés par Okalas [Okalas 05]. On peut imaginer deux mises en scène possible :
 - les différentes vérifications et validations sont proposées aux spécifieurs sous la forme d'outil et c'est le spécifieur qui décide de les utiliser,
 - l'utilisation des vérifications et validation est intégrée dans la définition des opérateurs et font partie des approches définies;
- nos "outils de validation et vérification" reposent sur l'utilisation du prouveur de B dans des cas particuliers bien identifiés. Une étude des obligations de preuve générées dans les cas d'échec peuvent apporter des aides à la construction des spécifications UML de départ. C'est un des objectifs du travail de thèse de Inès Mouakher.

BIBLIOGRAPHIE

- [Abadi 96] M. Abadi et L. Cardelli. *A Theory of Objects*. Springer, 1996.
- [Abrial 84] J. R. Abrial. Spécifier ou comment matérialiser l'abstrait. *Technique et Science Informatique*, 3(3):201–219, 1984.
- [Abrial 96a] J. R. Abrial. Extending B without Changing it (for Developing Distributed Systems). H. Habrias, éditeur, *Putting Into Practice Methods and Tools for Information System Design - 1st Conference on the B Method*, November 1996.
- [Abrial 96b] J. R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.
- [Abrial 98] J. R. Abrial et L. Mussat. Introducing Dynamic Constraints in B. D. Bert, éditeur, *B'98: Recent Advances in the Development and Use of the B Method - 2nd International B Conference*, numéro 1393 in Lecture Notes in Computer Science, Montpellier (F), April 1998. Springer-Verlag.
- [Abrial 00] J. R. Abrial. B : 2000 et plus. *École Jeunes chercheurs en programmation*, Ecole Normale Supérieure de Lyon, Janvier 2000.
- [Alencar 91] A. J. Alencar et J. A. Goguen. OOZE: An Object-Oriented Z Environment. P. America, éditeur, *Proceedings of ECCOP'91*, numéro 512 in Lecture Notes in Computer Science. Springer-Verlag, 1991.
- [Ambert 02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, N. Vacelet et M. Utting. BZ-TT: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming. *Formal Approaches to Testing of Software Workshop*, 2002.
- [André 95] P. André. *Méthodes formelles et à objets pour le développement du logiciel: Etudes et propositions*. Thèse de Doctorat, Université de Rennes I, Juillet 1995.
- [Arnold 92] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Collection Etudes et recherches en informatique. Masson, 1992.
- [B-C 96] B-Core(UK) Ltd, Oxford (UK). *B-Toolkit User's Manual*, 1996. Release 3.2.
- [Barradas 05] H. R. Barradas et D. Bert. Propriétés dynamiques avec hypothèses d'équité en B événementiel. *Technique et Science Informatiques, RSTI*, 2005.
- [Behm 99] P. Behm, P. Benoit et J. M. Meynadier. METEOR: A Successful Application of B in a Large Project. *Integrated Formal Methods, IFM99*, volume 1708, série LNCS, pages 369–387. Springer Verlag, 1999.
- [Bert 95] D. Bert, R. Echahed, P. Jacquet, M. L. Potet et J. C. Reynaud. Spécification, Généricité, Prototypage: Aspects du langage LPG. *Technique et Science Informatique*, 14(9):1097–1129, 1995.
- [Boehm 82] B. W. Boehm. Les facteurs du coût logiciel. *Technique et Science Informatique*, 1(1):5–24, 1982.

- [Booch 94] G. Booch. *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc, second édition, 1994. ISBN 0-8053-5340-2.
- [Booch 98] G. Booch, J. Rumbaugh et I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998. ISBN 0-201-57168-4.
- [Buchi 99] M. Buchi et R. Back. Compositional Symmetric Sharing in B. *FM'99, Formal Methods Conference*, volume 1708, série *LNCS*, pages 431–451. Springer Verlag, 1999.
- [Cansell 03] D. Cansell et J. R. Abrial. Click'n prove: Interactive proofs within set theory. *Theorem Proving in Higher Order Logics*, pages 1–24, 2003.
- [Casais 93] E. Casais, C. Lewerent, T. Lindner et W. Franz. Formal Methods and Object-Orientation. *Tools Europe'93*. Prentice-Hall, 1993.
- [Cavarra 04] A. Cavarra, E. Riccobene et P. Scandurra. A framework to simulate UML models: moving from a semi-formal to a formal environment. *Proceedings of the 2004 ACM Symposium in Applied Computing*, pages 1519–1523. ACM press, 2004.
- [Choppy 88] C. Choppy et M. C. Gaudel. Impact des spécifications formelles sur le développement de logiciel. *BIGRE + GLOBULE*, 58:3–11, 1988.
- [Clearsy] Clearsy. *B4free*. Available at <http://www.b4free.com>.
- [CoF 99] The CoFI Task Group on Language Design. *CASL: The Common Algebraic Specification Language Summary*, July 1999. Release 1.0.
- [Cusack 91] E. Cusack. Inheritance in Object Oriented Z. *ECOOP'91*, volume 512, série *LNCS*. Springer Verlag, 1991.
- [Dietrich 00] F. Dietrich. *Modelling and testing object-oriented communication services with temporal logic*. Thèse de Doctorat, Ecole Polytechnique Fédérale de Lausanne, 2000.
- [Dion 01] B. Dion. Vérification formelle de spécifications UML, septembre 2001. Présenté dans le séminaire de l'Informations - UML: Unified Modelling Language.
- [Distefano 00] D. Distefano, J-P. Katoen et A. Rensink. On a temporal logic for object-based systems. *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000*. Kluwer Academic Publishers, 2000.
- [Duke 90] D. Duke et R. Duke. Towards a Semantics for Object-Z. *VDM'90*, *LNCS*, pages 244–261. Springer Verlag, 1990.
- [Durr 92] E. H. Durr et J. V. Katwijk. VDM++ - A Formal Specification Language for Object-oriented Design. *Tools Europe'92, Technology of Object-Oriented Languages and Systems*, pages 63–78. Prentice-Hall, 1992.
- [Ehrig 85] H. Ehrig et B. Mahr. *Fundamentals of Algebraic Specification 1 (Equations and Initial Semantics)*, volume 6, série *ETACS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.
- [Finance 90] J. P. Finance, N. Lévy, J. Souquière et A. Valdenaire. SACSO: un environnement d'aide à la spécification. *Technique et Science Informatique*, 9(3):245–261, 1990.
- [Gallardo 02] M. M. Gallardo, P. Merino et E. Pimentel. Debugging UML Designs with Model Checking. *Journal Of Object Technology*, 1(2), 2002.
- [Goguen 96] J. Goguen et G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996. ISBN 0-262-07172-X.

-
- [Graham 97] I. Graham. *Méthodes orientées objet*. International Thomson publishing, seconde édition, 1997. ISBN 2-84180-984-6.
- [Gutttag 93] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet et J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, January 1993.
- [Harel 87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
- [Hayes 92] I. Hayes. *Specification Case Studies*. C.A.R Hoare Series. Prentice-Hall International, second édition, 1992.
- [Hazem 04] L. Hazem, N. Lévy et R. Marcano. UML2B : un outil pour la génération de modèles formels. *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, 2004.
- [Hinchey 95] M. G. Hinchey et J. P. Bowen. *Applications of Formal Methods*. Prentice Hall, 1995.
- [Hoare 85] C.A.R. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series. Prentice-Hall International, 1985.
- [Huet 99] G. Huet, G. Kahn et C. Paulin-Mohring. *The Coq Proof Assistant: A Tutorial*. Coq Project, December 1999. Release 6.3.1.
- [Jacobson 92] I. Jacobson, M. Christerson, P. Jonsson et G. Overgaard. *Object-Oriented Software Engineering : A Use case Driven Approach*. Addison Wesley, 1992.
- [Jones 90] C. B. Jones. *Systematic Software Development using VDM*. Prentice-Hall International, 1990. ISBN 0-13-880733-7.
- [Julliand 98] J. Julliand et F. Bellegarde. Extension des spécifications B pour décrire des propriétés dynamiques de systèmes réactifs. *AFADL'98 : Actes des journées AFADL*, Octobre 1998.
- [Kemmerer 85] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, 1985.
- [Kim 00] S. K. Kim et D. Carrington. A Formal Mapping between UML Models and Object-Z Specifications. *ZB 2000: Formal Specification and Development in Z and B*, volume 1878, série *Lecture Notes in Computer Science*. Springer Verlag, 2000.
- [Kneuper 89] R. Kneuper. *Symbolic Execution as a Tool for Validation of Specifications*. Thèse de Doctorat, University of Manchester, 1989.
- [Laleau 00] R. Laleau et F. Polack. Metamodels for Static Conceptual Modelling of Information System. *Workshop on Defining Precise Semantics of UML, ECOOP Conference*, Lecture Notes in Computer Science. Springer Verlag, 2000.
- [Lamsweerde 00] A.V. Lamsweerde. Formal specification: a roadmap. A. Finkelstein, éditeur, *The Future of Software Engineering*. ACM press, June 2000.
- [Lano 91] K. Lano. Z++, an Object-Oriented Extension to Z. J. E. Nicholls, éditeur, *Z User Workshop*, Workshops in Computing, pages 151–172, Oxford (UK), 1991. Springer-Verlag.
- [Lano 96] K. Lano. *The B Language and Method: A Guide to Practical Formal Development*. FACIT. Springer-Verlag, 1996. ISBN 3-540-76033-4.

- [Lano 04] K. Lano, D. Clark et K. Androutsopoulos. UML to B: Formal Verification of Object Oriented Models. *Integrated Formal Methods*, volume 2999, série *Lecture Notes in Computer Science*, pages 187–206. Springer Verlag, 2004.
- [Ledang 01] H. Ledang et J. Souquière. Modeling class operations in B: Application to UML behavioral diagrams. *IEEE International Conference on Automated Software Engineering*, pages 289–296. IEEE Computer Society, 2001.
- [LeDang 02a] H. LeDang. *Traduction Systématique de Spécifications UML en B*. Thèse de Doctorat, Université Nancy 2, LORIA, novembre 2002.
- [LeDang 02b] H. LeDang et J. Souquière. Contributions for modelling UML state-charts in B. *Third International Conference on Integrated Formal Methods*, volume 1456, série *LNCS*. Springer Verlag, 2002.
- [Ledang 02c] H. Ledang et J. Souquière. Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B. *9th Asia Pacific Software Engineering Conference (APSEC)*, Lecture Notes in Computer Science. Springer Verlag, 2002.
- [Legéard 02] B. Legéard, F. Peureux et M. Utting. Automated boundary testing from Z and B. *Formal Method Europe, FME'02*. Springer Verlag, 2002.
- [Leuschel 03] M. Leuschel et M. Butler. ProB: A model checker for B. *Integrated Formal Method, IFM'03*, Lecture Notes in Computer Science, pages 855–874. Springer Verlag, 2003.
- [Malioukov 98] A. Malioukov. An Object-Based Approach to the B Formal Method. D. Bert, éditeur, *B'98: Recent Advances in the Development and Use of the B Method - 2nd International B Conference*, numéro 1393 in Lecture Notes in Computer Science, Montpellier (F), April 1998. Springer-Verlag.
- [Manna 91] Z. Manna et A. Pnueli. Tools and rules for the practicing verifier. Rapport, Stanford University, juin 1991.
- [Marcano 01] R. Marcano et N. Lévy. Transformation d'annotations OCL en expression B. *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, 2001.
- [Marcano 02] R. Marcano et N. Lévy. Using B formal specifications for analysis and verification of UML/OCL models. *Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*, 2002.
- [Meyer 85] B. Meyer. On Formalism in Specifications. *IEEE Software*, 2(1):6–26, 1985.
- [Meyer 88] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.
- [Meyer 90] B. Meyer. *Conception et programmation par objets*. InterEdition, 1990. ISBN 2-7296-0272-0.
- [Meyer 92] B. Meyer. *Introduction à la théorie des langages de programmation*. InterEdition, 1992. ISBN 2-7296-0416-2.
- [Meyer 99] E. Meyer et J. Souquière. A systematic approach to transform OMT diagrams to a B specification. *Proceedings of the Formal Method Conference*, numéro 1708 in Lecture Notes in Computer Science, pages 875–895. Spring Verlag, 1999.
- [Meyer 01] E. Meyer. *Développements formels par objets : utilisation conjointe de B et UML*. Thèse de Doctorat, Université Nancy 2, LORIA, mars 2001.

-
- [Milner 89] R. Milner. *Communication and Concurrency*. C.A.R. Hoare Series. Prentice-Hall International, 1989.
- [Nguyen 98] H. P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. Thèse de Doctorat, Conservatoire National des Arts et Métiers - CEDRIC, Paris (F), Décembre 1998.
- [Okalas 05] D. O. Okalas, J. Souquières et J-P. Jacquot. Consistency in UML and B multi-view specifications. *Proceeding of the International Conference on Integrated Formal Methods, IFM'05*, numéro 3771 in LNCS, pages 386–405. Springer-Verlag, 2005.
- [OMG 03] OMG. *Unified Modeling Language*. OMG [http : //www.omg.org/docs/formal/03-03-01.pdf](http://www.omg.org/docs/formal/03-03-01.pdf), Version 1.5 March 2003.
- [OMG 04] OMG. UML 2.0 Superstructure Specification. Available at <http://www.omg.org>, 2004.
- [Oussalah 97] C. Oussalah. *Ingénierie objet – Concepts et techniques*. InterEdition, 1997. ISBN 2-7296-06402.
- [Paludetto 05] M. Paludetto, J. Delatour et A. Benzina. UML et réseaux de Petri. *Technique et Sciences Informatiques*, 24, 2005.
- [Paulson 94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Numéro 828 in LNCS. Springer Verlag, 1994.
- [Pickin 04] S. Pickin et J. M Jézéquel. Using UML Sequence Diagrams as the Basis for a Formal Test Description Language. *Integrated Formal Method, IFM'04*, Lecture Notes in Computer Science. Springer Verlag, 2004.
- [Potet 03] M.-L. Potet. Spécifications et développements structurés dans la méthode B. *Revue TSI*, 22(1):61—88, 2003.
- [Rapanotti 92] L. Rapanotti et A. Socorro. Introducing foops. Rapport no. PRG-TR-28-92, Oxford University Computing Laboratory, Keble Road, Oxford, OX1 3QD, UK, 1992.
- [Rumbaugh 94] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy et W. Lorensen. *OMT Modélisation et conception orientées objet*. Masson - Prentice Hall, 1994. ISBN 2-225-84684-7.
- [Satpathy 04] M. Satpathy, M. Leuschel et M. Butler. ProTest: An automatic test environment for B specifications. *International workshop on Model Based Testing*, 2004.
- [Schafer 01] T. Schafer et S. Metz. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science*, 47:1–13, 2001.
- [Shankar 93a] A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, 1993.
- [Shankar 93b] N. Shankar, S. Owre et J. M. Rushby. *PVS Tutorial*, February 1993. in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406.
- [Shen 01] W. Shen, K. Compton et J.K Huggins. A validation method for a uml model based on abstract state machines. *Formal Methods and Tools for Computer Science*, pages 220–223, 2001.
- [Smith 92a] G. Smith. *An Object-Oriented Approach to Formal Specification*. Thèse de Doctorat, Department of Computing Science - University of Queensland, 1992.

- [Smith 92b] G. Smith et R. Duke. Specifying Concurrent Systems Using Object-Z. *15th Australian Computer Science Conference*, 1992.
- [Spivey 92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd édition, 1992. ISBN 013-978529-9.
- [STE 98] STERIA - Technologies de l'Information, Aix-en-Provence (F). *Atelier B, Manuel Utilisateur*, 1998. Version 3.5.
- [Sykes 01] D. A. Sykes et J. D. McGregor. *Practical guide to testing object-oriented software*. Addison Wesley, 2001.
- [Tkach 98] D. Tkach et R. Puttick. *La technologie objet dans le développement d'applications*. vuibert, seconde édition, 1998. ISBN 2-7117-8612-9.
- [Truong 04a] N. T. Truong et J. Souquières. An approach for the verification of UML models using B. *11th International Conference of Engineering of Computer Based Systems (ECBS)*, pages 195–202. IEEE Computer Society press, 2004.
- [Truong 04b] N. T. Truong et J. Souquières. Validation des propriétés d'un scénario UML/OCL à partir de sa dérivation en B. *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 04)*, pages 99–113, 2004.
- [Truong 05a] N. T. Truong et J. Souquières. Validation of UML static diagrams using B. *International conference on Software Engineering Research and Practice (SERP'05)*, pages 915–927. CSREA press, 2005.
- [Truong 05b] N. T. Truong et J. Souquières. Verification of behavioral elements of UML models using B. *20th Annual ACM Symposium on Applied Computing (SAC'05)*, pages 1546–1552. ACM press, 2005.
- [Truong 06a] N. T. Truong et J. Souquières. Validation of UML scenarios using the B prover. Rapport, INRIA, March 2006.
- [Truong 06b] N. T. Truong et J. Souquières. Verification of UML model elements using B. *Journal of Information Science and Engineering (JISE)*, 22(2):359–375, 2006.
- [van Eijk 89] P. H. J. van Eijk, C. A. Vissers et M. Diaz, éditeurs. *The formal description technique LOTOS*. Elsevier Science Publishers B.V., 1989.
- [Warmer 99] J. Warmer et A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999. ISBN 0-201-37940-6.

Glossaire B

La syntaxe complète de la notation B est présentée dans le B-Book [Abrial 96b]. Nous présentons ci-dessous les notations minimales nécessaires à la compréhension du document.

A.1 Logique

Dans le tableau suivant, P et Q sont des prédicats, x est une variable, E et F sont des expressions.

Syntaxe	Notation
$P \vee Q$	$\neg P \Rightarrow Q$
$P \Leftrightarrow Q$	$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$
$\exists x.P$	$\neg \forall x.\neg P$
$E \mapsto F$	E, F

A.2 Les ensembles

Syntaxe	Notation	Condition
$s \subseteq t$	$s \in P(t)$	
$s \subset t$	$s \subseteq t \wedge s \neq t$	
$u \cup v$	$\{x \mid x \in s \wedge (x \in u \vee x \in v)\}$	$u \subseteq s \wedge v \subseteq s$
$u \cap v$	$\{x \mid x \in s \wedge (x \in u \wedge x \in v)\}$	$u \subseteq s \wedge v \subseteq s$
$u - v$	$\{x \mid x \in s \wedge (x \in u \wedge x \notin v)\}$	$u \subseteq s \wedge v \subseteq s$
$\{E\}$	$\{x \mid x \in s \wedge x = E\}$	$E \in s \quad x \setminus (s, E)$
L, E	$\{L\} \cup \{E\}$	$\{L\} \subseteq s \wedge E \in s$
\emptyset	BIG - BIG	
$\mathbb{P}_1(s)$	$\mathbb{P}(s) - \{\emptyset\}$	

s, t, u et v sont des ensembles, x est une variable, E est une expression, L est une liste d'expressions.

A.3 Les relations binaires

Dans le tableau suivant, u, v et w sont des ensembles, a, b et c sont des variables distinctes, et p, q, s et t sont définis comme suit:

$$p \in u \leftrightarrow v \quad q \in v \leftrightarrow w \quad s \subseteq u \quad t \subseteq v$$

Syntaxe	Définition
$u \leftrightarrow v$	$\mathbb{P}(u \times v)$
p^{-1}	$\{b, a \mid (b, a) \in v \times u \wedge (a, b) \in p\}$
$\text{dom}(p)$	$\{a \mid a \in u \wedge \exists b. (b \in v \wedge (a, b) \in p)\}$
$\text{ran}(p)$	$\text{dom}(p^{-1})$
$p; q$	$\{a, c \mid (a, c) \in u \times w \wedge \exists b. (b \in v \wedge (a, b) \in p \wedge (b, c) \in q)\}$
$q \circ p$	$p; q$
$\text{id}(u)$	$\{a, b \mid (a, b) \in u \times v \wedge a = b\}$
$s \triangleleft p$	$\text{id}(s); p$
$p \triangleright t$	$p; \text{id}(t)$
$s \triangleleft p$	$(\text{dom}(p) - s) \triangleleft p$
$p \triangleright t$	$p \triangleright (\text{ran}(p) - t)$

Dans le tableau suivant, s, t, u et v sont des ensembles, a, b et c sont des variables, et p, w, q, f, g, h et k sont définis comme suit: $p \in s \leftrightarrow t \quad w \subseteq s \quad q \in s \leftrightarrow t \quad f \in s \leftrightarrow t$
 $g \in s \leftrightarrow v \quad h \in s \leftrightarrow u \quad k \in t \leftrightarrow v$

Syntaxe	Définition
$p[w]$	$\{b \mid b \in t \wedge \exists a. (a \in w \wedge (a, b) \in p)\}$
$q \triangleleft p$	$\{a, b \mid (a, b) \in s \times t \wedge ((a, b) \in q \wedge a \notin \text{dom}(p)) \vee (a, b) \in p\}$
$\text{prj}_1(s, t)$	$\{a, b, c \mid (a, b, c) \in s \times t \times s \wedge c = a\}$
$\text{prj}_2(s, t)$	$\{a, b, c \mid (a, b, c) \in s \times t \times t \wedge c = b\}$
$h \parallel k$	$\{(a, b), (c, d) \mid ((a, b), (c, d)) \in (s \times t) \times (u \times v) \wedge (a, c) \in h \wedge (b, d) \in k\}$

A.4 Les fonctions

Dans le tableau suivant, s et t sont des ensembles, r et f sont des variables.

Syntaxe	Définition
$s \leftrightarrow t$	$\{r \mid r \in s \leftrightarrow t \wedge (r^{-1}; r) \subseteq id(t)\}$
$s \rightarrow t$	$\{f \mid f \in s \rightarrow t \wedge dom(f) = s\}$
$s \rightsquigarrow t$	$\{f \mid f \in s \rightsquigarrow t \wedge f^{-1} \in t \rightsquigarrow s\}$
$s \mapsto t$	$\{f \mid f \in s \mapsto t \cap s \rightarrow t\}$
$s \dashrightarrow t$	$\{f \mid f \in s \dashrightarrow t \wedge ran(f) = t\}$
$s \twoheadrightarrow t$	$\{f \mid f \in s \twoheadrightarrow t \cap s \rightarrow t\}$
$s \twoheadrightarrow t$	$\{f \mid f \in s \twoheadrightarrow t \wedge s \twoheadrightarrow t\}$
$s \rightsquigarrow t$	$\{f \mid f \in s \rightsquigarrow j \cap s \twoheadrightarrow t\}$

A.5 Les substitutions généralisées

Substitutions de base. Soient x et y des variables, E et F des expressions, S un ensemble, P et R des prédicats et G et H des substitutions.

Description	Substitution	Sémantique
Simple	$x := E$	$[x := E]R \Leftrightarrow R[E/x]$
Identité	$skip$	$[skip]R \Leftrightarrow R$
Deviens élément de	$x \in S$	$[x \in S]R \Leftrightarrow [\@x'.x' \in S \Rightarrow x := x']R$
Deviens tel que	$x : P$	$[x : P]R \Leftrightarrow [\@x'.x' : P \Rightarrow x := x']R$
Multiple	$x, y := E, F$	$[x, y := E, F]R \Leftrightarrow R[E, F/x, y]$
Séquencement	$G; H$	$[G; H]R \Leftrightarrow [G][H]R$
Préconditionnée	$P \mid G$	$[P \mid G]R \Leftrightarrow P \wedge [G]R$
Gardée	$P \Rightarrow G$	$[P \Rightarrow G]R \Leftrightarrow P \Rightarrow [G]R$
Choix borné	$G \parallel H$	$[G \parallel H]R \Leftrightarrow [G]R \wedge [H]R$
Choix non borné	$\@z.(G)$	$[\@zz.(G)]R \Leftrightarrow \forall z.([G]R)$

Notation étendue. Soient x une liste de variables, y une variable libre dans E , E une expression, P et R des prédicats et G et H des substitutions.

Description	Substitution	Définition
Bloc	begin G end	G
Conditionnelle	if P then G else H end	$(P \Rightarrow G) \parallel (\neg P \Rightarrow H)$
Choix borné	choice G or H end	$G \parallel H$
Choix non borné	any x where P then G end	$\@x.(P \Rightarrow G)$
Sélection	select P then G ... when R then H end	$(P \Rightarrow G) \parallel \dots \parallel (R \Rightarrow H)$
Variable locale	var x in G end	$\@x.(G)$
Définition locale	let y be $y = E$ in G end	$\@y.(y = E \Rightarrow G)$

La spécification du système de passage à niveau

B.1 Le composant *Types.mch*

MACHINE *Types*

SETS

OBJECTS;
LIGHT_STATUS = {*light_red*,*light_yellow*,*light_green*};
BARRIER_STATUS = {*barrier_open*,*barrier_close*}

CONSTANTS

CONTROL,*BARRIER*,*LIGHT*

PROPERTIES

$CONTROL \subseteq OBJECTS \wedge$
 $BARRIER \subseteq OBJECTS \wedge$
 $LIGHT \subseteq OBJECTS$

END

B.2 Le composant *Basic.mch*

MACHINE *Basic*

SEES *Types*

VARIABLES

control,*light*,*light_status*,*barrier*,*barrier_status*,
controlLight,*controlBarrier*

INVARIANT

$control \subseteq CONTROL \wedge$
 $light \subseteq LIGHT \wedge$
 $light_status \in light \rightarrow LIGHT_STATUS \wedge$
 $barrier \subseteq BARRIER \wedge$
 $barrier_status \in barrier \rightarrow BARRIER_STATUS \wedge$
 $controlLight \in control \leftrightarrow light \wedge$
 $controlBarrier \in control \leftrightarrow barrier$

INITIALISATION

control := \emptyset || *light* := \emptyset ||
barrier := \emptyset || *light_status* := \emptyset ||
barrier_status := \emptyset || *controlLight* := \emptyset ||
controlBarrier := \emptyset

OPERATIONS

li \leftarrow *lightOfControl*(*co*) =

```

pre
   $co \in control$ 
then
   $li := controlLight(co)$ 
end;

 $ba \leftarrow barrierOfControl(co) =$ 
pre
   $co \in control$ 
then
   $ba := controlBarrier(co)$ 
end;

 $co \leftarrow controlOfLight(li) =$ 
pre
   $li \in light$ 
then
   $co := controlLight^{-1}(li)$ 
end;

 $co \leftarrow controlOfBarrier(ba) =$ 
pre
   $ba \in barrier$ 
then
   $co := controlBarrier^{-1}(ba)$ 
end;

 $light\_lightYellow(li) =$ 
pre
   $li \in light \wedge$ 
   $light\_status(li) = light\_green$ 
then
   $light\_status(li) := light\_yellow$ 
end;

 $light\_lightRed(li) =$ 
pre
   $li \in light \wedge light\_status(li) = light\_yellow$ 
then
   $light\_status(li) := light\_red$ 
end;

 $barrier\_barrierClose(ba) =$ 
pre
   $ba \in barrier \wedge barrier\_status(ba) = barrier\_open$ 
then
   $barrier\_status(ba) := barrier\_close$ 
end;

 $control\_close(co) =$ 
pre
   $co \in control \wedge$ 
   $light\_status(controlLight(co)) = light\_green \wedge$ 
then
   $light\_status(controlLight(co)) := light\_red \parallel$ 
   $barrier\_status(controlBarrier(co)) := barrier\_close$ 
end
END

```

Printing the status of Basic.mch

	NbObv	NbPO	NbPRi	NbPRa	%Pr
Initialisation	0	7	0	7	100
lightOfControl	10	0	0	0	100
barrierOfControl	10	0	0	0	100
controlOfLight	10	0	0	0	100
controlOfBarrier	10	0	0	0	100
light_lightYellow	8	2	0	2	100
light_lightRed	8	2	0	2	100
barrier_barrierClose	8	2	0	2	100
control_close	6	4	0	4	100
Basic	70	17	0	17	100

B.3 Le composant System.mch

MACHINE *System*

SEES *Types*

VARIABLES

control, light, light_status, barrier, barrier_status,
controlLight, controlBarrier

INVARIANT

control \subseteq *CONTROL* \wedge
light \subseteq *LIGHT* \wedge
light_status \in *light* \rightarrow *LIGHT_STATUS* \wedge
barrier \subseteq *BARRIER* \wedge
barrier_status \in *barrier* \rightarrow *BARRIER_STATUS* \wedge
controlLight \in *control* \leftrightarrow *light* \wedge
controlBarrier \in *control* \leftrightarrow *barrier*

INITIALISATION

control := \emptyset || *light* := \emptyset ||
barrier := \emptyset || *light_status* := \emptyset ||
barrier_status := \emptyset || *controlLight* := \emptyset ||
controlBarrier := \emptyset

OPERATIONS

controlClose_sys(*co*) =
pre
co \in *control* \wedge
light_status(*controlLight*(*co*)) = *light_green* \wedge
then
light_status(*controlLight*(*co*)) := *light_red* ||
barrier_status(*controlBarrier*(*co*)) := *barrier_close*
end

END

Printing the status of System.mch

	NbObv	NbPO	NbPRi	NbPRa	%Pr
Initialisation	0	7	0	7	100
control_close_sys	6	4	0	4	100
System	6	11	0	11	100

B.4 Le composant System_imp.imp

IMPLEMENTATION *System_imp*

REFINES *System*

SEES *Types*

IMPORTS *Basic*

OPERATIONS

control_close_sys(co) =

```

var li, ba in
  li ← lightOfControl(co);
  ba ← barrierOfControl(co);
  light_lightYellow(li);
  light_lightRed(li);
  barrier_barrierClose(ba)
end

```

end

END

Printing the status of System_imp.imp

	NbObv	NbPO	NbPRi	NbPRa	%Pr
ValuesLemmas	1	0	0	0	100
InstanciadedConstraintsLemmas	0	0	0	0	100
Initialisation	3	0	0	0	100
control_close_sys	11	5	0	3	60
System_imp	15	5	0	3	60

La spécification du système de contrôle d'accès

C.1 Le composant *Card.mch*

```

MACHINE Card
SETS PIN = {1, 2, 3, 4, 5, 6};
       CODE = {a, b, c, d, e};
VARIABLES
       cards
INVARIANT
        $cards \in PIN \rightarrow CODE$ 
INITIALISATION
       cards :=  $\emptyset$ 
OPERATIONS

createCard(pin, code) =
pre
        $pin \in PIN \wedge code \in CODE$ 
then
       cards :=  $cards \cup \{pin \mapsto code\}$ 
end
END

```

C.2 Le composant *Controller.mch*

```

MACHINE Controller
INCLUDES Reader, Door
VARIABLES
       authorised_cards,
       inside_cards
INVARIANT
        $authorised\_cards \subseteq cards \wedge$ 
        $inside\_cards \subseteq authorised\_cards$ 
INITIALISATION
       authorised_cards :=  $\emptyset$  ||
       inside_cards :=  $\emptyset$ 
OPERATIONS
bb  $\leftarrow isAuthorised(ca)$  =
pre
       ca  $\in cards$ 
then
       bb :=  $(ca \in authorised\_cards - inside\_cards)$ 
end;

```

```
enter(ca) =  
  pre  
    ca ∈ (authorised_cards - inside_cards)  
  then  
    inside_cards := inside_cards ∪ {ca}  
  end  
END
```

C.3 Le composant Reader.mch

MACHINE *Reader*

INCLUDES *Card*

SETS *READER_STATE* = {no_card_inside, card_inside, card_ejected}

VARIABLES

current_card,
state

INVARIANT

current_card ∈ *cards* ∧
state ∈ *READER_STATE*

INITIALISATION

current_card := ∅ ||
state := no_card_inside

OPERATIONS

```
ca ← insertCard(pin, code) =  
pre  
  pin ∈ PIN ∧ code ∈ CODE ∧ state = no_card_inside  
then  
  current_card := {pin ↦ code} ||  
  state := card_inside ||  
  ca := {pin ↦ code}  
end;
```

```
ejectCard =  
pre  
  state = card_inside  
then  
  current_card := ∅ ||  
  state := card_ejected  
end
```

```
takeCard =  
pre  
  state := card_ejected  
then  
  state := no_card_inside  
end
```

END

C.4 Le composant Door.mch

```
MACHINE Door
SETS
    DOOR_STATE = {open,close}
VARIABLES
    door_state
INVARIANT
    door_state ∈ DOOR_STATE
INITIALISATION
    door_state := close
OPERATIONS
openDoor =
    pre
        door_state = close
    then
        door_state := open
    end;
closeDoor =
    pre
        door_state = open
    then
        door_state := close
    end
END
```


RÉSUMÉ

Le couplage des approches orientées objets avec la méthode B est une piste pour l'amélioration de l'activité de spécification et de développement de logiciels. La méthode B fournit des notations et des outils supports puissants permettant de modéliser et de vérifier des modèles. Les approches objets fournissent des mécanismes intéressants pour la structuration et le développement de gros systèmes. L'apport de notre travail de thèse contribue aux activités de couplage entre ces deux formalismes en utilisant le prouveur de B pour valider et vérifier des spécifications UML.

En étendant les schémas de dérivation d'UML vers B proposés dans des travaux précédents réalisés dans l'équipe de recherche Dédale, nous proposons une approche de dérivation en B de méta-modèles UML, de diagrammes statiques et de diagrammes dynamiques. L'objectif de cette proposition est de vérifier la sémantique et la cohérence entre différents diagrammes de spécifications UML.

Notre thèse apporte aussi une contribution au développement de spécifications objets en utilisant la méthode B. La première proposition concerne la prise en compte de certains types d'associations entre classes lors de la dérivation en B. La deuxième proposition concerne la validation de spécifications orientées objets décrites à l'aide de diagrammes de séquence UML2.0.

Mots clés : UML, B, vérification formelle, spécification orientée objets, validation, obligations de preuve.

The coupling of object-oriented approaches with the B method makes improvement the activities of software specification and development. The B method provides notations for the specification and powerful tools, allowing to specify and verify models. The object-oriented approaches provide interesting mechanisms for the structuring and the development of large systems. The contribution of this thesis deals with the activities of coupling between these two formalisms by using the B provers to validate and verify UML specifications.

By extending the derivation of UML to B of preceding works realised in the Dedale research group, we propose an approach of the derivation to B of the UML meta-models, the static diagrams and the dynamic diagrams. The aim of this proposition is to check semantics and coherence between different diagrams of UML specification.

Our thesis brings also a contribution to the development of objects oriented specifications using B. The first proposition concerns the taking into account of some types of association between classes during the derivation to B. The second relates the validation of object-oriented specifications described by UML2.0 sequence diagrams.

Keywords : UML, B, formal verification, object-oriented specification, validation, proof obligations.