Université d'Evry Val d'Essonne

Mémoire présenté pour l'obtention d'une habilitation à diriger des recherches

Spécialité Informatique

Conception et développement formels d'applications bases de données

par Régine Laleau

soutenu le 10 décembre 2002 devant la commission d'examen composée de

Président :	**Gilles Bernot** (Université d'Evry Val d'Essonne)
Rapporteurs :	**Claude Chrisment** (Université Paul Sabatier, Toulouse) **Yves Ledru** (Université Joseph Fourier, Grenoble) **Michel Léonard** (Université de Genève)
Examinateurs :	**Corine Cauvet** (Université de Aix-Marseille 3) **Henri Habrias** (Université de Nantes) **Jacques Julliand** (Université de Besançon) **Véronique Viguié Donzeau-Gouge** (CNAM, Paris)

Remerciements

Je remercie chaleureusement Gilles Bernot, professeur à l'université d'Evry Val d'Essonne, d'avoir accepté de présider le jury de mon habilitation. Son avis extérieur et ses encouragements m'ont été bien précieux.

Je tiens ensuite à remercier Claude Chrisment, professeur à l'université Paul Sabatier de Toulouse, Michel Léonard, professeur à l'université de Genève, Yves Ledru, professeur à l'université Joseph Fourier de Grenoble, d'avoir accepté sans hésitation d'être rapporteurs sur mes travaux. Leurs remarques et commentaires m'ont été fort utiles, je les remercie vivement de l'intérêt qu'ils m'ont accordé.

Que Corine Cauvet, professeur à l'université d'Aix-Marseille 3, Henri Habrias, professeur à l'université de Nantes, et Jacques Julliand, professeur à l'université de Besançon, soient également remerciés pour leur participation à mon jury et l'intérêt qu'ils ont témoigné à mes activités.

Enfin toute ma gratitude va à Véronique Viguié Donzeau-Gouge, professeur au CNAM, et Catherine Dubois, professeur à l'IIE, qui m'ont accueillie dans leur équipe. Leurs conseils, leur soutien et leur amitié, tout au long de ces dernières années, ont contribué à l'aboutissement de ces travaux.

Les travaux présentés dans ce mémoire auraient certainement pris une toute autre forme s'ils n'avaient pas été menés en collaboration avec Philippe Facon, professeur à l'IIE. C'est ensemble que nous avons posé les fondements de mes travaux de recherche, au cours de nos nombreuses discussions certes animées mais très enrichissantes. Sa disparition prématurée m'a laissé démunie. Qu'il trouve ici l'expression de ma plus profonde gratitude.

Une partie des travaux présentés dans ce mémoire et articles sont le fruit d'une collaboration avec deux chercheurs étrangers : Marc Frappier, professeur à l'université de Sherbrooke au Québec, et Fiona Polack, professeur à l'université de York en Grande-Bretagne. A l'occasion de l'écriture de ce mémoire, je suis très heureuse de les remercier pour les discussions passionnantes et fructueuses que nous avons eues. Travailler avec eux est un réel plaisir qui, je l'espère, va encore durer plusieurs années.

Je n'oublie pas non plus les étudiants que j'ai encadrés ou co-encadrés en DEA ou thèse : c'est eux qui nous permettent d'avancer en recherche. Plus particulièrement, je remercie les anciennes thésardes : Nadira Lammari, Phuong Nguyen et Amel Mammar, et les "petits" nouveaux : Frédéric Gervais et Mario Richard.

Je tiens à remercier les membres de l'équipe CPR du laboratoire CEDRIC pour les échanges très enrichissants et les remarques constructives sur mes travaux ainsi que pour l'ambiance très chaleureuse de nos réunions.

Mes remerciements vont également à tous mes collègues et le personnel de l'IIE. Tous m'ont permis d'une manière ou d'une autre de travailler dans une ambiance bien sympathique. Une mention spéciale à Mireille Jouve et Brigitte Grau, mes co-locataires, qui, bien que le thème de mes travaux soit éloigné de leur préoccupation (surtout Brigitte), ont toujours su m'écouter et me donner de nombreux conseils forts pertinents ; j'ai beaucoup apprécié leur soutien indéfectible et leur constante bonne humeur.

Enfin, ma famille : mon père, ma mère, trop tôt disparue, mes sœurs et plus particulièrement Charly, Mélanie et Christian qui m'ont supporté, dans tous les sens du terme, ces derniers mois ; mais qui me font continuellement comprendre que sans eux, la vie serait bien triste.

Table des matières

1 Introduction **5**
1.1 Apport des méthodes formelles pour la conception des systèmes d'information 6
1.2 Caractéristiques des systèmes d'information et des méthodes de conception associées 8
1.3 Contributions du mémoire 10

2 Formalisation des concepts des modèles sémantiques de données **13**
2.1 Pourquoi la méthode B? 14
2.2 Formalisation en B des concepts de base des modèles sémantiques de données 15
2.2.1 Identité d'objet, Classe, Attribut, Clé 16
2.2.2 Association 17
2.2.2.1 Lien et association 17
2.2.2.2 Classe association 18
2.3 Structuration de la spécification B 19
2.3.1 Classe 19
2.3.2 Association 19
2.4 Génération des opérations de base 22
2.4.1 Principes généraux 22
2.4.2 Exemples d'opérations de base générées 22
2.4.3 Prise en compte des contraintes d'intégrité 24
2.5 Héritage et lien de généralisation/spécialisation 24
2.5.1 Définition 24
2.5.2 Structuration de la spécification 26
2.6 Conclusion 27

3 Langage formel pour la méta-modélisation des systèmes d'information **29**
3.1 Un méta-modèle formel pour les SI 30
3.1.1 Représentation graphique de la syntaxe abstraite 31
3.1.2 Représentation formelle de la syntaxe abstraite et de la sémantique statique . 32
3.1.3 Sémantique dynamique du méta-modèle 34
3.2 Instantiation du méta-modèle pour une application 36
3.3 Conclusion 39

4 Spécification des propriétés fonctionnelles des transactions **41**
4.1 Diagrammes d'états-transitions et de collaborations 42
4.1.1 Diagrammes d'états-transitions 42
4.1.2 Diagrammes de collaborations 45
4.2 Spécification des opérations dans le diagramme de classes 48

4.3 Architecture robuste . . . 51
4.4 Bilan et perspectives . . . 52

5 Raffinement vers une implémentation relationnelle 55
5.1 Raffinement en B . . . 56
5.2 Présentation générale de la méthode . . . 57
5.2.1 Description du processus de développement global . . . 57
5.2.2 Définition des machines abstraites représentant les relations . . . 58
5.3 Raffinement des données et opérations de base de la spécification . . . 59
5.3.1 Identification des objets . . . 61
5.3.2 Elimination des associations . . . 62
5.3.3 Passage d'un modèle fonctionnel à un modèle ensembliste . . . 64
5.3.4 Implémentation des données . . . 64
5.3.5 Commentaires . . . 65
5.4 Raffinement des transactions . . . 66
5.5 A propos des preuves : Correction du processus de raffinement . . . 66
5.5.1 Conditions de réutilisation . . . 67
5.5.2 Application au domaine des bases de données . . . 68
5.6 Conclusion . . . 69

6 Combinaison de spécifications pour la modélisation du comportement des SI 71
6.1 Aperçu de la méthode EB^3 . . . 71
6.2 Sur la spécification du comportement dans les SI . . . 74
6.3 Vérification de la cohérence des spécifications B et EB^3 par raffinement . . . 75
6.4 Discussion - Perspectives . . . 78

7 Construction systématique de hiérarchies de généralisation/spécialisation 79

8 Conclusion 81

Bibliographie 83

A Compléments sur le langage B 91
A.1 Principaux opérateurs relationnels . . . 91
A.2 Principales substitutions du langage B . . . 92
A.3 Clauses d'assemblage des composants B . . . 93

Chapitre 1

Introduction

On peut regretter que le développement de logiciels ne se fasse pas selon les règles traditionnelles utilisées dans les autres disciplines de l'ingénieur où l'on élabore d'abord des modèles mathématiques sur lesquels il est possible de raisonner avant de construire les produits. En effet, à l'heure actuelle, seuls les logiciels dits critiques (au sens où des vies humaines ou des intérêts économiques sont en jeu) sont développés selon des règles rigoureuses [ITS91, CC98] qui conduisent à des produits fiables et sûrs. Même si des efforts ont été faits durant les vingt dernières années pour décrire de manière abstraite les logiciels, pour normaliser les concepts, les méthodes actuelles de conception de logiciels les plus utilisées n'offrent pas cette rigueur. En conséquence, la plupart du temps, les coûts et délais de développement ne sont pas respectés; ceci est dû principalement à une analyse imprécise des besoins des utilisateurs (des limites fonctionnelles des projets) ce qui entraîne des erreurs dans la réalisation des produits, erreurs qui ne peuvent être détectées que tardivement, au pire durant les tests.

Les méthodes formelles de développement de logiciels, utilisées jusqu'à présent essentiellement pour les logiciels critiques, apportent une solution partielle à ce problème et de plus en plus d'industriels y croient [CGR95, BBFM99, SH01]. Elles sont passées en quelques années du stade projets de recherche au stade produits industriels et sont alors supportées par des outils. De plus en plus de formations universitaires les mettent à leur programme. Il est maintenant possible d'envisager d'étendre leur domaine d'application.

Dans ce mémoire, je considère l'application des méthodes formelles pour le développement des systèmes d'information (SI). Les SI sont devenus indispensables au bon fonctionnement de la plupart des organisations, qu'elles soient publiques ou privées, commerciales ou industrielles. Ils sont au coeur de nombreuses applications dans les domaines classiques de l'assurance, la banque, ou la gestion et plus récemment dans des applications scientifiques ou techniques. La compétitivité, l'efficacité de ces organisations dépendent en grande partie de la validité et du bon fonctionnement de leurs SI. Il est difficile de donner une définition simple de ce qu'est un système d'information. Schématiquement un SI est un système informatisé qui rassemble l'ensemble des informations présentes au sein d'une organisation, sa mémoire, ainsi que les activités qui permettent de les manipuler. Les types d'utilisateurs des SI sont très hétérogènes et des interfaces ad-hoc doivent être développées si on veut que les SI soient utilisés correctement et facilement. En règle générale, le volume de données manipulées par un SI est assez important et, de par leur implication dans des processus opérationnels étalés dans le temps, ces données sont nécessairement persistantes. Ceci implique l'utilisation de systèmes de gestion de bases de données. C'est plus précisément ce type de SI que je considère.

Dans la suite de l'introduction, je vais commencer par une discussion sur le rôle des méthodes formelles dans le développement du logiciel. Puis je donnerai une brève présentation des caractéristiques des SI qui me permettront d'analyser les différentes catégories de méthodes formelles selon cet angle spécifique. Enfin je présenterai les principales contributions de mon travail.

1.1 Apport des méthodes formelles pour la conception des systèmes d'information

Il est tout d'abord nécessaire de bien définir le vocabulaire utilisé. Un langage, ou notation, est dit formel s'il a une syntaxe et une sémantique formelles, formel signifiant ici fondé sur des bases mathématiques. Un langage est dit semi-formel s'il a seulement une syntaxe formelle. Il est dit informel si ni sa syntaxe ni sa sémantique ne sont formelles. Par abus de langage, une spécification écrite dans un langage formel est appelée spécification formelle. Par méthode on entend généralement une suite d'activités, une démarche, qui conduit à l'élaboration d'un résultat. Une méthode de conception (ou développement) de logiciels permet de développer des logiciels à partir d'une spécification. Toute méthode devrait être accompagnée d'un guide qui indique comment conduire ces activités. Je qualifie une méthode de formelle s'il est possible de développer et prouver une implémentation conforme à la spécification.

L'aspect formel permet la déduction et donc des calculs de preuves de propriétés, ce qui permet de raisonner au niveau de la spécification et de développer des outils d'assistance au développement de logiciels.

Ne perdons pas de vue que l'un des objectifs finaux du processus de développement de logiciels est de fournir un ensemble de programmes qui sont décrits de manière formelle, les langages de programmation étant par nature formels dès qu'ils possèdent une sémantique formelle. A l'opposé, le résultat de la phase d'analyse des besoins est le cahier des charges. C'est une première spécification du système à construire qui par nature est informelle, car sa caractéristique essentielle est d'être mise au point avec les clients, qui eux n'utilisent pas nécessairement un langage mathématique. Passer de l'un à l'autre ne se fait pas généralement en une seule étape et implique la participation de plusieurs types d'intervenants. La spécification obtenue lors de la phase d'analyse doit décrire le système qui sera implémenté. Elle est réalisée par des analystes et son objectif est double. D'une part elle doit être validée par les clients et d'autre part elle constitue le point de départ du développement informatique proprement dit, c'est donc une forme de contrat avec les développeurs. Les caractéristiques essentielles d'une bonne spécification sont qu'elle doit être concise, claire, non-ambiguë afin d'éviter le plus d'échecs dans la suite du développement. Elle doit être abstraite et à chaque fois que possible non-déterministe afin de laisser aux développeurs le choix de la meilleure implémentation, compte-tenu de problèmes de coûts, d'efficacité, d'utilisabilité ou de maintenance.

En ingénierie des SI, les méthodes de conception classiques utilisent pour décrire les spécifications des notations graphiques. Les avantages non négligeables de ces notations sont qu'elles sont facilement compréhensibles par les clients et relativement concises: la spécification obtenue est une bonne synthèse du système. Mais bien souvent elles restent globalement semi-formelles, la sémantique n'étant pas toujours précise : plusieurs interprétations sont possibles. De plus, comme on ne peut pas tout exprimer graphiquement, l'obtention d'une specification complète nécessite que le concepteur accompagne les diagrammes par des commentaires rédigés le plus souvent en langue naturelle qui ne sont ensuite exploitables ni pour effectuer des vérifications de cohérence ni pour dériver de façon automatique une partie de l'implémentation; ceci, on le comprend bien, est assez

décourageant même pour les bonnes volontés. Plusieurs problèmes résultent de cet état de fait. Le premier est qu'on ne peut faire qu'une validation partielle avec les clients, la spécification étant la plupart du temps incomplète. Les seuls outils de vérification qu'il est possible de développer permettent uniquement la vérification de la correction de la syntaxe. Il y a aussi un risque de mauvaise interprétation de la spécification par les développeurs sans possibilité bien souvent de la découvrir avant la phase de tests.

On peut envisager une utilisation des notations formelles à deux niveaux. Un premier niveau consiste à utiliser un langage de spécifications formelles. L'idée est de dire que plutôt que d'attendre de produire une description formelle du système au niveau programme, mieux vaut le faire le plus tôt possible dans les phases de développement, c'est-à-dire au niveau spécification. Ceci éliminera la plupart des problèmes posés précédemment. En effet, il est alors possible d'établir formellement *certaines* propriétés et contraintes que le système doit vérifier et s'assurer que la spécification est cohérente vis à vis d'elles. La restriction vient du fait qu'on ne peut pas être sûr que *toutes* les propriétés et contraintes ont été identifiées par les clients eux-mêmes. La validation peut cependant être plus complète avec les clients. Des prototypes peuvent être développés, des animateurs de spécifications formelles existent, on peut même envisager de paraphraser les notations formelles pour les rendre compréhensibles [DV97]. Enfin, le document fourni aux développeurs est non ambigu, évitant ainsi les erreurs d'interprétation. Cela suppose néanmoins que ces développeurs connaissent les langages formels, c'est alors une question de formation.

Une utilisation plus poussée des notations formelles, et qui constitue le deuxième niveau d'utilisation, est de considérer une méthode formelle, c'est-à-dire que les transformations successives qui conduisent de la spécification au programme, appelées raffinement, sont aussi formelles. De manière générale, le processus de raffinement est une tâche manuelle, lourde et qui nécessite un niveau d'expertise assez élevé. Si cela se justifie facilement pour les systèmes critiques, il est plus difficile de le faire admettre pour des SI quelconques. Il est alors absolument nécessaire d'accompagner la méthode formelle d'outils qui automatisent le plus possible le raffinement.

Remarquons que même avec le premier niveau d'utilisation il est possible de s'assurer que l'implémentation satisfait la spécification : on peut générer des jeux de tests à partir de la spécification. Les résultats obtenus en faisant exécuter l'implémentation sur ces jeux de tests sont comparés avec les résultats attendus par la spécification.

Les langages de spécifications formelles se différencient d'une part par leur support mathématique et d'autre part par les aspects et types de contraintes du système qu'ils modélisent.

Une classification généralement admise des langages formels conduit aux quatre catégories suivantes : algébrique, ensembliste, logique d'ordre supérieur et dynamique.

Avec l'approche algébrique, le principe de base est d'encapsuler les structures de données et les fonctions qui les manipulent en utilisant le concept de type abstrait de données ou type abstrait algébrique. Un type abstrait algébrique est décrit par trois éléments : une sorte, la signature des fonctions qui agit sur la sorte et les axiomes. Une sorte est un nom qui sert à représenter un ensemble de valeurs. Les axiomes décrivent les propriétés des fonctions sous forme d'équations. Parmi les langages de spécifications algébriques les plus connus, on peut citer OBJ [GMP83], LARCH [GHW85], PLUSS [Bid86, Gau92], LPG [BE95] et plus récemment CASL [oLD99].

Dans une approche ensembliste, l'état du système est modélisé par des ensembles ou des constructeurs ensemblistes et par des contraintes exprimées en logique du premier ordre. Chaque opération est spécifiée par pre- et postconditions ou par commandes gardées qui définissent son effet sur l'état du système. Les langages Z [Abr80, Spi92], VDM [Jon90] et B [Abr96] sont les plus représentatifs de

cette catégorie. Des approches objets ont récemment été développées [Lan95, AR97], on peut citer OBJECT-Z[DKRS91] et VDM++ [LG96].

Coq [HKPM95], PVS [ORS95] et HOL [GM93] sont des langages basés sur les logiques d'ordre supérieur. Ils sont utilisés essentiellement comme outils de preuve pour les programmes et permettent de réaliser des preuves complexes.

L'approche dynamique ou événementielle a été utilisée à l'origine pour spécifier des systèmes communicants et en particulier les protocoles. Le principe est de décrire les événements qui sont échangés entre le système et son environnement ou entre les différents processus qui composent le système. On peut alors modéliser les synchronisations, les séquences possibles d'événements, et détecter les interblocages. Plusieurs types de langages relèvent de cette catégorie : les systèmes de transitions étiquetées [Plo81, WN95], les réseaux de Pétri, les algèbres de processus comme CSP [Hoa85] ou les logiques temporelles [Eme90, Sti92].

Chacune de ces catégories est plus ou moins bien appropriée à la vérification de certains types de propriétés. Les propriétés fonctionnelles décrivent le système en fonction de ses entrées-sorties. Les propriétés comportementales prennent en compte les problèmes de gestion de la concurrence, de vivacité, ... De manière générale, les systèmes sont multi-facettes, mais on peut quand même distinguer deux grandes catégories de systèmes [vL00]: les systèmes transactionnels et les systèmes réactifs. Pour les premiers, l'accent est plutôt mis sur la spécification des propriétés fonctionnelles et les langages a priori les mieux adaptés sont les trois premiers types décrits précédemment. Pour les seconds, ce sont plutôt les propriétés comportementales qui sont considérées et l'approche dynamique semble la mieux adaptée. Il existe bien sûr des approches hybrides qui cherchent à combiner un langage de l'approche dynamique avec un langage de l'approche algébrique ou ensembliste comme RAISE [GHH+95] ou LOTOS [vEVD89].

Notons qu'il existe une classe de propriétés, regroupant entre autre la maintenabilité, la performance ou la confidentialité d'un système et couramment appelées propriétés non-fonctionnelles, qui ne sont pas (ou peu) prises en compte par les langages formels existants.

Afin de déterminer le langage de spécification le mieux approprié aux SI, il est nécessaire de définir au préalable les caractéristiques de ces systèmes.

1.2 Caractéristiques des systèmes d'information et des méthodes de conception associées

La complexité des données et leur volume important nécessitent d'avoir des structures de données bien adaptées aux SI. Les informations sont en général représentées par des types auxquels est associé un nombre important d'instances, ce qui nécessite l'utilisation des types ensemble ou liste. De plus ces types sont très fortement reliés entre eux. Ceci s'exprime d'une part par la nécessité de modéliser explicitement les liens entre les types, liens qui peuvent être eux-mêmes porteurs d'information, et d'autre part par l'utilisation de contraintes d'intégrité qui peuvent être statiques ou dynamiques. Les contraintes statiques imposent des restrictions sur les données stockées, par exemple la valeur d'un attribut est unique pour l'ensemble des instances d'un type, les contraintes dynamiques imposent des restrictions sur le cycle de vie des données, par exemple la valeur d'un attribut ne peut pas décroître.

Un SI peut se décomposer fonctionnellement en trois parties : l'interface graphique, le comportement fonctionnel et la base de données. L'interface graphique doit être bien adaptée aux différents types d'utilisateurs. La base de données est gérée en général par un système de gestion de bases

de données (SGBD) et le comportement fonctionnel est défini par un ensemble de programmes d'application définissant les transactions sur la base de données qui peuvent être de deux types : les transactions d'interrogation qui recherchent des informations spécifiques contenues dans la base de données, les transactions de mise à jour qui modifient les données stockées. Pour ces dernières, une caractéristique essentielle est qu'elles doivent respecter la cohérence des données exprimée par les contraintes d'intégrité, c'est-à-dire qu'une transaction qui part d'un état de la base qui vérifie les contraintes d'intégrité doit restituer, après son exécution, un état de la base qui vérifie aussi ces contraintes. La complexité de ces transactions provient essentiellement du grand nombre de contraintes et de relations entre les données qu'il faut prendre en compte, les algorithmes étant relativement simples. De plus, comme le SI est partagé par plusieurs utilisateurs, l'exécution des transactions doit être concurrente sur les mêmes données, sans pour autant permettre la perte de données ou l'apparition d'incohérences.

Les SGBD existants offrent tous des mécanismes évolués qui assurent le contrôle de la concurrence, le stockage efficace des données et des langages d'interrogation qui permettent des recherches sophistiquées sur le contenu de la base. Le modèle relationnel de Codd [Cod70] est à l'heure actuelle le standard *de-facto* utilisé comme modèle de données dans la plupart des SGBD commercialisés, même si on assiste à l'émergence de modèles de données relationnel-objet (Oracle V8, ...) [BGV98, EN00]. Les SGBD purement orientés objets n'ont pas réussi à émerger, essentiellement à cause de problèmes de performances et d'absence de fondements théoriques du modèle objet supporté, contrairement au modèle relationnel basé sur la théorie des ensembles. Classiquement, l'architecture opérationnelle d'un SI repose sur un serveur (ou plusieurs) supportant un SGBD relationnel, qui gère l'exécution des transactions. Les programmes d'application et l'interface graphique sont développés sur des clients. La communication entre clients et serveurs est réalisée via un réseau en utilisant des requêtes de type SQL. En ce qui concerne les contraintes d'intégrité, seules les contraintes d'intégrité statiques sont gérées par les SGBD actuels, les techniques nécessaires à la prise en compte des contraintes dynamiques ne sont pas encore au point [EN00].

En conception de SI, l'accent a plutôt été mis sur la conception de la base de données, car celle-ci a toujours été considérée comme le pivot du futur SI. Ceci est encore renforcé par le fait que le modèle relationnel ne décrit que les données et que les SGBD relationnels ne prennent pas en compte les programmes d'application, ou seulement de façon récente et très partielle. Classiquement, les différentes phases de développement telles que préconisées par le rapport ANSI/SPARC/X3 [ANS75] sont l'élaboration du modèle conceptuel de données qui est indépendant du SGBD, la conception logique qui vise un SGBD cible et la conception physique qui permet de choisir la meilleure organisation physique des données et les meilleurs chemins pour y accéder. Logiquement cette dernière phase tient compte des transactions qui accèdent aux données. De nombreux travaux de recherche ont proposé différents types de modèles de données conceptuels, appelés aussi sémantiques [HK87, PM88]. Parmi les pionniers, on peut citer le modèle Entité-Association de Chen [Che76] et le modèle relationnel binaire de Abrial [Abr74] qui sont à la base des différents modèles de données proposés par les méthodes de conception comme Merise, OMT ou plus récemment UML. De nombreux modèles formels ont été définis sur des extensions de ces deux modèles [AH87, NH89, Hai90, BCN92, PS92]. Le caractère formel de ces modèles permet de définir des règles de dérivation d'un schéma conceptuel vers un schéma relationnel [Cer83]. Tous les outils de conception existants automatisent ce passage.

En revanche très peu de travaux de recherche ont porté sur la spécification formelle des transactions. L'argument était : cela relève plus du domaine du génie logiciel que de la conception de SI [EN00]. Dans les méthodes commercialisées, les transactions sont spécifiées à l'aide des formalismes graphiques des réseaux de Pétri (Merise) ou des diagrammes états-transitions (OMT, UML) mais

le caractère formel a disparu. Les règles actives [WC95, BGV98], extensions des triggers utilisés dans les SGBD relationnels, permettent de décrire les réactions du système à l'arrivée de certains événements. Elles ont principalement été utilisées pour la vérification et la validation de contraintes d'intégrité mais ne permettent pas de décrire un SI au niveau spécification de manière complète. Les quelques essais de définition de langages formels pour la spécification des transactions (Galileo [ACO85], Taxis [MBW80], Herm [Tha89]) sont restés au stade de prototypes de recherche. Les travaux autour du projet DAIDA [SSW91] sont certainement les plus poussés. Cependant le modèle sémantique considéré est de plus bas niveau que les modèles ci-dessus. Dans [Sch94], K. D. Schewe définit un nouveau langage de spécifications formelles dédié aux applications bases de données. Il est basé sur la logique intuitioniste et cherche à combiner des spécifications de type algébrique avec des spécifications de modules dans le style des machines abstraites de B. L'inconvénient majeur, comme toute définition de nouveau langage, est qu'il faut développer complètement tous les outils associés. Depuis peu, une partie de la communauté bases de données commence à considérer que les SI sont des logiciels comme les autres et forment un tout et donc qu'il est nécessaire de tenir compte de la spécification des transactions pour réaliser une bonne conception de la base de données. Certains préconisent même l'utilisation de langage de spécifications formelles, au moins pour les fonctions critiques d'un SI dont il faut garantir la bonne exécution [EN00].

1.3 Contributions du mémoire

Je me suis intéressée à l'aspect programmation des bases de données dès mes travaux de thèse qui ont consisté à définir un langage de programmation pour bases de données intégrant Pascal et un langage de requêtes "à la SQL" [Mao86]. Rejoignant l'équipe "Conception de Systèmes d'Information" du laboratoire CEDRIC, je me suis tournée vers la conception de programmes utilisant des bases de données. L'idée initiale de mon travail était d'offrir aux concepteurs un moyen de décrire formellement les transactions d'un SI au même niveau d'abstraction que les données. L'objectif est alors, d'une part, de pouvoir vérifier la cohérence du système dès le niveau conceptuel et d'autre part, de pouvoir dériver les programmes correspondant à ces transactions, de la même manière que le schéma d'une base de données est dérivé à partir du modèle conceptuel des données. Ces deux activités doivent être assistées par des outils qui automatisent la plus grande part possible.

Si on considère les trois éléments fonctionnels d'un SI présenté ci-dessus, je n'ai pas étudié la conception de l'interface graphique. C'est un axe de recherche à part entière même si des possibilités existent pour générer des interfaces standards à partir de spécifications. En ce qui concerne la partie base de données, mon objectif n'est pas de proposer une nouvelle méthode de conception, car comme je l'ai dit dans la section précédente, c'est déjà une activité de recherche passée, à moins de prendre en compte de nouveaux concepts comme les entrepôts de données ou les systèmes d'information liés au web [Mét99].

L'expérience de définition de nouveaux langages n'étant pas concluante, l'idée était de reprendre un langage formel existant qui a déjà fait ses preuves et qui est outillé. D'autre part, il me paraissait opportun de réutiliser le plus possible ce qui existe au niveau de la conception des données, qui est plutôt bien formalisé et qui est utilisé dans les méthodes de conception existantes (Merise, OMT, UML).

La première étape du travail a consisté à définir un noyau du modèle sémantique de données incluant les concepts les plus répandus et dont la sémantique est relativement bien définie et fait l'objet d'un large consensus dans la communauté. C'est un prérequis pour pouvoir choisir le langage formel

le mieux adapté à notre problématique. En collaboration avec l'équipe "Spécification et analyse du logiciel" du CEDRIC, dirigée par Philippe Facon, nous avons ensuite étudié plusieurs types de langages de spécifications formelles. Le choix s'étant finalement porté sur la méthode B, les concepts retenus ont été formalisés en B. Plus précisément, il existe deux approches possibles pour cette formalisation. L'une est implicite, par simulation des concepts, c'est-à-dire en définissant des règles de traduction qui donnent directement pour un diagramme la spécification B correspondante. L'autre est explicite, par formalisation du méta-modèle, puis instanciation de ce méta-modèle formalisé pour obtenir la spécification B d'un diagramme. Ces deux approches ont été étudiées et finalement c'est la première qui a été retenue comme je le justifierai dans le chapitre 3 de ce mémoire. Néanmoins, la deuxième approche a servi à définir les bases d'un outil support de la traduction d'une représentation graphique vers une représentation textuelle. Nous l'avons également développée récemment avec Fiona Polack de l'Université de York en Grande-Bretagne pour permettre d'une part de fournir une aide au concepteur pour la mise au point de ses opérations et d'autre part pour comparer notre approche avec les travaux du groupe pUML [CE97, CEKS01]qui propose une formalisation en Z ou en OCL du méta-modèle de UML.

Pour la spécification des transactions, plusieurs alternatives sont possibles. La première est de compléter directement la spécification B obtenue à partir du diagramme des données. L'inconvénient est qu'il n'y a plus une représentation homogène des différents éléments d'une spécification : certains sont représentés graphiquement et textuellement, d'autres uniquement textuellement. Ceci pose un problème d'évolution et de maintenance de la spécification, ce qui est assez gênant quand on sait qu'une spécification est rarement obtenue en une seule étape et qu'on procède plutôt de manière itérative et incrémentale.

Les deux autres solutions sont complémentaires. L'une consiste à considérer le diagramme de données comme un diagramme de classes (représentation objet de OMT ou UML) et de définir de nouvelles classes non plus comme support de données mais comme support pour définir les transactions. Les transactions sont alors écrites en utilisant le langage B. L'autre méthode consiste à reprendre les diagrammes des méthodes de conception classiques qui sont utilisés pour exprimer le comportement d'un système comme les diagrammes états-transitions, les diagrammes de séquences ou les diagrammes de collaborations. Contrairement aux diagrammes de données, ceux-ci ont une sémantique très imprécise et, selon le domaine d'application considéré, peuvent avoir des interprétations différentes. Un des apports du travail a été de définir les seuls concepts de ces diagrammes utiles pour la spécification de transactions et de leur donner une sémantique formelle. L'avantage de cette dernière solution est qu'elle offre une représentation de la spécification formelle permettant deux niveaux de lecture. La représentation graphique donne une bonne synthèse du système permettant d'avoir une vue d'ensemble des fonctionnalités du système. On peut alors se focaliser sur les détails de ces fonctionnalités en considérant la représentation textuelle sous forme de spécifications B. Ces travaux ont, en partie, été menés avec Phuong Nguyen [Ngu98] pendant sa thèse.

Un prototype supportant ces travaux a été réalisé. Il prend en entrée les diagrammes élaborés en utilisant l'atelier ROSE dédié à UML et les traduit en spécifications B. Ces spécifications peuvent être ensuite passées à l'Atelier B pour analyse et preuve. Il est écrit en OCaml. Il a été repris par une équipe de l'Inrets qui souhaite l'intégrer dans une plate-forme logiciel libre proposant différents composants ouverts dédiés à la méthode B [Mar01].

Une suite logique de ces travaux concerne la génération d'une implémentation relationnelle à partir des spécifications B obtenues. De manière générale, le processus de raffinement demeure une tâche manuelle, relativement lourde, en particulier en phase de preuve. Le but est de concevoir un assistant, dédié au domaine des applications bases de données, qui réduit le plus possible le

coût élevé de cette phase en automatisant la plus grande partie. Pour cela, et dans le cadre de la thèse d'Amel Mammar [Mam02], un ensemble de règles génériques de raffinement a été proposé. Les règles prennent en compte le diagramme de données et les opérations élémentaires sur ces données et permettent la définition du schéma de la base de données et du code des différentes opérations SQL (insertion et suppression). Ces règles ont été prouvées donnant lieu à des scénarios de preuves génériques et donc réutilisables par instanciation. L'implémentation ainsi obtenue est prouvée formellement correcte par rapport à la spécification initiale. L'outil décrit précédemment a été complété pour prendre en compte cette nouvelle étape de raffinement.

Récemment, je me suis intéressée à d'autres catégories de méthodes formelles que les méthodes ensemblistes et qui permettent de prendre en compte des propriétés du comportement complémentaires de celles de B. C'est un travail que je mène en collaboration avec Marc Frappier de l'Université de Sherbrooke au Québec. En effet, en B le comportement d'un SI est décrit par transitions d'états. Le langage que je considère, basé sur CSP et LOTOS décrit les enchaînements d'événements admissibles. Nous étudions plus particulièrement le problème de la vérification de la cohérence des deux spécifications entre elles.

Le mémoire est organisé de la façon suivante. Le chapitre 2 présente la démarche suivie pour choisir un langage de spécifications formelles adapté au SI : définition du noyau de base des concepts du modèle sémantique de données, étude des différents langages formels et choix d'un de ces langages, formalisation en B des concepts. Le chapitre 3 illustre l'approche par méta-modélisation. Le chapitre 4 aborde la spécification des transactions. Le chapitre 5 traite du raffinement vers une implémentation relationnelle. Le chapitre 6 fait le point sur la combinaison de différents styles de spécifications pour la modélisation du comportement des SI. Le chapitre 7 constitue une petite enclave aux différents points traités jusque là, à savoir l'utilisation de méthodes formelles existantes pour la spécification de SI. Partant du constat que la spécification des hiérarchies de lien de généralisation/spécialisation dans un schéma conceptuel de données reste une tâche très approximative, je me suis intéressée lors de la thèse de Nadira Lammari [Lam96] à la définition d'une méthode qui permet la construction systématique de ces hiérarchies d'héritages. Les notations et clauses du langage B utilisées dans ce mémoire sont présentées en annexe.

Chapitre 2

Formalisation des concepts des modèles sémantiques de données

Dans un système d'information tel que défini dans l'introduction, la représentation des données est primordiale. Dans toutes les méthodes de conception existantes, la spécification qu'on retrouve immanquablement est celle des données, et c'est d'ailleurs souvent la seule qui offre un certain degré de formalisation. C'est pour ces raisons que nous avons envisagé cet aspect là d'un système en premier. Au moment où Philippe Facon et moi-même avons commencé, des travaux sur la formalisation des concepts des méthodes classiques de conception comme SA ou OMT existaient déjà [Hal90, PWM93, LH94a]. Cependant, il n'était pas clair de savoir à quels types de systèmes ils s'adressaient. Il nous a donc paru opportun de dégager d'abord les aspects des SI que le modèle formel retenu devait représenter de manière naturelle et donc simple et non pas en introduisant des artifices de codage.

Avant de présenter ces aspects, il est nécessaire de préciser comment les données d'un SI sont spécifiées dans les modèles sémantiques de données. Un SI est représenté par un ensemble de classes (ou entités), chaque classe gérant un ensemble d'objets. Les objets sont connectés entre eux par des liens d'associations ou d'héritage. Des propriétés supplémentaires sur les données sont exprimées par les contraintes d'intégrité qui peuvent être relativement nombreuses. Rappelons que le volume des données gérées est très important. Les transactions sont elles aussi nombreuses mais leur complexité algorithmique est faible. Nous nous intéressons à la spécification des propriétés fonctionnelles du SI : le langage choisi doit permettre de vérifier que les transactions spécifiées satisfont bien les contraintes d'intégrité du SI.

Les aspects à considérer sont les suivants:

(i) Les transactions et les contraintes d'intégrité portent sur des ensembles d'objets existants et les liens qui les connectent. Il est donc important de pouvoir définir explicitement l'état d'un SI comme étant constitué de ces objets et liens.

(ii) Les associations permettent une description abstraite de liens entre les objets. Des contraintes d'intégrité peuvent être définies, soit intra-association, soit inter-association. Une association a donc bien une sémantique propre, il est alors nécessaire de considérer ce concept à part entière.

(iii) Il est important de définir des identités d'objet indépendamment de leur valeur, et donc du concept de clé comme dans le modèle relationnel.

(iv) Un SI contenant un grand nombre de données et d'opérations pour les manipuler, les modèles sémantiques utilisés pour les représenter offrent des mécanismes de structuration, mis en place par les classes, les associations et l'héritage. Cette structuration doit être respectée par le

modèle formel pour faciliter le passage entre les deux représentations et rendre la traduction lisible.

(v) Les ingénieurs en SI sont habitués à travailler avec des méthodes de conception, supportées par des outils, qui permettent la spécification et la génération automatique de schémas de bases de données. Le modèle formel doit être associé à une méthode de développement complète accompagnée elle aussi d'outils.

2.1 Pourquoi la méthode B?

Nous avons analysé les différentes catégories de langages de spécifications formelles présentés dans l'introduction au regard des propriétés que nous voulions considérer et des cinq critères présentés ci-dessus.

Les langages de l'approche algébrique sont certainement ceux qui ont les bases théoriques les plus poussées. Malheureusement ils ne se prêtent pas facilement à la spécification de SI, essentiellement car leur paradigme est assez éloigné de la représentation naturelle qu'on peut se faire d'un SI. En effet, elles n'ont pas de notion d'état, ce qui ne permet pas de construire un modèle explicite du système. De ce fait, les éléments modifiés par une opération doivent obligatoirement faire partie des paramètres d'entrée et de sortie de l'opération. De même, comme il n'y a pas de représentation des objets existants, on ne peut pas faire la différence entre les opérations sur les types (entiers, booléens, ...) et celles sur ces objets existants. Les contraintes d'intégrité ne s'expriment pas non plus très naturellement. Ces différentes constatations ont été étayées par plusieurs études de cas que nous avions demandées à des élèves-ingénieurs de l'IIE et qui ont fait que nous ne les avons pas retenues. On peut cependant noter que plusieurs travaux de recherche sont en cours, en particulier autour de CASL, pour faciliter la description et l'axiomatisation de la notion d'état.

De part leur domaine d'applications, les langages de l'approche logique ne nous ont pas paru bien adaptés. Ils relèvent encore du domaine de la recherche et sont essentiellement utilisés pour réaliser des preuves complexes. De plus, je ne pense pas qu'on ait besoin de toute la puissance d'expression de ces langages pour spécifier nos besoins.

Au regard des propriétés que nous voulions vérifier, nous n'avons pas considéré les langages de l'approche dynamique.

Les méthodes de l'approche ensembliste sont celles qui se prêtent le mieux à nos besoins. Elles sont caractérisées par la notion d'état. Un état est défini par des ensembles et des relations entre les ensembles. Il lui est associé un prédicat qui exprime des contraintes sur ces ensembles et relations. Ces méthodes reposent sur des bases théoriques simples: les notions d'ensembles, relations et fonctions sont des concepts mathématiques connus dont l'apprentissage est relativement aisé (si on ne considère pas les preuves). Ces notions ont déjà été utilisées pour formaliser les modèles sémantiques de données : une des premières tentatives d'abstraction des modèles de données a été proposée par Abrial [Abr74] avec le modèle relationnel binaire. La spécification formelle obtenue est relativement proche, au niveau de l'expressivité, d'une formulation graphique du type Entité/Association. Toutes ces méthodes proposent des mécanismes de structuration : modules, machines abstraites ou schémas. La plupart d'entre elles sont outillées.

Plusieurs méthodes relevant de l'approche ensembliste pouvaient convenir. Nous avons éliminé les méthodes formelles orientées objets comme VDM++ ou Object-Z. D'abord, quand nous avons commencé notre recherche, elles étaient relativement jeunes et bien souvent en cours de développement [LH94b]. Ensuite, et surtout, les points (i) et (ii) ne sont pas pris en compte de manière

naturelle [AR97, Dup00] ; en fait, elles sont mieux adaptées à la spécification de programmes écrits en langages à objets, or nous considérons la spécification de bases de données. Finalement, il reste les méthodes "classiques" telles VDM, Z et B. Nous avons éliminé Z car, comme le dit leur auteur commun [Abr95]: "Z est plus une notation pour écrire des spécifications qu'une méthode de développement complète". De plus il y avait, à l'époque, peu d'outils associés et c'étaient plutôt des prototypes de recherche. Ce n'était à priori pas évident de choisir entre VDM et B, c'est pourquoi nous les avons étudiées de manière plus approfondie [FL93, LHR95, FL96]. Les conclusions que nous avons tirées sont les suivantes:

- le modèle de types de B ("given sets", richesse de la notation relationnelle) pousse naturellement à adopter une approche où on sépare les identités d'objet de leurs caractéristiques. Certes un style similaire est possible en VDM, mais ce n'est pas le style habituel des spécifications VDM. A cet égard, B apparaît comme plus "orienté objet" et VDM plus "orienté valeur".
- l'homogénéité des constructeurs de B lui confère une grande souplesse: une séquence est une fonction, une fonction est une relation, une relation est un ensemble. De plus les expressions obtenues sont beaucoup plus concises.
- l'inclusion du typage dans l'invariant fournit des spécifications également plus concises en B qu'en VDM. En VDM une fonction est définie sur des types de données, il faut alors préciser son domaine et son codomaine de définition. En B, une fonction est définie sur le domaine et le codomaine, son type est inféré. Par exemple, supposons qu'on veuille modéliser un ensemble de commandes avec pour chacune un numéro. En VDM on devra dire d'une part que numéro est une fonction de type: $COMMANDE \rightarrow NAT$ puis (dans l'invariant) que le domaine de numéro est égal à l'ensemble *Commande* des commandes existantes ; en B on écrit uniquement que numéro est une fonction totale de *Commande* dans NAT (ce qui suffit pour typer). Cette situation est très courante en systèmes d'information car la plupart des contraintes s'expriment entre des objets existants.
- la notion de machine abstraite B fournit un moyen de structuration mieux adapté aux SI que la notion de module de VDM.
- la méthode B est supportée par des outils industriels : l'AtelierB [Cle] et le BToolkit [BC].
- Enfin le contexte de travail a fait que nous avions plus de compétences autour de B: J.R. Abrial intervenait à ce moment-là dans les enseignements du CNAM.

Depuis, cette étude a été corroborée par le livre [FH00]. Une étude de cas relevant du domaine des SI (un système de gestion de commandes) est spécifiée en utilisant quinze méthodes de spécification différentes qui constituent un échantillon complet des types (paradigmes) de méthodes existantes. Même si, comme le soulignent les auteurs du livre, le cas d'étude est simple et qu'on ne peut pas se contenter d'une seule étude pour en déduire un classement des méthodes, il apparaît très clairement que selon les caractéristiques du système et le type de propriétés qu'on veut modéliser telle méthode sera mieux adaptée. Il est donc fondamental de définir ce qu'on souhaite modéliser avant de choisir la méthode.

2.2 Formalisation en B des concepts de base des modèles sémantiques de données

L'objectif du travail n'était pas de proposer une nouvelle formalisation des modèles sémantiques de données. C'est un domaine qui a été largement étudié dans les années 80. Nous nous sommes donc appuyés sur plusieurs formalisations existantes de modèles sémantiques [Abr74, MBW80, HM81,

Shi81]. En revanche, les concepts des diagrammes de classes des méthodes OMT et UML, ou le modèle conceptuel des données de Merise, méthode très utilisée en France, n'ont pas de définition formelle, bien qu'elles se disent issues du modèle Entité/Association. Ce que nous nous sommes proposé de réaliser c'est de reprendre les notations graphiques de ces méthodes et de donner une définition formelle de leurs concepts en B. Dans la première version du travail, nous avions adopté une notation graphique dans le style de Merise. Comme depuis UML a été normalisé par l'OMG et devient le standard de facto, je préfère dans ce mémoire reprendre ses notations graphiques.

Je ne vais pas présenter ici tous les concepts que nous avons étudiés. Les principaux concepts sont présentés dans [FL96] et une étude détaillée se trouve dans [Ngu98]. Les concepts que j'ai retenus sont les concepts de base, à savoir : objet, classe, association et contraintes de multiplicité, lien de généralisation/spécialisation.

2.2.1 Identité d'objet, Classe, Attribut, Clé

La notion d'objet que nous utilisons relève de la modélisation du monde réel et de la représentation des connaissances. C'est donc une notion très large qui recouvre toute abstraction du monde réel qui peut être identifiée comme pertinente pour l'application ou plutôt le domaine d'application considéré. Le concept de classe est lui utilisé pour structurer les connaissances sur les objets : une classe regroupe les objets partageant des caractéristiques (attributs et opérations) communes (client, commande, produit,...). Un objet est une instance d'une classe.

Un aspect absolument fondamental de la modélisation par objets est la notion d'identité d'objet (point (iii)) : les objets existent indépendamment des différentes valeurs qui leur sont attachées. Une personne peut changer de nom, d'adresse, de numéro de téléphone, ... il s'agit toujours de la même personne. Une spécification formelle fidèle à l'approche par objets doit donc bien séparer l'existence des objets de leurs caractéristiques.

En fait, la formalisation de l'existence des objets (point (i)) ne requiert que des ensembles d'identités d'objet, sans aucune structure et donc pas d'autre opérateur que l'égalité. Cela correspond typiquement aux "given sets" de B, ensembles donnés sans aucune structure pour leurs éléments. La plupart des contraintes et des opérations portant sur des ensembles d'objets existants, il est important de les considérer à part entière.

Règle F-1
chaque classe e donne lieu à :
-l'ensemble des (identités d') objets possibles de e : given set Se
-l'ensemble des (identités d') objets existants de e : variable Ve
-l'invariant : $Ve \subseteq Se$

Un attribut est défini par son nom et son type, qui définit l'ensemble des valeurs possibles. Une valeur d'attribut ne peut être un identifiant d'objet (car c'est le rôle des associations d'exprimer de tels liens entre objets). Les types des attributs sont des types possibles en B : ils peuvent être simples (entiers, booléens, chaînes de caractères), complexes (en utilisant les constructeurs ensembles, séquences,...) ou non définis au niveau de la spécification (cela laisse la possibilité au spécifieur de différer la définition exacte du type lors du raffinement).

La formalisation des attributs se fait naturellement à l'aide de relations et fonctions entre les ensembles d'identités d'objet et les types des attributs. On obtient donc :

Règle F-2
chaque attribut a de type Ta de la classe e donne lieu à :
- la relation définissant les valeurs de a pour les objets de e : variable Va
- l'invariant de typage de Va : $Va \in Ve \leftrightarrow Ta$ (cas le moins contraint)

Selon que l'attribut est obligatoire ou optionnel, multi ou mono-valué, la relation peut devenir une fonction partielle ou totale. Si Ta est un type non-défini, on définit en B un given set.

Le concept de clé est formalisé par un invariant exprimant que les objets sont en injection avec leurs clés : si (a1, a2) est une clé, alors nous avons : $Va1 \otimes Va2 \in Ve \rightarrowtail Ta1 \times Ta2$.

Tout au long du mémoire, je vais m'appuyer sur un exemple, la gestion (simplifiée) d'un club video. Le diagramme des classes est représenté dans la figure 2.1. Un client peut emprunter des cassettes s'il n'est pas suspendu. Un objet de la classe *Cassette* représente non pas l'objet physique mais son type. L'attribut *Nbdispo* permet de connaître le nombre de copies de cassette disponibles pour l'emprunt. Une cassette appartient à une boutique. La notation $\{K\}$ associée à un attribut (ou un groupe d'attributs) signifie que l'attribut (ou le groupe d'attributs) est une clé.

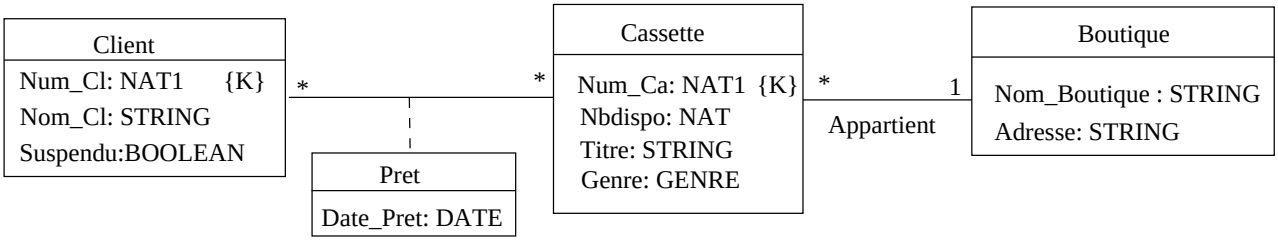

FIG. 2.1 – *Un exemple de diagramme de classes : le cas Club Video*

Prenons l'exemple de la classe *Cassette*, nous obtenons la spécification B suivante :

SETS
$CASSETTE$; $GENRE$;
VARIABLES
$Cassette$, Num_Ca, $Nbdispo$, $Titre$, $Genre$
INVARIANT
$$\begin{aligned} Cassette &\subseteq CASSETTE \ \wedge \\ Num_Ca &\in Cassette \rightarrowtail NAT1 \ \wedge \\ Nbdispo &\in Cassette \rightarrow NAT \ \wedge \\ Titre &\in Cassette \rightarrow STRING \ \wedge \\ Genre &\in Cassette \rightarrow GENRE \end{aligned}$$

2.2.2 Association

2.2.2.1 Lien et association

Le concept d'association est un concept très important dans les SI et nous y avons attaché un intérêt particulier (point (ii)). Il existe des liens entre objets, par exemple un lien de type *Appartient* entre une cassette et une boutique. Un type de lien est appelé association. Nous ne considérons que les associations binaires, c'est-à-dire qu'un lien concerne exactement deux objets. Ce n'est pas une restriction car une association n-aire peut toujours se ramener à une association

binaire en rajoutant des contraintes d'intégrité sur les associations et les classes concernées. Pour une association donnée, un lien est totalement identifié par la paire d'objets connectés par ce lien. Une association est caractérisée par son nom et ses contraintes de multiplicité avec les valeurs (nombre minimum..nombre maxixum de liens pour chaque objet) 1..1 (abrégé en 1) ou 1..n ou 0..1 ou 0..n (abrégé en *) ou m..n (n et m sont des nombres entiers représentant des multiplicités quelconques avec $n \geq m > 1$).

Comme une instance d'association (i.e. un lien) est identifiée par la paire d'objets connectés, il n'est pas nécessaire de fournir un ensemble des instances possibles comme pour une classe. La formalisation la plus directe d'une association est donc une relation entre les ensembles d'identités d'objets existants. Nous obtenons donc :

Règle F-3
chaque association r entre les classes e et f donne lieu à :
- une variable Vr, ensemble des paires d'instances de e et f liées par r
- l'invariant de typage : $Vr \in Ve \leftrightarrow Vf$ (cas le moins contraint)

Selon les multiplicités, cette relation, ou son inverse Vr^{-1}, peut être une fonction partielle, une injection, etc.

L'association *Appartient* du diagramme précédent donne :

VARIABLES
Appartient
INVARIANT
$Appartient \in Cassette \rightarrow Boutique$

2.2.2.2 Classe association

Dans la plupart des modèles sémantiques existants, les associations ne peuvent pas posséder d'attributs ni être elles-mêmes membres d'associations ou de liens de généralisation/spécialisation. On transforme alors systématiquement l'association en classe, avec des contraintes d'intégrité supplémentaire. En UML, on peut utiliser le concept de classe association. Dans nos premiers travaux, nous avions introduit le concept d'entité associative qui est très proche. Dans l'exemple de la figure 2.1, *Pret* est une classe association car elle possède un attribut.

La formalisation en B d'une classe association dépend de ses relations avec d'autres éléments du schéma.

Règle F-4.1
chaque classe association qui ne participe pas à d'autres associations ou à des liens de généralisation/spécialisation est modélisée comme une association classique en utilisant la Règle F-3 et ses attributs sont modélisés de la même manière que les attributs d'une classe en utilisant la Règle F-2.

C'est le cas de la classe association *Pret* de notre exemple.

Dans les autres cas, ce type de modélisation rendrait la spécification et l'utilisation de la classe association très lourde. Nous avons plutôt choisi de modéliser une classe association par une classe, deux associations qui la relie aux deux classes associées et un invariant supplémentaire qui traduit qu'une classe association est une association, c'est-à-dire que chaque couple d'instances des classes associées à la classe association détermine au maximum une instance de la classe association. Donc

l'ensemble d'instances de la classe association est en bijection avec une relation entre les ensembles d'instances des classes associées.

Règle F-4.2
- chaque classe association c définie entre les classes e_1 et e_2 et qui ne vérifie pas les conditions de la Règle F-4.1 est modélisée en utilisant la Règle F-1.
On obtient un given set Sc, une variable Vc et un invariant $Vc \subseteq Sc$.
- 2 associations r_1 et r_2 sont définies. Elles associent c respectivement aux classes e_1 et e_2.
Elles sont modélisées en utilisant la Règle F-3. On obtient 2 variables Vr_1 et Vr_2
et l'invariant $Vr_1 \in Vc \rightarrow Ve_1 \wedge Vr_2 \in Vc \rightarrow Ve_2$
- Un invariant supplémentaire est rajouté : $Vr_1 \otimes Vr_2 \in Vc \rightarrowtail Ve_1 \times Ve_2$

2.3 Structuration de la spécification B

La décomposition en unités faiblement couplées est indispensable pour maîtriser la complexité de tout système, qu'il s'agisse d'une spécification ou d'une réalisation. Elle doit en particulier faciliter la compréhension, l'évolutivité et la réutilisation de composants du système. En B l'unité de conception est une machine abstraite. Les machines abstraites sont liées les unes aux autres. Dans la suite de ce chapitre seuls sont utilisés le lien $USES$ qui permet d'accéder aux variables d'une autre machine en lecture uniquement et le lien $INCLUDES$ qui permet d'appeler, sous certaines restrictions, les opérations d'une autre machine (pour plus de précisions sur ces liens, se reporter à l'annexe A).

Dans cette partie, est étudiée la décomposition en machines d'une spécification obtenue par les règles précédentes. L'objectif est de suivre le plus fidèlement possible la structure initiale du diagramme de classes afin de retrouver facilement dans la spécification finale les éléments du schéma et de faciliter le va-et-vient entre la spécification et le schéma, favorisant ainsi une conception incrémentale et évolutive. Cela n'est pas toujours évident, car la logique sous-jacente aux mécanismes de structuration de B ont principalement pour but de séparer les preuves à effectuer, ce qui n'est pas forcément la même chose que de structurer le modèle d'objets (même si on peut penser qu'il y des liens étroits entre les deux). Dans la suite, je présente pour chaque concept la solution que nous avons adoptée en la comparant éventuellement aux autres approches existantes.

2.3.1 Classe

Une classe encapsulant à la fois les propriétés statiques et comportementales d'un ensemble d'objets, il semble naturel de faire correspondre à chaque classe une machine abstraite :

Règle M-1
A chaque classe e d'attributs $a1, \ldots, an$ correspond
une machine abstraite Me avec les variables Ve, $Va1, \ldots, Van$

2.3.2 Association

Le problème est de décider où mettre la variable dénotant l'association : dans la machine d'une des classes associées ou dans une troisième machine, propre à l'association?

Les solutions proposées dans l'état de l'art ont consisté soit à créer systématiquement une machine pour chaque association [NR94] ou à l'inverse à mettre systématiquement la variable association dans une machine classe [LH94a]. Aucune de ces solutions n'apparaît réellement satisfaisante : la première produit trop de machines, certaines inintéressantes (car les invariants importants sont dans les machines incluantes). A l'inverse, la seconde solution peut inclure trop d'information dans

la même machine : certaines associations peuvent présenter beaucoup d'opérations soit sur les liens eux-mêmes soit sur les attributs des liens, et sont donc plutôt des candidates naturelles à une machine indépendante ; en outre avec cette seconde solution des classes risquent de se retrouver dans la même machine afin d'éviter la circularité des liens $USES$ entre machines.

En fait une solution plus fine que les deux précédentes est nécessaire à l'obtention d'une bonne modularité. Nous utilisons le concept d'*associationfixe* des modèles sémantiques de données, proche de la notion CODASYL "retention is fixed" [COD71].

Une association entre les classes e et f est fixe pour e si l'ensemble des liens concernant chaque objet de e est créé en même temps que l'objet et ne peut être modifié ensuite (donc un lien ne peut être détruit qu'à la disparition de l'objet).

Dans notre exemple de club video, *Appartient* est une association fixe pour *Cassette*. En effet une cassette est définitivement liée à la même boutique : on ne peut pas modifier ses liens. En revanche on peut toujours rajouter de nouvelles cassettes dans une boutique : *Appartient* n'est donc pas fixe pour *Boutique*.

Nous créons une machine propre à l'association si et seulement si elle n'est pas fixe, c'est-à-dire qu'elle a des opérations sur les liens eux-mêmes, donc indépendantes des classes qu'elle relie. Nous obtenons alors la règle de structuration des associations :

Règle M-2
Si une association r entre les classes e et f est :
M-2.1 : non fixe (ni pour e, ni pour f), il lui correspond :
- une machine abstraite Mr, distincte de Me et Mf qui :
- utilise (lien $USES$) Me et Mf
- comprend Vr

M-2.2 : fixe pour e, non fixe pour f :
- la variable Vr est dans la machine Me (ainsi que les variables, invariants et opérations associés)
- Me utilise (lien $USES$) Mf

Remarques :

- Dans nos travaux précédents, nous avions rajouté dans le cas M-2.1 la définition d'une machine qui inclut les machines Mr, Me et Mf. Le but était de pouvoir vérifier les invariants associés aux contraintes de cardinalité. Je pense que c'est une machine qui est inutile car dans la spécification B finale il existera de toute manière une machine qui incluera ces trois machines, comme nous le verrons dans le chapitre qui traite de la spécification des transactions.
- La Règle M-2 ne s'applique pas dans le cas où r est fixe pour e et pour f.
  Dans ce cas, il n'y a pas de raison de choisir à priori entre Me et Mf pour contenir Vr ; nous laissons donc le choix au spécifieur ; une considération qui peut l'aider est : l'association est-elle obligatoire pour e et pas pour f? dans ce cas, il est préférable de mettre Vr dans Me (car la création d'instances de e doit être faite conjointement avec la création de liens r).
- Pour éviter la circularité des liens $USES$, nous pouvons avoir à regrouper deux classes dans une même machine (à cause de deux associations fixes entre les deux classes) mais ce cas est très rare.
- Dans le cas d'une classe association correspondant à la Règle F-4.1, c'est la Règle M-2.1 qui s'applique. Dans l'autre cas, rappelons que la classe association est traduite par une classe et deux associations qui en fait sont fixes pour la classe association. On obtient donc une machine

abstraite (pour la classe) dans laquelle sont aussi définies les deux associations.

Dans notre exemple, nous obtenons les quatre machines suivantes.

MACHINE $BasicBoutique$
SETS
$BOUTIQUE$
VARIABLES
$Boutique$, $Nom_Boutique$, $Adresse$
INVARIANT

$$\begin{array}{rcl} Boutique & \subseteq & BOUTIQUE \ \wedge \\ Nom_Boutique & \in & Boutique \rightarrowtail STRING \ \wedge \\ Adresse & \in & Boutique \rightarrow STRING \end{array}$$

OPERATIONS
...
END;

MACHINE $BasicCassette$
USES
$BasicBoutique$ /* nécessaire pour définir la variable $Appartient$ */
SETS
$CASSETTE; GENRE$
VARIABLES
$Cassette$, Num_Ca, $Nbdispo$, $Titre$, $Genre$, $Appartient$
INVARIANT

$$\begin{array}{rcl} Cassette & \subseteq & CASSETTE \ \wedge \\ Num_Ca & \in & Cassette \rightarrowtail NAT1 \ \wedge \\ Nbdispo & \in & Cassette \rightarrow NAT \ \wedge \\ Titre & \in & Cassette \rightarrow STRING \ \wedge \\ Genre & \in & Cassette \rightarrow GENRE \ \wedge \\ Appartient & \in & Cassette \rightarrow Boutique \end{array}$$

OPERATIONS
...
END;

MACHINE $BasicClient$
SETS
$CLIENT$
VARIABLES
$Client$,Num_Cl, Nom_Cl, $Suspendu$
INVARIANT

$$\begin{array}{rcl} Client & \subseteq & CLIENT \ \wedge \\ Num_Cl & \in & Client \rightarrowtail NAT1 \ \wedge \\ Nom_Cl & \in & Client \rightarrow STRING \ \wedge \\ Suspendu & \in & Client \rightarrow BOOLEAN \end{array}$$

OPERATIONS
...

END;

MACHINE $BasicPret$
USES
$BasicClient$,$BasicCassette$,$DATE$
VARIABLES
$Pret$,$Date_Pret$
INVARIANT
$Pret \in Client \leftrightarrow Cassette \wedge$
$Date_Pret \in Pret \rightarrow DATE$
OPERATIONS
...
END;

2.4 Génération des opérations de base

Dans un système d'information, on utilise très souvent des opérations appelées opérations de base, associées aux classes et associations, qui permettent l'ajout, la suppression d'instances de classe ou d'association et la modification de valeur d'attribut. Elles sont indépendantes des applications définies sur le système et sont appelées dans les transactions. Nous les générons automatiquement à partir du diagramme de classes et de sa traduction en B.

2.4.1 Principes généraux

Les opérations de base concernant une classe vont être définies dans la machine de cette classe car c'est là où les variables peuvent être modifiées directement. De même, les opérations concernant une association vont être spécifiées dans la machine où est définie l'association.

En B, chaque opération décrit son effet local sur les variables de la machine où elle est définie et non pas sur les variables représentant toutes les données du système. Concernant les obligations de preuve, les opérations de chaque machine doivent vérifier les invariants qui sont décrits en utilisant les variables de la machine elle-même.

Notre but est de spécifier des machines classes et associations (appelées machines de base) avec les opérations de base qui respectent les invariants correspondant aux multiplicités du diagramme de classes (même si certaines preuves ne peuvent être faites que dans une machine qui inclut les machines de base). Cela permet d'une part de mieux séparer la vérification des contraintes et d'autre part d'aider à la spécification des transactions. Afin d'atteindre cet objectif, nous avons choisi de renforcer les préconditions des opérations de base des machines.

Je ne vais pas détailler la génération de toutes les opérations de base. Le lecteur intéressé trouvera les détails dans [Ngu98]. Je vais juste donner des exemples d'opérations générées dans le cas du club video.

2.4.2 Exemples d'opérations de base générées

Les opérations générées concernent les classes et les associations. On distingue les opérations suivantes : ajout d'une instance ou d'un lien ; suppression d'un ensemble d'instances ou de liens ; modification de la valeur d'un attribut.

Opération d'ajout d'une instance dans une classe

Cette opération doit rétablir l'invariant contenu dans la machine associée à la classe c et qui comprend quatre types de prédicats :

- typage de la variable V_c associée à la classe,
- typage des attributs,
- typage des éventuelles clés de la classe,
- typage des éventuelles associations fixes auxquelles cette classe participe.

L'opération d'ajout a pour paramètres :

- l'instance cc à ajouter : $cc \in Sc - Vc$,
- la valeur de chaque attribut obligatoire Att. Si l'attribut considéré est monovalué (resp. multivalué) alors cette valeur est atomique (resp. ensembliste) et est du type de l'attribut.
- les valeurs des associations fixes définies sur cette classe (mêmes règles que pour un attribut de la classe).

Par exemple, l'opération d'ajout d'une cassette est spécifiée par :

$$
\begin{array}{l}
\mathsf{B_Add_Cassette}(ca,nu,ti,nb,ge,bo) \triangleq \\
\quad \text{PRE} \\
\qquad \begin{array}{rcl}
ca & \in & CASSETTE - Cassette \ \wedge \\
nu & \in & NAT1 - ran(Num_Ca) \ \wedge \\
ti & \in & STRING \ \wedge \\
nb & \in & NAT \ \wedge \\
ge & \in & GENRE \ \wedge \\
bo & \in & Boutique
\end{array} \\
\quad \text{THEN} \\
\qquad \begin{array}{rcl}
Cassette & := & Cassette \cup \{ca\} \,||\, \\
Num_Ca & := & Num_Ca \cup \{ca \mapsto nu\} \,||\, \\
Titre & := & Titre \cup \{ca \mapsto ti\} \,||\, \\
Nbdispo & := & Nbdispo \cup \{ca \mapsto nb\} \,||\, \\
Genre & := & Genre \cup \{ca \mapsto ge\} \,||\, \\
Appartient & := & Appartient \cup \{ca \mapsto bo\}
\end{array} \\
\quad \text{END}
\end{array}
$$

Cette spécification tient compte du fait que l'attribut Num_Ca est clé de la classe $Cassette$ et que l'association $Appartient$ est fixe pour $Cassette$ et donc définie dans la machine $BasicCassette$.

Opération de suppression de liens dans une association

Il existe différents types d'opérations de suppression de liens. En effet, un lien (ou un ensemble de liens) peut être identifié de trois manières : par l'objet source du lien, par l'objet cible du lien, par les objets source et cible du lien.

De manière générale, la suppression de liens d'une association consiste à soustraire l'ensemble de ces liens de l'association et aussi de l'ensemble des variables correspondant aux attributs de l'association.

Par exemple, l'opération de suppression des prêts associés à un ensemble de clients Cl est spécifiée dans la machine $BasicPret$ de la manière suivante :

$$
\begin{array}{l}
\mathsf{B_Sup_Pret_Client}(Cl) \triangleq \\
\quad \text{PRE} \\
\qquad Cl \subseteq Client \\
\quad \text{THEN} \\
\qquad Pret := Cl \lhd Pret \;||\; \\
\qquad Date_Pret := (Cl * Cassette) \lhd Date_Pret \\
\quad \text{END}
\end{array}
$$

2.4.3 Prise en compte des contraintes d'intégrité

Comme nous l'avons vu dans le paragraphe précédent, les opérations de base sont générées automatiquement à partir du diagramme de classes. On obtient ainsi un ensemble de machines de base contenant toutes les données du SI et qui sont entièrement prouvées. Le spécifieur les a ainsi à sa disposition pour écrire des opérations plus complexes. Les seules contraintes d'intégrité prises en compte dans ces machines de base sont les clés. Les autres contraintes qu'on pourrait prendre en compte sont les contraintes intra-classes. Il faudrait alors pouvoir déduire automatiquement de l'expression de la contrainte la précondition qu'il faut ajouter aux opérations de base. Dans [LP01b], nous avons proposé une première solution : le spécifieur peut préciser dans le diagramme de classes, pour chaque contrainte, quel type d'opération est concerné (insertion, suppression, ...) et l'expression booléenne qu'il faut ajouter à la précondition de cette opération. Une autre solution est de déduire automatiquement la précondition à partir de la spécification de l'opération et de l'invariant de la machine comme proposé dans [Led98]. Si ces informations ne sont pas disponibles, ces contraintes sont spécifiées dans la machine qui inclut la machine de base correspondant à la contrainte, comme nous le verrons au chapitre 4.

2.5 Héritage et lien de généralisation/spécialisation

2.5.1 Définition

Le concept d'héritage est un concept qui reste encore très controversé, tout simplement car plusieurs sémantiques peuvent y être rattachées [Tai96]. La plupart des chercheurs s'accorde sur une première distinction qui peut être faite selon le niveau d'abstraction auquel il est utilisé. Pour résumer, en phase de conception (ou spécification), l'héritage est un moyen de modéliser les liens de généralisation/spécialisation. En phase d'implémentation, et si on utilise un langage de programmation orienté objets, l'héritage permet de réutiliser le code des classes existantes pour en construire de nouvelles. C'est sur ce dernier point qu'il y a le plus de divergence au niveau de la sémantique, mais comme ce n'est pas mon sujet de préoccupation dans ce mémoire, je n'en dirais pas plus. Dans l'article [Tai96], le lecteur peut trouver une analyse approfondie de l'héritage et un nombre important de références.

Le concept de généralisation/spécialisation a été défini dans [SS77]. Partant d'un certain nombre de classes, appelées spécialisations ou sous-classes, la généralisation est un mécanisme d'abstraction permettant la description d'une nouvelle classe, appelée super-classe, qui regroupe les éléments communs des spécialisations et laissent dans chaque sous-classe les éléments spécifiques. Une classe Cs est considérée comme une spécialisation d'une classe Cg si toute entité appartenant à l'extension de la classe Cs appartient aussi à l'extension de Cg.

Ces concepts sont formalisés en raffinant la Règle F-1 par :

Règle F-5
Pour chaque sous-classe e d'une classe f :
- le "given set" (instances possibles) pour e est le même que pour f
- l'invariant supplémentaire : $Ve \subseteq Vf$

Il n'y a qu'un "given set" pour toute la hiérarchie d'héritages. Il est associé à la classe de plus haut niveau et représente donc l'ensemble de tous les objets possibles dans la hiérarchie. En revanche, il y a toujours une variable d'état par classe. $Ve \subseteq Sf$ est conséquence de l'invariant supplémentaire.

La figure 2.2 donne l'exemple d'une hiérarchie de liens de généralisation/spécialisation.

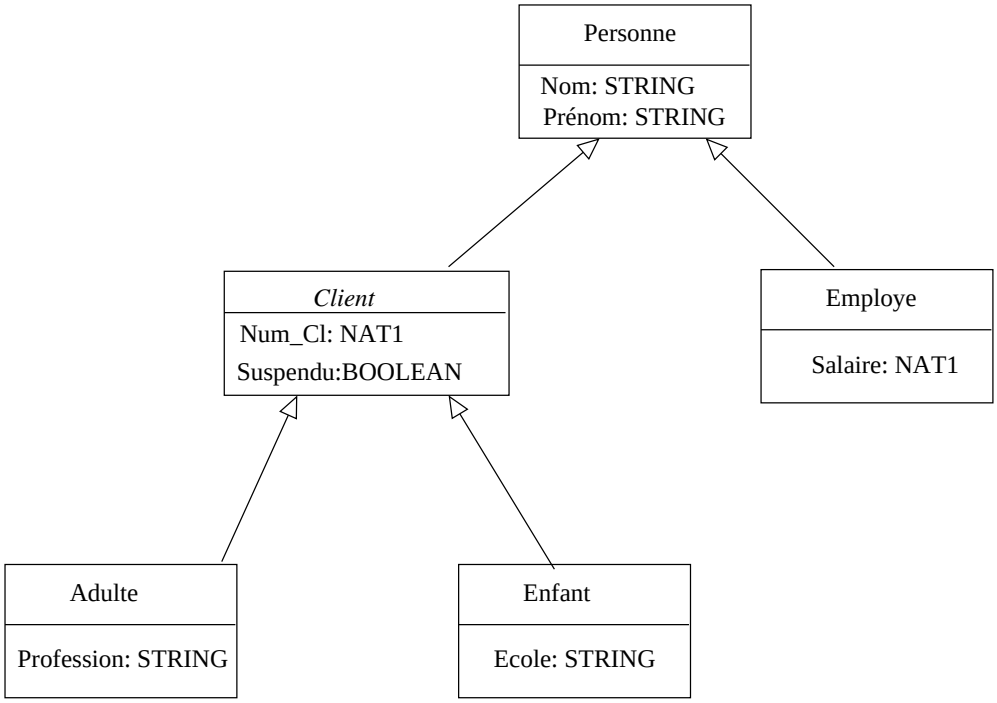

FIG. 2.2 – *Un exemple de hiérarchie de généralisation/spécialisation*

En appliquant la Règle F-5, on obtient :

$$Personne \subseteq PERSONNE \ \wedge$$
$$Employe \subseteq Personne \ \wedge \ Client \subseteq Personne \ \wedge$$
$$Adulte \subseteq Client \ \wedge \ Enfant \subseteq Client$$

On peut définir des contraintes additionnelles comme par exemple : la généralisation n'a pas d'instances propres (une telle classe est aussi appelée "classe abstraite"). Ces contraintes sont traduites par des invariants additionnels sur les variables décrivant les classes. Par exemple la classe abstraite $Client$ donne : $Client = Adulte \cup Enfant$. ($Adulte \subseteq Client \ \wedge \ Enfant \subseteq Client$ devient alors redondant). On peut exprimer de manière aussi directe une contrainte de disjonction : $Adulte \cap Enfant = \emptyset$

Les deux propriétés fondamentales du concept de généralisation/spécialisation, à savoir l'héritage d'attributs et du comportement, sont effectivement assurées comme conséquence de la contrainte

d'inclusion définie entre les variables associées à la sous-classe et à la super-classe. La première propriété signifie que tout attribut de la généralisation s'applique à toutes les instances de la hiérarchie. En effet, une fonction applicable sur un sur-ensemble de Ve est applicable sur Ve. Par exemple comme $nom \in Personne \rightarrow STRING$, *nom* est applicable à toute instance de *Client* et d'*Adulte*, puisque $Client \subseteq Personne$ et $Adulte \subseteq Client$. En revanche, comme $profession \in Adulte \rightarrow STRING$, *profession* n'est pas applicable en dehors de *Adulte*. La seconde propriété permet à tout objet d'une sous-classe d'avoir aussi le comportement défini dans la super-classe, c'est-à-dire que toute opération de la super-classe est applicable aux instances de la sous-classe.

Dans cette solution de formalisation, on ne peut pas modéliser l'héritage multiple sauf si chaque hiérarchie d'héritages a une seule classe racine. Dans ce cas, on a toujours un seul "given set" pour la hiérarchie. Sinon, dans le cas le plus général, il faudrait créer une classe fictive qui serait racine de toutes les hiérarchies d'héritages (et qui pourrait s'appeler "Object" comme dans beaucoup de langages de programmation orientés objets) et à laquelle serait associé le "given set" de la hiérarchie.

Quand nous avons commencé notre travail, nous avons voulu formaliser le concept d'héritage dans le sens lien de généralisation/spécialisation avec aussi héritage des opérations comme on le trouve dans les méthodes de conception objet comme UML ou OMT. Cependant, dans ces méthodes, ce concept n'a pas de sémantique formelle, l'héritage des opérations est purement syntaxique : les opérations doivent avoir la même signature. En fait, l'héritage dépend fortement des mécanismes propres à chaque langage de programmation orientés objets. Le problème est qu'il alors est très difficile de définir une sémantique pour un concept qui se veut à la fois indépendant des langages mais qui puisse quand même être représenté dans n'importe quel langage. J'aurais maintenant plutôt tendance à considérer ce type d'héritage comme un concept d'implémentation voire au mieux de raffinement. En effet, si on reprend l'objectif d'une spécification, il s'agit de décrire le problème à résoudre. L'implémentation et le raffinement ont pour but d'apporter des solutions à ce problème. Les mécanismes liés à l'héritage n'ont rien de conceptuel et ne servent pas à décrire un problème mais plutôt apporte une solution possible pour le résoudre. Dans l'article [FL96] et la thèse de Nguyen, nous avons essayé de résoudre l'héritage des opérations par une structuration particulière de la spécification B. Ce n'est pas une solution fausse. Mais elle est très lourde et ne résout que partiellement le problème. Je préfère ignorer cet aspect héritage d'opérations et proposer une autre solution de structuration.

2.5.2 Structuration de la spécification

Comme nous l'avons vu précédemment, lors de la structuration d'un diagramme de classes, nous avons essayé de respecter la structure de ce diagramme en associant une machine abstraite à chaque classe. Le concept d'héritage doit suivre le même principe. Il nous faut résoudre le même problème que pour les associations : si une classe e hérite d'une classe f, nous avons $Ve \subseteq Vf$, mais où mettre Ve et Vf ?

Comme pour les associations, certaines solutions ont déjà été proposées dans l'état de l'art :

- mettre toutes les variables d'une même hiérarchie d'héritage dans la même machine [LH94a] : cette solution fait perdre le bénéfice de la spécification incrémentale qu'est censé apporter l'héritage.
- mettre toutes les variables dans des machines différentes avec, si e hérite de f, un lien Me $INCLUDES$ Mf [NR94] ; cette solution présente un inconvénient rédhibitoire : comme une machine ne peut être incluse que dans une seule autre, une classe ne peut avoir qu'une seule sous-classe directe ...

Notre solution consiste également à conserver une machine différente par classe mais en utilisant des liens $USES$:

Règle M-3
Si la classe e hérite de la classe f alors la machine Me associée à e utilise (lien $USES$) la machine Mf associée à f et toutes les machines associées aux super-classes de f (le lien $USES$ n'est pas transitif).

L'invariant (présent dans Me) $Ve \subseteq Vf$, ne pourra être vérifié que dans une machine qui inclut les machines Me et Mf, comme c'est le cas pour certaines contraintes de multiplicités du diagramme de classes. Chaque machine contient les opérations de base générées à partir de la classe correspondante. Ainsi dans Me nous pouvons créer des instances de e (mais pas en tant qu'instances de f) et spécialiser des instances de f comme instances de e ou déqualifier des instances de e (elles restent instances de f). Dans la machine incluante, ces opérations peuvent être utilisées pour créer réellement des instances de e comme cas particulier d'instances de f (en appliquant en parallèle e-création et f-création) et vérifier ainsi l'invariant $Ve \subseteq Vf$.

2.6 Conclusion

Dans ce chapitre, j'ai décrit les principes de la solution que nous avons adoptée pour formaliser les concepts de base des modèles sémantiques de données en utilisant le langage B. Ont également été présentés la méthode pour structurer la spécification B obtenue et les mécanismes qui permettent de générer automatiquement les opérations de base qui manipulent les objets et les liens associés à chaque classe et association. Toutes les machines B ainsi obtenues ont été prouvées automatiquement par le prouveur de l'AtelierB.

Je n'ai pas présenté tous les concepts des modèles sémantiques. Le lecteur intéressé peut en trouver un complément dans la thèse de Phuong Nguyen [Ngu98]. Le concept d'agrégation/composition, concept important dans les méthodes à objets, a été étudié dans le stage de DEA de Mouhcine Sahbani [Sah99].

Si on revient à UML, je n'ai pas considéré tous les concepts qui existent dans un diagramme de classes UML. En effet, je considère que dans ces diagrammes on peut trouver des concepts qui ne relèvent pas d'une spécification mais plutôt d'une implémentation, comme par exemple la visibilité des variables. En fait je définis un sous-ensemble des concepts d'UML qui forment un langage bien adapté à la spécification des SI, que j'appellerai IS-UML dans la suite de ce mémoire. C'est à rapprocher de la notion de $Profile$ présenté dans [CKM+99] et maintenant préconisé par l'OMG.

Chapitre 3

Langage formel pour la méta-modélisation des systèmes d'information

Dans le chapitre précédent, j'ai présenté une formalisation des concepts des modèles sémantiques de données dans le langage de spécifications B. Cette approche par traduction est une formalisation indirecte ou implicite ; elle correspond à un plongement superficiel [BGG+92]. En effet, chaque *occurrence de concept* des modèles sémantiques de données est traduite par des éléments de la spécification formelle. L'approche est satisfaisante pour la majorité des concepts qui trouvent une traduction directe dans le langage formel, essentiellement dû au fait que B et les modèles sémantiques ont une base théorique assez proche. En revanche, elle soulève des problèmes pour les concepts liés à la structuration de la spécification ou ceux qui n'ont pas d'exact équivalent dans le langage formel comme l'héritage.

Une autre approche possible est de formaliser directement les *concepts* avec un langage formel. On donne ainsi une fois pour toutes la signification de chaque concept. On obtient alors un méta-modèle formel qui peut ensuite être directement utilisé : il suffit d'instancier chaque concept sur l'application à modéliser. Ainsi on ne formalisera pas comme dans l'approche par traduction une classe *Client* par un ensemble *Client* plus la collection de fonctions vers les attributs du client; on dira simplement que *Client* appartient à l'ensemble des classes du système, que *numéro de client* appartient à l'ensemble des attributs du système et est relié à l'instance *Client*, etc. Cette approche de formalisation est qualifiée d'explicite ou de directe. Elle correspond à un plongement en profondeur.

Actuellement, la syntaxe et la sémantique des notations UML sont également définies par méta-modélisation [OMG99]. Le méta-modèle est donné en termes de diagrammes de classes, d'expressions OCL [WK99] et de descriptions en langue naturelle. La sémantique de certains concepts reste alors très floue [FELR98] (et même parfois contradictoire comme par exemple pour le concept d'agrégation/composition [BHS99]). Mon objectif est de décrire un modèle formel du méta-modèle du langage IS-UML en utilisant un langage de spécifications formelles comme méta-langage. On obtient ainsi une sémantique du langage IS-UML. Plusieurs avantages bien connus en résultent. En particulier il va être possible de raisonner sur le méta-modèle et donc :

– La cohérence globale des différents composants servant à décrire une application peut être vérifiée : que ce soit la cohérence entre les concepts d'un même composant ou appartenant à

différents composants (diagrammes de classes, diagrammes d'états-transitions, ...).

- Il est possible de faire évoluer le méta-modèle tout en s'assurant que les nouvelles constructions sont cohérentes avec l'existant. C'est ce que nous avons fait en étendant le méta-modèle d'UML par la prise en compte des contraintes d'intégrité statique des SI.
- C'est une base solide pour la comparaison des différents modèles qu'on peut trouver dans les méthodes de conception existantes. En décrivant formellement le méta-modèle de chaque méthode, il est possible d'identifier leurs points communs et ceux divergents [BCPC+97] et de dépasser les problèmes terminologiques et les conventions graphiques. De plus, on peut définir des règles d'équivalence entre les concepts, sémantiquement proches, de ces méthodes permettant ainsi un meilleur apprentissage des méthodes et une reprise de l'existant si on est amené à changer de méthode.
- Il devient possible de séparer les raisonnements formels (preuves) sur le méta-modèle de ceux sur l'application modélisée. Il apparaît tout à fait normal de passer du temps à prouver certaines propriétés du méta-modèle, ces preuves étant faites une fois pour toutes et les propriétés obtenues étant valables pour tous les modèles d'applications.

De plus des outils implémentant cette sémantique peuvent être développés. La construction d'un outil implique qu'une sémantique des concepts utilisés soit définie. Les outils existants ont donc défini leur propre sémantique, pas toujours connue des utilisateurs d'ailleurs, et on peut arriver parfois à des situations aberrantes où, à partir d'un même diagramme de classes, le code généré n'est pas le même selon l'outil utilisé.

Dans la suite du chapitre, je vais présenter les principales caractéristiques du méta-modèle formel que nous avons défini, puis l'approche de construction de spécifications formelles par interprétation.

3.1 Un méta-modèle formel pour les SI

C'est un travail qui a commencé lors du stage de DEA de Fabrice Monge [Mon97], l'objectif étant la mise en œuvre de l'approche par méta-modélisation. Depuis, j'ai repris le travail, en collaboration avec Fiona Polack de l'Université de York en Grande-Bretagne, avec pour objectif cette fois de définir formellement le sous-ensemble de UML utilisé pour la spécification des SI (IS-UML). Nous tenions vraiment à travailler avec le standard UML décrit dans [OMG99], c'est pourquoi nous avons été amenées à modifier le méta-modèle présenté dans le rapport de DEA. Les différences sont essentiellement syntaxiques.

La méthode de définition de IS-UML est résumée par la figure 3.1. La syntaxe concrète décrit l'ensemble des symboles graphiques utilisés pour construire les diagrammes (boites, flèches, ...). La syntaxe abstraite définit l'ensemble des diagrammes valides. Elle associe à chaque constructeur graphique un constructeur de syntaxe abstraite défini par son nom, ses composants et ses relations avec les autres constructeurs de syntaxe abstraite. Elle est complétée par des contraintes sur les constructeurs qui restreignent l'ensemble des diagrammes valides. Ces contraintes constituent ce qu'on appelle couramment la sémantique statique.

La sémantique dynamique permet de préciser la signification de ces éléments. Elle peut être exprimée formellement en utilisant différentes techniques classiques [Win93] (dénotationnelle, opérationnelle, ...). Deux ensembles de règles de traduction permettent de faire le lien d'une part entre les syntaxes concrète et abstraite et d'autre part entre la syntaxe abstraite et la sémantique dynamique. Nous avons repris la syntaxe concrète de UML mais nous ne nous sommes pas intéressés à la spécification des règles syntaxiques.

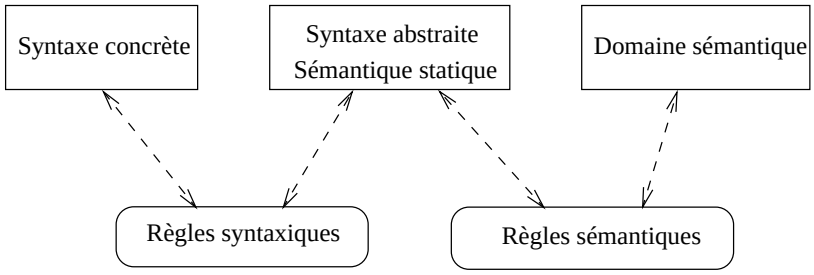

FIG. 3.1 – *Méthode de définition de IS-UML*

En UML, la syntaxe abstraite est définie par un méta-modèle qui est obtenu par auto-modélisation, c'est-à-dire en utilisant les diagrammes de classes de UML. La plupart des approches de formalisation du méta-modèle de UML [CE97, KC99, CEKS01, Con02] utilisent cette technique d'auto-modélisation qu'elles complètent par une spécification formelle écrite soit en OCL soit avec un langage de spécifications formelles comme Z ou Object-Z. Nous avons repris la même démarche pour décrire IS-UML et nous utilisons le langage B comme langage de spécifications formelles.

Dans la suite de la section, je donne la syntaxe abstraite, la sémantique statique et la sémantique dynamique des principaux concepts structurels. Le chapitre suivant fait de même pour les opérations. L'objectif de cette section n'est pas de définir de manière exhaustive tout le méta-modèle ni de comparer notre approche par rapport aux formalisations existantes, ce qu'on peut trouver dans les différents articles que nous avons écrits [Mon97, LP00, LP01a].

3.1.1 Représentation graphique de la syntaxe abstraite

Le diagramme de classes de la figure 3.2 reprend les principaux concepts du langage IS UML définis dans le chapitre 2. L'héritage est exprimé par l'association réflexive *subClass*. L'attribut *abstract* précise si une classe est abstraite, c'est-à-dire si elle n'a pas d'instances directes. Une association a deux extrémités, appelées aussi rôles, représentées par la classe *AssociationEnd* et qui sont reliées chacune à une classe par l'association *characClass*. La classe *Characteristic* regroupe les caractéristiques communes des classes *AssociationEnd* et *Attribute*. Les attributs $characMultMin$ et $characMultMax$ décrivent les multiplicités minimale et maximale d'un rôle d'une association ou d'un attribut. L'attribut *characChange* indique si la valeur de la caractéristique peut être modifiée ou si elle reste constante pendant toute sa durée de vie. La classe *Type* regroupe tous les types possibles des attributs, c'est-à-dire ceux définis par le concepteur et les types de base UML autorisés. Une classe association est représentée par un lien de l'association *assocClass* et une instance dans chacune des classes *Class* et *Association*.

J'ai également représenté dans ce méta-modèle les opérations générées à partir du diagramme de classes. Une opération est associée soit à une classe (*opClass*), soit à une association (*opAssoc*). Une opération qui met à jour un attribut est en plus associée à un attribut (*opAttrib*).

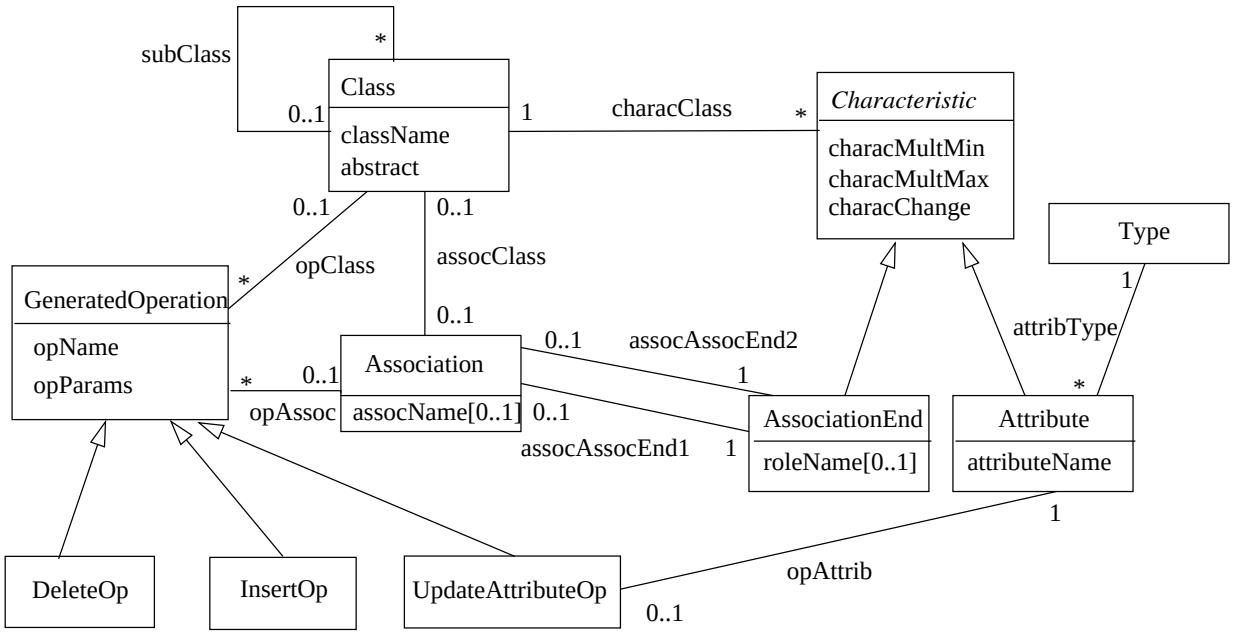

FIG. 3.2 – *Diagramme de classes du méta-modèle de IS-UML*

Par souci de clarté, je n'ai pas précisé sur le schéma le type des attributs des classes du méta-modèle. Ils sont décrits dans la section suivante.

3.1.2 Représentation formelle de la syntaxe abstraite et de la sémantique statique

Par application des règles de formalisation définies dans la section 2.2 du chapitre 2, on obtient une représentation formelle de la syntaxe abstraite en B, décrite par les tables 3.1, 3.2, 3.3.

$Class$	$\subseteq$	$CLASS$
$Association$	$\subseteq$	$ASSOCIATION$
$Characteristic$	$\subseteq$	$CHARACTERISTIC$
$Attribute$	$\subseteq$	$Characteristic$
$AssociationEnd$	$\subseteq$	$Characteristic$
$Type$	$\subseteq$	$TYPE$
$GeneratedOperation$	$\subseteq$	$OPERATION$
$DeleteOp$	$\subseteq$	$GeneratedOperation$
$InsertOp$	$\subseteq$	$GeneratedOperation$
$UpdateAttributeOp$	$\subseteq$	$GeneratedOperation$

TAB. 3.1 – *Variables et invariants de typage B représentant les classes du méta-modèle*

$subClass$	$\in$	$Class \nrightarrow Class$ [1]
$assocClass$	$\in$	$Association \rightarrowtail Class$
$assocAssocEnd1$	$\in$	$Association \rightarrowtail AssociationEnd$
$assocAssocEnd2$	$\in$	$Association \rightarrowtail AssociationEnd$
$characClass$	$\in$	$Characteristic \rightarrow Class$
$attribType$	$\in$	$Attribute \rightarrow Type$
$opClass$	$\in$	$GeneratedOperation \nrightarrow Class$
$opAssoc$	$\in$	$GeneratedOperation \nrightarrow Assoc$
$opAttrib$	$\in$	$UpdateAttributeOp \rightarrow Attribute$

TAB. 3.2 – *Variables et invariants B représentant les associations dans le méta-modèle*

$Class$	$className$	$\in$	$Class \rightarrowtail NAME$
	$abstract$	$\in$	$Class \rightarrow BOOLEAN$
$Characteristic$	$characMultMin$	$\in$	$Characteristic \rightarrow \mathbb{N}$
	$characMultMax$	$\in$	$Characteristic \rightarrow \mathbb{N}_1$
	$characChange$	$\in$	$Characteristic \rightarrow BOOLEAN$
$Attribute$	$attributeName$	$\in$	$Attribute \rightarrow NAME$
$AssociationEnd$	$roleName$	$\in$	$AssociationEnd \nrightarrow NAME$
$Association$	$assocName$	$\in$	$Association \nrightarrow NAME$
$GeneratedOperation$	$opName$	$\in$	$GeneratedOperation \rightarrowtail NAME$
	$opParams$	$\in$	$GeneratedOperation \rightarrow seq(TPARAM)$

TAB. 3.3 – *Variables et invariants B représentant les attributs du méta-modèle*

Le type de $opParams$, $seq(TPARAM)$, spécifie une liste ordonnée de paramètres. Un élément de $TPARAM$ est un triplet composé d'un nom, d'un type et d'un indicateur précisant si le paramètre correspondant est un paramètre d'entrée (i) ou de sortie (o) :

$$TPARAM \in (NAME \times TYPE \times \{i,o\})$$

Les différentes règles de la sémantique statique sont exprimées par des invariants supplémentaires comme par exemple celui qui interdit les cycles dans les hiérarchies d'héritage :

$$subClass^+ \cap id(Class) = \emptyset$$

ou celui exprimant que $Characteristic$ est une classe abstraite dont les sous-classes sont disjointes :

$$Attribute \cup AssociationEnd = Characteristic \wedge$$
$$Attribute \cap AssociationEnd = \emptyset$$

Terminons par l'invariant suivant :

$$\mathsf{ran}\ (assocAssocEnd1) \cap \mathsf{ran}\ (assocAssocEnd2) = \emptyset \quad \wedge$$
$$\mathsf{ran}\ (assocAssocEnd1) \cup \mathsf{ran}\ (assocAssocEnd2) = associationEnd$$

1. subClass(a)=b se lit "a est une sous-classe de b"

qui exprime que les extrémités d'une association sont différentes et que toute extrémité est nécessairement associée à une seule association.

3.1.3 Sémantique dynamique du méta-modèle

Une fois définie la représentation symbolique des éléments manipulés, on précise le domaine sémantique de ces éléments et les relations qui existent entre les deux. Dans notre cas, la sémantique introduit les notions d'objet, valeur et lien. Enfin, la sémantique des opérations permet de décrire comment le méta-modèle peut évoluer. Tout comme pour la syntaxe abstraite, cet aspect est spécifié en utilisant le langage B.

Domaine sémantique

On distingue l'ensemble des objets existants *Object* dont les éléments sont dans l'ensemble abstrait $OBJECT$:

$$Object \subseteq OBJECT$$

VAL représente toutes les valeurs possibles dans le modèle. Un objet est associé à plusieurs valeurs :

$$objVal \in Object \leftrightarrow VAL \ \wedge \ \mathsf{dom}(objVal) = Object$$

Un lien est représenté par une paire d'objets existants :

$$link \in Object \leftrightarrow Object$$

Liaison syntaxe abstraite et domaine sémantique

L'objectif est d'associer aux constructeurs syntaxiques des éléments du domaine sémantique. Une classe est associée à un ensemble d'objets, mais un objet est instance directe d'une seule classe :

$$objectClass \in Object \rightarrow Class$$

La définition de *link* est utilisée pour instancier une association comme un ensemble de liens entre objets :

$$linkAssociation \in Association \rightarrow link$$

Notons qu'il peut y avoir plusieurs associations entre les mêmes objets.

La fonction suivante associe les attributs à leur valeur :

$$attributeObjectValue \in Attribute \rightarrow objVal$$

Plusieurs propriétés sur la sémantique dynamique du méta-modèle peuvent être exprimées. J'en précise quelques-unes ci-dessous, après avoir donné deux définitions liées à l'héritage.

Soit *allClassObject* la relation qui définit pour une classe l'ensemble de ses instances directes ou non. Par définition de l'héritage sémantique, celui-ci est composé des instances directes et des instances indirectes qui viennent des sous-classes :

$$allClassObject \triangleq (subClass^*)^{-1} \ ; \ objectClass^{-1}$$

Pour une classe c, $allClassAttributes[\{c\}]$ désigne l'ensemble des attributs de la classe et de ses super-classes :

$$allClassAttributes[\{c\}] \triangleq \bigcup \ c' \ \cdot \ (c' \in subClass^*(c) \mid attribClass^{-1}\ [\{c'\}])$$

avec

$$attribClass \triangleq Attribute \lhd characClass$$

Une première propriété exprime que les liens qui instancient une association sont définis à partir des objets qui instancient les classes participant à l'association.

$$\begin{array}{l} linkAssociation \subseteq \\ ((assocAssocEnd1 \ ; \ characClass \ ; \ allClassObject) \otimes \\ \ (assocAssocEnd2 \ ; \ characClass \ ; \ allClassObject)) \end{array}$$

Une seconde propriété exprime qu'un objet ne peut prendre des valeurs que pour des attributs de sa classe (et de ses super-classes) :

$$\begin{array}{l} \forall c \in class \cdot \\ \mathsf{dom}(\bigcup(attribObjectValue\ [allClassAttributes[\{c\}]])) \subseteq objectClass^{-1}\ [\{c\}] \end{array}$$

D'autres propriétés expriment des contraintes sur les multiplicités des associations ou des attributs.

L'intérêt de l'approche est qu'il est facile de définir de manière rigoureuse de nouveaux concepts dans le méta-modèle. C'est ce que nous avons fait en étendant le méta-modèle avec la prise en compte des contraintes d'intégrité statique, concept très utilisé dans les SI [Mon97, LP01a].

Opérations sur le méta-modèle

La spécification du méta-modèle est complétée par les opérations qui permettent d'ajouter ou de supprimer des objets et des liens ou encore de modifier la valeur d'un attribut d'un objet. Toute transaction doit s'exprimer comme une composition de ces opérations.

Je vais présenter les opérations de mise à jour d'un attribut et de suppression d'un objet, je renvoie le lecteur à [Mon97] pour la spécification des autres opérations.

$\mathsf{Modifier_AttClassNonCle}(unAtt,unObj,ancVal,nouvVal) \triangleq$
PRE
$$\begin{array}{rcl} unObj & \in & Object \ \wedge \\ unAtt & \in & Attribute \ \wedge \\ ancVal & \in & VAL \ \wedge \\ nouvVal & \in & VAL \ \wedge \\ (unAtt \mapsto (unObj \mapsto ancVal)) & \in & attributeObjectValue \end{array}$$
THEN
$$attributeObjectValue := attributeObjectValue - \{unAtt \mapsto (unObj \mapsto ancVal)\} \cup \{unAtt \mapsto (unObj \mapsto nouvVal)\}$$
END
END

La destruction d'un objet peut entraîner la destruction d'objets qui lui sont liés par des liens d'associations et la destruction de ces liens eux-mêmes, selon la multiplicité minimale de ces associations.

$\mathsf{Detruire_Objet}(unObj) \triangleq$
PRE
$$unObj \in Object$$

THEN

$$
\begin{aligned}
Object &:= Object - Objets_Detruire[\{unObj\}] \;||\\
objVal &:= Objets_Detruire[\{unObj\}] \lhd objVal \;||\\
objectClass &:= Objets_Detruire[\{unObj\}] \lhd objectClass \;||\\
attributeObjectValue &:= attributeObjectValue\\
&\qquad \rhd (Objets_Detruire[\{unObj\}] \lhd objVal) \;||\\
link &:= link - Liens_Detruire(Objets_Detruire[\{unObj\}]) \;||\\
linkAssociation &:= linkAssociation\\
&\qquad \rhd Liens_Detruire(Objets_Detruire[\{unObj\}])
\end{aligned}
$$

END

END

La relation *Objets_Detruire* donne pour un objet l'ensemble des objets à détruire récursivement et la relation *Liens_Detruire* donne pour un objet l'ensemble des liens à détruire récursivement.

$$Objets_Detruire \triangleq Objets_Detruire_Direct^*$$

avec :

$$
\begin{aligned}
&Objets_Detruire_Direct \triangleq\\
&\quad (\bigcup aa \;.\; (aa \in Role_1_Oblig \mid\\
&\qquad (\lambda ob \,.\, ob \in \mathsf{ran}(linkAssociation(aa))\\
&\qquad \mid \{x \mid x \in Object \;\wedge\; linkAssociation(aa)[\{x\}] = \{ob\}\}))\\
&\quad \cup\\
&\quad (\bigcup aa \;.\; (aa \in Role_2_Oblig \mid\\
&\qquad (\lambda ob \,.\, ob \in \mathsf{dom}(linkAssociation(aa))\\
&\qquad \mid \{x \mid x \in Object \;\wedge\; linkAssociation(aa)^{-1}\,[\{x\}] = \{ob\}\}))
\end{aligned}
$$

$Role_i_Oblig$ est l'ensemble des associations dont l'extrémité i a une multiplicité minimale de 1 :

$$Role_i_Oblig = \mathsf{dom}(assocAssocEndi \rhd characMultMin^{-1}\,[\{1\}])$$

Finalement :

$$Liens_Detruire(ensObj) \triangleq (link \rhd ensObj) \cup (ensObj \lhd link)$$

Pour conclure sur cette section, la description formelle du langage IS-UML est donnée par une machine abstraite B qui inclut tous les éléments présentés dans la section.

3.2 Instantiation du méta-modèle pour une application

Cette partie a fait l'objet du stage de DEA de Fabrice Monge [Mon97]. Rappelons le principe de base de l'approche par méta-modélisation : conserver dans la spécification formelle les concepts du méta-modèle; on écrit donc une fois pour toutes une formalisation du méta-modèle. Les informations spécifiques à l'application modélisée doivent utiliser cette formalisation tout en en étant clairement séparées. Le schéma 3.3 illustre l'approche.

Tout d'abord, le méta-modèle est formalisé par une machine abstraite B qui contient la spécification du méta-modèle et des opérations élémentaires de gestion des instances (cf section précédente). Puis, pour chaque application, une traduction purement syntaxique est effectuée. Il s'agit tout simplement de dire que chaque élément du modèle de l'application est une instance de la construction du méta-modèle correspondante. Par exemple *Client* est instance de la classe *Class* du méta-modèle. Le problème est de définir comment une application peut utiliser le méta-modèle.

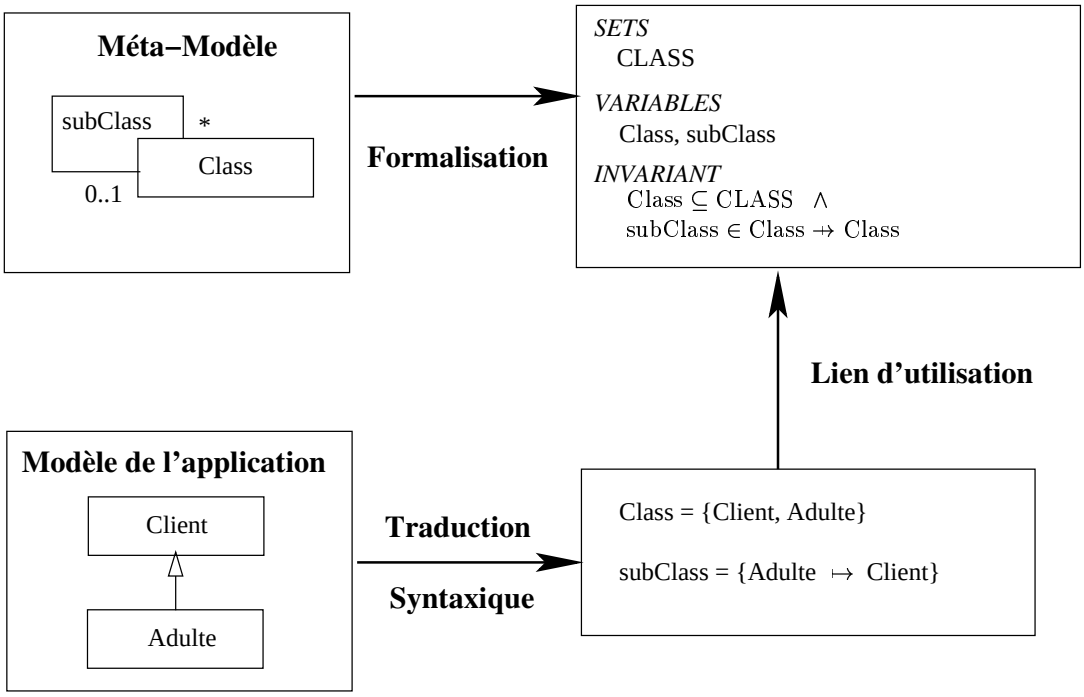

FIG. 3.3 – *Approche par méta-modélisation*

Les mécanismes de composition du langage B sont tels que le seul lien d'utilisation possible est le lien $INCLUDES$ car nous voulons pouvoir utiliser les opérations du méta-modèle dans les opérations de l'application. Nous avons alors été obligé de modifier la machine B obtenue dans la section précédente. La machine B $MetaModel$ présentée ci-dessous est maintenant paramétrée par tous les éléments du méta-modèle. La spécification d'une application est définie par une machine appelée *Application* qui inclut la machine $MetaModel$ et qui instancie les paramètres par les valeurs explicites de l'application. Prenons un exemple avec une partie du méta-modèle de la figure 3.2.

MACHINE $MetaModel$
$(OBJECT, CLASS, ASSOCIATION,$
$CHARACTERISTIC, ATTRIBUTE, ASSOCIATION_END$
$subClass, characClass, assocAssocEnd1, assocAssocEnd2)$

CONSTRAINTS

$$\begin{aligned} ASSOCIATION_END &\subseteq CHARACTERISTIC \ \wedge \\ ATTRIBUTE &\subseteq CHARACTERISTIC \ \wedge \\ subClass &\in CLASS \leftrightarrow CLASS \ \wedge \\ assocAssocEnd1 &\in ASSOCIATION \rightarrowtail ASSOCIATION_END \ \wedge \\ assocAssocEnd2 &\in ASSOCIATION \rightarrowtail ASSOCIATION_END \ \wedge \\ characClass &\in CHARACTERISTIC \leftrightarrow CLASS \end{aligned}$$

VARIABLES
$Object,$
$objectClass,$
$link,$

$linkAssociation$,

INVARIANT

$$\begin{aligned} Object &\subseteq OBJECT \ \wedge \\ objectClass &\in Object \rightarrow CLASS \ \wedge \\ link &\in Object \leftrightarrow Object \ \wedge \\ linkAssociation &\in ASSOCIATION \rightarrow link \end{aligned}$$

OPERATIONS

Detruire_Objet($unObjet$)$\triangleq$ ... ;

Detruire_Lien($unLien$) $\triangleq$ ... ;

Inserer_Objet($unObjet$,$uneClasse$)$\triangleq$ ... ;

Inserer_Lien($unLien$,$uneAssociation$) $\triangleq$ ... ;

...

END;

La machine *Application* reprend une partie de la figure 2.1, (les classes *Client* et *Cassette* et l'association *Pret*). Pour la lisibilité de la spécification, les paramètres réels ont le même nom que les paramètres formels. Ils sont instanciés dans les clauses SETS et PROPERTIES.

MACHINE *Application*

INCLUDES

$MetaModel$

($OBJECT$, $CLASS$, $ASSOCIATION$,
$CHARACTERISTIC$, $ATTRIBUTE$, $ASSOCIATION_END$
$subClass$, $characClass$, $assocAssocEnd1$, $assocAssocEnd2$)

SETS

$OBJECT$ = {o1,o2,o3};
$CLASS$ = {Client,Cassette};
$ASSOCIATION$ = {Pret};
$CHARACTERISTIC$ = {c1,c2};
$ASSOCIATION_END$ = {c1,c2};

CONSTANTS

$subClass$, $characClass$, $assocAssocEnd1$, $assocAssocEnd2$

PROPERTIES

$$
\begin{array}{rcl}
subClass & \in & CLASS \nrightarrow CLASS \ \wedge \\
assocAssocEnd1 & \in & ASSOCIATION \rightarrowtail ASSOCIATION_END \ \wedge \\
assocAssocEnd2 & \in & ASSOCIATION \rightarrowtail ASSOCIATION_END \ \wedge \\
characClass & \in & CHARACTERISTIC \nrightarrow CLASS \ \wedge \\
/*initialisation & des & constantes */ \\
subClass & = & \emptyset \ \wedge \\
assocAssocEnd1 & = & \{Pret \mapsto c1\} \ \wedge \\
assocAssocEnd2 & = & \{Pret \mapsto c2\} \ \wedge \\
characClass & = & \{c1 \mapsto Client, c2 \mapsto Cassette\}
\end{array}
$$

END;

A la lecture de ces machines apparaît un premier bémol à l'approche par méta-modélisation, dû aux contraintes de la méthode B. En effet, comme les paramètres d'une machine B ne peuvent pas être des variables de la machine incluante, ils sont alors déclarés comme des ensembles ou des constantes sur lesquelles sont définies des propriétés. Ainsi les variables qui représentent les classes du méta-modèle définies dans la table 3.1 sont confondues avec leur type. On perd également de la précision au niveau de la spécification des relations qui sont alors définies entre ces types, une fonction totale devenant une fonction partielle.

Un deuxième problème, plus grave, apparaît quand on essaie de structurer la machine *Application*. Si pour la répartition de la partie statique dans différents machines abstraites on arrive à une solution satisfaisante, il n'en est pas de même pour la répartition des opérations. Celle-ci implique que les instances de l'application (objets et liens) soient réparties dans les différentes machines abstraites, empêchant ainsi une vérification des contraintes sur l'ensemble de la base.

En conclusion sur l'approche par méta-modélisation, on peut dire que les résultats ne sont pas convaincants, essentiellement à cause des mécanismes de structuration d'une spécification qui se définissent formellement moins facilement qu'ils ne se simulent (approche par traduction). Je ne suis pas sûre que ce soit dû au langage de spécifications étudié. Certes, nous n'avons pas étudié aussi en détails d'autres langages formels mais une rapide étude de VDM et Z nous a amenés à la même conclusion. On peut donc se poser la question si oui ou non les mécanismes des langages de spécifications formelles actuels sont assez élaborés et adaptés pour permettre une totale séparation entre formalisation du méta-modèle et formalisation des applications. Reste la solution de créer *exnihilo* un langage spécifique qui répond à nos exigences, ce que nous nous sommes refusés de faire pour les raisons déjà exposées dans l'introduction de ce mémoire.

3.3 Conclusion

Concernant le type d'approche la mieux adaptée pour la spécification de SI, on peut conclure de l'étude que nous avons faite que l'approche par traduction est certainement une utilisation plus directe et plus naturelle d'un langage de spécifications formelles. Cette étude remonte maintenant à cinq ans. Depuis, d'autres langages de spécifications sont apparus, en particulier le langage FOC, développé conjointement au LIP6 et au CEDRIC [PDH02, FOC]. C'est un langage dédié au calcul formel mais d'autres applications pourraient être envisagées. Une de ses caractéristiques intéressantes est qu'il a un mécanisme de structuration proche des concepts objets.

Cependant, indépendamment de la spécification formelle des applications, la formalisation du méta-modèle des SI présente un intérêt propre : elle permet en particulier de clarifier et d'évaluer tous les (nombreux) concepts des SI. C'est ce que j'ai exposé dans la première partie de ce chapitre. Dans le cas de UML, on obtient un langage graphique formalisé, dédié au SI, ce qui est déjà en soi

un résultat intéressant. On obtient ainsi une spécification non ambiguë, sans nécessairement avoir besoin d'aller jusqu'à l'obtention d'une spécification formelle complète.

Un autre intérêt de cette formalisation du méta-modèle est que nous avons pu très rapidement développer un outil, implémenté en OCAML, dont les fonctions sont une ré-expression directe de la spécification du méta-modèle en OCAML [LM00, LM01]. Cet outil prend en entrée les diagrammes élaborés en utilisant l'atelier ROSE dédié à UML et les traduit en spécifications B. Ces spécifications peuvent être ensuite passées à l'Atelier B pour analyse et preuve.

C'est finalement un point de départ pour pouvoir naviguer entre les notations graphiques et textuelles. Il est en effet possible d'associer non seulement à chaque concept graphique les éléments du langage B correspondants mais aussi réciproquement à chaque élément de langage B les concepts graphiques où il est utilisé. Nous avons pour cela construit un méta-modèle du langage B dédié à la spécification des SI [LP02].

Chapitre 4

Spécification des propriétés fonctionnelles des transactions

Rappelons qu'un système d'information se décompose en trois parties: l'interface, la base de données et la partie fonctionnelle qui définit les transactions permettant de calculer les valeurs de sortie du système à partir des valeurs d'entrée et des données de la base. Les données d'un SI sont spécifiées à un haut niveau d'abstraction en utilisant un modèle sémantique. Ce n'est pas le cas des transactions qui sont spécifiées de manière informelle et incomplète, la seule description précise qu'on en a étant le code source lui-même. Mon objectif est d'utiliser un langage de spécifications formelles pour les décrire. Plus précisément, je m'intéresse à la description des mises à jour en utilisant le langage B. Il est alors possible de s'assurer de la cohérence des transactions, c'est-à-dire que les données et les contraintes d'intégrité du système sont préservées par ces mises à jour. Je ne traite pas pour le moment d'autres types de propriétés liées au comportement comme des propriétés de vivacité. Ceci a nécessairement un impact non seulement sur le choix du langage de spécifications comme je l'ai expliqué au chapitre 2, mais aussi sur la manière de construire la spécification et ses caractéristiques. Je reviendrai sur ce point dans la conclusion du chapitre. Enfin, dernier point à préciser : je modélise des systèmes déterministes, ce qui signifie que chaque événement externe au système va déclencher l'exécution d'une seule transaction.

Dans un premier temps, le diagramme des classes représentant la statique (les données du système) est défini et traduit en B. Puis les opérations de base, indépendante des transactions, sont générées automatiquement. Possédant cette connaissance, le spécifieur peut alors définir ses transactions qui prennent en compte les règles de gestion du système. Les transactions en général ne sont pas très complexes d'un point de vue algorithmique. La difficulté est plutôt de bien préciser toutes les opérations de mise à jour et les préconditions qui vont faire en sorte que les contraintes d'intégrité seront respectées.

En UML, comme dans toute méthode de conception, plusieurs diagrammes sont disponibles pour décrire l'aspect fonctionnel d'un système comme les diagrammes de séquences, de collaborations, d'activités ou d'états-transitions. Mais leur sémantique est encore moins bien définie que pour les données, principalement car ils se veulent universels, c'est-à-dire qu'ils doivent pouvoir servir à modéliser n'importe quel type d'applications (temps réel, systèmes réactifs, bases de données, ...) où des concepts tels que par exemple une garde, un événement n'ont pas toujours la même sémantique. Une autre solution est d'utiliser uniquement le diagramme de classes avec la possibilité de distinguer les classes qui servent à décrire ces fonctionnalités, des classes servant à décrire les données du système. Enfin, on peut aussi mixer les deux solutions, ce qui offre plus de souplesse au

spécifieur. Il est alors nécessaire de s'assurer de la cohérence globale de la spécification. C'est cette dernière solution que j'ai retenue.

Cette étude avait préalablement été réalisée pendant la thèse de Phuong Nguyen, avec la méthode OMT. Mais celle-ci offre moins de possibilités pour spécifier l'aspect fonctionnel que UML. C'est pourquoi j'ai repris l'étude avec UML, en reprenant bien sûr certaines idées, surtout en ce qui concerne l'architecture de la spécification formelle B qu'on cherche à obtenir.

La démarche de définition de la syntaxe de IS-UML pour la spécification des transactions est la même que celle définie dans le chapitre 3. Le méta-modèle définissant la syntaxe abstraite est spécifié graphiquement puis formellement en B. Différentes règles de la sémantique statique peuvent alors être exprimées. Pour la sémantique dynamique seul un plongement superficiel a été réalisé en B. L'inconvénient est qu'on ne sépare plus les deux aspects : formalisation du langage graphique IS-UML d'une part et traduction en spécifications formelles d'autre part. On ne peut donc pas utiliser les diagrammes UML seuls et raisonner sur eux pour prouver des propriétés. Trouver une manière de formaliser cette sémantique est un point que j'étudie avec Fiona Polack.

Ce chapitre résume les différents éléments du langage IS-UML pour la spécification des transactions. Les références [FLNM99, FLMP99, Lal00, LM01] donnent une description plus détaillée.

4.1 Diagrammes d'états-transitions et de collaborations

En UML, ces diagrammes sont utilisés à différentes phases du processus de développement d'un système. C'est intéressant pour garantir la traçabilité du développement mais malheureusement, il n'est pas clairement établi quels sont les concepts à utiliser à chaque niveau. D'un autre coté cela permet à chacun de le faire pour les propres besoins de ses applications.

4.1.1 Diagrammes d'états-transitions

Un diagramme d'états-transitions en UML est un graphe dont les noeuds sont appelés états et les arcs dirigés sont des transitions qui peuvent être étiquetés par des événements. Graphiquement, les diagrammes d'états-transitions sont similaires aux Statecharts introduits par Harel [Har87], mais leur sémantique est volontairement laissée informelle pour pouvoir modéliser n'importe quels types de systèmes. Mon objectif a été de définir les concepts utiles dans notre cadre et de leur donner une sémantique formelle.

Présentation et exemple

En IS-UML, un diagramme d'états-transitions est attaché à une classe, il est défini pour décrire les cycles de vie des objets de cette classe. Les événements sont donc les événements qui peuvent apparaître dans la vie d'un objet: création, suppression, modification de son état, c'est-à-dire des valeurs de ses attributs ou de ses liens avec des objets d'autres classes. Je rappelle que les opérations de base associées à une classe ou une association sont automatiquement définies à partir uniquement de la statique du système, plus précisément à partir des caractéristiques propres de la classe. Le diagramme d'états-transitions permet de prendre en compte les règles de gestion du système et les contraintes d'intégrité. Il sert donc d'abord à préciser sous quelles conditions les opérations de base de la classe sont appelées.

Les états décrivent les situations pérennes possibles dans la vie d'un objet. En conséquence un état est une abstraction des valeurs des attributs ou des liens de la classe et est défini par un prédicat. Une transition permet de faire passer un objet d'un état vers un autre état. On peut lui associer une

action qui sera exécutée lors de la transition. Cette action correspond nécessairement à une opération de base de la classe à laquelle est associé le diagramme. Une transition peut être déclenchée par un événement avec éventuellement une garde. Un événement appartient à un seul diagramme d'états-transitions mais peut déclencher plusieurs transitions dans un diagramme d'états-transitions. Si ces transitions partent toutes du même état alors les gardes doivent être disjointes car nous modélisons des systèmes déterministes.

Il y a deux états particuliers: l'état initial et l'état final. L'état initial est le point de départ du diagramme. Les transitions qui en sortent vont spécifier comment l'objet est créé. L'état final indique que l'objet n'existe plus. Les transitions qui y arrivent vont spécifier comment l'objet est supprimé du système.

UML définit quatre sortes d'événements : appel d'opération, émission d'un signal, événement temporel, événement conditionnel. Pour le type de propriétés que nous voulons vérifier, seul la première catégorie nous intéresse. Une occurrence d'événement est un appel d'opération et la garde associée est une précondition. Ceci signifie qu'à l'arrivée d'une occurrence d'événement, si la garde est fausse alors rien n'est garantit sur le résultat de l'opération déclenchée par cette occurrence. La vérification de la garde est faite par l'opération qui envoie l'événement, comme nous le verrons dans le paragraphe 4.3.

Nous avons éliminé d'autres concepts de UML qui soit ne nous semblaient pas utiles en phase de spécification, soit mal adaptés à notre domaine. Par exemple, le fait de pouvoir spécifier pour un état des actions d'entrée ou de sortie signifie que cette action sera réalisée à chaque fois qu'on entre ou qu'on sort de l'état. C'est plus une optimisation qui peut être prise en compte lors de la phase de conception. Nous avons aussi éliminé l'envoi d'événements vers des diagrammes d'états-transitions associés à d'autres classes lors de l'exécution d'une transition. Le fait qu'un événement concerne plusieurs classes est pris en compte dans les diagrammes de collaboration.

En revanche, pour le moment, d'autres concepts ne sont pas considérés et sont néanmoins intéressants comme par exemple les sous-diagrammes d'états-transitions (associés aux liens de composition du diagramme de classe) ou l'impact des hiérarchies de généralisation/spécialisation du diagramme de classes sur les diagrammes d'états-transitions des classes associées [Gra98].

Prenons l'exemple du club video. La figure 4.1 présente le diagramme d'états-transitions d'un client. A sa création, un client passe dans l'état *ClientEnRegle*. Il peut alors emprunter des cassettes, les rendre (ces opérations apparaissent dans le diagramme d'états-transitions de la classe *Cassette*). L'action déclenchée à l'arrivée de l'événement *SuppressionClient* dépend de l'état de ses emprunts. S'il n'en a pas, alors il est possible de le supprimer, sinon, il devient un *ClientSuspendu*, à qui par exemple il sera impossible d'emprunter. On peut imaginer qu'au bout d'un certain temps il pourra quand même être supprimé (événement *SuppressionClientSuspendu*).

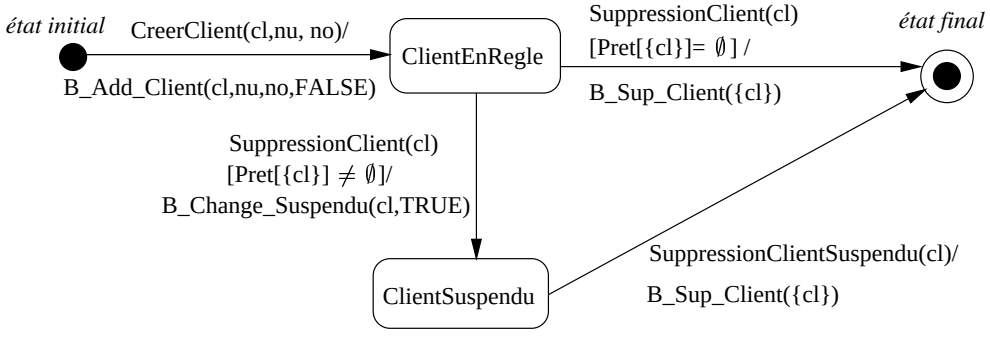

FIG. 4.1 – *Diagramme d'états-transitions de la classe Client*

UML n'impose pas de langage spécifique pour annoter les diagrammes. On peut le faire en langue naturelle ou en OCL [WK99]. Mais OCL pose encore des problèmes de sémantique [VJ99]. Comme nous voulons générer une spécification B il semble logique d'utiliser des notations B pour annoter les diagrammes. Le spécifieur va alors s'appuyer sur les notations B déjà obtenues lors de la traduction du diagramme de classes et la génération des opérations de base. On pourrait adopter un style "plus objet", en particulier pour la désignation des attributs ou associations liés à un objet comme proposé dans [SB01].

Formalisation et traduction en B

Le méta-modèle de la syntaxe abstraite est décrit graphiquement par un diagramme de classes (Figure 4.2), puis formellement en utilisant les règles de traduction du chapitre 2.

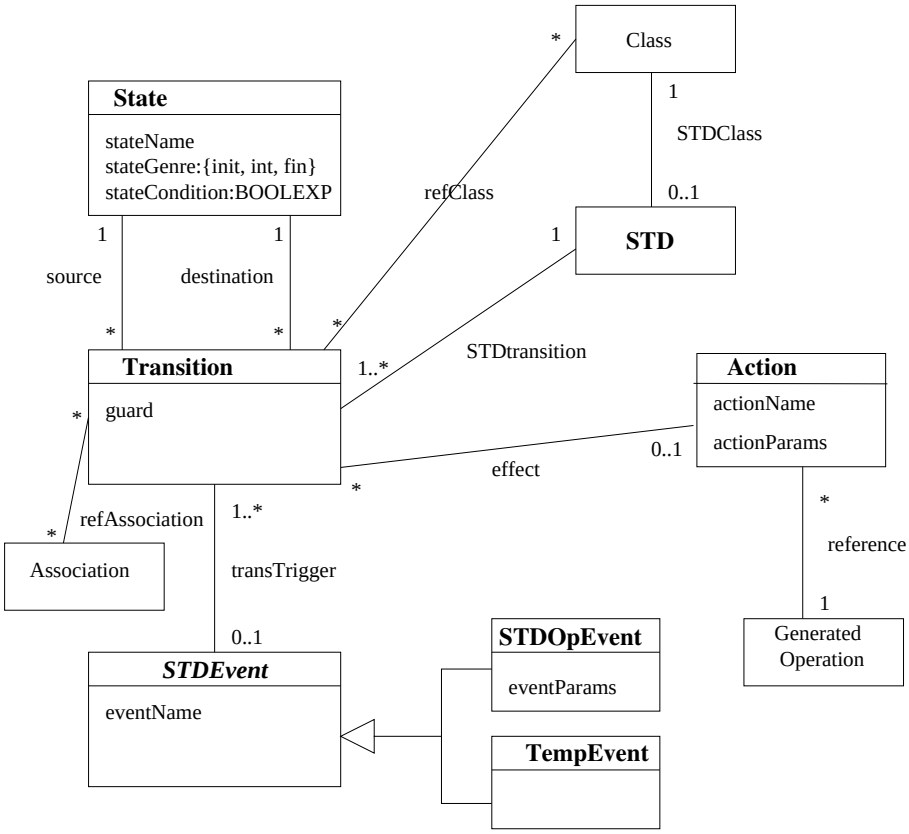

FIG. 4.2 – *Méta-modèle des diagramme d'états-transitions*

La sémantique d'un diagramme d'états-transitions est donnée par les règles de traduction en B de ses différents éléments. Pour chaque diagramme d'états-transitions attaché à une classe c est créée une machine abstraite $State_c$ qui inclut la machine Mc associée à la classe c et qui utilise, si nécessaire, les machines où sont déclarées les variables qui apparaissent dans les gardes. Chaque prédicat définissant un état est traduit par une définition dans cette machine. Chaque événement est traduit par une opération B qui appellent les opérations de base précisées dans les actions. C'est une opération dans le sens B "classique", c'est-à-dire spécifiée par pré/post-conditions. La sémantique d'un événement est donc totalement différente de celle d'un événement dans le B événementiel [AM98, Abr02].

La figure 4.3 donne les différentes possibilités de déclenchement de transitions par un événement.

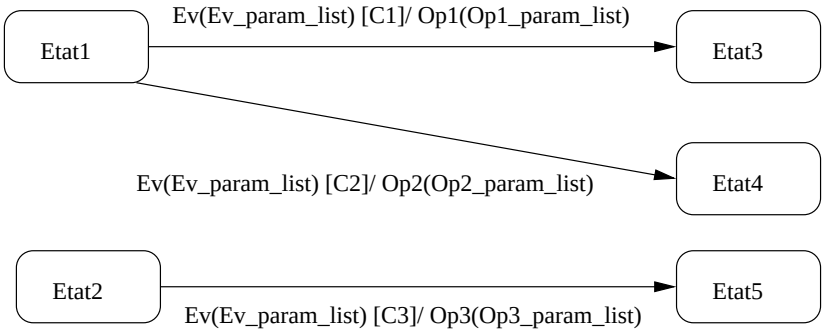

FIG. 4.3 – *Un événement déclenchant plusieurs transitions*

L'événement Ev est traduit par l'opération suivante :

$\mathsf{Ev}(ev_param_list) \triangleq$
PRE
Typage des paramètres $\wedge$
$((PredEtat_1 \wedge C_1 \wedge PC_Op_1) \vee$
$(PredEtat_1 \wedge C_2 \wedge PC_Op_2) \vee$
$(PredEtat_2 \wedge C_3 \wedge PC_Op_3))$
THEN
SELECT $(PredEtat_1 \wedge C_1)$ THEN $Op_1(Op_1_param_list)$
WHEN $(PredEtat_1 \wedge C_2)$ THEN $Op_2(Op_2_param_list)$
WHEN $(PredEtat_2 \wedge C_3)$ THEN $Op_3(Op_3_param_list)$
END
END

Avec :
- $PredEtat_i$ le prédicat associé à l'état $Etat_i$
- C_i la garde associée à la transition déclenchée par l'événement
- PC_Op_i la précondition de l'opération Op_i
- Op_i une opération de base de la classe c

4.1.2 Diagrammes de collaborations

En UML, le comportement global d'un système est décomposé en un ensemble de cas d'utilisation (*use cases*) qui permettent de définir les limites du système analysé et les différents acteurs. Chaque use case regroupe un ensemble de fonctionnalités (transactions) sémantiquement liées, comme par exemple *GestionDesPrets* ou *GestionDesClients* dans le système du club video. Une transaction appartient à un seul use case.

Présentation et exemple

En UML, un diagramme de collaborations est utilisé pour décrire comment différents objets participent et interagissent pour réaliser une tâche en modélisant les messages échangés entre eux.

Cela peut aller de la description d'un scénario à la manière dont est implémentée une opération.

En IS-UML, un diagramme de collaborations va permettre de préciser pour une transaction qui concerne plusieurs classes quelles opérations sont appelées et sous quelles conditions. Au niveau de la phase de spécification, c'est la seule utilisation possible de ce diagramme. En conséquence, le format général d'un diagramme est celui de la figure 4.4.

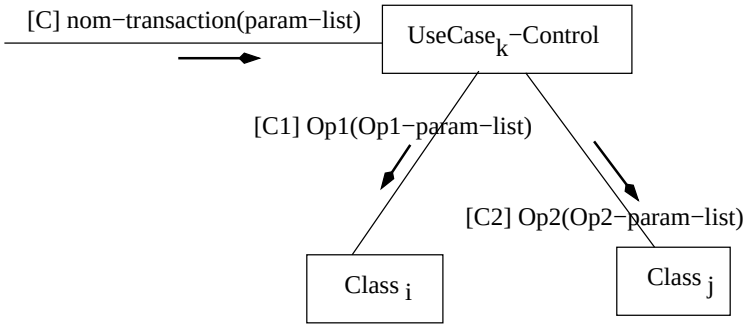

FIG. 4.4 – *Format général d'un diagramme de collaborations*

Le message[1] *nom-transaction(param_list)* correspondant à l'exécution d'une transaction arrive à une classe spéciale, *UseCase$_k$-Control*, qui gère toutes les transactions d'un même use case. Pour réaliser cette transaction, plusieurs messages sont envoyés aux classes concernées. Chaque message $Op_i(Op_i_param_list)$ correspond soit à un événement du diagramme d'états-transitions de la classe correspondante, soit à une opération de base de cette classe. La garde C a la même sémantique qu'une garde d'un diagramme d'états-transitions: c'est toujours une approche de spécification offensive. En revanche la sémantique des gardes C_i est différente : c'est une condition sur l'émission du message associé. Si la garde est vraie alors le message est effectivement émis sinon rien ne se passe.

Reprenons l'exemple du club video. Le diagramme de collaborations de la figure 4.5 décrit la transaction de radiation d'un client. Cette transaction consiste à envoyer le message *SuppressionClient* au diagramme d'états-transitions de la classe *Client*. Si le client a encore des prêts alors on va les supprimer et les cassettes empruntées sont considérées comme perdues (on n'incrémente pas l'attribut *Nbdispo*).

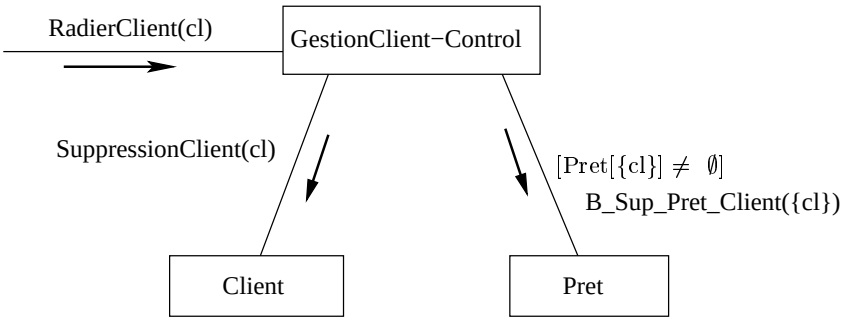

FIG. 4.5 – *Diagramme de collaborations de la transaction RadierClient*

Comme pour les diagrammes d'états-transitions, les diagrammes de collaborations sont annotés par des notations B. Le spécifieur va s'appuyer sur les notations B déjà obtenues lors de la traduction

1. En terminologie UML, on ne parle pas d'événements dans un diagramme de collaborations mais de messages

des diagrammes d'états-transitions, du diagramme de classes et la génération des opérations de base.

Formalisation et traduction en B

La syntaxe abstraite est définie par un méta-modèle décrit graphiquement par la figure 4.6) et traduit ensuite en B. La sémantique est donnée par les règles de traduction en B.

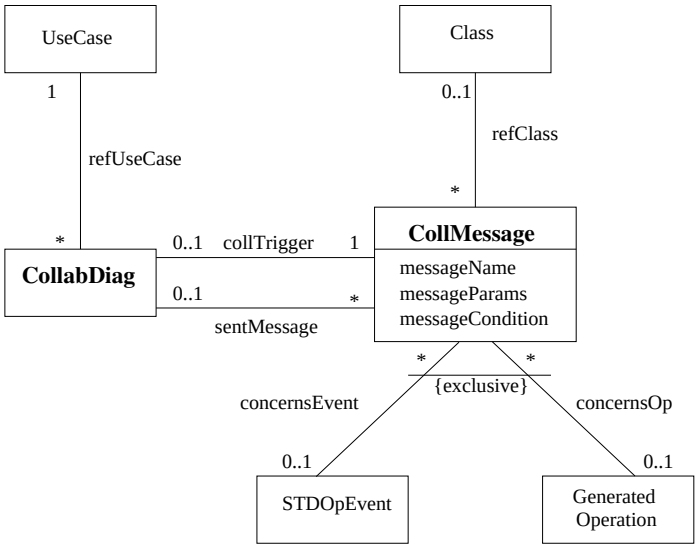

FIG. 4.6 – *Méta-modèle des diagrammes de collaborations*

Chaque diagramme est traduit par l'opération B suivante:

$\mathsf{nom\text{-}transaction}(param_list) \triangleq$
PRE
Typage des paramètres $\wedge\ C\ \wedge\ PC_Op_1\ \wedge\ PC_Op_2$
THEN
IF C_1 THEN $Op_1(Op_1_param_list)$ END ||
IF C_2 THEN $Op_2(Op_2_param_list)$ END
END

La transaction *RadierClient* est spécifiée de la manière suivante:

$\mathsf{RadierClient}(cl) \triangleq$
PRE
$cl \in Client\ \wedge\ PC_SuppressionClient\ \wedge$
$PC_B_Sup_Pret_Client$
THEN
$SuppressionClient(cl)$ ||
IF $Pret[\{cl\}] \neq \emptyset$ THEN $B_Sup_Pret_Client(\{cl\})$ END
END

L'ensemble des diagrammes de collaborations d'un use case est traduit dans une machine abstraite qui va inclure les machines dans lesquelles sont définies les opérations appelées par les transactions.

L'architecture de la spécification B obtenue après traduction des deux types de diagrammes est représentée dans la figure 4.7. Les trois niveaux de l'architecture correspondent aux trois niveaux d'abstraction de la description UML d'un système: diagramme de classes, diagrammes d'états-transitions et diagrammes de collaborations. Le premier niveau, obtenu entièrement de manière automatique, contient toutes les données du système et les opérations qui les manipulent. Le deuxième niveau contient les machines correspondant aux diagrammes d'états-transitions. Chaque machine est associée à un seul élément du diagramme de classes et décrit son cycle de vie. Elle contient également les éventuelles contraintes d'intégrité intra-classes. Le troisième niveau contient les machines définissant les transactions des use case et les contraintes d'intégrité inter-classes ou inter-associations.

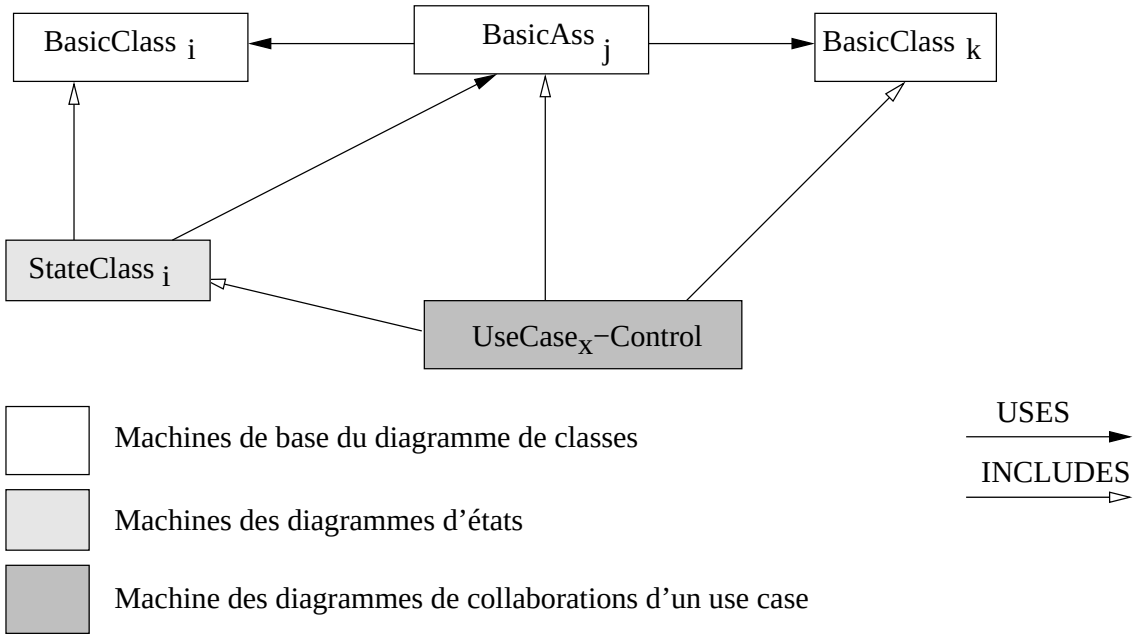

FIG. 4.7 – *Architecture globale de la spécification abstraite*

Les obligations de preuve générées par la méthode B permettent de s'assurer que les opérations correspondant aux transactions établissent bien l'invariant du système. Comme cet invariant contient les contraintes d'intégrité, on a bien obtenu une vérification des propriétés fonctionnelles du système.

Cette architecture offre une bonne compréhension des spécifications formelles B par rapport aux spécifications formelles graphiques de IS-UML. Chaque élément du SI est décrit dans une seule machine abstraite B.

4.2 Spécification des opérations dans le diagramme de classes

Cette section est en partie une contribution originale du mémoire. Elle découle de la constatation suivante issue des études de cas que j'ai pu mener. La plupart des transactions peuvent se spécifier en utilisant la méthode précédente. Cependant pour des transactions plus complexes ou qui ont des paramètres retour, elle n'est pas simple à utiliser. Nous avons commencé alors à réfléchir avec Fiona Polack à une proposition alternative exposée dans [LP01b]. Mais le lien avec les concepts d'UML n'étaient pas définis. Je me suis inspirée des travaux de Helen Treharne [Tre02] pour proposer la solution suivante.

Présentation et exemple

Il est possible de spécifier les transactions directement dans le diagramme de classes. Pour cela j'utilise une particularité très intéressante de UML, les notions de stéréotype et de propriété qui permettent de rajouter de nouveaux concepts. En fait un stéréotype permet de définir une nouvelle classe dans le méta-modèle et une propriété un nouvel attribut dans une classe du méta-modèle. Il existe déjà trois stéréotypes de classes dans UML qui correspondent aux trois parties qu'on retrouve dans un système. Les *Boundary* classes sont utilisées pour modéliser l'interface du système, les *Entity* classes sont les classes du diagramme de classes représentant la statique du système et les *Control* classes permettent d'exprimer et de coordonner les fonctionnalités du système. C'est dans ce dernier type de classes que les transactions sont spécifiées. Pour permettre de faciliter d'une part la traduction en B et d'autre part la correspondance avec les diagrammes de comportement, j'ai introduit deux nouveaux stéréotypes : les *Entity-Control* classes et les *Transaction-Control* classes. Les premières sont associées chacune à une seule *Entity* classe (ou association) et permettent de contrôler l'accès aux opérations de la classe (ou association). Les secondes décrivent effectivement les transactions du système et sont associées soit à des *Entity-Control* classes ou directement à des *Entity* classes (ou associations). Chaque *Transaction-Control* classe décrit les transactions d'un seul use case.

Sur notre exemple de video-club, considérons la transaction *EmpruntParGenre* qui consiste pour un client à emprunter n'importe quelle cassette appartenant à un genre donné. Comme elle concerne les deux classes[2] *Cassette* et $Pret$, elle est spécifiée dans une *Transaction-Control* classe, appelée *GestionDesPrets-Control*. Tout comme pour les diagrammes de comportement, UML n'impose pas de langage spécifique pour décrire les opérations. Nous conseillons donc d'utiliser les notations B pour permettre une traduction plus complète.

$ti \leftarrow$EmpruntParGenre$(cl,ge) \triangleq$
PRE
$cl \in Client \ \wedge \ ge \in GENRE \ \wedge$
$cl \in Suspendu^{-1}[\{FALSE\}] \ \wedge$
$\exists ca \in Cassette \cdot (Genre(ca) = ge \ \wedge \ Nbdispo(ca) \geq 1)$
THEN
ANY ca WHERE $ca \in Cassette \ \wedge \ Genre(ca) = ge \ \wedge \ Nbdispo(ca) \geq 1$
THEN
$Ajout_Pret(cl,ca) \ ||$
$B_Change_Nbdispo(ca,(Nbdispo(ca) - 1)) \ ||$
$ti := Titre(ca)$
END
END

L'opération de base $B_Change_NbDispo$ est dans la classe $Cassette$. L'opération $Ajout_Pret$ est une opération qui appelle l'opération B_Ajout_Pret de la classe-association $Pret$, elle définit les préconditions nécessaires à la vérification des multiplicités et des contraintes d'intégrité. Elle est spécifiée dans la *Entity-Control* classe associée à la classe-association $Pret$. On peut représenter ces différentes classes dans la figure 4.8. Il est évident que les liens qui relient les *Control* classes aux *Entity* classes ne sont pas des liens d'association comme dans le diagramme statique, ce sont plutôt des liens de type flux d'information. On distingue deux types de liens : utilisation des opéra-

2. Pour simplifier la notation, dans la suite du mémoire je continue d'appeler classe une *Entity* classe

tions (stéréotype «op») et utilisation des variables, en lecture seulement, (stéréotype «var»). Cette distinction est nécessaire pour la traduction en B.

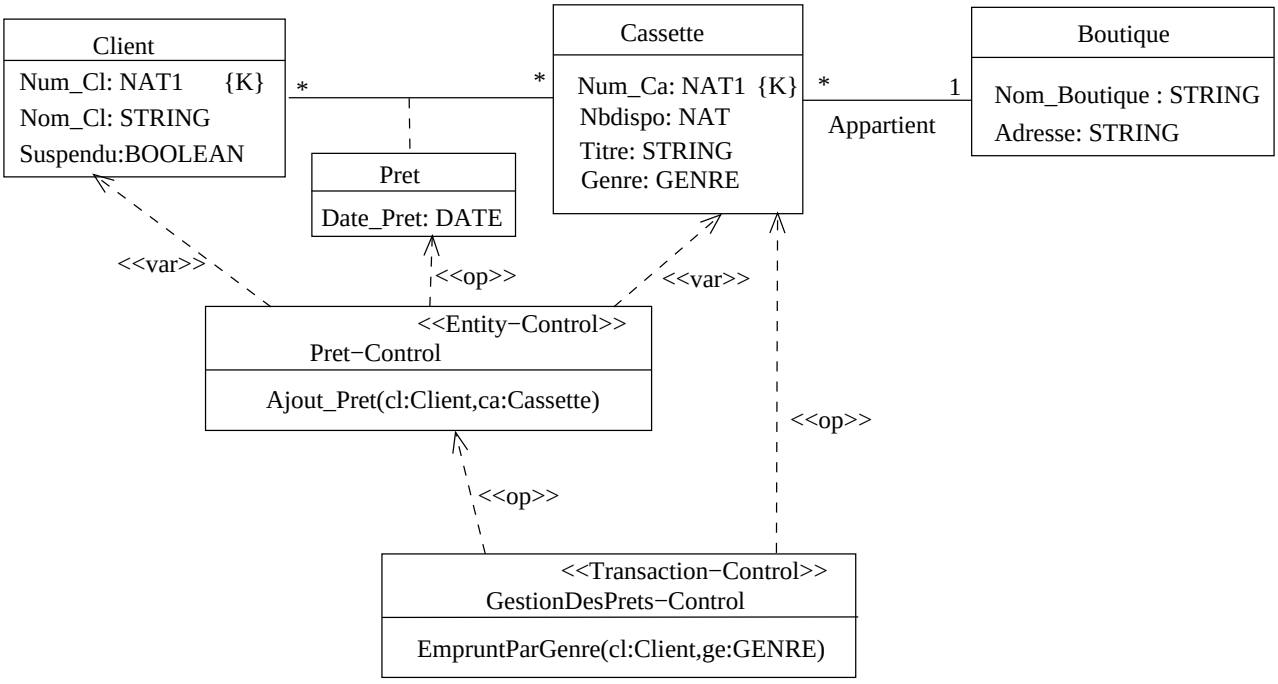

FIG. 4.8 – *Exemple de diagramme de classes avec des Control classes*

Formalisation et traduction en B

Tout comme pour les diagrammes d'états-transitions et de collaborations, je donne dans la figure 4.9 la représentation graphique de la syntaxe abstraite de cette partie de IS-UML.

Chaque *Entity-Control* classe est traduite en B par une machine qui inclut la machine de la classe associée et utilise les machines des classes dont elle lit les variables. Chaque *Transaction-Control* classe est traduite par une machine B qui inclut les machines associées aux *Entity-Control* classes et aux *Entity* classes dont elles utilisent les opérations ou les variables.

La sémantique d'une opération est donnée par sa traduction en B. Chaque opération d'une *Entity-Control* classe ou d'une *Transaction-Control* classe est traduite par une opération B. Il est clair que si l'opération dans le diagramme UML n'est pas spécifiée avec des notations B, on obtiendra seulement un squelette d'opération B lors de sa traduction.

Lien avec les diagrammes d'états-transitions et de collaborations

L'analogie est volontairement très forte entre d'une part une *Entity-Control* classe et un diagramme d'états-transitions et d'autre part une opération d'une *Transaction-Control* classe et un diagramme de collaborations. Ceci permet de mieux gérer la cohérence entre les différentes représentations d'un système. En fait la machine B correspondante est la même. Ceci signifie que chaque

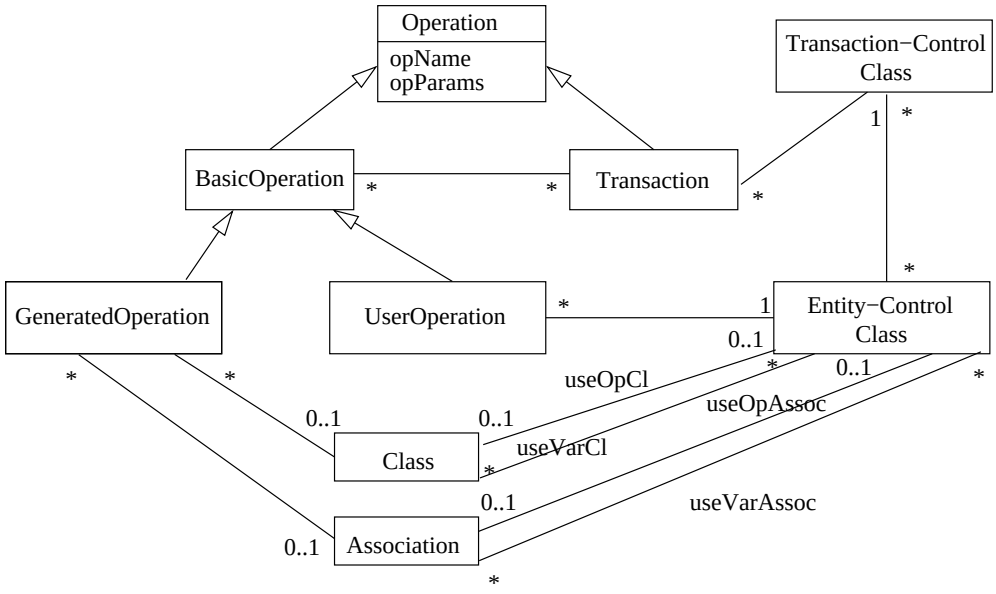

FIG. 4.9 – *Méta-modèle pour les transactions*

événement du diagramme d'états-transitions d'une classe c est traduit par une opération dans la *Entity-Control* classe associée à c. Son contenu est celui de l'opération B générée. De même chaque diagramme de collaborations est traduit par une opération dans la *Transaction-Control* classe du use case associé.

C'est une première étape qui permet d'établir des liens explicites et contrôlés entre les deux points de vue d'un système. Il est néanmoins nécessaire d'approfondir la méthode pour avoir une fusion plus complète entre les deux points de vue.

4.3 Architecture robuste

Rappelons que les opérations B ont été décrites dans une stratégie de programmation offensive : utilisation des préconditions. Pour chaque opération, on définit les conditions sous lesquelles cette opération doit être appelée afin de préserver les contraintes d'intégrité définies dans l'application. En utilisant l'AtelierB, la spécification ainsi obtenue peut être prouvée correcte. En effet, pour chaque opération Op, une obligation de preuve est automatiquement générée. Cette obligation de preuve permet de vérifier que, si la précondition Q de l'opération Op, et l'invariant Inv de la machine sont vérifiés, alors la substitution S de l'opération Op rétablit l'invariant Inv:

$$(Q \wedge Inv) \Rightarrow [S]Inv$$

Cependant, établir cette preuve n'interdit pas que l'opération puisse être appelée sans que sa précondition soit vérifiée. Le système pourra donc se retrouver dans un état incohérent. En effet, une précondition ne garantit un résultat (exécution correcte de la substitution associée) que si elle est vraie, dans le cas contraire le résultat devient imprévisible: violation des contraintes, non terminaison du calcul, infaisabilité, ... Par opposition à la précondition, une substitution conditionnée IF a toujours un comportement bien précis: selon la valeur booléenne de la condition, la branche THEN ou ELSE est exécutée. Un niveau supplémentaire de sécurité s'impose. Nous proposons pour cela de rajouter une machine (*Robuste_UseCase$_x$-Control*) qui inclut la machine *UseCase$_x$-Control* et qui permet d'assurer que chaque opération de cette dernière n'est appelée que si sa précondition

est vérifiée. A chaque opération de la machine *UseCase$_x$-Control* correspond une opération dans la machine *Robuste_UseCase$_x$-Control* dont la précondition correspond uniquement au typage des paramètres d'entrée. La machine *Robuste_UseCase$_x$-Control* contient alors **les transactions bases de données sûres**.

Reprenons l'exemple de transaction *EmpruntParGenre* de la page 49. L'opération robuste correspondante est spécifiée comme suit:

$(res,ti) \leftarrow$Robuste_EmpruntParGenre$(cl,ge) \triangleq$
PRE
$cl \in CLIENT \ \wedge \ ge \in GENRE$
THEN
IF $cl \in Client$ THEN
IF $cl \in Suspendu^{-1}[\{FALSE\}]$ THEN
IF $(\exists ca \in Cassette \cdot (Genre(ca) = ge \ \wedge \ Nbdispo(ca) \geq 1)$ THEN
$ti := EmpruntParGenre(cl,ge) \ ||$
$res := OK$
ELSE
$res := CassetteNonDisponible$
END
ELSE
$res := ClientSuspendu$
END
ELSE
$res := ClientInexistant$
END
END

Remarquons que nous supposons que le langage d'implémentation choisi est un langage typé. En effet, les ensembles abstraits $CLIENT$ et $GENRE$ qui typent les paramètres d'entrée sont traduits par des types du langage de programmation (cf chapitre 5). Pour être sûr qu'il n'y aura pas d'erreur à l'exécution de l'opération implémentée, il faut que les types des paramètres soient vérifiés à la compilation. C'est toujours le cas pour les langages de programmation qui implémentent les applications bases de données.

La construction de la machine *Robuste_UseCase$_x$-Control* n'est pas entièrement automatique. En effet, les différents messages d'erreurs ne peuvent être générés automatiquement car ils dépendent de la sémantique de la précondition associée. Le concepteur devra compléter manuellement ces parties de spécification. Il peut aussi définir, dans cette machine, ses propres stratégies de gestion d'erreurs.

On peut se demander pourquoi ne pas regrouper pour un use case les machines *UseCase$_x$-Control* et *Robuste_UseCase$_x$-Control*. C'est un choix de conception qui relève des pratiques utilisées dans le domaine des SI. La première machine contient la spécification des transactions: on se concentre uniquement sur les effets d'une transaction, les cas nominaux. La deuxième machine comprend les transactions sûres : on traite les cas d'erreurs, très nombreux en SI car liés aux contraintes d'intégrité qui elle-mêmes sont très nombreuses et aux règles de gestion.

4.4 Bilan et perspectives

Ce chapitre clôt la présentation du langage IS-UML qui permet de spécifier les données et les fonctionnalités du noyau d'un SI. L'objectif de ce langage est d'offrir une interface graphique

formelle, basée sur les notations de UML, qui soit le reflet de la spécification textuelle B. Les deux styles de notations textuelles et graphiques sont fortement imbriqués. On ne part pas de n'importe quels diagrammes UML pour obtenir du B et réciproquement, n'importe quelle spécification B ne peut pas être représentée graphiquement de cette manière.

On obtient un modèle mathématique de la spécification d'un SI, sur lequel il est possible de raisonner. Plusieurs niveaux de raisonnements existent. On peut montrer la cohérence entre les différents points de vue de la spécification (diagramme de classes d'une part, diagrammes d'états-transitions et de collaborations d'autre part). On peut montrer la correction de la spécification complète grâce aux preuves réalisées par un des outils B. On peut aussi construire des outils d'aide à la mise au point des transactions. Deux approches d'outils sont possibles. La première permet d'aider le spécifieur à spécifier ses transactions de telle sorte qu'elles respectent les contraintes d'intégrité, j'en ai présenté les principes dans [LP01b]. Une deuxième approche complémentaire permet d'aider le spécifieur à réaliser les preuves, en particulier en répercutant les erreurs décelées par le prouveur dans les diagrammes UML [LP02]. Ce sont des perspectives de travaux en cours.

IS-UML est le résultat de la combinaison des langages UML et B. Il a sa propre sémantique basée sur celle de B et celle de UML. Il définit implicitement une méthode de spécification qui fournit un guide pour construire une spécification B pour les SI. Ce guide est basé sur des pratiques classiques en analyse-conception de SI, utilisées aussi bien dans les outils AGL commerciaux que décrites dans les livres du domaine [BCN92, BP98, EN00]. Je n'ai pas remis en cause ces pratiques. Il est à peu près évident que si on donne le même cahier des charges à des personnes habituées à "faire du B classique", on n'obtiendra pas le même résultat. Il y a un compromis à réaliser entre faire du B pur et s'adapter aux habitudes des ingénieurs du domaine. C'est ce que je me suis proposée de définir.

Je suis consciente du fait que la spécification B obtenue n'est pas la plus abstraite possible. J'ai entrepris dans ce sens deux types de projets. Le premier consiste à utiliser un autre langage de spécifications que B pour spécifier les événements et les traces du système. J'en reparle plus en détails dans le chapitre 6. Le second est plus récent. Il s'agit de considérer l'évolution de B, le B événementiel [AM98, Abr02, Lop02]. C'est un travail mené en collaboration avec Nestor Lopez et Jean-Louis Boulanger. Dans cette méthode, ce sont les événements et les propriétés fondamentales du système qui sont modélisés. Plusieurs applications ont déjà été spécifiées avec cette méthode. Une des difficultés est de définir les propriétés fondamentales qui seront vérifiées par le système final. Par exemple, dans le domaine ferroviaire, les propriétés fondamentales sont des propriétés de sécurité, comme par exemple : "deux trains ne peuvent jamais se croiser sur la même voie" ou " les feux de signalisation ne sont pas tous au rouge en même temps". Dans le domaine des SI, il y a beaucoup de propriétés à satisfaire et il ne me paraît pas facile de mettre en évidence les propriétés fondamentales, ou tout simplement d'établir une hiérarchie de propriétés.

Chapitre 5

Raffinement vers une implémentation relationnelle

Les chapitres précédents ont décrit une méthode combinant UML et B qui permet d'obtenir la spécification formelle d'un SI. Il devient donc possible de vérifier la cohérence des transactions sans avoir besoin d'attendre la phase de tests. Je souhaitais aller plus loin et utiliser un processus formel de raffinement qui permet d'obtenir une spécification concrète dont on peut prouver qu'elle est conforme à la spécification abstraite. Une spécification concrète est caractérisée par le fait qu'elle est directement traduisible en un ensemble de programmes écrits dans le langage de programmation cible choisi. Il est donc nécessaire de définir les constructions de ce langage de programmation dans le langage de spécification, utilisé par le processus de raffinement, de telle sorte que la traduction soit uniquement syntaxique. Dans le cadre de ce travail, le résultat final est une implémentation relationnelle constituée d'un schéma de bases de données relationnelles décrit en SQL et d'un code écrit dans le langage de programmation JAVA auquel sont rajoutées des primitives d'accès à un SGBD relationnel.

Les principes de base du processus de raffinement sont d'une part de remplacer les variables abstraites par des variables concrètes et d'autre part de résoudre l'indéterminisme des substitutions abstraites. Le processus se réalise en plusieurs étapes. La correction du processus global est assurée par la preuve de chaque étape du raffinement. Elle repose sur les propriétés du langage d'implémentation cible choisi, c'est-à-dire, dans notre cas, les constructions SQL qui ont été définies.

Le processus de raffinement est implémenté en B, d'une part, pour rester cohérent avec le choix du langage de spécifications et d'autre part, pour bénéficier des outils associés. Par ailleurs, il existe déjà un algorithme qui permet de générer automatiquement le schéma d'une base de données relationnelles à partir d'un diagramme de classes [Cer83, BP98]. C'est un algorithme très utilisé et qui est implémenté dans la plupart des outils AGL. On va donc s'appuyer sur lui pour définir les étapes de raffinement des données.

L'autre aspect à considérer est la construction d'un outil, support du processus de raffinement. Il doit offrir une assistance pour le raffinement proprement dit mais aussi pour l'élaboration des preuves. En ce qui concerne ce dernier point, l'objectif est de tirer profit du caractère générique de l'algorithme de génération pour définir des briques de preuve de base qu'il faut pouvoir réutiliser pour l'établissement de preuves plus complexes. Il s'agit donc de définir des conditions de réutilisation, en fonction des différentes formes de substitutions qui sont combinées.

Ceci a fait l'objet de la thèse de Amel Mammar [Mam02] dont je vais présenter les points importants et originaux.

5.1 Raffinement en B

Un raffinement B agit sur les données et/ou les substitutions. Le raffinement des données consiste à introduire de nouvelles variables ou à remplacer les variables du composant raffiné par des variables plus proches de celles utilisées dans le langage d'implémentation choisi. Un prédicat J, appelé invariant de collage et défini dans l'invariant du composant raffinant, permet de typer les nouvelles variables introduites et d'exprimer la relation existante entre ces variables et les variables du composant raffiné. Tout raffinement de données implique le raffinement des substitutions en fonction des variables concrètes. Le raffinement algorithmique consiste à remplacer une substitution abstraite S par une substitution concrète T.

Pour prouver la correction d'un raffinement, nous utilisons la sémantique relationnelle des substitutions qui permet une meilleure traduction de l'idée intuitive qu'on peut avoir du raffinement et du changement d'état. Deux prédicats sont associés à chaque substitution S : le prédicat de terminaison Trm et le prédicat avant-après $Prd_{x,x'}$.

$$Trm_x = [S]true$$

$$Prd_{x,x'} = \neg[S](x' \neq x)$$

x représente l'ensemble des variables libres de la substitution S. Les symboles x et x' représentent respectivement l'état des variables avant et après l'exécution de la substitution S. Intuitivement, $Trm_x(S)$ définit les conditions assurant l'exécution correcte de S (S termine), et $Prd_{x,x'}(S)$ établit le lien entre les valeurs des variables avant et après l'exécution de S, i.e. entre x et x'. Les deux prédicats sont définis inductivement sur le type de la substitution [Abr96].

Pour toutes substitutions S et T, S est raffinée par T avec l'invariant de collage J, ($S \sqsubseteq_J T$), si et seulement si [1] :

(i) $\exists(u,v).J(u,v)$
(ii) $Trm_x(S) \wedge J(x,y) \Rightarrow (Trm_y(T) \wedge \forall y''.(Prd_{y,y''}(T) \Rightarrow \exists x'.(Prd_{x,x'}(S) \wedge J(x',y''))))$

Les ensembles de variables x et y désignent les variables libres de S et T respectivement. Intuitivement, cette définition signifie que :

- il existe au moins une interprétation (u,v) du couple de variables (x,y) qui satisfait l'invariant de collage J : J n'est pas une contradiction,
- si l'exécution de la substitution abstraite S termine, alors la substitution concrète T termine aussi, et pour tout résultat possible y'' de l'exécution de T alors, il existe un résultat x' de l'exécution de S tel que le couple (x', y'') vérifie l'invariant de collage J.

Un composant M_1 est raffiné par un composant M_2 si et seulement si les deux conditions suivantes sont vérifiées :

1. l'initialisation de M_2 raffine l'initialisation de M_1,
2. chaque opération de M_2 raffine l'opération correspondante dans M_1. Une opération op_1 et son raffinement op_2 ont la même signature.

1. Comme je n'en ai pas besoin dans la suite du chapitre et afin de simplifier la présentation, je n'ai pas tenu compte des paramètres des opérations dans l'expression des conditions.

L'implémentation est la dernière étape du processus de raffinement B. Elle produit une spécification concrète, c'est-à-dire directement traduisible dans le langage de programmation choisi, contenu dans un composant B appelé $IMPLEMENTATION$. Contrairement à un raffinement, une implémentation ne possède que des variables concrètes (directement implémentables). Chaque ensemble abstrait doit être valué. Les substitutions préconditionnées, simultanées, non-déterministes (e.g. ANY) ne sont pas admises. Seules celles qui ont une correspondance directe dans le langage de programmation choisi peuvent être utilisées. Si des variables de l'implémentation ne sont pas directement traduisibles dans le langage cible choisi, on peut importer (clause $IMPORTS$) des machines abstraites qui vont permettre de les implémenter.

5.2 Présentation générale de la méthode

Je rappelle que l'objectif de notre processus de raffinement est d'obtenir une spécification concrète qui puisse ensuite être traduite en une implémentation relationnelle. La spécification concrète va donc s'appuyer sur la spécification des constructions SQL utilisées.

5.2.1 Description du processus de développement global

Le processus, schématisé dans la figure 5.1, est constitué de deux étapes : raffinement d'une spécification abstraite d'un SI et traduction vers une implémentation relationnelle. La première étape se décompose elle-même en deux phases :

- La première phase consiste à raffiner les données et les opérations de base. C'est une phase entièrement automatique. Les données sont raffinées en utilisant l'algorithme décrit dans [BP98]. Le dernier niveau de raffinement est réalisé en important les machines abstraites qui implémente les données. Ces machines correspondent aux différentes relations du schéma relationnel. Chaque machine importée précise la structure d'une relation et les opérations standard qui permettent de la manipuler.
  Les substitutions des opérations de base sont raffinées en même temps: elles sont réécrites en fonction des variables du composant qui raffine. Si on considère l'architecture de la spécification présentée dans la figure 4.7 de la page 48, cette phase consiste à raffiner toutes les machines de base en même temps (c'est nécessaire car certaines variables sont partagées). Les machines $StateClass_i$ et $UseCase_x$-$Control$ ne sont pas modifiées. En effet, *toutes* les variables du SI sont définies dans les machines de base et, comme le processus de raffinement B est monotone et que les opérations de ces machines sont toujours exprimées comme une combinaison des opérations de base, leur raffinement se ramène au raffinement des opérations de base qui les composent.
- La seconde phase raffine les opérations des machines $StateClass_i$ et $UseCase_x$-$Control$. Dans une spécification, les opérations peuvent utiliser des substitutions non-déterministes non autorisées dans une implémentation. Cette phase nécessite l'intervention du spécifieur qui doit choisir une substitution concrète pour chaque substitution non-déterministe.

La seconde étape du processus de développement consiste à traduire les machines abstraites représentant les relations et la spécification concrète des opérations. Pour le moment, nous n'avons pas étudié la possibilité d'automatiser cette traduction.

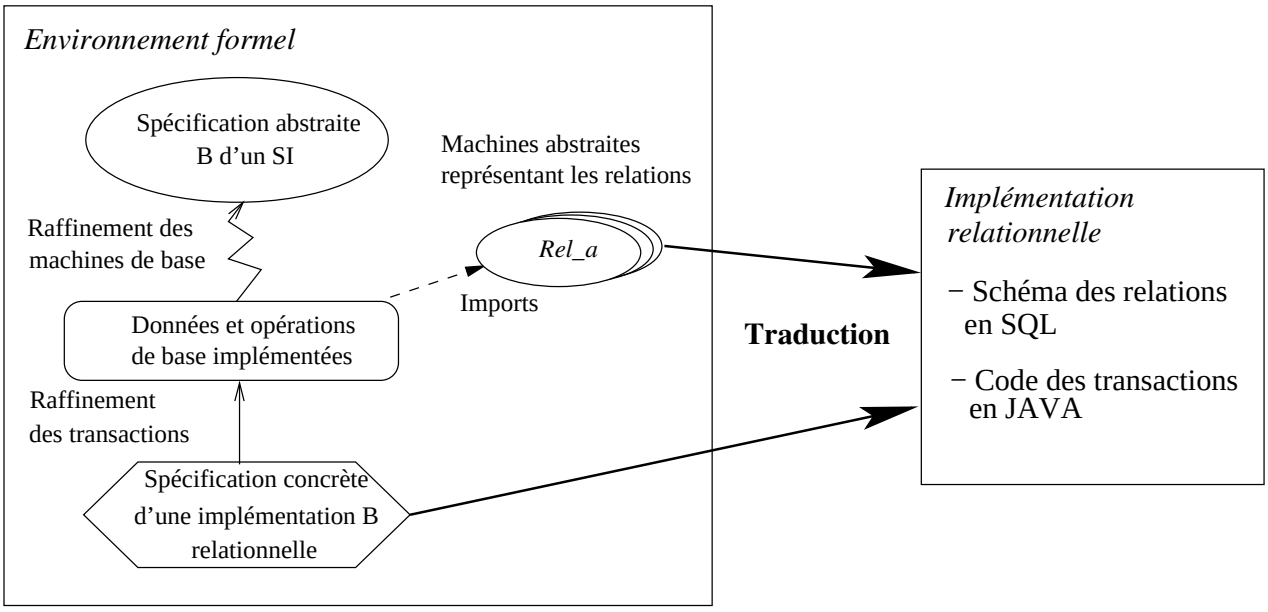

FIG. 5.1 – *Architecture du processus de développement*

5.2.2 Définition des machines abstraites représentant les relations

Ces machines abstraites contiennent la représentation, dans le langage de spécifications B, des constructions SQL utilisées dans l'implémentation relationnelle. Elles permettent d'identifier l'arrêt du processus de raffinement. Celui-ci termine quand les variables raffinées peuvent être implémentées par les variables de ces machines.

La structure de données de base du modèle relationnel est la relation, définie comme un sous-ensemble du produit cartésien du type de ses attributs. En B, une relation est représentée par le type de données *struct*. Chaque champ de ce constructeur correspond à un des attributs de la relation. Une machine abstraite spécifiant une relation contient une variable qui représente la relation et les opérations qui permettent de la manipuler (ajout d'un tuple, suppression d'un tuple ou d'un ensemble de tuples, mise à jour d'un attribut, ...).

Par exemple, la machine abstraite qui décrit une relation *Rel_cassette* est spécifiée comme suit :

MACHINE $Rel_cassette$
SETS
$GENRE$
VARIABLES
$Table_Cassette$
INVARIANT
$Table_Cassette \subseteq$
$struct(Num_Ca : NAT1, Nbdispo : NAT, Titre : STRING, Genre : GENRE,$
$Appartient_Mig : NAT)$

INITIALISATION
$Table_Cassette := \emptyset$
OPERATIONS
Insert_Cassette$(nu,ti,nb,ge,bo) \triangleq$

PRE
$nu \in NAT1 \wedge ti \in STRING \wedge nb \in NAT \wedge ge \in GENRE \wedge bo \in NAT$
THEN
$Table_Cassette := Table_Cassette \cup$ rec(nu,nb,ti,ge,bo)
END;
...
END

5.3 Raffinement des données et opérations de base de la spécification

Le passage d'une spécification d'un SI dans un modèle sémantique de données à une spécification dans le modèle relationnel implique la résolution des trois points suivants.

- La représentation de l'identification des données est différente dans les deux modèles. En relationnel, un tuple est identifié à partir de la *valeur* de certains de ses attributs qui constituent la clé de la relation. Dans les modèles sémantiques, un objet a une *identité* propre, indépendante de la valeur de ses attributs (cf chapitre 2, page 16). Le premier objectif du raffinement est de réaliser ce changement d'identification qui consiste à définir des clés dans chaque classe et à transformer les associations.
- Certains concepts des modèles sémantiques des données ne sont pas supportés par le modèle relationnel : l'héritage et les attributs de type complexe (séquence, Partie de, multivalué). Le second objectif du raffinement est de coder ces concepts en relationnel.
- Dans la spécification B initiale, nous avons une représentation fonctionnelle des structures de données : chaque attribut est défini comme une fonction de la variable correspondant à une classe ou une association vers le type de l'attribut. Le troisième objectif du raffinement est de passer à une représentation où chaque classe ou association est représentée comme un sous-ensemble du produit cartésien du type de ses attributs, comme décrite dans la section précédente.

Le schéma 5.2 décrit les différentes étapes du raffinement et précise l'ordre dans lequel s'effectuent les différents points évoqués ci-dessus.

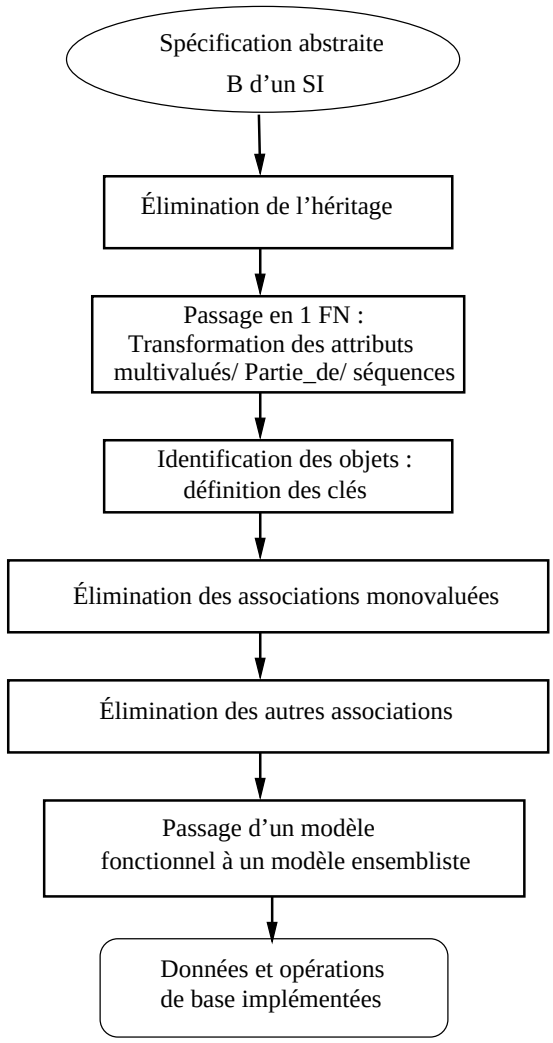

FIG. 5.2 – *Étapes du raffinement des données*

Chaque étape de raffinement est composée de deux types de règles complémentaires : raffinement des structures de données et raffinement des substitutions agissant sur les données raffinées.

- Règle de raffinement des structures de données *Nom_D* : cette règle permet de raffiner l'environnement abstrait *env_abs*, vérifiant la contrainte *C*, par l'environnement concret *env_conc*. Un environnement est défini par les ensembles et les variables abstraits qu'il contient. Les deux environnements sont reliés par l'invariant de collage *J(env_abs, env_conc)*. Cette règle est modélisée comme suit :

RULE *Nom_D* **IS**
ABSTRACT_ENV
env_abs
CONSTRAINT
C
CONCRETE_ENV
env_conc
INVARIANT
$J(env_abs, env_conc)$
END

- Règle de raffinement des substitutions *Nom_S*: cette règle permet le raffinement de la substitution abstraite *sub_abs*, agissant sur l'environnement abstrait *env_abs*, par la substitution concrète *sub_conc* qui agit sur l'environnement concret *env_conc*. Le lien entre la règle de raffinement des données et celle des substitutions est réalisé par l'invariant $J(env_abs, env_conc)$ figurant dans les deux règles. Comme pratiquement la substitution *sub_abs* est contenue dans une opération pouvant contenir d'autres substitutions, des contraintes supplémentaires C' sur ces substitutions peuvent être précisées.

RULE *Nom_S* **IS**
REFINES
sub_abs
CONSTRAINTS
$J(env_abs, env_conc) \wedge C'$
REFINEMENT
sub_conc
END

À chaque règle de raffinement de structure de données correspond une ou plusieurs règles de raffinement des substitutions. En effet, pour chaque type ou forme de substitution, opérant sur les données raffinées par la règle *Nom_D*, correspond une règle de raffinement permettant la réécriture de cette substitution dans l'environnement concret.

Pour chaque règle de raffinement de substitutions *Nom_S* associée à la règle de raffinement de structure de données *Nom_D*, les obligations de preuve définies dans la section 5.1 sont générées. Elles vérifient la correction de la substitution concrète *sub_conc* par rapport à la substitution abstraite *sub_abs* relativement à l'invariant de collage $J(env_abs, env_conc)$.

Je ne vais pas détailler toutes les règles, mais présenter les plus significatives en m'appuyant sur l'exemple du club vidéo.

5.3.1 Identification des objets

Dans le modèle relationnel, l'identification des différents objets se fait sur la base des valeurs d'un ou plusieurs attributs qui constituent la clé. Donc, une étape du processus de raffinement va consister à introduire cette notion de clé dans les classes qui n'en possèdent pas.

La classe *Boutique* du diagramme de classes de la page 17 n'a pas de clé : une nouvelle variable $Num_Boutique$ est rajoutée avec l'invariant

$$Num_Boutique \in Boutique \rightarrowtail TT_{Boutique}$$

$TT_{Boutique}$ est le type de la variable; c'est un ensemble abstrait dont la valeur sera donnée lors de la phase d'implémentation.

L'opération de base qui ajoute une nouvelle boutique :

B_Add_Boutique(bo,nom,ad)$\triangleq$
PRE
$$\begin{array}{rcl} bo & \in & BOUTIQUE - Boutique \ \wedge \\ nom & \in & STRING \ \wedge \\ ad & \in & STRING \end{array}$$
THEN
$$\begin{array}{rcl} Boutique & := & Boutique \cup \{bo\} \ || \\ Nom_Boutique & := & Nom_Boutiqua \cup \{bo \mapsto nom\} \ || \\ Adresse & := & Adresse \cup \{bo \mapsto ad\} \end{array}$$
END

est réécrite pour prendre en compte cette nouvelle variable et respecter l'invariant de collage :

B_Add_Boutique(bo,nom,ad)$\triangleq$
BEGIN
$Boutique := Boutique \cup \{bo\}$;
VAR nu IN
$$\begin{array}{rcl} nu & \longleftarrow & generer(TT_{Boutique} - ran(Num_Boutique)); \\ Num_Boutique & := & Num_Boutique \cup \{bo \mapsto nu\} \end{array}$$
END;
$$\begin{array}{rcl} Nom_Boutique & := & Nom_Boutique \cup \{bo \mapsto nom\}; \\ Adresse & := & Adresse \cup \{bo \mapsto ad\} \end{array}$$
END

L'opération *generer* est définie dans une machine abstraite paramétrée $GENERE(TT)$. Elle prend en entrée un sous-ensemble tt de TT et retourne une valeur aléatoire de tt. Dans l'exemple ci-dessus, tt est l'ensemble des valeurs de la clé $Num_Boutique$ non encore attribuées.

5.3.2 Elimination des associations

Une association est définie comme une relation entre des ensembles d'objets existants, c'est-à-dire entre des identités d'objet. L'objectif de cette phase de raffinement est de transformer cette relation pour qu'elle soit définie entre les valeurs des clés des classes associées. Le traitement est différent selon la multiplicité de l'association et le fait qu'elle possède ou non des attributs.

Associations monovaluées

Une association monovaluée est une association ayant au moins un rôle dont la borne supérieure de sa multiplicité est 1. Si l'association n'a pas d'attributs, celle-ci est éliminée en ajoutant un attribut supplémentaire à la classe associée à ce rôle monovalué et qui référence la clé de l'autre classe.

L'association *Appartient* est monovaluée pour *Cassette*. On raffine la variable correspondante (*Appartient*) par une nouvelle variable *Appartient_Mig* telle que :

$$Appartient_Mig \in Cassette \rightarrow TT_{Boutique} \quad \wedge \quad Appartient_Mig = Appartient; Num_Boutique$$

Le premier invariant type la variable *Appartient_Mig* alors que le second est l'invariant de collage qui établit la relation existant entre cette nouvelle variable et les variables abstraites.

Cette règle de raffinement remplace une variable de la spécification abstraite initiale par une nouvelle variable. Le raffinement des substitutions dans ce cas consiste à remplacer chaque substitution liée à la variable éliminée par une substitution sur la nouvelle variable qui satisfait l'invariant de collage. Intuitivement, ce raffinement consiste à remplacer chaque instance de *Boutique* par la valeur de sa clé. Par exemple, l'ajout d'un lien dans l'association *Appartient* spécifié par la substitution

$$Appartient := Appartient \cup \{ca \mapsto bb\}$$

est raffiné par

$$Appartient_Mig := Appartient_Mig \cup \{ca \mapsto Num_Boutique(bb)\}$$

Associations multivaluées

Une association, dont les deux rôles sont multivalués ou ayant des attributs, est transformée en une classe ayant, en plus des attributs de l'association, deux attributs supplémentaires (un pour chaque classe associée) liés par des contraintes référentielles aux clés des classes concernées par l'association. Ces deux attributs constituent une clé primaire de la classe ainsi créée.

L'association *Pret* entre *Client* et *Cassette* est multivaluée et possède des attributs. On définit deux nouvelles variables $Pret_{client}$ et $Pret_{cassette}$ telles que :

$$\begin{array}{rcl} Pret_{client} & = & Pret \lhd (prj_1(Client,Cassette); Num_Cl) \quad \wedge \\ Pret_{cassette} & = & Pret \lhd (prj_2(Client,Cassette); Num_Ca) \quad \wedge \\ (Pret_{client} \otimes Pret_{cassette}) & \in & Pret \rightarrowtail NAT1 \times NAT1 \end{array}$$

Les deux premiers invariants sont les invariants de collage : ils expriment chacun comment la nouvelle variable est définie. Par exemple le premier indique que $Pret_{client}$ associe à chaque couple (client,cassette) qui existe dans l'association *Pret* le numéro du client correspondant. Le dernier invariant spécifie que ($Pret_{client}$, $Pret_{cassette}$) est clé de la classe association *Pret*.

Cette règle de raffinement introduit deux nouvelles variables. Afin de préserver l'invariant (de collage) du composant raffinement, il est nécessaire de mettre à jour ces nouvelles variables à chaque mise à jour de la variable correspondant à l'association. Ainsi l'ajout d'un prêt spécifié par :

$$\begin{array}{rcl} Pret & := & Pret \cup \{cl \mapsto ca\}; \\ Date_{Pret} & := & Date_{Pret} \cup \{cl \mapsto ca \mapsto da\}; \end{array}$$

est raffiné par

$$\begin{array}{rcl} Pret & := & Pret \cup \{cl \mapsto ca\}; \\ Pret_{client} & := & Pret_{client} \cup \{cl \mapsto ca \mapsto Num_Cl(cl)\}; \\ Pret_{cassette} & := & Pret_{cassette} \cup \{cl \mapsto ca \mapsto Num_Ca(ca)\}; \\ Date_{Pret} & := & Date_{Pret} \cup \{cl \mapsto ca \mapsto da\}; \end{array}$$

5.3.3 Passage d'un modèle fonctionnel à un modèle ensembliste

Le but de cette étape est la définition des schémas des différentes relations. L'idée intuitive est de rassembler sous une variable les caractéristiques concernant une même classe. Cette variable sera par la suite traduite en une table SQL.

Par exemple, les variables Num_Ca, $Nbdispo$, $Titre$, $Genre$ et $Appartient_Mig$, qui sont des fonctions définies sur la variable $Cassette$, sont regroupées en une seule variable $T_Cassette$ définie comme suit:

$$\begin{array}{rcl} T_Cassette & \in & Cassette \rightarrow NAT1 \times NAT \times STRING \times GENRE \times TT_{Boutique} \wedge \\ T_Cassette & = & Num_Ca \otimes Nbdispo \otimes Titre \otimes Genre \otimes Appartient_Mig \end{array}$$

Le premier invariant est l'invariant de typage et le second l'invariant de collage.

Le raffinement des opérations consiste à regrouper l'ensemble des substitutions relatives à la mise à jour des fonctions en une seule substitution de mise à jour de la nouvelle variable. Par exemple l'opération $B_Add_Cassette(ca,nu,ti,nb,ge,bo)$ décrite à la page 23, une première fois raffinée par la règle décrite au paragraphe 5.3.2, se raffine finalement par :

$\mathsf{B_Add_Cassette}(ca,nu,ti,nb,ge,bo) \triangleq$
BEGIN
$$\begin{array}{rcl} Cassette & := & Cassette \cup \{ca\} \,||\\ T_Cassette & := & T_Cassette \cup \{ca \mapsto (nu \mapsto nb \mapsto ti \mapsto ge \mapsto Num_Boutique(bo))\} \end{array}$$
END

5.3.4 Implémentation des données

A l'issue de la phase précédente, les structures de données sont définies comme des fonctions d'un ensemble abstrait vers le produit cartésien des types des attributs. Il s'agit maintenant de définir concrètement comment ces structures sont définies dans le langage d'implémentation cible choisi. Cela consiste à faire le lien avec les machines abstraites représentant les relations, comme celle présentée à la page 58 et que j'appelle machines SQL dans la suite.

Valuation des ensembles abstraits

Avant cela, il faut donner une valeur aux ensembles abstraits. Rappelons qu'on peut trouver deux types d'ensembles abstraits : ceux associés aux classes et qui représentent les ensembles d'instances possibles et ceux définis comme types des attributs. Ces derniers sont identifiés avec des ensembles abstraits définis dans les machines SQL. Chaque ensemble abstrait associé à une classe C est identifié au type de la clé primaire de la classe. C'est la réalisation effective du passage d'un modèle par objets, où chaque objet est identifié de manière implicite, à un modèle par valeurs où chaque objet est identifié explicitement par la valeur de sa clé. Ainsi pour la classe $Cassette$, nous avons l'ensemble abstrait $CASSETTE$, la variable $Cassette$, telle que $Cassette \subseteq CASSETTE$, et l'attribut Num_Ca qui est clé, $Num_Ca \in Cassette \rightarrowtail NAT1$. La valuation de $CASSETTE$ est alors $CASSETTE = NAT1$, d'où $Cassette \subseteq NAT1$, ce qui signifie que chaque objet cassette est identifié à son numéro.

Implémentation des variables

Les machines SQL sont importées dans le composant $IMPLEMENTATION$ et le lien entre la variable définie dans chacune d'elles et les variables définies à l'étape précédente est exprimé dans l'invariant de collage du composant $IMPLEMENTATION$. Ainsi la variable $T_Cassette$ définie à la page 64 dans la section 5.3.3 est implémentée par la variable $Table_Cassette$ de la machine *Rel_cassette* de la page 58, avec l'invariant de collage :

$$id(Cassette) = (T_Cassette; prj_1(NAT1,NAT1); prj_1(NAT1,NAT1)) \quad \wedge$$

$$\forall(x_1,x_2,x_3,x_4,x_5).(x_1 \mapsto (x_1 \mapsto x_2 \mapsto x_3 \mapsto x_4 \mapsto x_5) \in T_Cassette$$
$$\Rightarrow rec(x_1,x_2,x_3,x_4,x_5) \in Table_Cassette) \quad \wedge$$

$$\forall(x_1,x_2,x_3,x_4,x_5).rec(x_1,x_2,x_3,x_4,x_5) \in Table_Cassette$$
$$\Rightarrow (x_1 \mapsto (x_1 \mapsto x_2 \mapsto x_3 \mapsto x_4 \mapsto x_5) \in T_Cassette)$$

Le premier prédicat, équivalent à $Num_Ca = id(Cassette)$, exprime l'identification de chaque objet par la valeur de sa clé. Les deux derniers prédicats établissent la correspondance bijective entre les éléments de $T_Cassette$ et de $Table_Cassette$.

Pratiquement, les machines SQL sont générées automatiquement à partir des variables définies à l'étape précédente.

Les substitutions de base qui apparaissent dans les opérations du composant $IMPLEMENTATION$ sont implémentées par des appels aux opérations des machines SQL. Un ensemble de règles a été défini pour chaque type de substitution. Ainsi l'opération $B_Add_Cassette$ est implémentée de la manière suivante :

$\mathsf{B_Add_Cassette}(ca,nu,ti,nb,ge,bo) \triangleq$
BEGIN
$Insert_Cassette(nu,ti,nb,ge,bo)$
END

5.3.5 Commentaires

Comme le processus de raffinement défini dans cette section est automatique, il faut s'assurer de certaines propriétés. Nous avons ainsi vérifié que l'algorithme termine, qu'il est déterministe et qu'il est valide, c'est-à-dire qu'il donne les mêmes résultats que l'algorithme de [BP98] pour le raffinement des données. Ces vérifications n'ont pas été faites formellement, mais elles s'appuient sur les caractéristiques des règles et sont relativement intuitives. Je reparlerai de la correction du processus de raffinement dans la section 5.5.

J'ai choisi de présenter les règles de raffinement qui sont les plus représentatives et finalement on se rend compte que leur spécification en B est assez naturelle. Cependant, il y a deux types de règles qui techniquement ont posé plus de difficultés : l'élimination des attributs de type *Séquence* et des attributs multivalués. La raison principale est que plusieurs solutions de raffinement étaient possibles et que certains choix ont impliqué des preuves très lourdes, voire conduisaient à des solutions qu'on ne pouvait pas raffiner dans la suite.

Pour rendre le processus déterministe, nous avons aussi été obligé de découper certaines étapes de l'algorithme classique.

5.4 Raffinement des transactions

Rappelons que les transactions n'ont pas été raffinées durant les phases précédentes. Elles ont juste été réécrites pour tenir compte des changements de variables effectués. Il faut maintenant supprimer le non-déterminisme qu'elles peuvent avoir. Ceci bien sûr ne peut être fait que par le spécifieur qui peut prendre en compte différents critères liés à l'implémentation comme par exemple le temps d'exécution ou la place mémoire ... Cette phase est donc essentiellement manuelle. Cependant, si les substitutions des opérations sont déterministes, on peut générer automatiquement les substitutions finales.

Il faut également réécrire les prédicats qui apparaissent dans les conditions des opérations. En effet, les variables de l'implémentation sont définies dans les machines SQL importées et on ne peut y accéder que par des opérations de lecture. Nous n'avons pas cherché à automatiser ce mécanisme, par manque de temps. Nous avons cependant donner les lignes directrices du processus. Plus de détails sur cette partie peuvent être trouvés dans la thèse.

5.5 A propos des preuves : Correction du processus de raffinement

La correction d'un développement par raffinement repose sur la preuve de correction de chaque étape de raffinement qui permet, de proche en proche, d'assurer la correction du code généré à la dernière étape.

Les règles de raffinement, présentées dans ce chapitre, ont été prouvées en utilisant le prouveur de l'AtelierB (version 3.5). Environ 75% des preuves l'ont été automatiquement sans rajout de règles dans le prouveur, l'échec du prouveur automatique à établir certaines preuves est dû essentiellement au manque de règles et de tactiques dans sa base.

Je l'ai dit dans l'introduction du chapitre, il est nécessaire non seulement d'automatiser le plus possible le processus de raffinement mais aussi de fournir une aide pour l'établissement des preuves. Ce qui veut dire : en pratique, pour une application donnée, comment le spécifieur va-t'il pouvoir réutiliser les preuves des règles de raffinement pour établir les preuves de raffinement de ses transactions? Un premier point est que la preuve de chaque règle ne dépend que de la règle elle-même, elle est indépendante des spécificités de l'application étudiée. Ce caractère générique des preuves permet donc leur réutilisation par instanciation. Notre objectif est alors de déterminer de quelle manière ces briques de preuve de base peuvent être réutilisées par le spécifieur pour prouver chaque étape du raffinement.

Autrement dit, soient S et T deux substitutions telles qu'il existe une preuve formelle Pr du raffinement de S par T ayant pour invariant de collage le prédicat $J : S \sqsubseteq_J T$. Le but est de définir comment cette preuve Pr peut être réutilisée pour prouver le raffinement d'une substitution plus complexe construite à partir de la substitution S. Cela n'est pas possible dans tous les cas. Nous avons défini des conditions de réutilisation, valables pour n'importe quelle spécification B, puis nous avons montré que, dans le cas des applications bases de données, ces conditions étaient pratiquement toujours vérifiées.

5.5.1 Conditions de réutilisation

En B, les substitutions étant construites de manière inductive, nous procédons donc par induction sur la forme de la substitution à raffiner. Cela se fait en 2 temps : d'abord on introduit un seul opérateur de B, puis on généralise à un nombre quelconque d'opérateurs.

Soient S_1,S_2 deux substitutions dont on connaît le raffinement et la preuve associée. Six cas de substitutions construites à partir d'un seul opérateur de B nous intéressent :

- Substitution Préconditionnée : $P|S_1$.
- Substitution Parallèle $S_1 \| S_2$: deux cas sont à distinguer :
  - le raffinement de $S_1 \| S_2$ par une substitution parallèle.
  - le raffinement de $S_1 \| S_2$ par une substitution séquentielle.
- Substitution Choix Borné : $S_1 [] S_2$.
- Substitution Conditionnée : **IF** P **THEN** S_1 **ELSE** S_2 **END**.
- Substitution séquence : $S_1 ; S_2$.
- Raffinement d'une substitution **ANY**.

Je ne vais pas détailler les différentes démonstrations qui nous ont permis de définir les conditions de réutilisation. Je donne le principe pour la substitution préconditionnée et présente de manière intuitive les différentes conditions.

Principe

Dans ce qui suit, chaque substitution ou prédicat est indexé par l'ensemble de ses variables libres.

Soient S_{a_1} et T_{a_2} deux substitutions, $J(a_1,a_2)$ un prédicat tel que :

$$S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2} \qquad (\mathrm{R}_1)$$

Soient P_{b_1}, Q_{b_2} et $J(a_1 \cup b_1,a_2 \cup b_2)$ trois prédicats tel que :

$$(P_{b_1}|S_{a_1}) \sqsubseteq_{J(a_1 \cup b_1,a_2 \cup b_2)} (Q_{b_2}|T_{a_2}) \qquad (\mathrm{R}_2)$$

On voudrait réutiliser la preuve du raffinement R_1 lors de la preuve du raffinement R_2.

Nous avons montré que les conditions nécessaires et suffisantes pour que cette réutilisation soit possible sont les suivantes : ($c_i = a_i \cup b_i$ et $d_i = b_i - a_i$)

i. $(P_{b_1} \wedge J(c_1,c_2)) \Rightarrow Q_{b_2}$

ii. $(P_{b_1} \wedge Trm(S_{a_1}) \wedge J(c_1,c_2)) \Rightarrow J(a_1,a_2)$

iii. Soit a'_2 une valeur de a_2. Toute valeur a'_1 de a_1 vérifiant (R_1) et

$$(P_{b_1} \wedge Trm(S_{a_1}) \wedge J(c_1,c_2) \wedge Prd(S_{a_1}) \wedge J(a'_1,a'_2) \wedge Prd(T_{a_2})) \Rightarrow J(a'_1 \cup d_1,a'_2 \cup d_2)$$

peut être réutilisée pour prouver (R_2).

Analyse des conditions

1. La condition (*i*) est indépendante de la réutilisation, c'est une condition nécessaire pour établir la correction du raffinement d'une substitution préconditionnée.
2. La condition (*ii*) est la première condition nécessaire de réutilisation. En effet, si l'objectif est de réutiliser la preuve de raffinement $S_{a_1} \sqsubseteq T_{a_2}$, alors il faut être capable d'exhiber dans l'environnement de preuve de $(P_{b_1}|S_{a_1} \sqsubseteq Q_{b_2}|T_{a_2})$ l'invariant de collage de $S_{a_1} \sqsubseteq T_{a_2}$.
3. La condition (*iii*) est la seconde condition nécessaire de réutilisation. Elle exprime que pour pouvoir réutiliser la preuve, il faut qu'au moins une des solutions de raffinement de $S_{a_1} \sqsubseteq T_{a_2}$ soit aussi solution du raffinement de $(P_{b_1}|S_{a_1} \sqsubseteq Q_{b_2}|T_{a_2})$, c'est-à-dire qu'elle vérifie l'invariant de collage du raffinement à prouver.
4. La réutilisation de la preuve de raffinement de $S_{a_1} \sqsubseteq T_{a_2}$ est réalisable pour toute valeur a_1' qui vérifie la condition (*iii*) : on n'exige pas que toutes les valeurs a_1' vérifient cette condition.
5. Si la substitution S_{a_1} est déterministe alors la condition (*iii*) est vérifiée. En effet, il existe une seule valeur a_1' qui satisfait le raffinement $(S_{a_1} \sqsubseteq_{J(a_1,a_2)} T_{a_2})$, celle-ci est déterminée par $Prd(S_{a_1})$. De plus, une solution possible du raffinement $(P_{b_1}|S_{a_1} \sqsubseteq_{J(c_1,c_2)} Q_{b_2}|T_{a_2})$ doit satisfaire le $Prd(S_{a_1})$ (à moins que le raffinement soit une tautologie) qui lui-même n'admet qu'une seule solution : les deux raffinements admettent donc le même espace de solutions.
6. Si le prédicat $J(a_1,a_2)$[2] est fonctionnel en a_1 alors la condition (*iii*) est vérifiée. En effet, une solution possible du raffinement $(P_{b_1}|S_{a_1} \sqsubseteq_{J(c_1,c_2)} Q_{b_2}|T_{a_2})$ doit satisfaire le prédicat $J(c_1,c_2)$. D'après (*ii*) cette solution satisfait le prédicat $J(a_1,a_2)$. Or $J(a_1,a_2)$ n'admet qu'une valeur unique de a_1 pour toute valeur de a_2 : cette solution unique est donc solution de $(P_{b_1}|S_{a_1} \sqsubseteq_{J(c_1,c_2)} Q_{b_2}|T_{a_2})$.

Après notre étude, il s'avère que les conditions de réutilisation sont toujours celles-ci, quelque soient les substitutions. Dans le cas du raffinement d'une substitution parallèle par une autre substitution parallèle, on a une condition supplémentaire : les ensembles des variables libres et modifiées des deux substitutions parallèles doivent être disjoints.

Nous avons ensuite proposé une méthode qui généralise aux cas de substitutions complexes, c'est-à-dire construites à partir de plusieurs opérateurs de B.

5.5.2 Application au domaine des bases de données

Il suffit de montrer que les substitutions et les invariants de collage de la base de règles que nous avons définie vérifient bien les conditions requises pour la réutilisation. En effet, le processus de raffinement défini étant fermé, seules les règles existantes dans la base peuvent être utilisées. Il reste la condition liée au raffinement d'une substitution parallèle par une autre substitution parallèle qui, elle, dépend entièrement de la façon dont l'utilisateur spécifie sa substitution abstraite. En particulier, cette condition est non satisfaite pour une transaction qui simultanément modifie la valeur d'un attribut et utilise sa valeur "avant" pour la mise à jour d'un autre attribut. Dans ce cas, la preuve est nécessairement établie sans réutilisation. Néanmoins en pratique, ce type de transactions est peu fréquent.

La condition (*ii*) relative à l'exhibition de l'invariant de collage de la règle de raffinement est vérifiée car le raffinement étant automatique, le processus génère automatiquement l'invariant de collage de la règle appliquée. En d'autres termes, l'invariant de collage de chaque étape est la conjonction des invariants de collage des différentes instanciations de la règle appliquée.

2. $J(a_1,a_2)$ est fonctionnel en a_1 si pour toute valeur de a_2 il existe une seule valeur de a_1 qui vérifie ce prédicat.

La condition (*iii*) est aussi vérifiée car tous les invariants de collage de nos règles de raffinement sont fonctionnels. Or nous avons montré que dans chaque cas de constructeur B, cette caractéristique est suffisante pour satisfaire la condition.

5.6 Conclusion

La thèse a rempli son objectif principal qui était de proposer un processus formel de raffinement produisant une spécification concrète traduisible en une implémentation relationnelle (schéma SQL de la base de données et code des transactions). Nous avons utilisé le processus de raffinement de la méthode B. Le processus étant formel, les transactions générées sont correctes par rapport à la spécification (modulo l'établissement de la preuve de correction du traducteur vers le langage d'implémentation cible choisi). En particulier, comme les contraintes d'intégrité ont été définies dans la spécification, on est sûr que les transactions les vérifient. En conséquence, il est possible de ne pas déclarer ces contraintes dans le SGBD, on économise ainsi leur temps de vérification, qui n'est en général pas négligeable.

Plus nous avancions dans la thèse, plus nous nous sommes rendues compte qu'il était vraiment nécessaire de développer un outil qui automatise le processus. En effet, une des caractéristiques des SI est le nombre très important de variables de la spécification initiale pour une application donnée, ce qui rend le processus de raffinement très lourd, en particulier la ré-écriture à chaque étape des substitutions des opérations de base et la réalisation des preuves associées. Un prototype a été construit en OCAML pour montrer la faisabilité d'un tel outil : il est possible d'automatiser tout le processus de raffinement des données et des opérations de base, des choix par défaut pouvant être réalisés. Il reste au spécifieur à raffiner les transactions. Cependant, partant du constat que, pour la plupart des transactions, l'algorithmique est peu complexe et avec peu d'indéterminisme, il devrait être possible de traiter de manière automatique ces transactions. Dans les autres cas, on pourrait envisager de définir des raffinements par défaut ou au moins des patrons de raffinement.

En ce qui concerne les preuves, la thèse a donné des résultats intéressants : les preuves de base peuvent être réutilisées par instanciation et des conditions de réutilisation des preuves ont été définies. Il s'avère que dans le domaine des SI, ces conditions sont la plupart du temps vérifiées. Il reste maintenant à continuer le développement de l'outil pour prendre en compte cet aspect.

Je rappelle que les travaux décrits dans la thèse s'arrête au niveau implémentation de B. Un certain nombre de règles de traduction vers un schéma de base de données SQL ont été données mais c'est incomplet. Nous avons montré sur une étude de cas qu'il était possible de traduire (à la main) le niveau de B en une application écrite en JAVA + SQL. Il serait intéressant d'aller plus loin et de pouvoir construire un traducteur automatique.

Chapitre 6

Combinaison de spécifications pour la modélisation du comportement des SI

La méthode définie dans les chapitres précédents permet de spécifier l'effet des transactions sur l'état du SI et de s'assurer de leur correction. Cependant il est difficile d'exprimer certaines propriétés comportementales comme par exemple l'enchaînement des transactions. D'autre part, comme je l'ai mentionné dans la conclusion du chapitre 4, le niveau d'abstraction de la spécification obtenue n'est pas aussi élevé qu'on pourrait le souhaiter. En effet, y apparaissent des éléments dont on n'a pas besoin dans un premier temps pour spécifier un SI en vue d'une validation avec les utilisateurs: description des attributs des classes et associations, de l'effet des transactions par combinaison des opérations de base, ... Marc Frappier, professeur à l'université de Sherbrooke au Québec, a développé, avec Richard Saint-Denis, la méthode EB^3 [FSD99], une adaptation de CSP et LOTOS dédiée à la spécification des SI. Le comportement d'un SI est décrit en termes de traces d'événements admissibles, ce qui facilite l'expression et la vérification de contraintes d'ordonnancement sur les événements. La spécification obtenue peut être une réponse aux critiques énoncées ci-dessus. Les deux types de spécifications me semblent donc complémentaires et l'objectif de notre collaboration est d'étudier leur combinaison. C'est un travail en cours, je vais présenter les résultats que nous avons obtenus jusqu'à présent et les axes de recherche que nous comptons développer.

Je commencerai d'abord par présenter rapidement le langage EB^3, puis je discuterai de différentes techniques existantes pour spécifier le comportement d'un SI. Enfin j'exposerai le processus qui permet de vérifier la cohérence entre une spécification EB^3 et une spécification B obtenue par ma méthode.

6.1 Aperçu de la méthode EB^3

La méthode EB^3 et ses fondements théoriques sont décrits dans le document [FSD99]. Je vais résumer les caractéristiques essentielles en les illustrant à travers l'exemple du club vidéo. Une spécification EB^3 est composée de quatre parties:

1. le diagramme de classes qui définit les classes, les associations et les événements externes associés.
2. l'espace d'entrées-sorties qui définit la signature des événements externes.
3. l'expression de processus qui définit l'ensemble des traces valides du SI.
4. les règles qui définissent les sorties en fonction des traces valides.

Le premier composant utilise les concepts des modèles sémantiques de données : classes et associations. Pour chaque élément du diagramme sont rajoutés les événements externes du SI qui le concernent. Le diagramme de la figure 6.1 reprend le diagramme de classes de la page 17. Pour des raisons pédagogiques, j'ai modifié la multiplicité de l'association *Pret* : elle devient 0..1 pour une cassette.

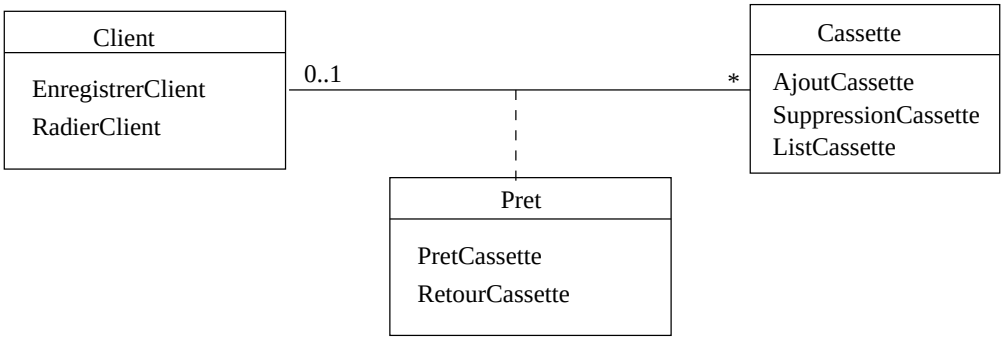

FIG. 6.1 – *Diagramme de classes d'une spécification* EB[3]

Les événements externes sont ceux qui déclenchent les transactions [1] dans le SI. Ils sont spécifiés par leur signature. Par exemple, voici celle des événements EnregistrerClient et ListCassette :

$M_1 \triangleq$ EnregistrerClient(cl : $CLIENT$,n : $STRING$) : ($result$: $STRING$)

$M_2 \triangleq$ ListCassette(ca : $CASSETTE$) : ($result$: $STRING$,$titre$: $STRING$,$NbPret$: NAT)

Le paramètre *result* rend compte à chaque fois du résultat de l'exécution de l'événement.

Les contraintes d'ordonnancement des entrées définissent les traces valides du SI. Celles-ci sont décrites par une expression de processus. Le langage utilisé est un sous-ensemble de CSP et LOTOS, adapté à la spécification des SI. Globalement, il correspond aux expressions régulières avec la composition parallèle, la synchronisation et l'appel de processus.

Les traces valides du SI sont une combinaison des traces valides de chaque classe. On commence par spécifier l'expression de processus pour un objet de chaque classe. Par exemple l'expression de processus d'un objet de la classe *Cassette* est définie par :

Entity Type **Cassette**(ca : $CASSETTE$) $\triangleq$
AjoutCassette(ca,_) .
(
emprunt(_,ca)*
|||
ListCassette(ca)*
) .
SuppressionCassette(ca)

Cette expression signifie que le premier événement d'entrée pour une cassette est AjoutCassette. Il est suivi (opérateur de concaténation .) par des événements qui résultent de l'entrelacement de deux expressions de processus. L'opérateur d'entrelacement (|||) d'expressions de processus représente les différentes possibilités de combinaison de ces expressions. La première expression de l'entrelacement du processus *Cassette* signifie qu'une cassette peut être empruntée 0 ou plusieurs fois (opérateur *);

1. plus précisément les transactions bases de données sûres telles que définies dans le chapitre 4

la seconde exprime que l'événement d'entrée ListCassette peut apparaître également 0 ou plusieurs fois. Enfin, le dernier événement possible pour une cassette est SuppressionCassette. Le processus *emprunt* est un processus auxiliaire utilisé dans d'autres expressions de processus et défini par :

Auxiliary process **emprunt**(cl : *CLIENT*,ca : *CASSETTE*) $\triangleq$
PretCassette(cl,ca) **.**
RetourCassette(ca)

L'expression de processus de la classe *Client* est définie par :

Entity Type **Client**(cl : *CLIENT*) $\triangleq$
EnregistrerClient(cl,_)**.**
($|||\, x : x \in CASSETTE$: **emprunt**(cl,x)*)**.**
RadierClient(cl)

La sous-expression définie par l'opérateur d'entrelacement signifie qu'un client peut emprunter plusieurs cassettes en même temps.

Une trace valide d'une expression de processus est n'importe quelle séquence d'événements que le processus peut exécuter. La trace suivante est un exemple de trace valide du client cl_1 :

EnregistrerClient(cl_1,n_1) **.** PretCassette(cl_1,ca_1) **.** PretCassette(cl_1,ca_2) **.** RetourCassette(ca_1)

L'expression de processus décrivant les traces valides pour l'ensemble des objets d'une classe est une généralisation de celle décrivant les traces possibles pour *un* objet de la classe:

$$|||\, x : x \in CLASSE_i : \mathbf{Classe}_i(x)$$

L'ensemble des traces valides pour un SI est spécifié en utilisant l'opérateur de synchronisation (||). Celui-ci signifie que plusieurs expressions de processus synchronisées exécutent simultanément les événements qu'elles partagent.

$$(|||\, x : x \in CLASSE_i : \mathbf{Classe}_i(x)) \quad || \quad (|||\, x : x \in CLASSE_j : \mathbf{Classe}_j(x)) \quad || \quad (\cdots)$$

Inversement, j'aurai besoin par la suite de décomposer une trace valide du système en un ensemble de traces valides associées aux classes du système. Ainsi, si s est une trace valide du système et $Cl1$,...,Cln sont les classes du diagramme de classes, le prédicat $class_traces(s,T_{Cl1},...,T_{Cln})$ est vrai si et seulement si les traces T_{Cl1},...,T_{Cln} associées aux classes $Cl1$,...,Cln forment une décomposition correcte de s.

Les sorties sont exprimées à partir des traces valides du SI en utilisant des opérateurs sur les traces.

- L'expression $t\lceil\lceil\{event_label\}$ est la restriction de la trace t aux événements dont le nom est spécifié dans la liste $event_label$.
- La longueur d'une trace t est définie par l'expression $\#t$.
- $param_i(e)$ désigne la valeur du paramètre $param_i$ de l'événement e.
- $label(e)$ désigne le nom de l'événement e.

Par exemple, s étant une trace valide du système, si le prédicat $class_traces(s,T_{Cassette},T_{Client})$ est vrai, alors la sortie $NbPret$ associé à l'événement d'entrée ListCassette est spécifiée en utilisant la fonction :

$$NbPret(ca) \quad = \quad \#\,(T_{Cassette}(ca)\lceil\lceil\{\text{PretCassette}\})$$

Un point que j'ai trouvé intéressant et qui rejoint mes préoccupations : une représentation graphique UML des notations EB^3 a été définie dans [Ric01]. Ce travail va même plus loin car il étudie également l'intégration de la méthode dans un processus de développement de logiciels orienté objets.

6.2 Sur la spécification du comportement dans les SI

Le paradigme le plus couramment utilisé pour spécifier le comportement des SI est celui des transitions d'états. Une spécification basée sur ce paradigme consiste en un espace d'état, défini par des variables d'état, le plus souvent décrites par les concepts des modèles sémantiques de données, et des opérations qui définissent les transitions d'états, c'est-à-dire les modifications des variables d'état. A la réception (arrivée) d'un événement externe au SI, une opération est appelée pour calculer le nouvel état du SI et produire une sortie si nécessaire. Il existe plusieurs langages pour décrire ce type de spécifications, comme par exemple les diagrammes états-transitions ou les machines à états, ou les langages de l'approche ensembliste (Z, VDM ou B) ou encore UML/OCL. La méthode que j'ai décrite dans les précédents chapitres de ce document relève de ce paradigme.

Une des difficultés avec ce type de spécifications est la vérification de propriétés comme les contraintes d'ordonnancement des événements. Pour vérifier qu'une séquence donnée d'événements d'entrée $e_1, \cdots, e_n$ est possible, il faut vérifier que l'exécution des opérations $op_{e_1}, \cdots, op_{e_n}$ associées aux événements est possible, ce qui signifie qu'il faut vérifier que l'exécution de op_{e_i} établit la précondition de $op_{e_{i+1}}$. Il est évident qu'une vérification manuelle de ces propriétés est fastidieuse et sujette à erreurs. De plus, il faudrait disposer d'un moyen systématique d'énumérer les séquences correctes.

Les algèbres de processus comme CCS ou CSP permettent facilement de spécifier les contraintes d'ordonnancement des événements d'entrée mais sont moins bien adaptées pour prendre en compte certaines spécificités des SI. En particulier, il est difficile de représenter la complexité de la structure des SI (nombreuses entités reliées par des associations) et d'exprimer les sorties correspondant aux événements d'entrée. C'est pour palier ces faiblesses que la méthode EB^3 a été définie.

Mais il reste quand même que les spécifications EB^3 sont très abstraites et que de passer à une implémentation classique bases de données n'est pas immédiat. Ce qui n'est pas le cas pour les spécifications obtenues par transition d'états comme je l'ai montré au chapitre 5. D'où l'idée qui nous est venue de combiner ces deux types de spécifications.

L'utilisation conjointe de plusieurs paradigmes est un axe de recherche très actif en ce moment. Deux approches sont possibles. L'approche homogène consiste à prendre une méthode d'un paradigme et à l'étendre pour pouvoir exprimer différents types de propriétés. C'est le cas de RAISE [GHH$^+$95]. Le principal reproche est le manque de lisibilité de la spécification obtenue. L'approche hétérogène combine des méthodes de différents paradigmes. Comme exemples on peut citer LOTOS [vEVD89] (algèbre de processus et spécifications algébriques), ZCCS [GS97] (algèbre de processus et Z), csp2B [But00] (algèbre de processus et B) ou KORRIGAN [Poi00] (spécifications algébriques, systèmes de transitions et logique temporelle). Il est alors nécessaire de pouvoir lier les différentes spécifications en définissant des règles de cohérence.

Nous avons choisi de combiner EB^3 avec la méthode B. D'après les remarques du paragraphe précédent, il apparaît assez naturel de considérer qu'une spécification transition d'états est un raffinement d'une spécification algèbre de processus, raffinement avec le sens de "plus proche d'une implémentation". Pour rester dans le même cadre de travail tout au long du processus de développement du SI, nous utilisons le raffinement de la méthode B. Pour cela la spécification EB^3 est d'abord traduite en une spécification B équivalente.

C'est bien sûr un début de travail, nous avons expérimenté notre idée sur une étude de cas pour en étudier la faisabilité. Mais il apparaît que c'est une piste prometteuse que nous comptons approfondir. Je vais dresser les grandes lignes du processus de raffinement qui permet de vérifier la cohérence des deux spécifications dans la section suivante. L'article [FL01] est plus détaillé et présente en particulier les preuves de raffinement.

6.3 Vérification de la cohérence des spécifications B et EB^3 par raffinement

Pour fixer le vocabulaire, j'utilise le terme $Trace-based$ pour qualifier une spécification EB^3 et le terme $State-based$ pour qualifier une spécification obtenue par ma méthode.

Traduction de la spécification EB^3 en une spécification B équivalente

L'état de la spécification est composé d'une seule variable qui représente la valeur courante de la trace du système; il y a une opération B par événement externe de l'espace d'entrées-sorties et son contenu se résume à ajouter l'événement correspondant à la trace du système. Une trace est naturellement représentée en B par une séquence. Nous avons bien évidement besoin d'exprimer la sémantique de EB^3 en B. Celle-ci est contenue dans une machine annexe, appelée $ProcessAlgebraDef$. Nous ne l'avons pas encore écrite mais cela ne devrait pas poser de problèmes : il s'agit de coder en B les règles d'inférence qui définissent le langage EB^3. Une autre machine, appelée $VideoClub$EB^3$Spec$, contient les définitions liées à la spécification EB^3 considérée, comme par exemple la définition de l'ensemble, appelé $Valid$, des traces valides construites à partir de l'expression de processus définissant la spécification.

Sur l'exemple du club vidéo on obtient la machine B suivante :

MACHINE $VideoClub_Trace-based$
SEES
$ProcessAlgebraDef$, $VideoClub$EB^3$Spec$
SETS
$CLIENT$,$CASSETTE$
VARIABLES
$systemTrace$
INVARIANT
$systemTrace \in Valid$

INITIALISATION
$systemTrace := []$

OPERATIONS
$result \longleftarrow$ **EnregistrerClient**($client_p$,nom_p) $\triangleq$
LET $traceAfter$ BE
$traceAfter = systemTrace \leftarrow \langle$EnregistrerClient,$client_p$,$nom_p\rangle$
IN
PRE
$client_p \in CLIENT \wedge$
$nom_p \in STRING$

```
        THEN
            IF traceAfter ∈ Valid
                THEN
                    systemTrace := traceAfter ||
                          result := OK
                ELSE
                    result := ERREUR
            END
    END;
...
END
```

Pour simplifier la présentation, je considère que la spécification B $State-based$ de l'exemple du club vidéo est définie dans une seule machine. La seule différence par rapport aux différents éléments de spécification que j'ai donnés dans les chapitres précédents vient du changement de multiplicité de l'association $Pret$. La variable B associée devient une fonction partielle de $Cassette$ dans $Client$.

Pour établir que la spécification B $State-based$ est un raffinement de la spécification B $Trace-based$, il faut définir la relation entre les variables représentant l'état du SI de la première et la variable représentant la trace de la seconde. Cette relation est spécifiée dans l'invariant de la machine $State-based$ et constitue l'invariant de collage. Nous avons introduit un niveau intermédiaire de raffinement pour simplifier l'expression de cet invariant et donc les preuves. La trace globale du système est découpée en traces élémentaires associées aux classes du diagramme de classes. Le tableau de la figure 6.2 donne une idée intuitive des valeurs des variables des trois machines B considérées.

Machine $Trace-based$	Machine intermédiaire	Machine $State-based$
trace du système	traces des classes	classes/associations
AjoutCassette(ca,t) . EnregistrerClient(cl,n) . PretCassette(cl,ca)	**cassette** ca : AjoutCassette(ca,t) . PretCassette(cl,ca) **client** cl : EnregistrerClient(cl,n) . PretCassette(cl,ca)	$Cassette = \{ca\}$ $Titre = \{ca \mapsto t\}$ $Client = \{cl\}$ $Nom_Cl = \{cl \mapsto n\}$ $Pret = \{ca \mapsto cl\}$

FIG. 6.2 – *Valeurs des variables d'état des trois machines B*

Spécification de la machine intermédiaire

Dans cette machine sont définies les variables qui associent à chaque classe sa trace :

$$T_{Cassette} \in CASSETTE \nrightarrow seq(I) \ \ \wedge$$
$$T_{Client} \in CLIENT \nrightarrow seq(I)$$

L'ensemble I est l'espace d'entrées-sorties.

L'invariant de collage qui relie ces variables à la variable $systemTrace$ établit que la décomposition de $systemTrace$ est correcte, c'est-à-dire que le prédicat $class_traces(systemTrace, T_{Cassette}, T_{Client})$ doit être vrai.

Chaque opération consiste à ajouter l'événement correspondant à l'opération à chaque trace de classe concernée par l'événement. Par exemple, l'événement PretCassette implique une mise à jour des traces de *Client* et *Cassette*.

$$
\begin{array}{l}
result \longleftarrow \mathbf{PretCassette}(client_p, cassette_p) \triangleq \\
\quad \text{PRE} \\
\qquad client_p \in CLIENT \ \wedge \\
\qquad cassette_p \in CASSETTE \\
\quad \text{THEN} \\
\qquad \text{IF} \\
\qquad\qquad activeClient\ (client_p) \ \wedge \\
\qquad\qquad activeCassette\ (cassette_p) \ \wedge \\
\qquad\qquad \neg enPret\ (cassette_p) \\
\qquad\quad \text{THEN} \\
\qquad\qquad T_{Client}(client_p) := T_{Client}(client_p) \leftarrow \\
\qquad\qquad\qquad\qquad \langle \mathsf{PretCassette}, client_p, cassette_p \rangle \ || \\
\qquad\qquad T_{Cassette}(cassette_p) := T_{Cassette}(cassette_p) \leftarrow \\
\qquad\qquad\qquad\qquad \langle \mathsf{PretCassette}, client_p, cassette_p \rangle \ || \\
\qquad\qquad\qquad result := OK \\
\qquad\quad \text{ELSE} \\
\qquad\qquad result := ERREUR \\
\qquad \text{END} \\
\quad \text{END;}
\end{array}
$$

La condition de la substitution IF est la seule partie vraiment créative de cette opération. En fait c'est elle qui traduit les contraintes d'ordonnancement de la spécification EB^3. Dans l'exemple, si on analyse l'expression de processus de la classe *Client*, un événement PretCassette doit toujours être précédé par un événement EnregistrerClient et jamais par un événement RadierClient. C'est ce qu'exprime le prédicat suivant :

$$activeClient(c) \triangleq c \in dom(T_{Client}) \ \wedge \ label(last(T_{Client}(c))) \neq \mathsf{RadierClient}$$

En analysant l'expression de processus de la classe *Cassette*, on définit de la même manière le prédicat $activeCassette$. On s'aperçoit aussi que la cassette ne doit pas être empruntée, ce qui se traduit par :

$$enPret(c) \triangleq label(last(T_{Cassette}(c) \overleftarrow{\prod} \{\mathsf{PretCassette}, \mathsf{RetourCassette}\})) = \mathsf{PretCassette}$$

Expression de l'invariant de collage entre la machine intermédiaire et la machine *State – based*

Cet invariant exprime comment les variables de la machine *State – based* qui correspondent aux différents éléments du diagramme de classes sont obtenues à partir des variables de la machine

intermédiaire. Pour une variable correspondant à une classe, on utilise un prédicat qui définit l'ensemble des objets existants de la classe et dont la définition est similaire à celle de *activeClient*. Ainsi, la variable *Client* est définie par :

$$Client = \{cl \mid activeClient(cl)\}$$

Pour une association, il faut prendre en compte les multiplicités. Par exemple, la variable correspondant à l'association *Pret* se définit par :

$$Pret = \lambda\, ca \cdot (activeCassette(ca) \ \wedge \ enPret(ca) \mid emprunteur(ca))$$

où l'emprunteur d'une cassette est le client du dernier prêt :

$$emprunteur(c) \triangleq cl(last(T_{Cassette}(c) \lceil\!\rceil \{\mathsf{PretCassette}\}))$$

Les opérations de la machine $State - based$ ne sont bien sûr pas modifiées. On est sûr de leur correction en établissant, non plus la preuve par rapport à l'invariant de la machine elle-même, mais la preuve de raffinement par rapport à l'opération correspondante dans la machine intermédiaire.

6.4 Discussion - Perspectives

Tout d'abord, quelques mots sur les preuves que nous avons faites. Elles ne sont pas très complexes et surtout ce qui est intéressant c'est que chaque preuve est "locale", c'est-à-dire qu'elle ne dépend que des classes et/ou des associations sur lesquelles travaillent l'opération. Ceci signifie que le nombre d'éléments du diagramme de classes n'a pas d'impact sur la complexité de la preuve. Comme on part d'une trace globale du système, ce n'était pas un point évident au départ de notre recherche.

Il est évident que le processus ne pourra pas être complètement automatisé. Je l'ai souligné dans la section précédente, il y a certains éléments des raffinements qui doivent être "inventés" par le spécifieur (conditions du IF dans les opérations de la machine intermédiaire, expressions des associations de la dernière machine, ...). Mais il est possible de fournir un support méthodologique sous forme d'heuristiques et d'un guide. Pour aller plus loin, de la même manière que des patrons peuvent être définis pour la spécification des SI, nous pensons que des patrons de raffinement peuvent être définis. C'est un axe que nous comptons développer.

Un autre axe de recherche est d'étudier la possibilité de générer un système d'information exécutable à partir d'une spécification formelle EB^3 sans passer par la spécification B $State - based$. C'est un projet que nous avons défini dans [FFLR02] et qui fait l'objet de la thèse de Mario Richard que nous encadrons conjointement.

Chapitre 7

Construction systématique de hiérarchies de généralisation/spécialisation

Le concept d'héritage est un des concepts fondamentaux de la conception par objets. Comme je l'ai souligné à la section 2.5 du chapitre 2, il est maintenant clairement reconnu par la communauté scientifique qu'il faut distinguer l'héritage d'implémentation qui est un mécanisme des langages de programmation par objets de celui de conception qui permet de modéliser les liens de généralisation/spécialisation. C'est ce dernier type d'héritage que je considère. Dans le chapitre 2 je m'intéresse à la formalisation de l'héritage et je pars du principe que les hiérarchies d'héritage sont bien conçues. Or obtenir ce résultat n'est pas une tâche facile. Aucune des méthodes d'analyse-conception existantes n'offre un mécanisme rigoureux pour y arriver, c'est laissé à l'appréciation, l'expérience du concepteur. La thèse de Nadira Lammari [Lam96] propose une solution dont les principes généraux sont présentés dans la suite. Je joins en annexe de ce manuscrit un article [LLJ99] qui décrit avec plus de détails que ci-dessous les travaux de la thèse.

La construction systématique de hiérarchies d'héritage repose sur la définition d'un nouveau type de contraintes spécifiées entre les caractéristiques (attributs et opérations) d'une classe et appelées *contraintes d'applicabilité*. Nous avons étendu le concept de contraintes sur les valeurs nulles définies uniquement sur les attributs pour prendre en compte les opérations d'une classe. Les contraintes d'applicabilité utilisent le concept de *domaine d'applicabilité*. Pour un attribut, c'est l'ensemble des instances possibles d'une classe où l'attribut a ou peut avoir une valeur. Pour une opération, c'est l'ensemble des instances possibles d'une classe qui peuvent être créées, supprimées, modifiées ou interrogées. Les contraintes d'applicabilité permettent de comparer les domaines d'applicabilité : elles expriment des relations d'inclusion, d'égalité ou de disjonction entre ces domaines. Ces contraintes ont été formellement définies dans la thèse et dans le rapport de recherche [LLJC94].

Ensuite nous avons défini deux mécanismes de réorganisation des hiérarchies d'héritage, basés sur la spécification des contraintes d'applicabilité. Le premier est un *mécanisme de décomposition*. A partir d'un diagramme de classes quelconque, il permet d'obtenir un diagramme canonique où chaque classe ne contient que des attributs ou opérations applicables à toutes les instances de la classe. Généralement, le diagramme de classes initial contient des attributs optionnels (dans le sens : jamais applicable à certaines instances de la classe) qui peuvent masquer des liens d'héritage. C'est une technique très connue et utilisée lorsque l'on fait de la rétro-conception d'une base de données relationnelle vers un diagramme objets. Ce caractère optionnel, ainsi que d'éventuels liens de généralisation/spécialisation présents dans le diagramme, sont traduits par des contraintes d'applicabilité.

D'autres contraintes sont définies par le concepteur, en particulier les contraintes entre attributs et opérations. Intuitivement, le mécanisme agit de la manière suivante. Les caractéristiques ayant même domaine d'applicabilité sont regroupées dans une même classe. Ainsi une classe définit un unique domaine d'applicabilité. Si entre deux classes il existe une relation d'inclusion entre leurs domaines d'applicabilité alors un lien de spécialisation est défini entre les deux classes.

Le second mécanisme est un *mécanisme de restructuration*. Il prend en entrée un diagramme canonique et permet de fusionner des classes appartenant à une même hiérarchie d'héritage. Les liens d'héritage supprimés sont remplacés par des contraintes d'applicabilité qui pourront lors de l'implémentation de la base de données être traduites par des contraintes d'intégrité. L'intérêt de ce mécanisme est qu'il permet d'avoir un diagramme plus compact et donc plus lisible et facile à maintenir qu'un diagramme canonique, tout en étant quand même porteur de la même sémantique.

Un outil, supportant ces deux mécanismes, a été développé par Didier Renée lors de son mémoire d'ingénieur CNAM [Ren98]. Les mécanismes étant indépendants des méthodes de conception existantes, il peut être intégré dans n'importe quel AGL.

Nadira Lammari est actuellement maître de conférences au CNAM et membre de l'équipe "Ingénierie des Systèmes d'Information et d'aide à la Décision" du CEDRIC. Elle poursuit ses travaux de recherche sur ce sujet, plus précisément sur l'utilisation des mécanismes à d'autres fins que la conception de hiérarchies d'héritage. Un premier domaine est la rétro-conception de bases de données relationnelles. Les articles [Lam99, ACWL99] proposent une approche de rétro-conception permettant d'extraire des hiérarchies de généralisation/spécialisation à partir d'une base de données relationnelle. Les deux mécanismes ont aussi été utilisés pour la conception et la maintenance de hiérarchies de concepts d'une ontologie [LM02]. Ces dernières sont à l'heure actuelle très utilisées pour la recherche sur le web.

Une perspective intéressante de poursuite de notre travail est de pouvoir extraire automatiquement de la spécification formelle des opérations certaines contraintes d'applicabilité entre attributs et méthodes ou méthodes entre elles.

Chapitre 8

Conclusion

L'objectif de mes travaux de recherche était de spécifier le comportement fonctionnel des transactions d'une base de données au même niveau d'abstraction que les données décrites par les modèles sémantiques, en utilisant une méthode de spécifications formelles existante afin de pouvoir bénéficier de tous les outils de raisonnement associés. On obtient ainsi une spécification globale d'un SI dont la cohérence données-traitements est vérifiée. J'ai montré ensuite qu'il était possible d'aller plus loin en générant, par un processus de raffinement, une implémentation relationnelle à partir de la spécification formelle obtenue. Le processus étant formel, les transactions générées sont correctes par rapport à la spécification.

Ces travaux ont été réalisés en utilisant la méthode B dont le paradigme de spécification, basé sur les transitions d'états, est bien adaptée aux types de systèmes et aux types de propriétés considérés. La seule contrainte imposée par l'utilisation de B est liée à son mécanisme de structuration des spécifications qui est différent de celui qu'on utilise habituellement en SI.

Plus récemment, je me suis tournée vers des méthodes formelles qui permettent d'exprimer et de valider d'autres types de propriétés que les propriétés fonctionnelles, plus liées à l'aspect dynamique des SI. Il est alors nécessaire de définir des règles qui permettent d'assurer la cohérence globale du système.

Il y a plusieurs points sur l'utilisation des méthodes formelles pour les SI que je n'ai pas abordés. En particulier, je n'ai pas considéré l'intégration des méthodes formelles avec un processus d'analyse-conception de systèmes. C'est un point important car c'est une des raisons principales de la mauvaise intégration des méthodes formelles dans les grands projets. C'est ce que propose Michel Lemoine [Lem00] dans son habilitation. Je pense qu'il faut avoir une grande expérience de projets industriels, que je ne possède pas.

En revanche, il me paraît essentiel de réfléchir à des moyens qui peuvent faciliter l'utilisation des méthodes formelles en général. Ceci permettrait aux donneurs d'ordre, qui bien souvent ne connaissent pas les notations formelles, de valider les spécifications et aux ingénieurs logiciels, qui n'ont pas nécessairement une culture mathématique très approfondie, de manipuler les outils de preuve. Ceci nécessite de définir des interfaces de manipulation des notations formelles plus conviviales, et en particulier graphiques. Le langage IS-UML que j'ai défini est un point de départ. L'objectif final est de pouvoir cacher la spécification formelle textuelle et de la rendre ainsi transparente au spécifieur. Il faut alors pouvoir remonter les problèmes détectés par les outils de preuve vers les diagrammes UML et aller ainsi vers des outils de preuve graphiques.

Un autre axe concerne la réutilisation de composants de spécification. Aujourd'hui, réutiliser en génie logiciel consiste essentiellement à réutiliser des morceaux de programmes fournis dans des

bibliothèques ou des environnements de développement. Les recherches actuelles visent à exploiter la réutilisation dans les phases amont du processus de développement de logiciels, en analyse et conception. L'objectif est alors de permettre aux concepteurs de travailler à un niveau d'abstraction élevé et donc de réutiliser davantage les connaissances acquises. Il existe déjà des propositions de composants UML [GHJV95, Fow97] mais leurs définitions restent informelles et donc peu exploitables par des outils. L'idée est de définir une méthode formelle outillée qui permet de réutiliser des composants de spécification. Ce travail a été commencé lors du stage de DEA de Frédéric Gervais [Ger02].

Bibliographie

[Abr74] J.R. Abrial. Data semantics. In Klimbie and Koffeman, editors, *Database Management.* North-Holland, 1974.

[Abr80] J.R. Abrial. The specification language Z: Basic library. Technical report, Oxford University, Programming Research Group, 1980.

[Abr95] J.R. Abrial. The Bug Book. Publication privée du "B User Group", October 1995.

[Abr96] J-R. Abrial. *The B-Book.* CUP, 1996.

[Abr02] J.R. Abrial. Modélisation formelle par événements. In *Journées du groupe B du GDR ALP, CNAM, Paris, France*, juin 2002.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. GALILEO: A strongly typed interactive conceptual language. *ACM Transactions on Database Systems*, 10(2), 1985.

[ACWL99] J. Akoka, I. Comyn-Wattiau, and N. Lammari. Relational database reverse engineering: Elicitation of generalization hierarchies. In *Workshop REIS'99 of E/R'99, Paris, France*, volume 1727 of *LNCS*. Springer Verlag, 1999.

[AH87] S. Abiteboul and R. Hull. IFO: a formal semantic database model. *ACM Transactions on Database Systems*, 12(4), 1987.

[AM98] J-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B98: Second International B Conference, Montpellier, France*, volume 1393 of *LNCS*, pages 83–128. Springer Verlag, April 1998.

[ANS75] ANSI/X3/SPARC. American National Standard Institute study group on DBMS, interim report. Technical report, FDT 7:2 bulletin of ACM SIGMOD, 1975.

[AR97] P. Andre and J.C. Roger. Objets et spécifications formelles. In C. Oussalah et alii, editor, *Ingéniérie objet: concepts et techniques.* InterEditions, Masson, 1997.

[BBFM99] P. Behm, P. Benoit, A. Faivre, and J.M. Meynadier. Météor: A successful application of B in a large project. In *FM99: World Congress on Formal Methods, Toulouse, France*, volume 1708 of *LNCS*, pages 369–387. Springer-Verlag, September 1999.

[BC] B-Core. B-Toolkit, on-line manual. Oxford, UK, available at http://www.b-core.com.

[BCN92] C. Batini, S. Ceri, and S. Navathe. *Conceptual Database Design: an Entity-Relationship Approach.* Benjamin/Cummings Publishing Company, 1992.

[BCPC+97] F. Bounaas, M. Chabre-Peccoud, P.Y. Cunin, J.P. Giraudin, P. Morat, and D. Rieu. Objets et méta-modélisation. In C. Oussalah et alii, editor, *Ingéniérie objet: concepts et techniques.* InterEditions, Masson, 1997.

[BE95] D. Bert and R. Echahed. On the operational semantics of the algebraic and logic language LPG. In *Recent Trends in Data Types Specification, 10th Workshop on Specification of Abstract Data Types*, volume 906 of *LNCS*, pages 132–152. Springer-Verlag, 1995.

[BGG+92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.

[BGV98] M. Bouzeghoub, G. Gardarin, and P. Valduriez. *Les Objets*. Eyrolles, 2 edition, 1998.

[BHS99] F. Barbier and Henderson-Sellers. Black and white diamonds. In *UML99*, volume 1723 of *LNCS*, pages 550–565. Springer-Verlag, 1999.

[Bid86] M. Bidoit. The PLUSS specification language. In Klaus Drosten, Hans-Dieter Ehrich, Martin Gogolla, and Udo W. Lipeck, editors, *Proceedings 4th Workshop on Abstract Data Type*. University of Braunschweig, Department of Computer Science, Informatik-Bericht Nr. 86-09, 1986.

[BP98] M. Blaha and W. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1998.

[But00] M. Butler. CSP2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12(4):182–198, 2000.

[CC98] Common criteria. Version 2.0, http://www.commoncriteria.org/index.html, 1998.

[CE97] T. Clark and A. Evans. Foundations of the Unified Modeling Language. In *Second Northern Formal Methods Workshop, Ilkley, UK*, volume 1241 of *LNCS*. Springer Verlag, 1997.

[CEKS01] A. Clark, A. Evans, S. Kent, and P. Sammut. The MMF approach to engineering object-oriented design languages. In *LTDA2001: Workshop on Language Descriptions, Tools and Applications*, April 2001.

[Cer83] S. Ceri. *Methodologies and Tools for Database Design*. Elsevier Science, 1983.

[CGR95] D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, 1995.

[Che76] P.P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1), 1976.

[CKM+99] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. Defining UML family members using prefaces. In C. Mingins and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems, TOOLS'99 Pacific*. IEEE Computer Society, 1999.

[Cle] Clearsy. Atelier B, manuel de référence. available at http://www.atelierb.societe.com.

[Cod70] E. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6), 1970.

[COD71] CODASYL. Feature analysis of generalized data base management systems. Technical report, System Commitee, ACM, New York, 1971.

[Con02] The Unambiguous UML Consortium. 2nd revised submission to UML2 infrastructure RFP. OMG document ad/2002-09-21, available at http://www.2uworks.org, September 2002.

[DKRS91] R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91-1, St. Lucia 4072, Australia, 1991.

[Dup00] Sophie Dupuy. *Couplage de notations semi-formelles et formelles pour la spécification des systèmes d'information*. PhD thesis, Université Joseph Fourier, Grenoble, France, September 2000.

[DV97] D. Diallo and A. Vailly. Paraphrasage d'une spécification formelle B, un premier essai. In *15ème Congrès INFORSID'97, Toulouse, France*, Juin 1997.

[Eme90] E. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science. volume B : Formal Models and Semantics*. Elsevier, 1990.

[EN00] R.A. Elmasri and S. Navathe. *Fundamental of Database Systems*. Addison-Wesley, 3 edition, 2000.

[FELR98] R. France, A. Evans, K. Lano, and B. Rumpe. Developing the UML as a formal modeling notation. *Computer Standards and Interfaces*, (19):325–334, 1998.

[FFLR02] M. Frappier, B. Fraikin, R. Laleau, and M. Richard. APIS : Automatic production of information systems. In *International AAAI 2002 Spring Symposium : Logic-Based Program Synthesis: State-of-the-Art and Future Trends, Stanford, USA*, March 2002.

[FH00] M. Frappier and H. Habrias. *Software Specification Methods. An Overview Using a Case Study*. Facit series, Springer, 2000.

[FL93] P. Facon and R. Laleau. Modeling semantic data concepts with VDM. In *Fifth International Conference on Software Engineering and Knowledge Engineering, San Francisco, USA*, Juin 1993.

[FL96] P. Facon and R. Laleau. Des modèles d'objets aux spécifications formelles ensemblistes. *Ingéniérie des Systèmes d'Information*, 4(2):239–276, 1996.

[FL01] M. Frappier and R. Laleau. Proving the refinement of scenarios into object-oriented models. Technical Report 277, CEDRIC Laboratory, CNAM, Evry, France, 2001.

[FLMP99] P. Facon, R. Laleau, A. Mammar, and F. Polack. Formal specification of the UML metamodel for building rigorous CAISE tools. Technical report, CEDRIC Laboratory, CNAM, Paris, September 1999.

[FLNM99] P. Facon, R. Laleau, H. P. Nguyen, and A. Mammar. Combining UML with the B formal method for the specification of database applications. Technical report, CEDRIC Laboratory, CNAM, Paris, September 1999.

[FOC] The FOC project. available at http://calfor.lip6.fr/ foc/.

[Fow97] M. Fowler. *Analysis Patterns : Reusable Object Models*. Addison-Wesley, 1997.

[FSD99] M. Frappier and R. St-Denis. Specifying information systems through structured input-output traces. Technical report, Département de mathéematiques et d'informatique, Université de Sherbrooke, Sherbrooke (Québec), Canada J1K 2R1, 1999.

[Gau92] M.-C. Gaudel. Structuring and modularizing algebraic specifications: The PLUSS specification language, evolutions and perspectives. In *STACS92*, volume 557 of *LNCS*, pages 3–18. Springer-Verlag, 1992.

[Ger02] F. Gervais. Réutilisation de composants de spécification en B. Master's thesis, DEA IIE(CNAM)-University of Evry-INT, Evry, France, Jullet 2002.

[GHH+95] C. George, A.E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J.S. Pedersen. *The RAISE Development Method*. Prentice Hall, 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[GHW85] J.V. Guttag, J.J. Horning, and J.M. Wing. The LARCH family of specification languages. *IEEE Software*, 2(5), 1985.

[GM93] M. Gordon and T.F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[GMP83] J.A. Goguen, J. Meseguer, and D. Plaisted. Programming with parameterized abstract objects in OBJ. *Theory and Practice of Software Technology*, 1983.

[Gra98] A. Le Grand. Specialization of object lifecycles. In *International Conference on Object-Oriented Information Systems, Paris, France*. Springer Verlag, September 1998.

[GS97] A.J. Galloway and W. Stoddart. An operational semantics for ZCCS. In M.G. Hinchey and Shaoying Liu, editors, *IFCEM'97: International Conference on Formal Engineering Methods*. IEEE Computer Society Press, 1997.

[Hab01] H. Habrias. *Spécification formelle avec B*. Hermes, 2001.

[Hai90] J.-L. Hainaut. Entity-relationship models : formal specification and comparison. In H. Kangassalo, editor, *ER'90: 9th International Conference Entity-Relationship Approach : the core of conceptual modelling*. Elsevier Science Publishers, 1990.

[Hal90] A. Hall. Using Z as a specification calculus for object-oriented systems. In *VDM'90: 3rd International Conference, Kiel, Germany*, volume 428 of *LNCS*. Springer-Verlag, Avril 1990.

[Har87] D. Harel. A visual formalism for complex systems. *Science of Computer Programming*, 8(3), 1987.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3), 1987.

[HKPM95] G. Huet, G. Kahn, and C. Paulin-Mohring. The COQ proof assistant, a tutorial, v5.10. Technical report, INRIA Rocquencourt, CNRS-ENS Lyon, France, 1995.

[HM81] M. Hammer and D. McLeod. Data description with SDM: a semantic database model. *ACM Transactions on Database Systems*, 6(3), 1981.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, 1985.

[ITS91] Information technology security evaluation criteria. Version 1.2, Office for Official Publications of the European Communities, http://www.cesg.gov.uk/assurance/iacs/itsec/index.htm, 1991.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 2 edition, 1990.

[KC99] S.K. Kim and D. Carrington. Formalizing the UML class diagram using OBJECT-Z. In *UML99*, volume 1723 of *LNCS*, pages 83–98. Springer-Verlag, 1999.

[Lal00] R. Laleau. On the interest of combining UML with the B formal method for the specification of database applications. In *ICEIS2000: 2nd International Conference on Enterprise Information Systems, Stafford, UK*, July 2000.

[Lam96] N. Lammari. *Réorganisation des hiérarchies d'héritages dans un schéma conceptuel objet*. PhD thesis, CEDRIC Laboratory, CNAM, Evry, France, October 1996.

[Lam99] N. Lammari. An algorithm to extract IS-A inheritance hierarchy from a relational database. In *E/R'99: International Conference on Entity/Relationship, Paris, France*, volume 1728 of *LNCS*. Springer Verlag, 1999.

[Lan95] K. Lano. *Formal Object-Oriented Development*. Springer-Verlag, 1995.

[Led98] Y. Ledru. Identifying preconditions with the Z/EVES theorem prover. In *ASE: 13th IEEE Conference on Automated Software Engineering*. IEEE Computer Society Press, October 1998.

[Lem00] M. Lemoine. *Contribution à l'utilisation des méthodes formelles: définition d'un processus évolutionnaire*. Habilitation à diriger des recherches, Université de Toulouse, France, 2000.

[LG96] K. Lano and S. Goldsack. Integrated formal and object-oriented methods : the VDM++ approach. In *Methods Integration Workshop, Leeds, UK*, eWiCS. Springer Verlag, March 1996.

[LH94a] K. Lano and H.Haugthon. Improving the process of system specification and refinement in B. In D. TILL, editor, *6th Refinement Workshop*, Workshops in Computing. Springer-Verlag, 1994.

[LH94b] K. Lano and H.Haugthon. *Object-Oriented Specification Case Studies*. Prentice Hall, 1994.

[LHR95] R. Laleau and N. Hadj-Rabia. Génération automatique de spécifications formelles VDM á partir d'un schéma conceptuel de données. In *INFORSID'95, Grenoble, France*, June 1995.

[LLJ99] N. Lammari, R. Laleau, and M. Jouve. Processus d'optimisation conceptuelle d'un schéma orienté-objet. *L'Objet*, 5(1):29–56, 1999.

[LLJC94] N. Lammari, R. Laleau, M. Jouve, and X. Castellani. Dépendances d'existence : définitions formelles et utilisation pour rechercher des sous-classes. Technical report, num 13, CEDRIC Laboratory, Paris, France, 1994.

[LM00] R. Laleau and A. Mammar. An overview of a method and its support tool for generating B specifications from UML notations. In *ASE: 15th IEEE Conference on Automated Software Engineering, Grenoble, France*. IEEE Computer Society Press, September 2000.

[LM01] R. Laleau and A. Mammar. An automatic generation of B specifications from well-defined UML notations for database applications. In *International Symposium On Programming System, Algiers, Algeria*, 2001.

[LM02] N. Lammari and E. Métais. Automatic help for building and maintaining ontology. In *7th Workshop on Applications of Natural Language to Information Systems (NLDB'02), Stockholm, Sweden*, June 2002.

[Lop02] Nestor Lopez. *Spécification formelle de systèmes complexes : méthodes et techniques*. PhD thesis, CEDRIC Laboratory, CNAM, Paris, France, January 2002.

[LP00] R. Laleau and F. Polack. Metamodels for static conceptual modelling of information systems. In *Workshop "Defining Precise Semantics for UML" of the International Conference ECOOP, Cannes, France*, 2000.

[LP01a] R. Laleau and F. Polack. A rigorous metamodel for UML static conceptual modelling of information systems. In *CAiSE2001: 13th International Conference on Advanced Information Systems Engineering, Interlaken, Switzerland*, volume 2068 of *LNCS*. Springer Verlag, June 2001.

[LP01b] R. Laleau and F. Polack. Specification of integrity-preserving operations in information systems by using a formal UML- based language. *Information and Software Technology*, 43:693–704, 2001.

[LP02] R. Laleau and F. Polack. Coming and going from UML to B : a proposal to support traceability in rigorous IS development. In *ZB2002: Formal Specification and Development in B and Z. 2nd International Conference of B and Z Users, Grenoble, France*, volume 2272 of *LNCS*. Springer-Verlag, January 2002.

[Mam02] Amel Mammar. *Une méthode et un environnement pour le développement formel d'applications bases de données*. PhD thesis, CEDRIC Laboratory, CNAM, Evry, France, November 2002.

[Mao86] R. Laleau-Le Maoult. *Etude d'un langage de programmation relationnel. Extension à la gestion de transactions imbriquées.* PhD thesis, CNAM, Paris, France, December 1986.

[Mar01] G. Mariano. Bcaml : plate-forme logicielle d'expérimentations. In *Journées "Outils pour et autour de B", INRETS-ESTAS, Villeneuve d'Ascq, France*, Octobre 2001.

[MBW80] J. Mylopoulos, P. Bernstein, and H. Wong. A language facility for designing database-intensive applications. *ACM Transactions on Database Systems*, 5(2), 1980.

[Mon97] F. Monge. Formalisation du méta modèle des méthodes graphiques d'analyse et conception orientées objet. Master's thesis, DEA IIE(CNAM)-University of Evry-INT, Evry, September 1997.

[Mét99] E. Métais. *The Role of Knowledge and Reasoning Mechanisms in Database Design Tools.* Habilitation à diriger des recherches, Université de Versailles Saint-Quentin, France, 1999.

[Ngu98] H. P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi–formelles.* PhD thesis, CEDRIC Laboratory, CNAM, Evry, December 1998. available at www.iie.cnam.fr/~ laleau/thesePHNguyen.ps-gz.

[NH89] G.M. Nijssen and T.A. Halpin. *Conceptual schema and relational database design, a fact oriented approach.* Prentice Hall, New Jersey, 1989.

[NR94] N. Nagui-Raïss. A formal software specification tool using the Entity/Relationship model". In *ER'94: 13th International Conference Entity-Relationship Approach, Manchester, UK*, LNCS. Springer-Verlag, December 1994.

[oLD99] The CoFI Task Group on Language Design. The common algebraic specification language summary. Technical report, IFIP WG 1.3, 1999.

[OMG99] OMG. *Unified Modeling Language Specification*, 1999. available at http://www.omg.org.

[ORS95] S. Owre, J. Rushby, and N. Shankar. Formal verification for fault-tolerant architectures : Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(5):107–125, 1995.

[PDH02] V. Prevosto, D. Doligez, and T. Hardin. Algebraic structure and dependent records. In Sofiène Tahar César Muñoz and Víctor Carreño, editors, *Proceedings of TPHOLs 02.* Springer-Verlag, August 2002.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, CS Department, University of Aarhus, 1981.

[PM88] J. Peckham and F. Maryanski. Semantic data models. *ACM Computing Surveys*, 20(3), 1988.

[Poi00] P. Poizat. *KORRIGAN: un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes.* PhD thesis, Université de Nantes, France, November 2000.

[PS92] C. Parent and S. Spaccapietra. ERC+: An object-based entity relationship approach. In R. Zicari P. Loucopoulos, editor, *Conceptual Modeling, Databases and CASE: an Integrated View of Information Systems Development.* John Wiley and Sons, 1992.

[PWM93] F. Polack, M. Whiston, and K. Mander. The SAZ project: Integrating SSADM and Z. In *FME93: First International Symposium of Formal Methods Europe, Odense, Denmark*, volume 670 of *LNCS*, pages 541–557. Springer Verlag, April 1993.

[Ren98] D. Renée. *Un outil de réorganisation des hiérarchies d'héritages sémantiques.* Mémoire d'ingénieur, CNAM, Paris, France, 1998.

[Ric01] M. Richard. Intégration de la méthode EB^3 dans un processus logiciel orienté objets. Master's thesis, Département de mathématiques et d'informatique, université de Sherbrooke, Québec, Canada, 2001.

[Sah99] M. Sahbani. Formalisation des relations de composition et d'agrégation des modèles sémantiques. Master's thesis, DEA IIE(CNAM)-University of Evry-INT, Evry, September 1999.

[SB01] C. Snook and M. Butler. Using UML class diagrams for constructing B specifications. Technical report, University of Southampton Declarative Systems and Software Engineering Group, UK, 2001. available at http://www.ecs.soton.ac.uk/ mjb/publications/.

[Sch94] K.D. Schewe. *Specification of Data-Intensive Application Systems.* Habilitation Thesis, Brandenburg University of Technology, Cottbus, Germany, 1994.

[SH01] C. Snook and R. Harrison. Practitioners' views on the use of formals methods: an industrial survey by structured interview. *Information and Software Technology*, 43:275–283, 2001.

[Shi81] D. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1), 1981.

[Spi92] J.M. Spivey. *The Z Notation - A Reference Manual.* Prentice Hall, 2 edition, 1992.

[SS77] J.M. Smith and D.C.P. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2(2), 1977.

[SSW91] K.D. Schewe, J.W. Schmidt, and I. Wetzel. Specification and refinement in an integrated database application environment. In *VDM'91: 4th International Conference, Noordwijkerhout, Holland*, LNCS. Springer-Verlag, October 1991.

[Sti92] C. Stirling. Modal and temporal logics. In T.S.E. Maibaum S. Abramsky, D.M. Gabbay, editor, *Handbook of Logic in Computer Science. volume 2 : Background: Computational Structures.* Oxford Science Publications, 1992.

[Tai96] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):439,479, september 1996.

[Tha89] B. Thalheim. The Higher-Order Entity-Relationship Model and DB2. In *MFDBS'89*, volume 364 of *LNCS*. Springer-Verlag, 1989.

[Tre02] H. Treharne. Supplementing a UML development process with B. In *FME02: Formal Methods Europe, Copenhagen, Denmark*, LNCS. Springer-Verlag, July 2002.

[vEVD89] P.H.J. van Eijk, C.A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS.* Elsevier Science Publishers, 1989.

[VJ99] M. Vaziri and D. Jackson. Some shortcomings of OCL, the Object Constraint Language of UML. Response to Object Management Group's Request for Information on UML 2.0, 1999. available at http://sdg.lcs.mit.edu/ dnj/.

[vL00] A. van Lamsweerde. Formal specification : a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering.* ACM Press, 2000.

[WC95] J. Widom and S. Ceri. *Active Database Systems.* Morgan-Kaufmann, CA, USA, 1995.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing. MIT Press, Cambridge Mass., 1993.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language.* Addison-Wesley, 1999.

[WN95] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science. volume 4 : Semantic Modelling.* Oxford University Press, 1995.

Annexe A

Compléments sur le langage B

A.1 Principaux opérateurs relationnels

Nom	Syntaxe	Définition	Condition
Produit cartésien	$s \times t$	$\{x \mapsto y \mid x \in s \wedge y \in t\}$	
Ensemble des parties de S	$\mathbb{P}$ S	$\{x \mid x \subseteq S\}$	
Union distribuée (1)	$\bigcup S$	$\{x \mid \exists\ y.\ (y \in S \wedge x \in y)\}$	*S=ens. d'ensembles*
Union distribuée (2)	$\bigcup z.(P \mid E)$	$\{x \mid \exists\ z.(P \wedge x \in E)\}$	*P est un prédicat, E une expression ensembliste*
Intersection distribuée (1)	$\bigcap S$	$\{x \mid \forall\ y.\ (y \in S \wedge x \in y)\}$	*S=ens. d'ensembles*
Intersection distribuée (2)	$\bigcap z.(P \mid E)$	$\{x \mid \forall\ z.(P \Rightarrow x \in E)\}$	*P est un prédicat, E une expression ensembliste*
Relation	$s \leftrightarrow t$	$\mathbb{P}(s \times t)$	
Inverse	r^{-1}	$\{x \mapsto y \mid y \mapsto x \in r\}$	$r \in s \leftrightarrow t$
Domaine	$dom(r)$	$\{x \mid x \in s \wedge \exists\ y.\ (y \in t \wedge x \mapsto y \in r)\}$	$r \in s \leftrightarrow t$
Codomaine	$ran(r)$	$dom(r^{-1})$	$r \in s \leftrightarrow t$
Composition	$(q;r)$	$\{x \mapsto y \mid (x \mapsto y) \in s \times u \wedge \exists\ z.(z \in t \wedge x \mapsto z \in q \wedge z \mapsto y \in r)\}$	$q \in s \leftrightarrow t \wedge r \in t \leftrightarrow u$
Relation Identité	$id(X)$	$\{x \mapsto y \mid x \in X \wedge y \in X \wedge x = y\}$	*X est un ensemble*
Itération	r^n	$r^0 = Id(s),\ r^{n+1} = r; r^n$	$r \in s \leftrightarrow s$
Fermeture transitive et réflexive	r^*	$r^* = \bigcup n.(n \in \mathbb{N} \mid r^n)$	$r \in s \leftrightarrow s$
Fermeture transitive	r^+	$r^+ = \bigcup n.(n \in \mathbb{N}_1 \mid r^n)$	$r \in s \leftrightarrow s$
Opérateur d'abstraction	$\lambda\, x.$ $(x \in s \wedge P \mid E)$	$\{x \mapsto y \mid (x,y) \in s \times t \wedge P \wedge y = E\}$	*P est un prédicat, E une expression telle que :* $\forall\, x.(x \in s \Rightarrow E \in t)$

Nom	Syntaxe	Définition	Condition
Restriction	$u \lhd r$	$\{x \mapsto y \mid (x, y) \in r \wedge x \in u\}$	$r \in s \leftrightarrow t \wedge u \subseteq s$
Corestriction	$r \rhd u$	$\{x \mapsto y \mid (x, y) \in r \wedge y \in u\}$	$r \in s \leftrightarrow t \wedge u \subseteq t$
Anti-Restriction	$u \lhd\!\!\!- r$	$\{x \mapsto y \mid (x, y) \in r \wedge x \notin u\}$	$r \in s \leftrightarrow t \wedge u \subseteq s$
Anti-Corestriction	$r \rhd\!\!\!- u$	$\{x \mapsto y \mid (x, y) \in r \wedge y \notin u\}$	$r \in s \leftrightarrow t \wedge u \subseteq t$
Image	$r[w]$	$\{y \mid y \in t \wedge \exists x.\ (x \in w \wedge x \mapsto y \in r)\}$	$r \in s \leftrightarrow t \wedge w \subseteq s$
Produit direct	$f \otimes g$	$\{x \mapsto (y \mapsto z) \mid ((x, y, z) \in s \times u \times v \wedge x \mapsto y \in f \wedge x \mapsto z \in g)\}$	$f \in s \leftrightarrow u \wedge g \in s \leftrightarrow v$
Fonction partielle	$s \nrightarrow t$	$\{f \mid f \in s \leftrightarrow t \wedge \forall x, y, z.$ $(x \mapsto y \in f \wedge x \mapsto z \in f \Rightarrow y = z)\}$	
Fonction totale	$s \rightarrow t$	$\{f \mid f \in s \nrightarrow t \wedge dom(f) = s\}$	
Injection partielle	$s \rightarrowtail t$	$\{f \mid f \in s \nrightarrow t \wedge f^{-1} \in t \nrightarrow s\}$	
Injection totale	$s \rightarrowtail t$	$s \rightarrowtail t \cap s \rightarrow t$	
Surjection partielle	$s \twoheadrightarrow t$	$\{f \mid f \in s \nrightarrow t \wedge ran(f) = t\}$	
Surjection totale	$s \twoheadrightarrow t$	$s \twoheadrightarrow t \cap s \rightarrow t$	
Bijection	$s \rightarrowtail\!\!\!\!\twoheadrightarrow t$	$s \rightarrowtail t \cap s \twoheadrightarrow t$	
Ens. des séquences sur T	$seq(T)$	$\bigcup (1..n \rightarrow T)$ avec $n \in \mathbb{N}$	
Séquence	$[a,b,...,c]$	$\{1 \mapsto a, 2 \mapsto b, ..., n \mapsto c\}$	

A.2 Principales substitutions du langage B

Nom	Notation B	Conditions de définition	Signification
Skip	*skip*		Ne rien faire
Substitution élémentaire	$x := E$	x est une variable E est une expression	Substituer E par x
Précondition	**PRE** P **THEN** S **END** (notée $P \mid S$)	P est un prédicat S est une substitution	S'assurer de P et exécuter S
IF	**IF** P **THEN** S **ELSE** T **END**	P est un prédicat S et T deux substitutions	Si P est vrai, exécuter S sinon exécuter T
Garde	**SELECT** P **THEN** S **END**	P est un prédicat S est une substitution	exécuter S si P est vrai.
ANY	**ANY** X **WHERE** P **THEN** S **END**	X est une liste de variables P est un prédicat S est une substitution	Sélectionner une valeur de X qui vérifie le prédicat P et exécuter S
Simultanée	$S \mid\mid T$	S et T sont des substitutions	exécuter S et T en même temps
Choix	$S \,[]\, T$	S et T sont des substitutions	exécuter S ou T
Séquence	$S \,;\, T$	S et T sont des substitutions	exécuter S puis T

Remarques :

- **IF** P **THEN** S **END** = **IF** P **THEN** S **ELSE** *Skip* **END**.
- La syntaxe complète de la substitution **SELECT** est (**SELECT** P **THEN** S **WHEN** Q **THEN** T **ELSE** U **END**) dont la sémantique est : exécuter une des substitutions dont le prédicat est vrai. Si tous les prédicats sont faux, on exécute la substitution associée à la partie **ELSE**. Une écriture possible de cette substitution est: (**SELECT** P **THEN** S **END**)[](**SELECT** Q **THEN** T **END**)[](**SELECT** $(\neg P \wedge \neg Q)$ **THEN** U **END**)

A.3 Clauses d'assemblage des composants B

La méthode B offre des mécanismes de structuration d'une spécification, d'un raffinement et d'une implémentation. Dans cette section, je ne présente que les clauses utilisées dans ce mémoire. Une description complète est disponible dans [Abr96, Hab01].

Au niveau des machines abstraites :

- **INCLUDES** : Conceptuellement, les ensembles et les variables des machines incluses sont ajoutés à ceux de la machine incluante. Le tableau suivant précise la visibilité des ensembles, variables et opérations d'une machine N incluse dans une machine M :

Clauses de M / Constituants de N	INVARIANT	OPERATIONS
Ensembles	visible	visible
Variables	visible	en lecture
Opérations		visible

  Bien que les opérations d'une machine incluse soient visibles dans la machine incluante, une opération de la machine incluante ne peut faire appel qu'à une seule opération de la machine incluse.

  La clause **INCLUDES** est transitive. Si une machine M_1 inclut une machine M_2 qui à son tour inclut une machine M_3, alors les ensembles et les variables de M_3 sont visibles à partir de M_1.

- **USES** : La clause **USES** est exclusivement employée pour le partage de données entre machines. Le tableau suivant précise la visibilité des ensembles, variables et opérations d'une machine N utilisée dans une machine M :

Clauses de M / Constituants de N	INVARIANT	OPERATIONS
Ensembles	visible	visible
Variables	visible	en lecture
Opérations		

  La clause **USES** n'est pas transitive.

- **SEES** : Les variables d'une machine M vue par une machine N ne sont visibles en lecture que dans les opérations de M. Contrairement à la clause **USES**, les opérations de lecture de la machine M peuvent être utilisées dans celles de N. Le tableau suivant précise la visibilité des ensembles, variables et opérations d'une machine N vue dans une machine M :

Clauses de M / Constituants de N	INVARIANT	OPERATIONS
Ensembles	visible	visible
Variables		en lecture
Opérations		de lecture

  La clause **SEES** n'est pas transitive.

Les raffinements peuvent être construits à partir des machines abstraites par les clauses $INCLUDES$ et $SEES$.

Les implémentations peuvent être construites à partir des machines abstraites par les clauses $IMPORTS$ et $SEES$. La clause $IMPORTS$ est l'équivalent de la clause $INCLUDES$, au niveau de l'implémentation ; la clause $SEES$ de l'implémentation est l'équivalent de la clause $SEES$ entre machines abstraites. La différence vient de ce que dans une implémentation, le principe d'encapsulation est respecté, c'est-à-dire que les variables d'une machine N ne sont pas visibles dans les opérations de l'implémentation qui importe ou voit N.