Combinaison de spécifications formelles pour la modélisation des systèmes d'information

Frédéric Gervais

Conservatoire National des Arts et Métiers
Université de Sherbrooke

THÈSE

présentée pour l'obtention des grades de :

Docteur en Sciences du CNAM,
spécialité **Informatique**

Philosophiæ Doctor en Informatique
de l'Université de Sherbrooke

dans le cadre d'une cotutelle de thèse France-Québec

par

Frédéric GERVAIS

Titre de la thèse :

Combinaison de spécifications formelles pour la modélisation des systèmes d'information

soutenue le 4 décembre 2006 devant le jury suivant :

Présidente du jury :	Isabelle Comyn-Wattiau, CNAM, Paris
Rapporteurs :	Yves Ledru, Université Joseph Fourier, Grenoble
	Ali Mili, New Jersey Institute of Technology, USA
Examinateurs :	Richard St-Denis, Université de Sherbrooke, Canada
	Anatol Slissenko, Université Paris 12, Créteil
Directeurs de thèse :	Véronique Donzeau-Gouge, CNAM, Paris
	Marc Frappier, Université de Sherbrooke, Canada
	Régine Laleau, Université Paris 12, Créteil

Remerciements

Je remercie tout d'abord tous les membres du jury pour leur disponibilité et ce, malgré les contraintes liées à la cotutelle et aux emplois du temps chargés de chacun. Je remercie Yves Ledru et Ali Mili d'avoir rapporté cette thèse et pour le temps qu'ils ont consacré à l'analyse de ce document. Je tiens à remercier aussi Isabelle Comyn-Wattiau, Richard St-Denis et Anatol Slissenko d'avoir accepté de participer à mon jury de soutenance.

Toute ma gratitude va à Jean-Luc Kors pour son soutien et pour sa disponibilité à rechercher un support technique de qualité pour la soutenance.

Je tiens à remercier le Ministère des Affaires Etrangères en France et le Ministère des Relations Internationales au Québec pour leur soutien financier. Cette cotutelle France/Québec a été pour moi une expérience très enrichissante.

Je n'aurais probablement jamais fait une thèse en informatique sans le soutien et la confiance de Régine Laleau, qui m'a initié au monde de la recherche pendant mon stage de DEA, avant de devenir ma co-directrice de thèse. Je remercie Régine qui m'a fait découvrir l'intérêt des méthodes formelles et le monde des systèmes d'information. Ses conseils et sa bonne humeur ont été déterminants dans l'aboutissement de mon travail.

Je remercie Marc Frappier, mon co-directeur de thèse, de m'avoir donné la chance de découvrir le Québec et d'avoir partagé avec moi sa passion pour le génie logiciel et pour la synthèse automatique de programmes. Dans son domaine, Marc est un peu le Steve Bauer des méthodes formelles dans les systèmes d'information. Son soutien indéfectible, sa bonne humeur et ses encouragements m'ont permis de travailler dans les meilleures conditions.

Je remercie Véronique Donzeau-Gouge et Catherine Dubois de m'avoir accueilli au sein de l'équipe CPR du CEDRIC depuis mon stage de DEA en 2002. Malgré son agenda surchargé, Véronique m'a sans cesse apporté son soutien et donné de nombreux conseils. Je lui suis très reconnaissant d'avoir accepté d'être ma co-directrice de thèse au CNAM. Je remercie Catherine tout d'abord comme enseignante ; elle m'a fait découvrir les langages formels et les techniques de preuve. Je la remercie aussi pour les nombreuses discussions que nous avons pu avoir par la suite (parfois même dans un couloir !) et pour l'attention qu'elle a portée à ma thèse. Je remercie enfin tous les enseignants, techniciens et personnels administratifs qui m'ont accueilli chaleureusement à l'ENSIIE (anciennement IIE) pendant plus de quatre ans. Je pense en particulier à : Florent Chavand, Brigitte Grau, Olivier Hubert, Mireille Jouve, Eric Lejeune, Blandine et Jean-Marc Pasquier, Anne-Marie Pavageau, Olivier Pons, Xavier Urbain et Françoise Vuillermet. Un grand merci aussi à Sandrine Blazy qui m'a fait découvrir la méthode B et qui m'a encadré avec Régine pendant mon stage de DEA.

Dans le cadre de la cotutelle, j'ai passé 18 mois de ma thèse à l'Université de Sherbrooke. Je remercie tous les membres du Département d'informatique de l'Université de Sherbrooke pour leur accueil et leur gentillesse. Je pense en particulier à : François Boivin, Lise Charbonneau, Sylvain Giroux, Froduald Kabenza, Luc Lavoie, Lynn Le Brun, André Mayers, Hélène Pigot, Chantal Proulx et Marie-France Roy.

Je tiens à remercier tous les membres du LACL qui m'ont accueilli au sein de leur laboratoire depuis que je suis ATER à l'Université Paris 12. Je pense notamment à : Danièle Beauquier, Alexis Bès, Joël Brunet, Patrick Cégielski,

Joëlle Cohen, Catalin Dima, Marie Duflot, Elisabeth Pelz, Farida Semmak et l'indispensable Flore Tsila.

Les études récentes montrent que la recherche française n'est pas assez reconnue dans le monde. La lourdeur administrative en est probablement en partie responsable. Je tiens à remercier ici le personnel administratif qui m'a aidé et conseillé dans les démarches administratives, malgré la rigidité des règlements et la multiplication des instances responsables. J'espère que les décideurs prendront un jour une direction différente et mettront en place une sorte de "guichet unique", avec un seul interlocuteur qui aurait la responsabilité de gérer les démarches entre les différentes administrations, afin de simplifier la vie du doctorant et de réduire la charge aujourd'hui trop importante consacrée à ces efforts infructeux.

Au fil de mes nombreux déplacements entre la France et le Québec, j'ai rencontré plusieurs voisins de bureau qui ont supporté mes moments d'euphorie et de doute. Je les remercie pour leur soutien moral et pour les nombreuses discussions qui ont permis d'animer mes journées de travail. Je dis un grand merci à : Olivier Boite, Matthieu Carlier, Karima Djebali, Jean-Frédéric Etienne, Jérôme Grandguillot, Amel Mammar, Jean-Marc Mota et Pierre Rousseau (ENSIIE) ; Panawé Batanado, Daniel Côté, Larbi Djaider, Benoît Fraikin et Nicolas Richard (Université de Sherbrooke) ; Frédéric Gava et Louis Gesbert (Université Paris 12) ; Nuno Amálio, Akram Idani, Mario Richard et Tarek Sadani (un peu partout).

Dans une vie antérieure, comme j'ai l'habitude de dire pour évoquer mes années d'études en mathématiques, j'ai fait un stage au sein du LAMI de l'Ecole Nationale des Ponts et Chaussées. Je tiens à remercier ici les personnes qui m'ont côtoyé à cette époque. Je pense notamment à Thien Phu Le, que j'ai retrouvé plus tard par hasard à Sherbrooke (comme le monde est petit).

Je remercie tous mes amis qui, malgré l'éloignement physique, ont gardé le contact avec moi. Lors de nos retrouvailles régulières, on se rend vite compte que la distance ne compte pas. Merci à : Arnaud et Virginie, Arnaud et Mélanie, Isabelle et Lionel, Florence, Frédéric, François, Nicolas et Stéphanie.

Bien sûr, je tiens à remercier toute ma famille qui m'a soutenu et encouragé. Mes parents m'ont donné la chance de suivre la voie que je souhaitais et je leur en suis extrêmement reconnaissant : Papa, Maman, je vous remercie pour votre soutien et pour vos encouragements. Mon frère et ma soeur m'ont également été d'un grand soutien, notamment par nos échanges de courriels lors de mes séjours au Québec. Merci aussi à Nonna Carmen. Enfin, je n'oublie pas mes neveux, qui commençaient à s'impatienter à l'idée de lire le "livre" de Tonton.

J'ai une pensée particulière pour mon grand-père Quirino qui était très curieux au sujet de ma thèse et que j'ai eu la chance de revoir une dernière fois malgré mes nombreux déplacements. Je lui dédie aujourd'hui cette thèse : Ciao Quirino !

Table des matières

Table des Matières iii

Introduction 1

I Contexte et état de l'art 7

1 Systèmes d'information et méthodes formelles 9
1.1 Contexte scientifique . . . 9
1.2 Introduction aux SI . . . 10
1.2.1 Composantes des SI . . . 11
1.2.2 Conception des SI . . . 11
1.3 Propriétés des SI . . . 12
1.3.1 Quels types de propriétés ? . . . 13
1.3.2 Et les spécifications formelles ? . . . 14
1.4 Deux exemples de spécification formelle des SI . . . 15
1.4.1 UML-B-SQL . . . 15
1.4.2 EB^3 . . . 16
1.4.3 Discussion . . . 17

2 Langages formels et applications 19
2.1 Brefs rappels sur les méthodes formelles . . . 19
2.1.1 Langages basés sur les états et sur les événements . . . 19
2.1.2 Sémantique . . . 20
2.1.3 Vérification . . . 22
2.1.4 Raffinement . . . 23
2.2 Langages basés sur les états . . . 23
2.2.1 Action Systems . . . 23
2.2.2 Langage Z . . . 25
2.2.3 Object-Z . . . 27
2.2.4 Langage B . . . 29
2.2.5 B événementiel . . . 32
2.3 Langage basé sur les événements : CSP . . . 33

3 Combinaison de spécifications formelles 39
3.1 Critères de comparaison . . . 39
3.2 csp2B . . . 41

3.2.1 Syntaxe . . . 41
3.2.2 Spécification . . . 42
3.2.3 Outil csp2B . . . 44
3.2.4 Vérification et raffinement . . . 46
3.2.5 Adéquation avec les SI . . . 47
3.3 CSP || B . . . 48
3.3.1 Syntaxe . . . 48
3.3.2 Spécification . . . 49
3.3.3 Vérification et raffinement . . . 51
3.3.4 Adéquation avec les SI . . . 53
3.4 CSP-OZ . . . 53
3.4.1 Syntaxe . . . 54
3.4.2 Spécification . . . 55
3.4.3 Vérification et raffinement . . . 58
3.4.4 Adéquation avec les SI . . . 58
3.5 Circus . . . 59
3.5.1 Syntaxe . . . 59
3.5.2 Spécification . . . 60
3.5.2.1 Déclaration . . . 60
3.5.2.2 Action principale . . . 62
3.5.3 Vérification et raffinement . . . 62
3.5.4 Adéquation avec les SI . . . 63
3.6 PLTL et B événementiel . . . 64
3.6.1 Syntaxe . . . 64
3.6.2 Spécification . . . 65
3.6.3 Vérification et raffinement . . . 66
3.6.4 Adéquation avec les SI . . . 67
3.7 Comparaison des approches . . . 68
3.7.1 Synthèse des approches présentées . . . 69
3.7.2 Discussion . . . 70
3.7.3 Autres approches de combinaisons . . . 72

4 Langage EB³ 75
4.1 Spécification en EB³ . . . 75
4.1.1 Exemple . . . 76
4.1.2 Traces valides d'un SI . . . 78
4.1.2.1 Syntaxe . . . 78
4.1.2.2 Expression de processus . . . 78
4.1.2.3 Patrons de processus . . . 80
4.1.3 Règles d'entrée-sortie . . . 81
4.2 Définitions d'attributs . . . 82
4.2.1 Langage . . . 82
4.2.1.1 Définition de clé . . . 83
4.2.1.2 Définition d'attribut non clé . . . 84
4.2.1.3 Définition des rôles dans les associations . . . 87
4.2.2 Exécution des définitions d'attributs . . . 88
4.3 Conclusion . . . 90

II Intégration d'EB^3 et B **93**

5 Vers une combinaison des approches EB^3 et B **95**
5.1 Motivations et rappel du problème . . . 95
5.2 Preuve de propriétés EB^3 sur des spécifications B . . . 96
5.2.1 Exemple . . . 97
5.2.2 Vérification de propriétés EB^3 . . . 99
5.2.3 Discussion . . . 100
5.3 De EB^3-B vers EB^4 . . . 101

6 Traduction vers B **103**
6.1 Algorithme général . . . 103
6.2 Partie statique . . . 104
6.3 Génération des substitutions B . . . 105
6.3.1 Substitutions pour l'initialisation . . . 106
6.3.1.1 Définition de clé . . . 106
6.3.1.2 Définition d'attribut non clé . . . 106
6.3.2 Génération des opérations . . . 107
6.3.2.1 Algorithme général . . . 107
6.3.2.2 Analyse des clauses d'entrée . . . 109
6.3.2.3 Génération des substitutions . . . 112
6.3.2.4 Réécriture des formules de substitution . . . 115
6.4 Exemple de la bibliothèque . . . 118
6.5 Analyse et discussion . . . 120

7 Méthode EB^4 **123**
7.1 Description générale d'EB^4 . . . 123
7.1.1 Description des principales étapes . . . 124
7.1.2 Mise en œuvre . . . 125
7.2 Propriétés statiques du SI . . . 125
7.2.1 Prise en compte des gardes des actions EB^3 . . . 126
7.2.2 Identification des conditions *CS* . . . 127
7.2.2.1 Règles de calcul . . . 127
7.2.2.2 Exemple de la bibliothèque . . . 130
7.2.2.3 Spécification et vérification de propriétés de sûreté 130
7.2.2.4 Génération des gardes . . . 131
7.2.3 Discussion sur les gardes . . . 131
7.3 Propriétés dynamiques . . . 132
7.3.1 Raffinement . . . 133
7.3.2 Définition de patrons de raffinement . . . 135
7.3.2.1 Patron de base pour les types d'entité et associations . . . 135
7.3.2.2 Patron pour les entités participant à des associations binaires . . . 136
7.3.2.3 Analyse de la preuve . . . 136
7.3.2.4 Exemple de la bibliothèque . . . 137
7.4 Conclusions . . . 138

8 Synthèse de l'approche EB^4 139
8.1 Contributions . . . 139
8.2 Discussion . . . 140
8.2.1 Comparaison avec les autres approches de combinaison . . 141
8.2.2 Limites . . . 142
8.2.2.1 Inconvénients . . . 142
8.2.2.2 Améliorations possibles . . . 144
8.3 Perspectives . . . 145

III Synthèse de transactions 147

9 Génération de transactions BD relationnelles 149
9.1 Projet APIS . . . 149
9.1.1 Composantes du projet APIS . . . 149
9.1.2 Exemple d'application d'EB^{3}TG . . . 150
9.2 Algorithmes de génération des transactions . . . 152
9.2.1 Algorithme général . . . 153
9.2.2 Création des tables de la BD . . . 154
9.2.3 Initialisation . . . 154
9.2.4 Analyse des clauses d'entrée . . . 154
9.2.4.1 Algorithme . . . 154
9.2.4.2 Calcul de $Att(\mathsf{a})$ . . . 155
9.2.4.3 Calcul de $K_D(b)$ et $K_{IU}(b)$ . . . 155
9.2.4.4 Définition de variables et de tables temporaires . 156
9.2.5 Définition des transactions . . . 157
9.3 Conclusions et perspectives . . . 159

10 Patrons SELECT 161
10.1 Introduction . . . 161
10.2 Patrons de base pour prédicats atomiques . . . 162
10.2.1 Patron $k = p$ . . . 162
10.2.2 Patron $k = g(\vec{p})$ . . . 163
10.2.3 Patron $k = g(\vec{k}, \vec{p})$, avec $k \notin \vec{k}$ . . . 163
10.2.4 Patron $k\ op\ g(\vec{k}, \vec{p})$, avec $k \in \vec{k}$ . . . 164
10.2.5 Patron $f(\vec{k^c}, \vec{k^1}, \vec{p^1})\ op\ g(\vec{k^c}, \vec{k^2}, \vec{p^2})$ . . . 165
10.2.6 Patron $k \in aRole(\vec{p})$ . . . 166
10.2.7 Patron $k \in eKey()$ . . . 166
10.3 Patrons étendus . . . 167
10.4 Conjonction des prédicats . . . 167
10.5 Analyse et discussion . . . 169

11 Mise en œuvre 171
11.1 Correction de la synthèse de transactions . . . 171
11.1.1 Modèles utilisés dans la preuve . . . 172
11.1.2 Idée de la preuve . . . 173
11.2 Outil . . . 176
11.2.1 Description de l'outil EB^{3}TG . . . 176
11.2.2 Forme des transactions générées par EB^{3}TG . . . 178

11.2.3 Conclusion 180
11.3 Synthèse de l'approche 182
11.3.1 Autres travaux 182
11.3.2 Contributions et perspectives 182

Conclusion **187**

Bibliographie **197**

Table des Figures **200**

Annexes **203**

A Glossaire des notions utilisées **203**

B Langage B **207**

C Exemple de la bibliothèque **209**
C.1 Cas d'étude 209
C.2 Spécification avec EB^3 210
C.2.1 Diagramme ER 210
C.2.2 Expression de processus 211
C.2.3 Règles d'entrée-sortie 213
C.2.4 Définitions d'attributs 213
C.3 Traduction en B 215
C.3.1 Conditions *CS* 219
C.3.2 Conditions *CD* 220
C.4 Synthèse de transactions 220
C.4.1 Fichiers source 221
C.4.2 Programmes générés 224

D Liste des publications **233**

Introduction

> "Il y a deux façons de construire une spécification qui semble correcte : la première est de la faire si simple qu'il n'y a évidemment aucune erreur ; la seconde est de la rendre si compliquée qu'il n'y a aucune erreur évidente."
>
> — Tony Hoare

Motivations

Un système d'information (SI) est un système informatisé qui rassemble l'ensemble des informations présentes au sein d'une organisation, sa mémoire, et les activités qui permettent de les manipuler. Les SI sont aujourd'hui indispensables dans la plupart des entreprises et organismes privés ou publics. Comme il serait impossible de considérer tous les types de SI, nous étudions dans cette thèse les SI qui sont développés autour de systèmes de gestion de bases de données. Les bases de données permettent d'assurer la pérennité des données enregistrées et facilitent leur consultation et modification. De tels systèmes sont caractérisés par des structures de données complexes et par des opérations impliquant des volumes de données importants. Toutefois, les opérations elles-mêmes ne sont pas complexes d'un point de vue algorithmique. Les données peuvent être modifiées ou accédées par de nombreux utilisateurs en concurrence. Enfin, les SI n'ont pas des contraintes fortes au niveau des temps de réponse, mais requièrent dans tous les cas une réponse qui, le cas échéant, peut être un message d'erreur.

Notre domaine de recherche se situe au niveau conceptuel du SI. Cette partie concerne la définition du modèle qui représente les données et les transactions du système. Le succès du développement d'un SI dépend en grande partie de la validité et de la qualité de la spécification de ce niveau d'abstraction. Les autres niveaux de description du système, comme le niveau interne (appelé aussi physique) qui définit la structure de stockage des données, ne sont pas étudiés dans cette thèse, même si des problèmes importants comme l'organisation des fichiers ou la concurrence d'accès à la base de données y sont traités et font l'objet de recherche par ailleurs.

Une spécification peut être définie comme une description des comportements attendus d'un système. Dans le cycle de développement d'un logiciel, la spécification initiale est importante, car elle constitue la description de référence entre le client et le concepteur du logiciel. Elle permet de définir les attentes du client et de fixer les objectifs du concepteur. Les méthodes actuelles de conception des SI reposent généralement sur la modélisation du système avec plusieurs vues complémentaires. Les langages de spécification utilisés dans ces méthodes

ne sont généralement pas formels, autrement dit, leur sémantique n'est pas définie avec des mathématiques. Comme les vues peuvent décrire plusieurs fois les mêmes propriétés du système à réaliser, la cohérence du modèle est difficile à vérifier.

Les méthodes de spécification formelle sont utilisées en génie logiciel pour raisonner sur des modèles mathématiques. L'intérêt est de pouvoir prouver ou vérifier des propriétés sur ces modèles. Malgré les coûts supplémentaires liés au travail d'analyse et de vérification, l'utilisation de telles méthodes est de plus en plus justifiée pour des logiciels qui impliquent des données ou des conditions de sécurité critiques, car elles permettent d'assurer leur bon fonctionnement et évitent ainsi des risques d'erreur. Dans le domaine des SI, plusieurs approches ont été développées [Ngu98, MS99, DLCP00, LM00] afin de formaliser les diagrammes utilisés pour spécifier les SI, mais leur application reste toutefois marginale. Pourtant, nous sommes convaincu que l'utilisation de notations et de techniques formelles constitue une réponse efficace aux problèmes de validation et de vérification de la cohérence des modèles des SI. Notre approche se veut pragmatique et le développement d'outils d'assistance et de techniques de synthèse automatique de programmes fait partie de notre stratégie pour minimiser les efforts sur les phases comme le codage, qui peuvent demander beaucoup de ressources si elles ne sont pas automatisées, mais qui ne requièrent pas une grande expertise. Les phases d'analyse et de conception sont plus critiques et le succès du développement du système en dépend.

L'objectif d'une spécification est de décrire les propriétés d'un système. Il existe plusieurs paradigmes possibles, qui proposent des descriptions de natures différentes. En premier lieu, le vocabulaire est différent suivant le type de paradigme ; par exemple, les termes "opération", "événement" et "fonction" désignent des concepts à peu près équivalents. Les propriétés qui peuvent s'exprimer dans le langage diffèrent d'un paradigme à l'autre. Par exemple, une spécification de type algébrique, comme en CASL [ABK+02], permet d'exprimer les axiomes du système et les propriétés vérifiées par chaque fonction sous forme d'équations. Une spécification de type algèbre de processus, comme en CSP [Hoa85], décrit le comportement d'un système sous forme d'ordonnancement, d'entrelacement et de synchronisation entre les événements. Une spécification de type transition d'états, comme en B [Abr96], décrit d'une part les propriétés d'invariance du système et d'autre part les préconditions et postconditions de chaque opération. Il faut toutefois faire une distinction entre spécification et vérification. Même si une propriété n'est pas exprimée dans une spécification, cette dernière peut être satisfaite par le modèle représenté par la spécification.

Une des difficultés actuelles est de spécifier formellement le comportement des SI. Un des paradigmes les plus couramment utilisés est celui des transitions d'états. Une spécification consiste alors en un espace d'états défini par des variables d'état et en des opérations qui définissent les transitions d'états. À la réception d'un événement externe au SI, une opération est appelée afin de calculer le nouvel état du SI et éventuellement de produire une sortie. Plusieurs langages existent pour décrire des spécifications basées sur les transitions d'états, comme les diagrammes états-transitions, les machines à états ou bien les langages basés sur la théorie des ensembles comme Z [Spi92] ou B [Abr96]. Certaines propriétés, comme les contraintes d'ordonnancement des événements, sont toutefois moins naturelles à décrire en utilisant une approche basée sur

les transitions d'états. Les algèbres de processus par exemple permettent de spécifier directement ce type de propriétés, mais ne sont pas bien adaptées pour prendre en compte les principales caractéristiques des SI. Les approches basées sur les événements et les approches basées sur les transitions d'états semblent donc complémentaires pour modéliser les SI.

Combinaison de spécifications formelles

Deux méthodes nous semblent plus particulièrement adaptées pour spécifier formellement des SI : B et EB^3. La méthode B [Abr96] est à la fois un langage et une méthode de spécification formelle. Elle a l'avantage de traiter toutes les phases du cycle de conception, depuis l'analyse des besoins jusqu'à l'implémentation finale. Elle est de plus très bien outillée. Ces raisons nous ont conduit à considérer B plutôt que d'autres langages du même type comme VDM ou Z, qui ne couvrent pas tous ces aspects. Le langage B est basé sur la notion de machine abstraite qui est fondée sur les notions d'état et de propriétés d'invariance. Les outils associés à la méthode permettent d'une part de vérifier la correction des machines spécifiées et d'autre part de prouver des propriétés sur les spécifications obtenues.

La méthode EB^3 [FSD03] a été spécialement développée pour la spécification des SI. Elle s'inspire des approches basées sur les événements comme CSP, CCS ou LOTOS tout en simplifiant leur sémantique et leur utilisation. Le langage EB^3 est de plus orienté sur les traces d'événements ce qui rend les propriétés d'ordonnancement plus explicites. La méthode est également outillée, avec notamment un interpréteur [FF02] des expressions de processus EB^3. Si la méthode B permet de bien prendre en compte les structures de données d'un SI, elle est en revanche inadéquate pour considérer des propriétés temporelles ou d'ordonnancement des opérations. Si la méthode EB^3 est intéressante pour décrire et vérifier des propriétés dynamiques, il est en revanche plus difficile de prendre en compte les actions et les modifications d'un état particulier, ainsi que les propriétés d'invariance de l'espace d'états. Nous avons donc choisi de coupler ces deux langages formels.

Notre objectif a été d'utiliser les langages formels EB^3 et B de manière plus intégrée, afin de prendre en compte à la fois les propriétés statiques et dynamiques des SI. Dans ce but, nous nous sommes intéressé à une série d'exemples de combinaisons de spécifications formelles afin d'analyser les avantages, les inconvénients et les contraintes liés à ce type d'approches. Pour les comparer, nous avons utilisé un petit exemple de référence dont les opérations sont caractéristiques des SI. La combinaison de spécifications formelles peut apporter de nombreux avantages. Les modèles sont plus riches et permettent de mieux représenter les aspects statiques et dynamiques des systèmes. Les possibilités de vérification sont nombreuses. Cependant, le caractère formel peut aussi avoir ses inconvénients. L'intégration de plusieurs méthodes formelles est en effet difficile. Les spécifications peuvent d'une part être contradictoires ou redondantes. Elles sont d'autre part difficiles à analyser et à comprendre, car la définition sémantique de l'intégration pose souvent problème. Le principal problème à résoudre concerne le niveau d'intégration d'une approche par rapport à l'autre. Si deux langages sont peu intégrés, les possibilités d'analyse et de vérification seront limitées. Si, au contraire, les deux langages sont unifiés pour en créer un

nouveau, la principale difficulté sera la définition d'une nouvelle sémantique.

Méthode EB^4

Notre principal leitmotiv tout au long de cette thèse a été le suivant : pour modéliser un maximum d'exigences, une solution consiste à spécifier deux fois le même système dans deux langages différents et à prouver ensuite que les deux spécifications sont cohérentes. Dans la méthode que nous proposons, EB^4, le SI est dans un premier temps spécifié en EB^3. Ensuite, un raffinement est utilisé pour passer de la spécification EB^3 à une spécification en B. Enfin, la cohérence de l'ensemble du modèle est assurée en ajustant la spécification EB^3 en fonction des propriétés décrites en B.

Informellement, le raffinement permet de passer d'une spécification qui décrit ce que le système doit faire et satisfaire, à une description qui indique quels sont les algorithmes qui permettent de résoudre les problèmes et de respecter les exigences de la spécification initiale. Dans la pratique, il est souvent réalisé en plusieurs étapes de raffinement successives, qui sont chacune plus simple à valider qu'un seul et unique pas de raffinement. Une étape de raffinement consiste alors à réécrire une spécification donnée, de manière à la rendre plus déterministe et plus proche d'une implémentation. Le raffinement n'est pas automatisable en général, car des choix d'implémentation sont réalisés à chaque étape au niveau des variables d'état et des algorithmes utilisés. Enfin, une description obtenue par raffinement a la propriété de conserver le même comportement que la spécification raffinée.

Dans la méthode EB^4, la relation de raffinement définie dans la méthode B est utilisée pour raffiner la spécification initiale en EB^3 du SI en une spécification B. Cependant, la mise en œuvre d'un tel raffinement est difficile. En effet, une spécification EB^3 ne contient, par définition, qu'une seule variable d'état, à savoir la trace courante du système, alors que la description B peut contenir plusieurs variables d'état. En particulier, si on souhaite spécifier et vérifier par la suite des propriétés statiques sur les données, les variables d'état en B doivent être choisies de manière à représenter adéquatement le modèle de données du SI. Les différentes étapes de la méthode EB^4 permettent d'obtenir progressivement et systématiquement une grande partie du raffinement en B. Elles permettent également de compléter la spécification EB^3 en fonction des nouvelles propriétés statiques décrites et prouvées sur le modèle B. Une intervention humaine reste évidemment nécessaire sur quelques étapes critiques, comme la spécification initiale en EB^3 du SI, la preuve de propriétés d'ordonnancement qui ne font pas partie des propriétés type que nous avons identifiées et cataloguées, et la spécification des propriétés statiques à vérifier sur le modèle B.

Par conséquent, la méthode EB^4 permet de spécifier le SI avec deux vues orthogonales et complémentaires, en EB^3 et en B. La spécification EB^3 met en avant les propriétés d'ordonnancement du système, tandis que la spécification B permet de prendre en compte le modèle de données. Les différentes étapes de traduction et de raffinement définies dans la méthode ont pour objectif de faciliter les tâches de spécification et de vérification du concepteur.

Une fois que le SI a été spécifié, il existe deux alternatives pour implémenter le système. On peut utiliser les techniques de raffinement développées en B dans le cadre des SI [Mam02]. Dans cette thèse, nous avons choisi de définir

de nouvelles techniques de synthèse automatique, afin de générer automatiquement des transactions de bases de données relationnelles qui correspondent au modèle de données décrit en EB^3. Ce travail s'insère dans le cadre du projet APIS [FFLR02], qui a pour objectif de générer automatiquement des SI à partir de spécifications EB^3. La traduction a l'avantage d'être entièrement automatique. D'autre part, nous avons souhaité garder la possibilité d'utiliser les techniques de raffinement B pour les cas où le concepteur voudrait faire d'autres choix d'implémentation que ceux proposés dans la synthèse automatique. Ainsi, les techniques présentées dans cette thèse aboutissent à la définition d'une méthode complète pour la spécification des SI, qui peut être utilisée au sein du projet APIS ou bien indépendamment.

Contributions et plan de la thèse

Les principales contributions de cette thèse sont les suivantes :

1. **État de l'art détaillé des combinaisons de spécifications formelles transitions d'états/événements.** L'approche consistant à combiner des spécifications formelles de type transitions d'états et de type événements apporte une solution au problème de la modélisation du comportement dans les SI. Il existe de nombreux exemples dans la littérature, dans des domaines d'application autres que les SI, de combinaisons qui prennent en compte à la fois les propriétés statiques et dynamiques dans le processus de spécification. L'étude de l'état de l'art nous a permis d'estimer le niveau d'intégration nécessaire pour définir EB^4, notre méthode de spécification formelle des SI.
2. **Définition précise du langage d'attributs d'**EB^3**.** Une spécification EB^3 est composée de plusieurs parties. Les définitions d'attributs permettent de définir la dynamique des attributs du SI. Une des contributions de cette thèse a été de définir avec précision la syntaxe du langage de définition des attributs dans EB^3.
3. **Définition de règles de traduction vers B pour les attributs.** Dans le but d'utiliser les avantages du langage B concernant la spécification et la vérification de propriétés statiques des SI, nous avons défini des règles de traduction afin de générer des spécifications B à partir des définitions d'attributs EB^3.
4. **Vérification de contraintes d'intégrité statiques en B.** Grâce aux spécifications B obtenues à partir des définitions d'attributs EB^3, de nouvelles propriétés statiques peuvent être spécifiées et prouvées sur le modèle B. Dans cette thèse, nous avons défini plusieurs techniques de vérification de propriétés statiques exprimées en B. En particulier, des règles de réécriture et de simplification sont proposées afin de générer les préconditions des opérations.
5. **Définition d'une approche globale couplant les langages EB^3 et B.** Nous avons exploré différentes pistes afin de définir un processus de spécification intégrant EB^3 et B. Pour considérer le plus vite possible les aspects dynamiques, la spécification du système débute par la description en EB^3. Ensuite, la partie B permet de spécifier et de vérifier des

contraintes d'intégrité statiques sur les données du système. Dans cette thèse, plusieurs étapes ont été proposées afin de définir la méthode EB^4.

6. **Définition de règles de traduction pour obtenir des transactions de bases de données relationnelles qui correspondent aux définitions d'attributs EB^3.** L'objectif du projet APIS est de générer automatiquement des SI à partir de spécifications EB^3. Dans le cadre de l'approche EB^4, nous avons défini des règles de traduction afin de générer des transactions de bases de données relationnelles qui correspondent aux définitions d'attributs EB^3.

La thèse se décompose en trois parties. Dans la première partie, nous présentons le contexte et l'état de l'art. Le chapitre 1 introduit la problématique en présentant les SI et en discutant des difficultés liées à l'utilisation de langages formels pour spécifier les SI. Ensuite, le chapitre 2 présente les définitions et les principaux langages utilisés dans la suite de cette thèse. Les principales approches de combinaisons de spécifications de type transitions d'états avec des spécifications de type événements sont présentées et analysées dans le chapitre 3. Enfin, le chapitre 4 est une introduction au langage EB^3.

Dans la seconde partie de la thèse, nous nous intéressons à l'intégration des langages EB^3 et B. Le chapitre 5 constitue une introduction à cette problématique. Dans le chapitre 6, nous présentons nos algorithmes de traduction des définitions d'attributs en B. Le chapitre 7 est consacré à notre nouvelle méthode, EB^4, et à la description de ses principales étapes. Enfin, le chapitre 8 conclut cette partie avec des analyses et des perspectives concernant l'approche EB^4.

Dans la dernière partie, nous nous intéressons à la solution que nous proposons pour implémenter les transactions d'un SI spécifié avec la méthode EB^4 ou, plus généralement, avec le langage EB^3. Le chapitre 9 présente nos algorithmes de génération de transactions de bases de données relationnelles à partir des définitions d'attributs EB^3. Dans le chapitre 10, nous montrons comment générer des requêtes SQL qui retrouvent les tuples caractérisés par les prédicats du premier ordre utilisés dans les définitions d'attributs EB^3. Enfin, le chapitre 11 conclut cette partie avec la présentation d'un outil qui implémente les algorithmes des chapitres 9 et 10.

Première partie

Contexte et état de l'art

Chapitre 1

Systèmes d'information et méthodes formelles

> "Les problèmes de la vie réelle, ce sont ceux qui restent une fois que vous avez retiré toutes les solutions connues."
>
> — Edsger W. Dijkstra

Ce chapitre est une introduction aux SI et aux problèmes liés à la modélisation de tels systèmes avec des techniques formelles. Les chapitres suivants sont consacrés à l'état de l'art afin de situer notre problèmatique par rapport aux travaux existants. Dans le chapitre 2, nous introduisons les principales définitions qui nous serviront tout au long de cette thèse. Nous y présentons notamment les différents langages formels, ainsi que des concepts importants des méthodes formelles, comme la vérification, la preuve et le raffinement. Comme notre objectif est de coupler deux langages formels dont les paradigmes sont différents, le chapitre 3 présente un état de l'art sur les combinaisons de spécifications formelles de type transitions d'états avec des spécifications de type événementiel. Enfin, le chapitre 4 présente le langage EB^3, qui nous a servi de support pour définir la méthode EB^4.

1.1 Contexte scientifique

Nos intérêts portent sur la spécification formelle des systèmes d'information (SI). Pour bien comprendre l'utilité des techniques formelles dans le cadre de ces systèmes, nous devons dans un premier temps fixer la définition de ce que nous appelons des "systèmes d'information", car il n'existe pas de consensus dans la littérature concernant ce terme. De plus, nous avons besoin d'une définition précise des SI si on souhaite par la suite les modéliser avec des méthodes formelles.

Généralement, un SI est un système logiciel qui permet à une organisation de rassembler et de manipuler des informations. Cependant, les propriétés et les applications de ces systèmes sont nombreuses et il serait difficile de les considérer toutes. Au CEDRIC, Philippe Facon et Régine Laleau se sont intéressés aux SI qui sont développés autour de systèmes de gestion de bases de données (SGBD).

Nos travaux portent sur ce type de SI. Dans la section 1.2, nous introduisons cette définition des SI et nous présentons leurs principales caractéristiques.

Par définition, une notation est dite *formelle* si elle est fondée sur des modèles mathématiques. Un *langage formel* a une syntaxe et une sémantique formelles. Par abus de langage, une spécification écrite dans un langage formel est appelée une spécification formelle. Si un langage a seulement une syntaxe formelle, il est dit *semi-formel*. Une *méthode formelle* est une méthode de développement de logiciels qui s'appuie sur des techniques et sur des langages formels.

L'utilisation de méthodes formelles pour développer des SI [Ngu98, MS99, DLCP00, Mam02, FSD03] est justifiée par la sûreté ou par la valeur marchande des données manipulées par des organismes comme des banques, des compagnies d'assurance, ou bien des industries de haute technologie. Les principaux travaux de formalisation concernent la structuration des données du système ; ils permettent ainsi de coupler des notations graphiques, intuitives mais semi-formelles avec des spécifications formelles qui détaillent les imprécisions des représentations graphiques.

Néanmoins, la spécification de la dynamique des SI mérite aussi une attention particulière. Dans les bases de données, les transactions permettent de manipuler les données à travers des opérations de requête ou de mise à jour. L'ordonnancement des transactions peut jouer un rôle important dans la préservation des contraintes d'intégrité du SI. La section 1.3 présente les intérêts de la prise en compte du comportement dans la modélisation des SI. Plus généralement, nous indiquons aussi dans cette section quels types de propriétés nous intéressent dans le cadre des SI.

Nos travaux se basent enfin sur des approches existantes. Nous les présentons dans la section 1.4. En France, au sein de l'équipe CPR du CEDRIC, une sémantique B a tout d'abord été définie afin de formaliser des notations semi-formelles comme OMT ou UML. L'idée est de supprimer toute ambiguïté due à l'absence de sémantique formelle dans ce type de diagrammes. Ces travaux ont abouti à la création d'une nouvelle méthode, appelée UML-B-SQL, qui permet d'une part de générer des spécifications B à partir de diagrammes UML et d'autre part de raffiner la spécification obtenue en une implémentation Java/SQL. L'utilisation d'un formalisme comme B est intéressante notamment pour spécifier des propriétés statiques des SI.

De l'autre côté de l'Atlantique, l'équipe GRIL du Département d'informatique de l'Université de Sherbrooke, au Québec, a développé dans la même période un nouveau langage appelé EB^3 dans le but de spécifier des SI du même type que ceux étudiés par le CEDRIC. L'idée est de considérer les SI comme des boîtes noires et d'utiliser une approche événementielle pour décrire leur comportement. Le langage EB^3 est basé sur une algèbre de processus ; il est ainsi possible d'exprimer des propriétés dynamiques comme des propriétés d'ordonnancement sur les opérations du SI. Comme les deux approches semblaient complémentaires, le principal objectif de cette thèse a été de faire le lien entre elles afin de bénéficier à la fois des avantages des deux formes de modélisation.

1.2 Introduction aux SI

Un *système d'information* (SI) est un système informatisé qui rassemble l'ensemble des informations présentes au sein d'une organisation, sa mémoire,

et les activités qui permettent de les manipuler [Lal02]. Il est caractérisé par :

- l'utilisation de nombreuses données,
- des relations complexes entre les structures de données,
- des utilisateurs hétérogènes qui peuvent agir en concurrence,
- des opérations impliquant plusieurs structures de données, utilisant un volume important de données, tout en préservant l'intégrité des données modifiées,
- une communication adaptée des requêtes des utilisateurs et une gestion des messages d'erreur en cas d'appel invalide des opérations.

Une conséquence importante est le choix d'une structure de données adaptée dans le processus de modélisation des SI. En revanche, les algorithmes des opérations ne sont pas complexes.

1.2.1 Composantes des SI

Un SI est généralement constitué de trois parties :

1. une interface graphique,
2. une base de données,
3. un ensemble de transactions.

Si le concept d'interface graphique est assez courant, il est en revanche difficile de définir avec précision les notions de base de données et de transaction.

D'après la définition de Gardarin [Gar99], une *base de données* est un ensemble de données interrogeables modélisant les objets d'une partie du monde réel et qui sert de support à une application informatique. Par conséquent, la base de données n'a d'intérêt que si elle peut être interrogée et modifiée.

Une *transaction* est une application sur la base de données qui permet d'y faire des interrogations ou des mises à jour. Elle vérifie en général les propriétés appelées ACID (*Atomicity, Consistency, Isolation, Durability*) [EN04] :

- une transaction est atomique, car elle est exécutée dans son ensemble ou pas du tout,
- la base de données doit rester cohérente après l'exécution d'une transaction,
- chaque transaction est indépendante et n'interfère pas avec les autres transactions,
- et les modifications d'une transaction qui a été exécutée perdurent même en cas de panne de la base de données.

Les livres de Gardarin [Gar99], Ullman [Ull88] et Elmasri [EN04] sont des références standard concernant les fondements et les principes des bases de données.

1.2.2 Conception des SI

Les méthodes de conception des SI peuvent être classées en trois catégories : les méthodes fonctionnelles, les méthodes systémiques et les méthodes orientées objet. Ces méthodes se fondent sur des vues et des concepts différents pour modéliser le SI.

Méthodes fonctionnelles. Les méthodes fonctionnelles s'appuient sur les techniques de décomposition cartésienne et sur les représentations des flots de données. L'approche est dite fonctionnelle, car elle identifie un SI à une fonction globale de gestion qui est ensuite décomposée en des fonctions plus détaillées, et ainsi de suite. Le modèle du SI est représenté à l'aide de diagrammes de flots de données [YC79] et de diagrammes de structure. Les méthodes de conception les plus connues appartenant à cette catégorie sont la méthodologie Gane et Sarson [GS79] et SADT [MM88]. Ces méthodes de la première génération (années 60) ne reposent pas sur des fondements théoriques et les modélisations ne sont pas formelles.

Méthodes systémiques. Les méthodes systémiques se focalisent sur la modélisation des données. Les modèles permettent de représenter à la fois les informations du système et les relations entre elles. Il existe plusieurs niveaux d'abstraction dans ces méthodes, depuis l'analyse du système, en passant par les niveaux conceptuel, logique et physique. La modélisation la plus abstraite des données et des traitements sur ces données est réalisée par des schémas conceptuels de données et de traitements. La modélisation conceptuelle des données repose généralement sur le modèle relationnel et sur le modèle entité-association. Les modèles de traitement dépendent des méthodes de conception. Parmi les plus connues, on peut citer : Merise [TRC83], Remora [RFB88] et Axial [Pel86]. Les modèles utilisés dans ces approches comme les entités-associations ne sont pas formels. Certaines méthodes ont toutefois étendu les concepts pour les rendre formels.

Méthodes orientées objet. Les méthodes orientées objet modélisent les SI autour d'entités appelées des classes qui représentent un état et qui définissent des opérations (ou méthodes). Les classes sont reliées entre elles par des associations. Une conception orientée objet est constituée de trois modèles. Le modèle statique permet de représenter les classes et leurs associations. Le modèle dynamique représente le comportement de chaque type d'objet. Enfin, le modèle d'interaction permet de représenter les flux de messages entre objets. Les approches appartenant à cette catégorie sont par exemple : OOD [Boo94], OOSE [Jac94], OMT [RBP+91] et UML [Obj06]. Les notations graphiques utilisées dans ces approches constituent à la fois un avantage et un inconvénient. Elles permettent une meilleure compréhension des modèles mais leur sémantique n'est pas précise. Les langages graphiques sont en effet considérés comme *semi*-formels.

S'il existe des méthodes pour concevoir le modèle de données en s'appuyant sur des langages formels (voir section 1.4), la spécification du comportement fonctionnel du SI reste en revanche, dans la plupart des cas, informelle.

1.3 Propriétés des SI

La modélisation du comportement reste un défi important dans l'ingénierie des SI. Le comportement fonctionnel d'un SI est essentiellement défini par des transactions sur les bases de données : transaction d'interrogation pour interroger la base de données et transaction de mise à jour pour modifier cette base. La cohérence des données est garantie par des contraintes d'intégrité de deux

types : les contraintes statiques, qui imposent des restrictions sur les données de la base, et des contraintes dynamiques, qui imposent des restrictions sur le cycle de vie des données.

Dans les bases de données actives [WC96], le comportement peut être modélisé par des règles de transformation sur les structures de données. Ces règles actives, qui sont des extensions des déclencheurs (*triggers*), permettent de décrire les réactions du système lorsqu'un événement se produit. Elles sont de la forme événement-condition-action (ECA). Lorsqu'un événement E survient, si la condition C est vérifiée, alors l'action A est exécutée. Cette approche reste toutefois limitée, car elle est difficile à mettre en œuvre sur l'ensemble des transactions d'une base de données.

En outre, la plupart des méthodes de conception actuelles des SI ne considèrent pas la définition des transactions au niveau de l'analyse, mais plutôt dans les phases successives du développement des SI.

1.3.1 Quels types de propriétés ?

On peut distinguer plusieurs types de propriétés à spécifier ou à vérifier sur les SI.

Les propriétés *statiques* permettent d'assurer la cohérence du système et l'intégrité des informations : elles s'expriment par l'intermédiaire de contraintes d'intégrité dans les bases de données. Dans les approches formelles basées sur les transitions d'états, les contraintes d'intégrité sont spécifiées sous la forme d'invariants ou bien de gardes sur les opérations.

Une propriété est dite *dynamique* si elle traite de l'occurrence et de l'ordonnancement des événements. Cette définition comprend aussi bien les propriétés de sécurité et de vivacité, que les propriétés d'ordonnancement du type "a suivi d'un nombre arbitraire de b et suivi d'un c".

Une propriété de *sûreté* est une propriété de la forme : "quelque chose de mauvais n'arrive jamais". Dans le cas de méthodes formelles basées sur les transitions d'états, un invariant constitue une propriété de sûreté pour le système. Dans le cas de spécifications exprimées sous forme de traces d'événements, une trace valide du système est une trace qui respecte les propriétés de sûreté.

Une propriété de *vivacité* est une propriété de la forme : "quelque chose de bien arrivera nécessairement". Ce type de propriétés se vérifie en analysant toutes les traces valides ou bien l'ensemble de toutes les séquences possibles de transitions d'états.

Enfin, il faut bien cerner le mode d'expression des propriétés du comportement d'un système. À la limite, tous les formalismes permettent d'exprimer une propriété dynamique, car on peut considérer la propriété comme étant un système, et la spécifier avec ce formalisme. Les langages basés sur les événements (par exemple, les logiques temporelles, les algèbres de processus, les automates, les expressions régulières et les grammaires) sont souvent bien adaptés pour spécifier de manière explicite les propriétés dynamiques. Toutefois, certaines propriétés dynamiques sont plus faciles à spécifier avec un invariant sur l'espace d'états. Par exemple, spécifier que deux emprunteurs ne peuvent emprunter un livre en même temps s'exprime plus facilement par un invariant (du genre la variable emprunteur est une fonction de livre vers membre) que par une expression de processus. En fait, on pourrait considérer un invariant sur l'espace d'états comme un cas particulier de propriété dynamique.

Pour définir la notion de propriété de manière plus précise, on peut utiliser la notion de système de transitions sur un espace d'états défini par des variables. Dans la suite, les systèmes de transitions étiquetées (ou LTS) nous serviront à représenter ou à expliquer nos exemples.

1.3.2 Et les spécifications formelles ?

Les principales techniques utilisées [AMF00] pour modéliser le comportement dans les SI, comme les réseaux de Petri, les modèles entités-associations étendus ou les approches orientées objet, sont limitées. En outre, elles sont généralement appliquées une fois que le système est défini, pour vérifier que les propriétés voulues sont satisfaites. Pour assurer que ces propriétés restent vraies dans le temps, certaines méthodes de conception proposent aussi la définition de règles de déclenchement (*triggers*) qui empêchent ou modifient les demandes de mises à jour qui pourraient violer les contraintes d'intégrité. Toutes ces stratégies sont toutefois plutôt défensives, ce qui limite les possibilités du système.

Dans le cas des réseaux de Petri, le comportement du système ne peut être que partiellement décrit et simulé. Un réseau de Petri [Pet81] est un graphe biparti composé d'une part de places qui représentent les variables logiques du système, et d'autre part de transitions qui définissent les transitions ou les actions du système. Dans la modélisation d'une règle de type ECA (voir l'introduction de la section 1.3), les places du réseau peuvent être utilisées pour représenter les événements et les conditions, tandis que les transitions représentent les actions. La complexité des réseaux de Petri limite néanmoins leur utilisation et cette technique est plutôt utilisée pour modéliser une partie seulement des transactions.

Le modèle entités-associations permet de modéliser les contraintes d'intégrité d'un SI. Dans certaines approches, il est étendu afin de représenter des transitions ou des règles actives. Par exemple, le modèle ER^2 (entités-associations-événements-règles) [Tan92] permet de considérer les événements et les règles, qui sont représentés comme des objets. Un réseau de Petri coloré [Jen96], c'est-à-dire un réseau de Petri dont les places sont paramétrées, est utilisé pour modéliser les flots de contrôle entre processus. Si ce modèle permet de représenter facilement les relations entre règles et objets, il ne permet pas de représenter des règles autres que celles concernant les opérations sur les données.

Dans les méthodes orientées objet, les transactions sont modélisées grâce à des diagrammes états-transitions. Ils décrivent le comportement en termes d'états et de transitions sur ces états. Les effets d'une règle de transformation sur les objets ne sont pas toujours visibles avec cette approche. De plus, elles utilisent un formalisme graphique, semi-formel.

En conclusion, les approches actuelles ne privilégient pas l'aspect formel pour spécifier les propriétés dynamiques. De plus, il est souvent difficile d'intégrer rapidement les aspects statiques et dynamiques lors du processus de développement des SI. Les spécifications concernant les propriétés dynamiques peuvent en effet être en contradiction avec les structures de données. Comme les aspects statiques sont déjà spécifiés, les contraintes dynamiques sont dépendantes de la modélisation de la statique. Dans le cas d'approches non formelles ou semi-formelles, cette intégration tardive des aspects dynamiques peut être une source non négligeable d'erreurs.

1.4 Deux exemples de spécification formelle des SI

Pour mieux comprendre les intérêts de l'utilisation des approches formelles pour spécifier les SI, nous présentons maintenant deux exemples de méthodes qui ont été développées dans cette optique : UML-B-SQL et EB^3. L'approche UML-B-SQL permet de formaliser les descriptions semi-formelles des diagrammes de classes UML en utilisant le langage de spécification formel B. Les propriétés dynamiques du SI modélisé ne sont pas facilement exprimables par cette approche. Le langage EB^3, qui a été créé pour spécifier des SI, permet de bien spécifier les traces des actions des différentes entités du système. Les propriétés concernant les états et les données sont en revanche plus difficiles à vérifier.

1.4.1 UML-B-SQL

L'approche UML-B-SQL [Ngu98, Mam02] permet de spécifier un SI à partir de diagrammes UML qui sont ensuite traduits en B pour rendre la spécification formelle. Le langage B sera présenté en détail dans le chapitre 2. La spécification obtenue peut ensuite être raffinée pour obtenir du code Java/SQL.

Le langage UML. Le langage UML [Obj06] est un langage graphique orienté objet, issu de l'unification de plusieurs méthodes. La description graphique de UML représente un sérieux avantage, car les diagrammes sont plus faciles à appréhender pour le concepteur, et c'est la raison pour laquelle les approches graphiques sont couramment utilisées dans l'industrie. Toutefois, un langage comme UML est considéré comme *semi*-formel, car la sémantique des notations graphiques utilisées n'est pas précise.

Le langage UML offre plusieurs sortes de diagrammes afin de couvrir les différentes étapes du processus de développement d'un système, depuis l'analyse des besoins jusqu'à l'implémentation. Dans l'approche UML-B-SQL, les diagrammes suivants sont utilisés :

- le diagramme de classes représente les aspects statiques et structurels,
- les diagrammes d'états-transitions permettent de décrire le comportement des objets d'une classe donnée,
- enfin, les diagrammes de collaborations représentent les fonctionnalités du système.

Si la description d'un système selon plusieurs vues facilite la compréhension du fonctionnement pour l'utilisateur, elle peut également rendre la spécification globale incohérente. Comme les diagrammes ne sont pas formels, il n'est pas possible de faire des vérifications. L'utilisation de règles de traduction pour définir chaque notation graphique en une notation mathématique permet d'obtenir une spécification formelle du système et de réaliser des preuves ou des vérifications de propriétés.

Spécification UML-B. La spécification en UML et B du SI est réalisée de la manière suivante :

- La première étape consiste à décrire les aspects statiques du SI avec un diagramme de classes UML. Les contraintes qui ne peuvent pas être ex-

primées de façon précise avec les notations graphiques sont spécifiées en B.
- Les règles de traduction définies dans [Ngu98, Mam02] permettent ensuite de générer des spécifications B à partir du diagramme de classes établi lors de la première étape.
- L'étape suivante consiste à décrire les transactions du système à l'aide des diagrammes d'états-transitions et de collaborations. Les expressions B de la première étape sont alors utilisées pour annoter ces nouveaux diagrammes et ainsi assurer leur traduction automatique. Cette étape peut nécessiter l'ajout de nouveaux attributs dans le diagramme de classes et par conséquent la réitération des deux premières étapes du processus.
- Les spécifications B correspondant aux diagrammes d'états-transitions et de collaborations sont ensuite générées automatiquement par un outil.
- La méthode permet de générer des obligations de preuves afin d'assurer la satisfaction des propriétés d'invariance (propriétés statiques).

Conclusion. La méthode de spécification UML-B-SQL permet ainsi d'obtenir une spécification formelle en B d'un SI. Seules les propriétés statiques du système sont effectivement prises en compte par cette méthode, car l'utilisation de B ne permet pas la spécification de propriétés dynamiques, telles que les contraintes temporelles ou les propriétés de vivacité.

1.4.2 EB^3

Comme le langage et la méthodologie autour d'EB^3 seront détaillés dans le chapitre 4, cette section ne décrit que brièvement EB^3 afin d'illustrer l'intérêt de l'approche et la comparer à la méthode UML-B-SQL. EB^3 (qui est le sigle de "Entity-Based Black Box") [FSD03] est à la fois un langage formel et une méthode de spécification dédiés à la modélisation des SI.

Une méthode dédiée aux SI. Le langage EB^3 est un langage formel de spécification qui est fondé sur la notion de trace, sur les algèbres de processus et sur la notion d'entité de la méthode JSD [Jac83]. En EB^3, les séquences possibles des événements du SI, appelées *traces*, sont décrites grâce à une algèbre de processus. La syntaxe des expressions de processus emprunte les principaux opérateurs de CSP [Hoa85], CCS [Mil89] et LOTOS [BB87], mais leur sémantique a été adaptée et des concepts comme l'action interne ou le non-déterminisme n'ont pas été pris en compte en EB^3, car ils ne sont pas utiles dans le cadre des SI.

Pour des raisons historiques, EB^3 a repris une partie des termes utilisés dans JSD. Ainsi, les notions classiques de classe et d'objet des modèles orientés objet sont appelées respectivement en EB^3 des *types d'entité* et des *entités*. Les méthodes de chaque type d'entité et association constituent les *actions* du SI. Les expressions de processus permettent de décrire les traces associées à chaque type d'entité et association.

Le langage EB^3 a été créé dans le cadre du projet APIS [FFLR02], qui a pour objectif de générer des SI directement à partir de spécifications formelles. Pour ce faire, plusieurs outils ont été développés. Un interpréteur des expressions de processus, appelé EB^{3}PAI, a notamment été implémenté [FF02]. Les objectifs et les réalisations du projet APIS sont détaillés dans le chapitre 4.

Conclusion. La méthode EB^3 est formelle et modulaire. Elle est de plus basée sur les traces d'événements, ce qui permet de prendre en compte certaines spécificités des SI. Comme elle est basée sur les traces, elle ne permet pas de bien représenter les conséquences d'une action sur un état particulier du système. En particulier, les propriétés fonctionnelles sont difficiles à vérifier avec cette approche.

1.4.3 Discussion

L'exemple de ces deux approches de spécification formelle montre que les langages formels ne sont pas toujours adaptés pour prendre en compte à la fois les propriétés statiques et dynamiques d'un système. D'une part, les approches de spécification basées sur les transitions d'états, comme B, permettent de décrire facilement certaines propriétés spécifiques aux SI, comme les relations complexes entre les larges structures de données du système, mais elles rendent aussi la vérification des propriétés d'ordonnancement des événements difficile. D'autre part, si la spécification des contraintes d'ordonnancement est plus facile en utilisant des spécifications basées sur les événements comme EB^3, les caractéristiques des SI sont au contraire plus difficiles à prendre en compte. Ces deux approches de spécification semblent donc complémentaires pour modéliser les SI.

Le choix de B par rapport à des approches comme VDM ou Z est justifié par une méthode de spécification complète et outillée qui couvre tout le processus de conception du système. La méthode UML-B-SQL montre également que B est adapté pour formaliser des diagrammes semi-formels UML qui sont couramment utilisés dans les méthodes de conception industrielles des SI. La méthode EB^3 est historiquement une méthode dédiée à la spécification formelle des SI puisqu'elle a été créée dans ce but. Chaque méthode peut être considérée, indépendamment l'une de l'autre, comme une bonne approche pour concevoir de manière formelle des SI. Les outils associés à B et la grande souplesse qui entoure les possibilités d'évolution d'EB^3 sont deux autres atouts indéniables. Pour ces raisons, les langages EB^3 et B semblent être des candidats naturels à une intégration afin de prendre en compte le maximum de propriétés caractéristiques des SI.

Chapitre 2

Langages formels et applications

> "L'ouïe de l'oie de Louis a ouï. Ah oui ? Et qu'a ouï l'ouïe de l'oie de Louis ? Elle a ouï ce que toute oie oit ! "
>
> — Raymond Devos

Pour améliorer la compréhension des nombreuses notions utilisées dans la thèse, ce chapitre est consacré aux principaux langages formels de spécification qui nous intéressent (sections 2.2 et 2.3). Nous présentons également les concepts de sémantique (section 2.1.2), de vérification (section 2.1.3) et de raffinement (section 2.1.4). Un glossaire des principaux termes utilisés est donné en annexe A.

2.1 Brefs rappels sur les méthodes formelles

Nous rappelons qu'une méthode de spécification est dite formelle lorsqu'elle utilise un ou plusieurs langages formels de spécification.

2.1.1 Langages basés sur les états et sur les événements

Il est possible de classer les méthodes de spécification formelles en deux groupes :

- les approches basées sur les états,
- et celles basées sur les événements.

Les méthodes basées sur les états représentent le système à travers deux modèles complémentaires : la partie statique permet de décrire les entités constituant le système et leurs états, tandis que la partie dynamique modélise les changements d'états que le système peut effectuer par l'intermédiaire d'opérations ou d'actions. Des propriétés d'invariance sont souvent définies sur le système pour assurer la cohérence du système. Les langages s'appuyant sur les états sont par exemple : Statechart [Har87], Esterel [BG92], ASM [Gur93], Action Systems [BKS83], VDM [Jon90], Z [Spi92], Object-Z [Smi00], B [Abr96] et B événementiel [AM98]. Nous illustrons dans la section 2.2 les approches Action Systems, Z, Object-Z, B et B événementiel.

Les approches basées sur les événements représentent le système à travers des processus ou des agents, qui sont des entités indépendantes qui communiquent entre elles ou avec l'extérieur du système. Ces méthodes permettent de modéliser le comportement du système à l'aide de séquences ou d'arbres d'événements. Parmi les exemples de formalismes basés sur les événements, on peut citer : les réseaux de Petri [Pet81], LOTOS [BB87], CCS [Mil89], CSP [Hoa85] et EB^3 [FSD03]. Dans la section 2.3, nous présentons le langage CSP.

Il existe également des approches différentes ou plus hybrides. Il y a par exemple les approches algébriques comme CASL [ABK+02] ou Larch [GH93], ou bien des méthodes combinant plusieurs aspects comme RAISE [Geo91] ou LOTOS [BB87]. Dans la suite, nous nous concentrerons sur les méthodes basées sur les états et sur les événements.

2.1.2 Sémantique

La *sémantique* d'un langage est la définition de ce que les expressions de ce langage signifient. Elle permet en effet d'interpréter dans un modèle les symboles de la syntaxe utilisée.

Une approche possible consiste à définir le sens des expressions d'un langage en décrivant comment elles seraient exécutées. Cette approche de la sémantique est dite opérationnelle. Les systèmes de transitions étiquetées permettent ainsi de représenter les états et les transitions d'états possibles d'un système.

Les langages de type algèbre de processus comme CSP sont plutôt représentés par l'observation de leur comportement. Cette approche de la sémantique est alors dite dénotationnelle. Elle permet de relier un programme à son comportement.

Système de transitions étiquetées. Les systèmes de transitions étiquetées ou *labelled transition systems* (LTS) [Mil90] permettent de modéliser le comportement de systèmes basés sur les états lors de leur exécution.

Le système de transitions étiquetées $(A, S, \longrightarrow, R)$ d'un système P est défini par la donnée de quatre éléments :

- A est l'alphabet, c'est-à-dire l'ensemble des événements possibles du système,
- S est l'ensemble des états possibles,
- $\longrightarrow$ est la relation de transition d'états, avec $\longrightarrow \subseteq S \times A \times S$,
- et R est l'ensemble des états initiaux, avec $R \subseteq S$ et $R \neq \varnothing$.

La relation $\longrightarrow$ est définie par un ensemble de règles d'inférence, de la forme :

$$\frac{H_1 \ \ldots \ H_n}{G}$$

où les H_i sont des hypothèses et G est la conclusion. Ces règles permettent de définir le comportement de chaque opérateur de la syntaxe du langage.

Le meilleur moyen de représenter un LTS est un diagramme. La figure 2.1 est un exemple de LTS. Les cercles représentent les états du système tandis que que les flèches modélisent les transitions d'états. L'état initial est marqué par le symbole ">". Dans cet exemple, l'alphabet du LTS est :

$$A = \{B, C, D, E, F, G, H, I, J, K, L\}$$

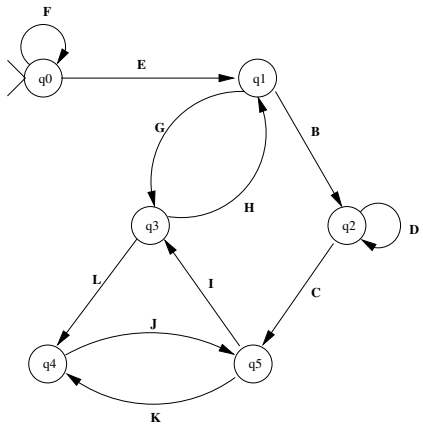

FIG. 2.1 – Exemple de LTS

L'ensemble des états possibles est :

$$S = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

L'ensemble des états initiaux est :

$$R = \{q_0\}$$

Dans l'exemple de la figure 2.1, on a bien :

$$(q_0, E, q_1) \in \longrightarrow$$

La transition (q_0, E, q_1) est aussi dénotée par :

$$q_0 \xrightarrow{E} q_1$$

Chaque transition $q \xrightarrow{action} r$ de $\longrightarrow$ est représentée par une flèche dans le diagramme.

Comme le LTS permet de représenter aisément le comportement d'un système, il sera utilisé dans les chapitres suivants pour améliorer la compréhension du lecteur.

Sémantique dénotationnelle. Les modèles sémantiques présentés dans ce paragraphe constituent les modèles de base [Ros97] usuellement utilisés dans la communauté des algèbres de processus pour représenter les langages à processus comme CSP. La sémantique dénotationnelle d'un processus est un ensemble d'observations.

Le modèle le plus simple est celui des traces : un processus P est représenté par $traces(P)$, l'ensemble de toutes les séquences d'événements possibles que ce processus peut exécuter. Le *modèle des traces* identifie un processus avec l'ensemble de ses traces.

Le *modèle des échecs stables* identifie un processus avec l'ensemble de ses traces et l'ensemble de ses échecs stables. Un *échec stable* du processus P est

une paire trace/refus (tr, X) où tr est une séquence d'événements que P peut exécuter en rejoignant un état stable et X est l'ensemble des événements que P peut refuser d'exécuter à partir de cet état stable. L'ensemble des échecs stables d'un processus P est dénoté par $\mathit{failures}(P)$. Si tr est une trace de P telle que $(tr, \Sigma) \in \mathit{failures}(P)$, où Σ est l'ensemble de tous les événements, alors P atteint un état dans lequel aucun événement n'est exécutable et dans ce cas, P est bloqué. Un processus est dit *libre de blocage* s'il n'existe aucun ensemble des échecs stables de la sorte.

Un autre modèle permet de prendre en compte les divergences d'un processus. Un *état divergent* est un état du processus dans lequel il est possible d'exécuter une infinité d'événements internes[1]. Une *divergence* du processus P est une séquence d'événements tr telle que P atteint un état divergent après avoir exécuté tr. L'ensemble des divergences d'un processus P est dénoté par $\mathit{divergences}(P)$. Un processus est dit *libre de divergence* si l'ensemble de ses divergences est vide. Un *échec* du processus P est une paire trace/refus. Le *modèle des échecs-divergences* identifie un processus avec l'ensemble de ses traces, l'ensemble de ses divergences et l'ensemble de ses échecs.

2.1.3 Vérification

Outre la possibilité de décrire des systèmes de façon précise, sans ambiguïté aucune, les sémantiques formelles permettent également de pouvoir raisonner sur des modèles mathématiques. Cet aspect est important dans le cadre de la vérification. En génie logiciel, la *vérification* est l'étape qui permet de s'assurer que le logiciel implémente correctement certaines propriétés. Elle se distingue de la *validation* qui, de son côté, permet de vérifier que le produit obtenu répond aux exigences du cahier des charges. Par conséquent, la vérification garantit que le logiciel est développé correctement, alors que la validation assure que le logiciel correspond au produit souhaité.

Il existe principalement deux pistes pour vérifier une propriété avec des méthodes formelles : le *model-checking* et la preuve. Le *model-checking*, parfois appelé aussi vérification, consiste à parcourir exhaustivement l'espace d'états d'un modèle fini du système à vérifier. Les propriétés à vérifier sont généralement exprimées en logique temporelle. Un contre-exemple est généré lorsque la propriété n'est pas vérifiée par le modèle considéré. La difficulté consiste essentiellement à définir un modèle fini assez représentatif d'un système qui a priori n'a aucune raison d'être fini. Un *model-checker* est le terme qui désigne un outil appliquant une méthode de *model-checking*.

La preuve (ou *theorem proving*) désigne un ensemble de techniques qui permettent de prouver des propriétés à partir des axiomes du système et d'un ensemble de règles d'inférence. Des lemmes intermédiaires peuvent être définis et prouvés afin de décomposer la preuve d'une propriété complexe. La preuve peut être manuelle ou automatisée. Cependant, ce type de vérification demande souvent une intervention humaine lorsque des patrons de preuve (souvent appelés des stratégies) ne sont plus capables de résoudre les calculs d'inférence. Le principal avantage de la preuve est son caractère universel puisqu'elle permet de considérer tous les cas possibles. Ainsi, lorsqu'une propriété est prouvée, on est assuré de sa correction, pourvu que le modèle soit valide.

[1] Les événements internes sont des événements qu'un processus peut exécuter mais qui ne sont pas observables par d'autres processus.

2.1.4 Raffinement

Pour comprendre la notion de raffinement, nous devons tout d'abord introduire le concept de *correction* d'un système par rapport à une spécification. Intuitivement, un système logiciel est "correct" lorsqu'il est conforme à sa spécification. Pour formaliser la notion de correction, on a besoin d'une représentation du logiciel afin de pouvoir la comparer avec sa spécification initiale. On appelle *implémentation* une description qui peut être traduite automatiquement en un programme exécutable sur un ordinateur. D'après la logique de Hoare [Hoa69], la spécification d'une expression quelconque S d'une implémentation peut être définie par une précondition P et une postcondition Q. La précondition P permet de caractériser les états possibles avant l'exécution de S, tandis que la postcondition Q est vérifiée par les états possibles après l'exécution de S. Dans ce cas, une implémentation est *totalement correcte* vis-à-vis de sa spécification si, à partir de tout état initial vérifiant la précondition, l'implémentation termine et fait passer le système dans un état satisfaisant la postcondition. Si la terminaison n'est pas assurée, la correction est alors seulement *partielle*.

Comme il est difficile de vérifier directement si une implémentation est correcte vis-à-vis de sa spécification, le raffinement est une activité qui permet de construire progressivement des implémentations correctes. Il consiste à dériver par étapes successives une spécification initiale, en vérifiant que chaque transformation préserve la correction vis-à-vis de la spécification précédente. Il existe, suivant les langages utilisés, de nombreuses notions de raffinement qui ne sont pas toujours équivalentes. La méthode la plus courante est la recherche d'une relation de simulation entre la spécification concrète et la spécification abstraite. Elle est inspirée de techniques classiques de vérification en sémantique opérationnelle [Mil89], qui consistent à relier les espaces d'états de deux LTS par des relations de simulation (*e.g.,* s'il existe une relation dans chaque sens, on parle alors de *bisimulation*). Dans le cadre du raffinement, les relations de simulation dépendent de la sémantique donnée aux spécifications et aux implémentations.

La notion de raffinement a été tout d'abord introduite par Wirth [Wir71] et Dijkstra [Dij76] dans les années 1970, puis formalisée par Back [Bac78, Bac88] dans les années 1980. Plusieurs travaux ont ensuite développé cette notion, en particulier Abadi et Lamport [AL88], Back [BvW98], Morgan [Mor88, Mor90], Morris [Mor87] et Abrial [Abr96].

2.2 Langages basés sur les états

Quatre approches basées sur les états sont présentées dans les sections suivantes : Action Systems, Z, Object-Z, B et B événementiel.

2.2.1 Action Systems

Les Action Systems sont basés sur les systèmes de transitions (qui sont des LTS sans étiquette). Plusieurs versions existent, comme celles de Back [BKS83] ou UNITY [MC88]. Nous avons pris comme convention dans cette thèse d'utiliser l'expression "Action Systems" pour se référer aux travaux de Back. Les

systèmes qui de manière générale s'inspirent des travaux de Back sont désignés par *systèmes d'actions*.

Un système d'actions est composé :

- d'un espace d'états, défini par des variables d'états,
- des actions qui définissent une initialisation et des transitions sur ces variables d'états.

Dans l'approche Action Systems de Back [BKS83], les actions sont décrites par le langage de commandes gardées de Dijkstra [Dij76].

Le langage de Dijkstra est associé à un calcul de plus faible précondition. Dans ce langage, les programmes agissent sur des variables, sont séquentiels et terminent. Le langage de commandes gardées de Dijkstra comprend notamment des affectations de variables, des compositions séquentielles, des conditionnelles de type **IF THEN ELSE END** et des boucles **DO END**. Le comportement d'un programme est spécifié avec une condition (appelée *précondition*) sur les valeurs des variables avant l'exécution du programme et une condition sur ces valeurs après l'exécution du programme (*postcondition*).

Afin d'assurer la terminaison d'un programme, Dijkstra a défini une sémantique de plus faible précondition qui introduit l'opérateur wp. Si S est une commande et $post$ est une postcondition de S, alors :

$$wp(S, post)$$

représente tous les états initiaux à partir desquels il est assuré d'atteindre la postcondition $post$ en exécutant S. Ainsi S satisfait les conditions $(pre, post)$ si :

$$pre \Rightarrow wp(S, post)$$

Plusieurs travaux sont basés sur le langage de commandes gardées de Dijkstra. Le calcul de raffinement introduit par Back [BvW98] est notamment une extension du calcul de plus faible précondition de Dijkstra. Les raffinements de données de Z et B sont également inspirés des travaux de Dijkstra et de Back.

Les actions dans les Action Systems de Back sont de la forme :

$$g \longrightarrow com$$

où g est une garde représentant des contraintes sur les variables d'états et com est une commande ou un appel de programme.

Une action $g \longrightarrow com$ est exécutable dans tout état satisfaisant la garde g. Le système d'actions commence par exécuter l'initialisation. Ensuite, les actions du programme sont analysées par l'évaluation de leur garde et une action est choisie parmi les actions exécutables pour être exécutée. Le système termine lorsqu'il atteint un état dans lequel plus aucune action n'est exécutable. Le système peut diverger si l'initialisation ou bien une des actions échoue.

Si le système termine, le programme est de la forme :

$$I\,;\ \textbf{do}\ A_0 \square A_1 \square ...\ \textbf{od}$$

où I désigne la commande d'initialisation et les A_i représentent les actions du système. Le symbole $\square$ est un choix non déterministe des actions. L'expression **DO OD** est un pseudo-langage pour désigner une boucle de type **WHILE** true **DO END**.

Une contrainte des Action Systems de Back est la terminaison du système uniquement lorsqu'il n'y a plus d'action exécutable. Une variante possible est fournie par UNITY [MC88] : les actions sont déterministes, n'échouent pas et sont toujours exécutables. Le système peut donc terminer à tout moment. Les propriétés des systèmes d'actions sont souvent exprimées à l'aide de la logique temporelle [Pnu77].

2.2.2 Langage Z

Le langage Z est basé sur la théorie des ensembles et sur la logique du premier ordre. [Spi92] est un manuel de référence complet sur le langage Z. Le *schéma* est la notion de base des spécifications Z. Un schéma est une boîte contenant des descriptions utilisant les notations Z. Les schémas sont utilisés pour décrire les états d'un système, les états initiaux ou bien les opérations.

Statique. La partie statique permet de définir les états et les relations d'invariant qui sont préservées lors des transitions d'états. Elle est décrite en Z sous la forme d'un schéma d'état.

Le *Birthday Book* est un exemple connu de système qui permet de retenir les dates d'anniversaire [Spi92]. Les types de base de ce système sont : $[NAME, DATE]$. Le schéma *BirthdayBook*, qui permet de définir les aspects statiques du système, est dénoté en Z par :

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}\, birthday$$

La première partie du schéma correspond à la déclaration des variables *known* et *birthday*. La seconde partie est la définition de l'invariant. La variable *known* est ici égale au domaine de la variable *birthday*.

Dynamique. Les aspects dynamiques concernent les opérations, les relations entre les entrées et les sorties, et les changements d'états.

Dans l'exemple précédent, l'opération *AddBirthday* permet de rajouter une date d'anniversaire dans le système :

AddBirthday

$$\Delta BirthdayBook$$
$$n? : NAME$$
$$d? : DATE$$
$$n? \notin known$$
$$birthday' = birthday \cup \{n? \mapsto d?\}$$

La notation Δ indique que l'état du schéma *BirthdayBook* sera modifié par cette opération. La notation ? signifie que les variables $n?$ et $d?$ sont des paramètres d'entrée de l'opération. Le prédicat $n? \notin known$ est la précondition de l'opération. La notation $birthday'$ indique un changement d'état de la variable

birthday par exécution de l'opération. En l'occurrence, l'opération a pour effet de rajouter un élément à la variable *birthday*. Un paramètre de sortie d'une opération est dénoté en Z par le nom de la variable suivi d'un !.

Pour initialiser, un schéma *InitBirthdayBook* est défini :

InitBirthdayBook

$$BirthdayBook$$

$$known = \varnothing$$

La variable *known* est ici initialisée par l'ensemble vide.

Raffinement de données. Le *raffinement* [DW96] permet de remplacer les types de données abstraits d'une spécification Z par des types de données plus concrets. Le raffinement d'un schéma d'état est, en outre, accompagné des raffinements des opérations qui modifient l'état du schéma raffiné. Les opérations sont par conséquent à nouveau spécifiées en utilisant les nouveaux types de données définis dans le raffinement du schéma du système.

Par exemple, *BirthdayBook* peut être raffiné par le schéma (voir [Spi92]) :

*BirthdayBook*1

$$names : \mathbb{N}_1 \rightarrow NAME$$
$$dates : \mathbb{N}_1 \rightarrow DATE$$
$$hwm : \mathbb{N}_1$$

$$\forall i, j : 1..hwm \bullet i \neq j \Rightarrow names(i) \neq names(j)$$

Ce schéma définit les nouvelles variables *names*, *dates* et *hwm*. Cette nouvelle spécification est plus concrète, car les variables sont désormais représentées par des tableaux de données.

Le schéma *Abs* définit la *relation d'abstraction* entre les variables abstraites (*known* et *birthday*) et les variables concrètes (*names*, *dates* et *hwm*) :

Abs

$$BirthdayBook$$
$$BirthdayBook1$$

$$known = \{i : 1..hwm \bullet names(i)\}$$
$$\forall i : 1..hwm \bullet$$
$$\quad birthday(names(i)) = dates(i)$$

La variable abstraite *known* est remplacée dans le raffinement par un ensemble de noms. La variable concrète *hwm* représente le nombre de noms disponibles dans le carnet et *names* est un tableau de noms. La variable abstraite *birthday* permet de relier les éléments du tableau de noms *names* aux éléments du tableau de dates *dates*.

Les opérations Z sont raffinées en spécifiant les opérations définies dans les schémas abstraits avec les types de données concrets. Par exemple, dans [Spi92], l'opération *AddBirthday* est raffinée par :

*AddBirthday*1
$\Delta BirthdayBook1$
$n? : NAME$
$d? : DATE$

$\forall i : a..hwm \bullet n? \neq names(i)$
$hwm' = hwm+1$
$names' = names \oplus \{hwm' \mapsto n?\}$
$dates' = dates \oplus \{hwm' \mapsto d?\}$

L'opération a les mêmes paramètres d'entrée et de sortie que dans la spécification abstraite. L'ajout d'une date d'anniversaire dans le carnet est désormais spécifié en incrémentant la variable *hwm* de 1, et en surchargeant les variables *names* et *dates* pour compléter les tableaux de données par le nouveau nom et la nouvelle date respectivement.

2.2.3 Object-Z

Object-Z est une extension du langage Z qui permet de spécifier des systèmes dans un style orienté objet [Smi00]. Dans une spécification Z, il est difficile de déterminer les conséquences des opérations sur un schéma d'état donné, car les schémas d'opération peuvent agir sur les états de plusieurs schémas d'état. La notion de classe est introduite pour regrouper dans un même schéma toutes les opérations la concernant.

Notion de classe. Dans Object-Z, la structure appelée *classe* permet de décrire à la fois un schéma d'état et des schémas d'opération qui agissent sur cet état. Une classe Object-Z est représentée par une boîte contenant :

- la liste des classes héritées,
- des définitions de types,
- des définitions de constantes,
- un schéma d'état,
- un schéma d'état initial,
- et des schémas d'opération.

Le schéma d'état ne porte généralement pas de nom dans une classe Object-Z.

Par exemple, la classe *Queue[T]* [SD01] (voir figure 2.2) définit une file d'attente de type FIFO. Elle est représentée comme une séquence d'éléments dont le type est défini en paramètre de la classe. La file d'attente a une capacité maximale *max*. Deux opérations sont définies, *Join* et *Leave*. Elles permettent respectivement d'ajouter et de retirer un élément de la file. Contrairement aux schémas d'opération en Z, une Δ-liste des variables modifiées est spécifiée pour chaque opération. Enfin, la classe *Queue[T]* n'hérite pas d'autres classes.

Héritage et instanciation. L'*héritage* permet à une classe de considérer les définitions d'une autre classe, en particulier les définitions de type, de constante et les schémas. L'*instanciation* permet de renommer les variables, les types et les constantes d'une classe.

Les deux mécanismes sont généralement liés. Par exemple, la définition d'un canal de transmission de messages avec pertes (lossy channel system) peut être

Queue[T]

$max : \mathbb{N}$

$items : \text{seq } T$

$\#items \leqslant max$

INIT

$items = \langle\,\rangle$

Join

$\Delta(items)$

$item? : T$

$\#items < max$

$items' = \langle item? \rangle \frown items$

Leave

$\Delta(items)$

$item! : T$

$items \neq \langle\,\rangle$

$items = items' \frown \langle item! \rangle$

FIG. 2.2 – Classe Queue[T]

considérée comme l'instanciation et l'héritage de la classe *Queue[T]*. Si *MSG* est l'ensemble de tous les messages possibles, alors :

MsgChannel

$Queue[MSG][msgs/items, msg?/item?, msg!/item!]$

Lose

$\Delta(msgs)$

$msgs' = tail\ msgs$

est la classe Object-Z définissant un canal de transmission de messages avec pertes. Les variables *items*, *item*? et *item*! sont renommées par *msgs*, *msg*? et *msg*! respectivement. Une nouvelle opération, *Lose*, est en outre définie.

2.2.4 Langage B

B [Abr96] est à la fois un langage et une méthode puisqu'il permet de spécifier un système depuis l'analyse jusqu'à l'implémentation. Il dispose en outre de nombreux outils pour assister l'utilisateur. Le langage B est un langage de spécification formel, basé sur la notion de machine abstraite. Les fondements théoriques de la méthode sont détaillés dans [Abr96].

Machine abstraite. Une *machine abstraite* représente un état spécifié par une partie statique (à l'aide de variables d'état et des propriétés d'invariance) et une partie dynamique (à l'aide d'opérations). Le langage pour la description de la statique repose sur la théorie des ensembles et sur la logique du premier ordre. Les variables sont ainsi typées par des ensembles et les invariants sont spécifiés à l'aide de conjonctions de prédicats du premier ordre. L'état de la machine abstraite ne peut être modifié que par des opérations. Le langage permettant d'exprimer la partie dynamique est un langage de substitutions généralisées. Il permet de décrire les opérations qui font évoluer l'état du système modélisé. Lors des phases initiales de spécification, le langage est abstrait : les instructions des opérations utilisent des préconditions et de l'indéterminisme. Les différentes clauses d'une machine abstraite B sont présentées dans le tableau 2.1.

La figure 2.3 présente la machine *Exemple_Machine*, qui est un exemple de machine abstraite spécifiée avec le langage B. On remarque que, dans le corps de l'opération abstraite **change**, la substitution est spécifiée à l'aide de la commande CHOICE qui n'est pas déterministe. Dans ce cas, la variable abstraite x peut être substituée par $x+2$ ou par $x-2$. La précondition (PRE) permet de faire respecter l'invariant (voir la clause INVARIANT) si l'opération **change** est exécutée. Les principaux opérateurs du langage de substitutions sont présentés dans l'annexe B.

Raffinement. Une machine abstraite B est ensuite raffinée. Cette phase permet de passer d'une structure abstraite à une structure proche du code. Le raffinement B se fait en plusieurs étapes successives. Les préconditions des opérations deviennent alors de plus en plus larges et les instructions de plus en plus déterministes. Les machines issues du raffinement ne contiennent alors

TAB. 2.1 – Clauses d'une machine abstraite en B

Clauses	Description
MACHINE	Nom et paramètres de la machine
CONSTRAINTS	Définition des propriétés des paramètres de la machine
SETS	Liste des ensembles abstraits et définition des ensembles énumérés
CONSTANTS	Liste des constantes de la machine
PROPERTIES	Définition des propriétés des constantes et des ensembles
VARIABLES	Liste des variables d'état de la machine
INVARIANT	Définition des types et des propriétés des variables
DEFINITIONS	Liste d'abréviations pour les prédicats, les expressions ou les substitutions
INITIALISATION	Initialisation des variables d'état
OPERATIONS	Liste des opérations de la machine

MACHINE *Exemple_Machine*
VARIABLES x
INVARIANT $x \in 0..20$
INITIALISATION $x := 10$
OPERATIONS
  change $=$
  PRE
    $x + 2 \leq 20 \wedge$
    $x - 2 \geq 0$
  THEN
    CHOICE $x := x + 2$
    OR $x := x - 2$
    END
  END
END

FIG. 2.3 – Machine abstraite B : *Exemple_Machine*

REFINEMENT *Exemple_Refinement*
REFINES *Exemple_Machine*
VARIABLES y
INVARIANT
$y \in 0 .. 10 \wedge$
$x = 2 \times y$
INITIALISATION $y := 5$
OPERATIONS
change =
BEGIN
$y := y + 1$
END
END

FIG. 2.4 – Raffinement en B : *Exemple_Refinement*

ni précondition, ni indéterminisme. Par exemple, la figure 2.4 fournit un raffinement possible de la machine *Exemple_Machine*. On a introduit une variable concrète y. L'invariant $x = 2 \times y$ permet de relier cette nouvelle variable y avec la variable abstraite x de la machine abstraite. Cet invariant est appelé un invariant de *collage*. Cette nouvelle machine est plus concrète que *Exemple_Machine*, puisque l'opération **change** est désormais déterministe et sans précondition.

Outil. À la différence des approches précédentes, B possède un outil très puissant qui couvre toutes les phases de la méthode de conception. L'Atelier B [Cle06], commercialisé par la société Clearsy, est en effet un environnement permettant de gérer des projets en langage B. Il offre différentes fonctionnalités :

- automatisation de certaines tâches (vérification syntaxique, génération automatique de théorèmes à démontrer, traduction de B vers C, C++, ...),
- aide à la preuve pour démontrer automatiquement des théorèmes,
- aide au développement.

Plus précisément, le prouveur de l'Atelier B permet de vérifier quatre points importants des spécifications B :

- au niveau de la machine : la dynamique doit respecter la statique,
- au niveau de l'initialisation : l'initialisation (clause INITIALISATION) établit l'invariant (clause INVARIANT),
- au niveau des opérations : chaque opération (clause OPERATIONS) doit préserver les propriétés d'invariance (clause INVARIANT),
- l'Atelier B permet enfin de prouver la correction du raffinement par rapport au modèle initial.

Le prouveur B est un outil de preuve interactif. Dans un premier temps, il permet de générer les obligations de preuve (de la forme : hypothèses $\Rightarrow$ conclusion) qui sont classées selon deux catégories : les obligations dont la preuve est évidente (notamment dans les cas où la conclusion fait partie des hypothèses) et les autres. L'Atelier B dispose alors d'un prouveur automatique de force variable. La force est un compromis entre l'efficacité et la rapidité du prouveur. Si, malgré l'exécution du prouveur automatique, il reste encore des obligations à prouver, l'utilisateur doit les prouver interactivement en utilisant les tactiques

du prouveur.

Bien plus qu'un langage, B est une méthode de spécification complète. L'environnement d'applications Atelier B peut être considéré comme faisant partie intégrante de la méthode B.

2.2.5 B événementiel

Le B événementiel [AM98] est une extension du langage B qui permet de spécifier des systèmes événementiels. Les principales différences avec B sont d'une part la considération d'un système fermé pour représenter l'ensemble des composants dans un seul modèle et d'autre part la définition du comportement sous la forme d'événements et non par des opérations comme en B. L'objectif est de prendre en compte l'ensemble du système.

Systèmes événementiels. Un événement est défini en B événementiel par une garde, *i.e.,* une condition bloquante qui assure la cohérence du système en cas d'exécution de l'événement, et d'une action exprimée à l'aide du langage de substitutions généralisées comme en B. Un événement est de la forme générale :

ANY $x, y, ...$ WHERE
  $P(x, y, ..., v, w, ...)$
THEN
  $S(x, y, ..., v, w, ...)$
END

avec $x, y, ...$ des variables locales et $v, w, ...$ des constantes ou des variables d'état du système d'événements. Dans cet exemple, P est la garde et S est l'action. Lorsqu'aucune variable locale n'est définie, l'expression d'un événement se simplifie par :

SELECT $P(v, w, ...)$
THEN $S(v, w, ...)$
END

Par exemple, la figure 2.5 présente le système *Exemple_System*, qui est une spécification en B événementiel. Dans cet exemple, après l'initialisation du système, seul **change0to1** peut être exécuté car sa garde est satisfaite. Si cet événement est exécuté, alors la valeur de la variable x devient égale à 1. Dans ce cas, un seul événement peut alors être exécuté : il s'agit de **change1to2**. Si x reste inchangée par l'exécution de **change1to2**, alors l'événement peut être exécuté de nouveau, sinon aucun événement ne peut être exécuté.

Raffinement. Le raffinement défini en B événementiel permet non seulement de modifier les espaces d'états comme en B, mais aussi de rajouter de nouveaux événements. Cependant, seules les nouvelles variables concrètes qui ne dépendent pas des variables plus abstraites peuvent être modifiées par ces nouveaux événements. Le raffinement de données est exprimé, comme en B, par un invariant de collage. Par exemple, le système *Exemple_System* peut être raffiné par le système décrit dans la figure 2.6. Dans ce raffinement, une nouvelle variable d'état *compteur* est définie. Cette variable peut être modifiée uniquement par le nouvel événement appelé **incremente**, qui l'incrémente de 1. En B

```
EVENT SYSTEM Exemple_System
VARIABLES x
INVARIANT x ∈ 0..2
INITIALISATION x := 0
EVENTS
    change0to1 =
    SELECT
        x = 0
    THEN
        x := 1
    END;

    change1to2 =
    SELECT
        x = 1
    THEN
        CHOICE x := 1
        OR x := 2
        END
    END
END
```

FIG. 2.5 – Système d'événements B : *Exemple_System*

événementiel, le raffinement a pour effet de renforcer les gardes des événements. Ainsi, les gardes de **change0to1** et **change1to2** ont été renforcées par la condition sur la variable *compteur*. De plus, les événements deviennent de plus en plus déterministes à chaque étape de raffinement. Par exemple, la substitution CHOICE a été remplacée par un des choix possibles dans l'événement **change1to2**.

2.3 Langage basé sur les événements : CSP

Le langage CSP (Communicating Sequential Processes) [Hoa85] est une notation utilisée pour décrire des systèmes concurrents. Le langage est supporté par quelques outils, comme FDR [For97], qui permettent d'analyser et de vérifier les spécifications en cours ou existantes. CSP a été inventé par C.A.R. Hoare et développé à l'Université de Oxford dans les années 80.

En CSP, les *processus* sont des entités, indépendantes les unes des autres, mais qui peuvent communiquer entre elles. Un processus peut exécuter des *événements* (ou *actions*). Les événements permettent de décrire le comportement des processus. L'ensemble des événements que le processus P peut exécuter est appelé son *alphabet* (ou *interface*) et est dénoté par $\alpha(P)$. Le comportement le plus simple d'un processus est de ne rien faire : un tel processus est dénoté par *STOP*.

REFINEMENT *Exemple_Refinement*
REFINES *Exemple_System*
VARIABLES $y, compteur$
INVARIANT $y = x \wedge compteur \in \mathbb{N}$
INITIALISATION $y, compteur := 0, 0$
EVENTS
  change0to1 =
  SELECT
    $y = 0 \wedge compteur \leq 10$
  THEN
    $y := 1$
  END ;

  change1to2 =
  SELECT
    $y = 1 \wedge compteur \geq 5$
  THEN
    $y := 2$
  END ;

  incremente =
  SELECT
    $compteur \leq 10$
  THEN
    $compteur := compteur+1$
  END
END

FIG. 2.6 – Raffinement en B événementiel : *Exemple_Refinement*

Préfixe. Le *préfixe* ($\rightarrow$) permet de définir un processus en explicitant les événements qu'il peut exécuter. Si a est un événement et P un processus, alors :

$$a \rightarrow P$$

est le processus qui peut exécuter a et se comporte ensuite comme le processus P. L'opérateur de préfixe est toujours utilisé sous cette forme avec un événement à la gauche de $\rightarrow$ et un processus à droite. Il est possible d'enchaîner les préfixes. Dans ce cas, les parenthèses sont implicites et portent sur les processus à droite de la flèche (associativité à droite). Par exemple,

$$a \rightarrow b \rightarrow Q$$

correspond à l'expression de processus :

$$a \rightarrow (b \rightarrow Q)$$

Récursivité. Les définitions *récursives* permettent de décrire en CSP des processus qui agissent sans fin. Par exemple,

$$Alternative = On \rightarrow Off \rightarrow Alternative$$

est un processus défini de manière récursive. Il exécute les événements *On* et *Off* en alternance. Les processus peuvent ainsi être définis par mutuelle récursion. Par exemple,

$$\begin{aligned} Light\_Off &= On \rightarrow Light\_On \\ Light\_On &= Off \rightarrow Light\_Off \end{aligned}$$

Opérateurs de choix. Il existe plusieurs opérateurs de choix en CSP. Le plus simple, dénoté par $|$, agit sur les processus de la forme $a \rightarrow P$. Le processus :

$$a \rightarrow P \mid b \rightarrow Q$$

peut soit exécuter l'événement a et se comporter ensuite comme le processus P, soit exécuter l'événement b et se comporter ensuite comme le processus Q. Les préfixes sont nécessairement distincts ($a \neq b$). Il est possible d'utiliser $|$ avec plus de deux choix :

$$a \rightarrow P \mid b \rightarrow Q \mid ... \mid z \rightarrow R$$

L'opérateur $|$ se restreint aux processus de la forme $a \rightarrow P$. Il existe en CSP deux autres opérateurs de choix sur les processus : choix externe ($\Box$) et choix interne ($\sqcap$). Le processus $P \Box Q$ ($P \sqcap Q$ respectivement) peut exécuter tout événement que P ou Q peut exécuter. Après l'exécution de ce premier événement, le comportement de $P \Box Q$ ($P \sqcap Q$ respectivement) est soit celui de P, soit celui de Q, selon sur quel processus agissait le premier événement. Le choix est dit *externe*, si le choix du premier événement dépend d'un autre processus. Le choix est dit *interne*, si le choix est non déterministe. Pour définir les choix internes, CSP introduit la notion d'*événement interne*, dénotée par τ, qui représente un événement du processus qui n'est pas observable par les autres processus du système considéré.

Événements cachés. Il est possible à partir d'un processus P de *cacher* certains événements de son alphabet. Si A est un ensemble d'événements et si P est un processus, alors :

$$P \setminus A$$

est le processus dont les événements appartenant à A deviennent des événements internes du processus. Par exemple, pour cacher l'événement a du processus P dans la transition :

$$P \xrightarrow{a} P'$$

on définit le processus $P \setminus \{a\}$ et la transition $\xrightarrow{a}$ devient :

$$P \setminus \{a\} \xrightarrow{\tau} P'$$

Composition parallèle. Pour décrire plusieurs processus en concurrence, CSP introduit l'opérateur de *composition parallèle* entre processus. Si A et B sont les alphabets des processus P et Q respectivement, alors :

$$P_A||_B Q$$

est la composition parallèle de P et de Q. Dans ce cas, P peut uniquement exécuter les événements de A, Q ceux de B, et tous les événements de l'intersection de A et de B sont exécutés par P et Q en synchronisation. Par exemple, supposons que les processus P et Q sont définis par :

$$\begin{aligned} P &= a \to c \to P' \\ Q &= b \to c \to Q' \end{aligned}$$

avec $A = \{a, c\}$ et $B = \{b, c\}$. Dans ce cas, les transitions

$$P_A||_B Q \xrightarrow{a} (c \to P')_A||_B Q$$

et

$$P_A||_B Q \xrightarrow{b} P_A||_B (c \to Q')$$

sont possibles. Dans le premier cas, il est ensuite possible d'agir sur Q :

$$(c \to P')_A||_B Q \xrightarrow{b} (c \to P')_A||_B (c \to Q')$$

Il est à présent possible d'exécuter c qui demande la synchronisation des processus :

$$(c \to P')_A||_B (c \to Q') \xrightarrow{c} P'_A||_B Q'$$

Par abus de notation, les alphabets des processus ne sont parfois pas précisés dans la composition parallèle. Par exemple, la composition parallèle des processus P et Q est dénotée par :

$$P \,||\, Q$$

Entrelacement. L'opérateur d'*entrelacement* (|||) est similaire à l'opérateur de composition parallèle, mais sans synchronisation. On note :

$$P \ ||| \ Q$$

l'entrelacement des processus P et Q. Les événements de P sont alors exécutés indépendamment de l'exécution des événements de Q. Par exemple, si P et Q sont définis par :

$$\begin{aligned} P &= a \rightarrow b \rightarrow P' \\ Q &= c \rightarrow Q' \end{aligned}$$

Les séquences d'événements possibles pour $P \ ||| \ Q$ sont : (a, b, c), (a, c, b) et (c, a, b). Ces séquences sont appelées les *traces* du processus $P \ ||| \ Q$.

Entrées et sorties. Un événement en CSP peut être représenté par un message passant par un canal. Les événements sont donc de la forme $c.v$, où c est le nom d'un *canal* et v est la *valeur* du message passant par le canal c. Le *type* d'un canal est l'ensemble des valeurs possibles pouvant passer par ce canal.

Les valeurs des messages peuvent être exprimées en CSP en termes d'*entrées* ou de *sorties* des canaux dans les processus avec préfixe. Le processus :

$$c\,!v \rightarrow P$$

est un processus dont la valeur v est une sortie du canal c et qui se comporte ensuite comme le processus P. Dans cette expression de processus, v doit vérifier que $v \in T$, avec T le type du canal c.

De même,

$$c\,?x : T \rightarrow P(x)$$

est un processus dont le paramètre x de type T est une entrée du canal c et qui se comporte ensuite comme le processus paramétré $P(x)$.

Quantification. Les opérateurs de choix interne et externe, de composition parallèle et d'entrelacement ont une forme plus générale pour considérer des combinaisons d'un nombre quelconque de processus. Si I est un ensemble fini d'indices, alors :

$$\sqcap_{i \in I} P_i$$

définit le choix interne entre les processus P_i, avec $i \in I$. De même,

$$\Box_{i \in I} P_i$$

est le choix externe entre tous les processus P_i, avec $i \in I$. Si A_i est l'interface de P_i, pour chaque $i \in I$, alors :

$$||_{A_i}^{i \in I} \ P_i$$

est la composition parallèle de tous les P_i. L'entrelacement de plusieurs processus P_i, avec $i \in I$, est dénoté par :

$$|||_{i \in I} \ P_i$$

Raffinement. La sémantique de CSP repose sur l'observation des effets des processus : modèles des traces, des échecs stables et des traces-divergences (voir section 2.1.2). Le raffinement consiste alors à calculer et à comparer les modèles sémantiques de deux processus. Par exemple, dans le cas du modèle des échecs-divergences, si P et Q sont deux processus, alors Q raffine P si :

$$\begin{aligned} \mathit{failures}(Q) &\subseteq \mathit{failures}(P) \\ \mathit{divergences}(Q) &\subseteq \mathit{divergences}(P) \end{aligned}$$

Dans cet exemple, il n'est pas utile de comparer $\mathit{traces}(P)$ et $\mathit{traces}(Q)$, car par définition des échecs stables :

$$\mathit{failures}(Q) \subseteq \mathit{failures}(P) \Rightarrow \mathit{traces}(Q) \subseteq \mathit{traces}(P)$$

Chapitre 3

Combinaison de spécifications formelles

> "Vos lacunes en mathématiques ne doivent pas vous inquiéter ... Je peux vous assurer que les miennes sont encore plus importantes ! "
>
> — Albert Einstein

Dans ce chapitre, nous allons étudier des exemples d'utilisation de combinaisons de spécifications formelles pour modéliser des systèmes et pour vérifier des propriétés.

3.1 Critères de comparaison

Afin de comparer les différentes approches, nous avons utilisé un exemple de référence. Il s'agit d'un système qui crée et supprime des produits. Le LTS de cet exemple est représenté par la figure 3.1. On remarque qu'un produit ne peut être supprimé que s'il a été créé auparavant et qu'un nouveau produit ne peut être créé que s'il n'y a plus aucun autre produit. Ces fortes propriétés d'ordonnancement devront être respectées par les différentes approches. L'opération `DisplayProduct`, qui peut être exécutée lorsque la machine n'est pas à l'arrêt, retourne le produit courant ou bien la valeur spécifique "NULL", lorsque le dernier produit a été supprimé.

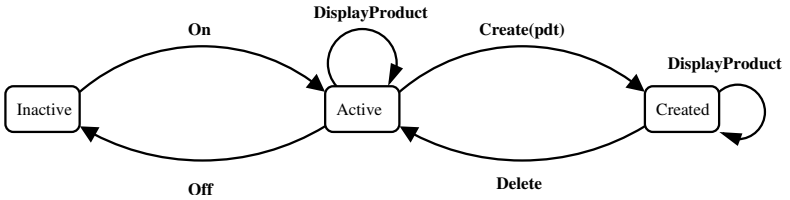

FIG. 3.1 – LTS de l'exemple

Cet exemple nous permet de considérer à la fois des propriétés statiques et dynamiques caractéristiques des SI. En effet, on y retrouve les types suivants d'exigences :

- quelques opérations, comme `Create` et `Delete`, modifient les variables d'état du système ; il est donc possible de considérer des propriétés statiques comme des propriétés d'invariance sur les états.
- le système est caractérisé par des contraintes d'ordonnancement fortes ; par conséquent, on peut considérer des propriétés dynamiques comme des propriétés de vivacité.
- l'opération `DisplayProduct` retourne la valeur d'une variable ; elle peut donc représenter une requête sur la valeur d'un attribut.

L'exemple proposé contient en fait le plus grand ensemble d'exigences, typiques des SI, commun à toutes les approches étudiées dans les sections suivantes. Cela dit, il n'est pas tout à fait réaliste pour un SI. En particulier, il ne permet pas l'existence de plusieurs produits à la fois. Nous indiquons cependant, pour chaque approche, s'il est effectivement possible de traiter la création de plusieurs produits.

Les critères de comparaison que nous considérons sont les suivants :

- syntaxe spécifique utilisée dans l'approche : nous décrivons les syntaxes particulières ainsi que les restrictions des langages utilisés.
- spécification de l'exemple de comparaison : nous spécifions avec chaque approche un système qui respecte le LTS de la figure 3.1.
- cohérence : comme les combinaisons peuvent introduire des redondances ou des contradictions, nous indiquons les techniques utilisées pour valider le modèle dans son ensemble. Nous précisons aussi quelle est la sémantique considérée pour chaque approche.
- vérification : nous présentons les techniques utilisées ainsi que les éventuels outils d'assistance.
- raffinement : nous indiquons s'il existe des relations de raffinement spécifiques à l'approche.
- adéquation avec les SI : nous analysons les avantages et les limites des combinaisons pour le contexte particulier des SI.

Les exemples de combinaisons de spécifications basées sur les états avec des spécifications basées sur les événements peuvent être classés en trois groupes suivant les objectifs de chaque approche :

1. **Spécification en plusieurs parties :** Dans ce cas, deux langages (l'un basé sur les états et l'autre, sur les événements) sont utilisés conjointement pour spécifier le comportement du système, comme avec csp2B (section 3.2) et CSP || B (section 3.3). Chaque langage fournit une partie ou un aspect de la spécification. Par conséquent, les deux parties sont requises pour retrouver la spécification complète. Ce type de combinaison préserve les caractéristiques de chaque langage, ce qui permet de réutiliser aisément les outils existants. Les deux parties de la spécification sont reliées entre elles par une traduction ou par l'assimilation de deux structures.

2. **Création d'un nouveau langage :** Dans ce cas, un nouveau langage est défini pour intégrer dans une même sémantique les structures ou les opérateurs des paradigmes basés sur les états et de ceux basés sur les événements. Le langage permet alors d'unifier leur sémantique et de réutiliser le maximum de notations et de techniques intéressantes. Par exemple, le langage CSP-OZ (section 3.4) est inspiré de CSP et Object-Z, alors que Circus (section 3.5) est un langage basé sur Z, CSP et le calcul du raffinement.

3. **Vérification de propriétés :** Dans ce dernier cas, une spécification complète est exprimée dans un premier langage et le second langage est utilisé pour décrire des propriétés à prouver ou à vérifier sur le premier. Par exemple, des formules de logique temporelle PLTL sont vérifiées sur des systèmes d'événements B (section 3.6).

3.2 csp2B

csp2B [But99] appartient au premier groupe de combinaisons (spécification en plusieurs parties). L'idée est de traduire une spécification CSP (voir section 2.3) en une machine B (section 2.2.4). Pour des raisons sémantiques, un sous-ensemble seulement d'opérateurs CSP est utilisé. La description CSP permet de spécifier l'ordonnancement des opérations d'une machine B ; cette dernière est dite "conjointe" (*conjoined* en anglais) à la spécification CSP. L'outil csp2B peut traduire la description CSP en une machine B standard qui réutilise la machine B conjointe, grâce au lien INCLUDES qui est défini dans la méthode B pour modulariser les spécifications. Le but de l'approche csp2B est de spécifier le contrôle en CSP et les données en B. Ainsi, les propriétés de vivacité peuvent être vérifiées avec les outils CSP existants et les propriétés de sûreté peuvent être prouvées en B. Le raffinement utilisé dans l'approche csp2B consiste à traduire les spécifications CSP en B et à utiliser la relation de raffinement de la méthode B. L'outil csp2B fournit un lien unidirectionnel (de CSP vers B) entre les deux parties de la spécification d'un système.

3.2.1 Syntaxe

La partie CSP du modèle est décrite comme une machine avec les clauses suivantes :

- MACHINE : nom de la spécification,
- CONJOINS : nom de la machine "conjointe", qui est une machine B dont l'ordonnancement des opérations est contraint par la machine CSP,
- ALPHABET : liste d'événements,
- PROCESS : description du processus principal.

Les opérateurs CSP suivants sont utilisés pour décrire le processus de la machine CSP : $\rightarrow$ (préfixe), $\Box$ (choix externe), || (composition parallèle), ||| (entrelacement), *SKIP* (processus qui ne fait rien). Les deux premiers opérateurs sont utilisés sans aucune restriction. La composition parallèle est utilisée uniquement pour les expressions les plus à l'extérieur. L'entrelacement s'applique uniquement sur plusieurs instances du même processus. Tout appel récursif de processus est préfixé par un événement. Le langage csp2B accepte également des processus de la forme **IF** c **THEN** P_1 **ELSE** P_2, avec c une condition et P_1 et P_2 des processus. La puissance et l'expressivité de CSP ne sont donc pas entièrement utilisées par l'approche csp2B, puisque la quantification des expressions est notamment limitée.

La partie B du modèle est spécifiée par une machine B standard, sans aucune restriction sur le langage.

3.2.2 Spécification

La machine *Example_Act* permet de spécifier en B les types de données ainsi que les opérations de notre exemple de référence :

MACHINE *Example_Act*
SETS *PRODUCTS*
CONSTANTS *Null*
PROPERTIES $Null \in PRODUCTS$
VARIABLES *Product*
INVARIANT $Product \in PRODUCTS$
INITIALISATION $Product := Null$
OPERATIONS
  Create_Act(*pdt*) =
  PRE $pdt \neq Null$
  THEN
    $Product := pdt$
  END ;

  Delete_Act =
  BEGIN
    $Product := Null$
  END ;

  $pdt \longleftarrow$ **DisplayProduct_Act** =
  BEGIN
    $pdt := Product$
  END
END

Cette machine est définie comme une machine B classique. L'état du système est représenté par la variable d'état *Product*. Cette dernière représente le produit existant, s'il en existe effectivement un. Sinon, sa valeur est *Null*, qui est définie comme une constante de l'ensemble *PRODUCTS* (voir les clauses CONSTANTS et PROPERTIES). La variable d'état *Product* appartient à l'ensemble abstrait *PRODUCTS*, qui représente tous les produits possibles. L'invariant de la machine doit être préservé par les trois opérations définies dans la machine : **Create_Act**, **Delete_Act** et **DisplayProduct_Act**. La première a pour effet d'affecter le nouveau produit *pdt* à la variable *Product*. Dans ce cas, le paramètre d'entrée *pdt* ne doit pas être égal à la constante *Null*. La précondition de l'opération vérifie effectivement que ce n'est pas le cas. L'opération **Delete_Act** donne la valeur *Null* à la variable *Product*, car le produit a été supprimé. Quand la précondition d'une opération B est toujours vraie, cette dernière est remplacée par le mot-clé BEGIN. Enfin, l'opération **DisplayProduct_Act** retourne le produit existant, le cas échéant.

Une description CSP est maintenant utilisée pour contraindre les opérations de la machine *Example_Act*. En CSP, le processus suivant serait suffisant pour spécifier le comportement du système :

$$Main = On \rightarrow Active$$
$$Active = (ProductCycles \;|||\; Requests) \Box (Off \rightarrow Main)$$

$$Requests = (DisplayProduct!pdt \rightarrow Requests) \Box SKIP$$
$$ProductCycles = Create?pdt \rightarrow Delete \rightarrow Active$$

Trois sous-processus sont définis dans *Main* : *Active*, *Requests* et *ProductCycles*. À l'état initial du processus *Main*, seule l'action *On* est exécutable. Une fois cette action exécutée, le processus se retrouve à l'état[1] *Active*. Dans cet état, il y a deux alternatives : soit une ou plusieurs créations de produit sont effectuées, soit l'action *Off* est exécutée et le système retourne à l'état initial. Dans le premier cas, une ou plusieurs requêtes (action *DisplayProduct*) sont entrelacées avec le cycle de vie des produits qui sont créés, les uns après les autres (processus *ProductCycles*). Le processus *Main* a le même système de transition que la figure 3.1. Cependant, on ne peut pas écrire une telle expression en csp2B, car l'opérateur d'entrelacement ne peut être utilisé que pour les expressions de plus haut niveau. Par conséquent, il existe deux options pour modéliser le comportement du système avec csp2B : soit l'action *DisplayProduct* n'est pas contrainte par la partie CSP, soit elle doit être explicitement entrelacée dans les séquences d'actions correspondantes.

Si l'action *DisplayProduct* n'est pas contrainte par le processus CSP, alors la partie CSP de la spécification csp2B est la suivante :

MACHINE *Example*
CONJOINS *Example_Act*
SETS *PRODUCTS*
ALPHABET
  $On,\ Off, Create(pdt : PRODUCTS), Delete$
PROCESS $EX = Inactive$
CONSTRAINS *Create*, *Delete*
WHERE
  $Inactive = On \rightarrow Active$
  $Active = (Create?pdt \rightarrow Delete \rightarrow Active) \Box (Off \rightarrow Inactive)$
END

Dans la sémantique de csp2B, une action *op* de la clause CONSTRAINS correspond à l'opération **op_Act** de la machine B qui est conjointe. La description CSP ci-dessus définit un processus principal appelé *EX* pour contraindre les opérations **Create_Act** et **Delete_Act**. Deux sous-processus sont également définis : *Inactive* et *Active*. Le processus *EX* a exactement le même comportement que le processus *Main* décrit ci-dessus, excepté pour l'action *DisplayProduct* qui n'est pas prise en compte par *EX*. Notons que l'action *Create* est considérée dans la partie CSP comme un canal de communication parce que l'opération B correspondante a un paramètre d'entrée. Le type de ce dernier est déclaré dans la clause ALPHABET.

Si on souhaite tenir compte de l'action *DisplayProduct* dans la machine CSP, alors le processus *Active* est défini de manière à éviter l'utilisation de l'entrelacement :

$$Active = (Create?pdt \rightarrow Created) \Box (DisplayProduct!pdt \rightarrow Active) \Box (Off \rightarrow Inactive)$$
$$Created = (Delete \rightarrow Active) \Box (DisplayProduct!pdt \rightarrow Created)$$

[1] Une expression de processus est souvent appelée un "état" en CSP, par référence au système de transitions qui peut être associé à une expression de processus.

L'état intermédiaire entre la création et la suppression d'un produit est alors explicitement défini en CSP et l'action *DisplayProduct* est indiquée à chaque fois qu'elle peut être exécutée. Toutefois, cette solution ne serait pas optimale pour des spécifications avec de longues séquences d'actions. En effet, supposons par exemple que les processus P et Q sont définis par :

$$P = a_1 \rightarrow ... \rightarrow a_{10} \rightarrow P$$
$$Q = b_1 \rightarrow ... \rightarrow b_{10} \rightarrow Q$$

La réécriture de $P \;|||\; Q$ en une expression de processus qui n'utilise pas d'entrelacement devient alors fastidieuse.

3.2.3 Outil csp2B

L'outil csp2B permet de traduire une description CSP définie comme dans la section 3.2.1 en une machine B. Par exemple, la traduction de la machine CSP qui ne contraint pas l'action *DisplayProduct* est la suivante :

MACHINE *Example*
INCLUDES *Example_Act*
SETS $PRODUCTS$; $EXState = \{Inactive, Active, Active\_1\}$
VARIABLES EX
INVARIANT $EX \in EXState$
INITIALISATION $EX := Inactive$
OPERATIONS
On =
SELECT $EX = Inactive$
THEN
$EX := Active$
END ;

Off =
SELECT $EX = Active$
THEN
$EX := Inactive$
END ;

Create(pdt) =
PRE
$pdt \neq Null$
THEN
SELECT $EX = Active$
THEN $EX := Active\_1$
END
||
SELECT $EX = Active$
THEN **Create_Act**(pdt)
END
END ;

Delete =

BEGIN
SELECT $EX = Active\_1$
THEN $EX := Active$
END
‖
SELECT $EX = Active\_1$
THEN **Delete_Act**
END
END
END

Les différents états du processus *EX* sont définis dans l'ensemble *EXState* : *Inactive*, *Active* et *Active_1*. Ce dernier est un état implicite du processus *EX* qui est automatiquement généré par l'outil. Il correspond à l'état après avoir exécuté *Create*, mais avant d'exécuter *Delete* (*i.e.,* l'état appelé *Created* dans la figure 3.1). L'état courant du système est représenté par la variable *EX*. Les actions sont traduites en des opérations gardées, autrement dit, avec des substitutions B de la forme SELECT THEN END. Une opération gardée ne peut être exécutée que si le prédicat indiqué dans la clause SELECT est satisfait.

Le résultat de la traduction contient aussi une inclusion (clause INCLUDES) de la machine B conjointe indiquée dans la clause CONJOINS de la description CSP initiale. Le lien INCLUDES permet d'avoir accès en lecture aux variables et aux ensembles de cette machine B et de réutiliser ses opérations. Les spécifications obtenues sont le résultat d'une traduction automatique ; par conséquent, certaines spécifications peuvent être simplifiées. Par exemple, on peut définir plus simplement l'opération **Create** par :

Create(*pdt*) =
PRE $pdt \neq Null$
THEN
SELECT $EX = Active$
THEN
Create_Act(*pdt*) ‖
$EX := Active\_1$
END
END

La traduction automatique de la machine CSP qui contraint l'action *Display-Product* est similaire à la traduction ci-dessus, mais l'état *Active_1* est remplacé par *Created* et l'opération **DisplayProduct** est définie par :

pdt ⟵ **DisplayProduct** =
BEGIN
SELECT $EX = Active \lor EX = Created$
THEN *skip*
END
‖
SELECT $EX = Active \lor EX = Created$
THEN *pdt* ⟵ **DisplayProduct_Act**
END
END

La traduction de CSP vers B est justifiée par une approche opérationnelle de la sémantique. Les expressions de processus de la machine CSP sont dans un premier temps normalisées puis traduites en des LTS. Cette première traduction permet de retrouver les différents états implicites et explicites du processus. Les actions sont ensuites traduites en B par des opérations, dont les gardes sont retrouvées par analyse des LTS.

3.2.4 Vérification et raffinement

Comme les machines CSP se traduisent en des machines B grâce à csp2B, il est possible de vérifier certaines propriétés sur le processus en utilisant la clause ASSERTIONS de la machine B obtenue. La difficulté consiste à exprimer ces propriétés en B afin de pouvoir utiliser les outils supportant la méthode B, comme l'Atelier B. Par exemple, il est difficile de vérifier avec l'approche csp2B que l'action **Create** n'est exécutable que si l'état du système est préalablement passé à l'état *Active*, car ce type de propriétés s'exprime mal avec des prédicats du langage B.

L'utilisation de B constitue donc à la fois un avantage et un inconvénient. Les propriétés concernant les états sont assez faciles à exprimer en B, car une variable d'état est définie dans la machine B pour représenter les états du système. Les ordonnancements des événements sont en revanche difficiles à exprimer en B, car les actions sont traduites par des opérations et il n'est pas possible d'exprimer des prédicats sur les opérations en B. Il est toutefois possible d'utiliser les substitutions de ces opérations.

Le raffinement csp2B d'une machine CSP est défini grâce à un invariant d'abstraction déclaré dans la clause INVARIANT du raffinement csp2B. L'invariant d'abstraction permet d'exprimer les états de contrôle concrets du raffinement csp2B en fonction des états de contrôle abstraits de la machine CSP raffinée.

La figure 3.2 représente une méthode de raffinement possible en utilisant l'outil csp2B. Comme les variables d'état en B correspondent aux états des pro-

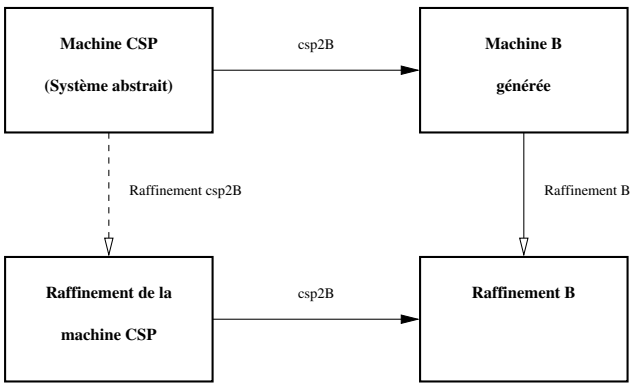

FIG. 3.2 – Méthode de raffinement csp2B

cessus en CSP, cette notion de raffinement csp2B est analogue au raffinement usuellement défini en B. Une traduction de la machine CSP avec l'outil csp2B

peut donc être utilisée pour expliciter les états implicites des processus et le raffinement classique en B de la machine obtenue peut servir à définir l'invariant du raffinement csp2B. Cet invariant est ensuite inclus dans l'invariant du raffinement csp2B comme invariant d'abstraction. Un exemple de raffinement est donné dans [But99]. Le raffinement csp2B est défini en fonction du raffinement B. L'approche ne propose donc pas de nouvelle notion de raffinement.

3.2.5 Adéquation avec les SI

Indépendamment de l'approche csp2B, le langage CSP n'est pas pratique pour spécifier les SI. Tout d'abord, les modèles de données des SI nécessitent la définition de structures basées sur les états comme les types de données abstraits ou bien les attributs de classes, alors que l'utilisation de variables d'état est limitée en CSP aux paramètres d'entrée-sortie au moyen de canaux de communication. En particulier, l'affectation de variables qui avait été définie aux débuts du langage CSP [Hoa85] n'est plus utilisée dans la pratique.

De plus, le processus est le concept fondamental de CSP, et chaque processus est utilisé pour communiquer avec les autres processus du système. Par conséquent, la représentation d'entités distinctes partageant les mêmes propriétés constitue un problème en CSP. En effet, si un processus est utilisé pour définir un type d'entité, alors comment peut-on considérer les communications entre les différentes entités du même type ? Et si un processus est défini pour chaque entité, alors quels sont les critères pour retrouver les entités du même type d'entité ?

Dans l'approche csp2B, il n'y a pas de concept de type d'entité, et la composition parallèle et l'entrelacement de CSP sont limités aux expressions de plus haut niveau. Toutefois, l'opérateur d'entrelacement peut être utilisé pour plusieurs instances du même processus. Plusieurs entités du même type peuvent donc être prises en compte si elles ont exactement le même comportement et s'il n'y a pas d'action en synchronisation avec les autres entités. Par exemple, pour permettre l'existence de plusieurs produits à la fois, le processus suivant peut être défini dans la partie CSP de csp2B :

$$\begin{array}{l} \text{PROCESS } \mathit{Main} \;=\; |||\;\; \mathit{pdt}.\mathit{ProductCycles}[\mathit{pdt}] \\ \text{WHERE} \\ \quad \mathit{ProductCycles}[\mathit{pdt}] \;=\; \mathit{Create}.\mathit{pdt} \rightarrow \mathit{Delete}.\mathit{pdt} \rightarrow \mathit{ProductCycles}[\mathit{pdt}] \end{array}$$

Comme l'entrelacement doit être défini dans le processus principal, les autres contraintes sur les actions *On* et *Off* ne sont pas prises en compte ici. De plus, le paramètre de quantification est spécifié comme un paramètre d'entrée des actions *Create* et *Delete*.

Le langage B permet de décrire le modèle de données d'un SI grâce aux variables d'état et aux propriétés d'invariance. En revanche, il est difficile de spécifier et d'analyser des propriétés dynamiques en B. Avec l'approche csp2B, les expressions de processus CSP décrivent le comportement des opérations B et l'outil traduit les descriptions CSP en B. La traduction est basée sur une sémantique opérationnelle de CSP, par l'intermédiaire de LTS.

L'utilisation d'opérations gardées en B n'est pas pratique d'un point de vue SI. En effet, une opération de la forme SELECT THEN END ne peut être exécutée que si le prédicat de la clause SELECT est satisfait. Sinon, l'opération n'est pas réalisable. Or, dans un SI, une opération doit toujours être disponible

et un message d'erreur est retourné si son exécution n'est pas autorisée (par exemple, s'il y a un risque de violation des contraintes d'intégrité).

En conclusion, l'approche csp2B propose une technique d'intégration de CSP et B qui apporte à la fois des avantages et des inconvénients. D'un côté, la traduction en B proposée limite l'expressivité des descriptions CSP parce que la sémantique de certains opérateurs de parallèlisme est difficile à définir avec des LTS. D'un autre côté, l'utilisation de B permet de prouver des propriétés basées sur les états comme des propriétés d'invariance. De plus, la traduction en B fournit une sémantique pour l'ensemble du modèle et permet d'éviter des contradictions entre les deux parties CSP et B de la spécification.

3.3 CSP || B

CSP || B [ST02] appartient au premier groupe de combinaisons (spécification en plusieurs parties). Un processus CSP (section 2.3) appelé *contrôleur d'exécution* est couplé à une machine B (section 2.2.4). L'exécution des opérations B est contrainte par le contrôleur CSP. Cette approche peut être appliquée à plusieurs machines B en parallèle. Chaque machine B est alors associée à un contrôleur CSP et les communications entre machines passent par les processus CSP. L'idée de l'approche CSP || B est de décomposer le système en plusieurs paires machine B/contrôleur CSP. Afin d'assurer la cohérence de l'ensemble de la spécification, il existe des conditions suffisantes à vérifier. Les propriétés de vivacité du système sont vérifiées à l'aide des outils existant en CSP, alors que les propriétés de sûreté sont prouvées en B. Dans l'approche CSP || B, les deux parties CSP et B sont requises pour modéliser un système.

3.3.1 Syntaxe

Le langage décrivant les processus est le suivant :

$$
\begin{array}{lll}
P & ::= & a \rightarrow P \mid c\,?x\langle E(x)\rangle \rightarrow P \mid d\,!v\{E(v)\} \rightarrow P \mid \\
 & & e\,\mathbf{?}\,v\,\mathbf{!}x\{E(x)\} \rightarrow P \mid P_1 \Box P_2 \mid P_1 \sqcap P_2 \mid \sqcap_{x|E(x)} P \mid \\
 & & \textit{if } b \textit{ then } P_1 \textit{ else } P_2 \textit{ end}
\end{array}
$$

où a est un événement, c est un canal de communication acceptant des entrées, d est un canal de communication acceptant des sorties, e est un canal de machine, x représente une variable de données, v une valeur de données, $E(x)$ est un prédicat sur x, b est une expression booléenne et P est une expression de processus.

Ce langage reprend la plupart des opérateurs couramment utilisés en CSP : $\rightarrow$ (préfixe), $\Box$ (choix externe), $\sqcap$ (choix interne). L'entrée d'une valeur x le long d'un canal c est dénotée par $c?x$. De même, la sortie d'une valeur v le long d'un canal d est notée $d!v$. Le prédicat $E(x)$ est utilisé pour représenter la garde (dénotée par $\langle E(x)\rangle$) ou l'assertion ($\{E(x)\}$) d'un événement.

Une garde est un prédicat sur les paramètres d'entrée qui doit être satisfait pour exécuter l'événement. Si la garde n'est pas satisfaite, alors le système est bloqué. Une assertion est un prédicat sur les paramètres de sortie qui est supposé être vrai si l'événement a été correctement exécuté. Si l'assertion n'est pas satisfaite, alors le système peut diverger. Dans ce cas,

– soit l'événement a avorté ; autrement dit, l'événement n'a pas été exécuté dans les bonnes conditions, et le système se retrouve alors dans un état qui n'était pas prévu dans la spécification,
– soit une boucle infinie d'événements internes a pris le contrôle du système.

Les opérateurs de composition parallèle et d'entrelacement ne sont pas pris en compte au niveau des contrôleurs CSP. La concurrence est en effet considérée à un niveau supérieur, entre les contrôleurs d'exécution. Si $P_1, ..., P_n$ est la liste de tous les contrôleurs, alors le comportement global du système est spécifié par : $P_1 \parallel ... \parallel P_n$.

Le langage permet de distinguer les communications impliquant uniquement des processus CSP (! et ?) des communications entre machines B et contrôleurs CSP (**!** et **?**). La figure 3.3 représente graphiquement les interactions possibles entre machines B et processus CSP.

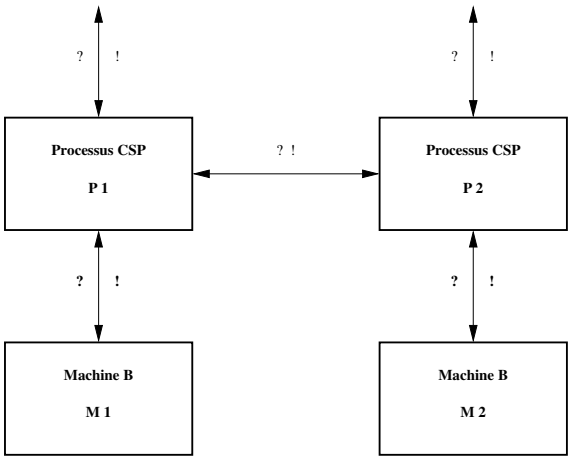

FIG. 3.3 – Interactions entre machines B et processus CSP

La partie B du modèle est spécifiée par une machine B standard, sans aucune restriction sur le langage.

3.3.2 Spécification

Notre exemple de référence (voir figure 3.1) est spécifié de la manière suivante. La machine B *ExampleData* décrit les états et les opérations du système, tandis que le processus *ExampleProc* est son contrôleur d'exécution associé. Les liens entre le contrôleur CSP et la machine B sont les opérations qui sont identifiées comme des canaux de communication dans le processus CSP. Par exemple, l'expression de processus $Create\_B\ ?x\{E(x)\} \rightarrow P$ exécute d'abord l'opération B **Create_B**(x), puis se comporte comme le processus P. Dans cette thèse, chaque opération B qui correspond à un canal CSP *op* est dénoté par **op_B**. Cependant, cette convention n'est pas obligatoire dans l'approche CSP || B et nous l'utilisons uniquement pour améliorer la compréhension du lecteur.

La figure 3.4 représente les interactions entre *ExampleData* et *ExampleProc*. Les événements *On* et *Off* sont appelés par l'environnement extérieur. L'opération *Create* est appelée par l'environnement pour indiquer le produit

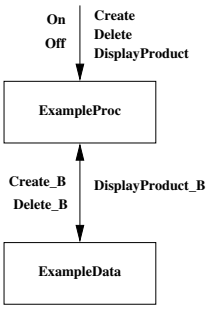

FIG. 3.4 – Lien entre la machine B et le processus CSP

pdt à créer. Elle contraint l'opération B **Create_B** dans le processus *Example-Proc*. De même, l'opération *Delete*, qui est appelée par l'environnement pour supprimer le produit existant, contraint l'opération **Delete_B** de la machine B. Enfin, l'opération *Display-Product* est appelée par l'environnement pour afficher le produit courant, si ce dernier existe.

La machine *ExampleData* est définie comme la machine *Example_Act* de la section 3.2 :

MACHINE *ExampleData*
SETS *PRODUCTS*
CONSTANTS *Null*
PROPERTIES $Null \in PRODUCTS$
VARIABLES *Product*
INVARIANT $Product \in PRODUCTS$
INITIALISATION $Product := Null$
OPERATIONS
  Create_B(*pdt*) =
  PRE $pdt \neq Null$
  THEN
    $Product := pdt$
  END ;

  Delete_B =
  BEGIN
    $Product := Null$
  END ;

  $pdt \longleftarrow$ **DisplayProduct_B** =
  BEGIN
    $pdt := Product$
  END
END

Comme dans csp2B (voir section 3.2), l'opérateur d'entrelacement n'est pas autorisé dans l'approche standard de CSP || B, décrite notamment dans [TS99, TS00, ST02]. Toutefois, l'entrelacement et la composition parallèle ont été utilisés en CSP || B dans [ETLF04] (voir section 3.3.4). Le processus CSP qui

permet de contrôler la machine *ExampleData* est défini par :

$$
\begin{aligned}
ExampleProc &= Inactive \\
Inactive &= On \rightarrow Active(Null) \\
Active(P) &= (ProductCycles(P) \,|||\, Requests(P)) \\
&\quad \Box(Off \rightarrow Inactive) \\
Requests(P) &= (DisplayProduct\ !pdt\{pdt \in P\} \rightarrow \\
&\quad DisplayProduct\_B\ \mathbf{!}pdt \rightarrow Requests(P)) \Box SKIP \\
ProductCycles(P) &= Create\ ?pdt < pdt \neq Null > \\
&\quad \rightarrow Create\_B\ \mathbf{?}pdt \rightarrow ProductCycles1(pdt) \\
ProductCycles1(P) &= Delete \rightarrow Delete\_B \rightarrow ProductCycles(Null)
\end{aligned}
$$

Ce processus a pour paramètre P qui représente l'ensemble des produits courants. Dans cet exemple, soit P est un singleton, soit on a $P = \{Null\}$ si le produit n'existe pas.

L'état initial du processus *ExampleProc* est *Inactive*. Quand il est à l'état *Active*, le processus décrit la séquence valide d'actions. Les gardes, comme le prédicat $pdt \neq Null$ dans l'expression $Create\ ?pdt < pdt \neq Null >$, sont utilisées pour définir les valeurs acceptées par les paramètres des opérations B. Les assertions, comme le prédicat $pdt \in P$ dans $DisplayProduct\ !pdt\{pdt \in P\}$, sont des propriétés qui doivent être satisfaites par les paramètres de sortie des opérations. Toute exécution d'une opération B est précédée par une action en CSP qui la contrôle. Par exemple, l'action *DisplayProduct* contrôle l'opération **DisplayProduct_B**.

À l'état *Inactive*, seule l'action *On* peut être exécutée. Ensuite, le système est dans l'état *Active*. Une fois de plus, deux options sont possibles. Soit l'environnement demande la création d'un nouveau produit avec l'action *CreateProduct* (avec le symbole "?"), soit le système retourne à l'état *Inactive* avec l'action *Off*. Dans le premier cas, l'opération B **Create_B** est exécutée et un état intermédiaire est défini avec *ProductCycles1*. On remarque que la valeur du paramètre P est le produit créé *pdt*. Par conséquent, la seule action possible est alors *Delete* qui contraint l'opération B **Delete_B**, et le système retourne à l'état *ProductCycles* avec $P = Null$. À tout moment, l'action *DisplayProduct* peut être entrelacée par l'environnement de manière à exécuter l'opération B **DisplayProduct_B**.

3.3.3 Vérification et raffinement

Le principal problème de l'approche CSP || B est la cohérence du modèle, autrement dit, les parties B et CSP de la spécification ne doivent pas être contradictoires. Par exemple, les contrôleurs CSP ne doivent introduire ni divergence, ni blocage lors de l'exécution des opérations B. Une technique de vérification est proposée dans l'approche CSP || B afin d'assurer cette notion de cohérence. Elle consiste en deux étapes :

1. vérifier que, pour toute machine M_i contrôlée par un processus P_i, le couple $P_i \,||\, M_i$ est libre de divergence,
2. vérifier que la combinaison des contrôleurs $||_i\ P_i$ est libre de blocage.

Les auteurs de l'approche CSP || B ont montré dans [ST02] que ces deux étapes étaient suffisantes pour assurer la cohérence du modèle lorsque les contrôleurs utilisent la syntaxe décrite dans la section 3.3.1. Comme notre exemple ne concerne qu'une paire machine B/processus CSP, nous n'illustrons que le premier cas dans cette thèse. Le deuxième cas est décrit dans [ST02].

Dans CSP || B, les opérations B sont de la forme PRE THEN END. Elles sont donc exécutables à tout moment, même lorsque les préconditions ne sont pas vérifiées. L'exécution d'opérations B n'entraîne donc pas de blocage. En revanche, si une opération est appelée alors que ses préconditions ne sont pas vérifiées, le comportement de l'opération n'est pas prévu par la spécification et par conséquent, elle peut entraîner une divergence du système.

Pour montrer qu'un couple $P_i \parallel M_i$ est libre de divergence, il faut vérifier que, pour chaque appel d'opération B de M_i dans le processus P_i, les préconditions de l'opération sont vérifiées. Cette propriété est assurée grâce à un invariant dans chaque boucle récursive du contrôleur P_i, noté CLI, qui vérifie, pour tout appel récursif $S(p)$ dans P_i, que :

$$CLI \wedge I \Rightarrow [\{BBODY_{S(p)}\}]CLI$$

où la séquence d'opérations $\{BBODY_{S(p)}\}$ est issue de la traduction en B de l'expression de processus associée à $S(p)$ et I est l'invariant de la machine M_i. Cette propriété signifie qu'à chaque appel récursif du processus, la séquence d'opérations B correspondantes préserve l'invariant de boucle CLI. La traduction des expressions de processus en B est définie par des règles de traduction décrites dans [TS99].

Pour illustrer la vérification de l'absence de divergence, nous considérons l'exemple de processus CSP sans entrelacement décrit dans la section 3.3.2. L'invariant de boucle CLI est alors de la forme $c \in 0..2$, où c est une variable de contrôle du système qui correspond aux états 0 (*Inactive*), 1 (*Active*) ou 2 (*Created*). La détermination de cet invariant n'est pas toujours immédiate et demande souvent une analyse du système. Dans ce cas, l'utilisation d'outils comme FDR [For97] peut s'avérer pratique.

Dans le cas de plusieurs combinaisons $P_i \parallel M_i$ en concurrence, la cohérence est vérifiée en analysant les blocages de la combinaison $||_i\ P(i)$. Schneider et Treharne montrent que si chaque $P_i \parallel M_i$ est libre de divergence, il suffit alors de vérifier que la combinaison $||_i\ P(i)$ est libre de blocage pour prouver la cohérence du système combiné dans son ensemble [ST02]. Ces propriétés sont possibles grâce aux gardes et aux assertions qui permettent d'analyser et de traiter les couples $P_i \parallel M_i$ et la combinaison $||_i\ P(i)$ indépendamment les uns des autres.

L'analyse des blocages de $||_i\ P(i)$ est un exercice difficile et indépendant du cadre des combinaisons de spécifications formelles. Le principal apport de cette méthode est la compositionnalité de la vérification. La possibilité de vérifier chaque système combiné $P_i \parallel M_i$ et ensuite la combinaison $||_i\ P(i)$ de manière indépendante permet de simplifier les analyses et les calculs.

Le raffinement n'est pas explicitement traité dans cette approche. Dans le cadre de la méthode B, il est toutefois possible de raffiner la machine M d'une combinaison $P \parallel M$.

3.3.4 Adéquation avec les SI

Nous avons déjà mis l'accent sur les difficultés de l'utilisation de CSP pour spécifier les SI (voir section 3.2.5). Dans l'approche CSP || B, les machines B sont associées à des processus CSP qui contrôlent l'exécution de leurs opérations. La spécification est donc divisée en deux parties : CSP et B. Comme dans l'approche csp2B, le langage CSP est restreint. En particulier, l'entrelacement, la composition parallèle et la quantification sont limités. Contrairement à csp2B, une sémantique n'est pas définie pour simuler les expressions de processus CSP en B.

CSP || B diffère de csp2B dans la manière de contrôler les opérations B :

- Dans csp2B, les opérations B sont gardées (*i.e.,* elles sont de la forme SELECT THEN END). Par conséquent, une opération B ne peut être exécutée que si la garde est satisfaite. Les gardes des opérations sont générées à l'aide de la traduction des processus CSP en B.
- Dans CSP || B, les opérations B ont des préconditions (*i.e.,* elle sont de la forme PRE THEN END). Ainsi, une opération B est toujours disponible, mais le système peut diverger si une opération est exécutée en dehors de ses préconditions. Il faut donc vérifier que les préconditions sont cohérentes par rapport aux processus CSP.

L'approche CSP || B fournit des techniques pour vérifier la cohérence du modèle, en vérifiant que chaque paire machine B/processus CSP n'introduit pas de divergence. Elle fournit aussi des techniques pour vérifier l'absence de blocage dans l'ensemble du système.

Pour spécifier des SI, l'approche CSP || B a besoin de quelques extensions [ETLF04], principalement à cause de la distinction entre les contrôleurs de CSP et les machines à états de B. Si chaque type d'entité est représenté par une machine B, alors les interactions entre les entités du même type ne peuvent pas être prises en comptes par la partie CSP, puisqu'il n'y a qu'un contrôleur par machine B. Pour considérer des interactions entre des entités du même type, le contrôleur doit être décrit par des expressions CSP comprenant des opérateurs d'entrelacement et de composition parallèle et pouvant être quantifiées. Dans ce cas, il est possible de spécifier des SI avec CSP || B [ETLF04], mais de nouvelles obligations de vérification doivent être définies afin de prendre en compte ces nouveaux opérateurs. De plus, les SI contiennent généralement des milliers d'entités pour chaque type d'entité. La vérification peut donc entraîner une explosion combinatoire de l'espace d'états à analyser.

3.4 CSP-OZ

CSP-OZ [Fis00] appartient au deuxième groupe de combinaisons (création d'un nouveau langage). CSP-OZ est un langage de spécification formel, orienté objet, et qui intègre la plupart des concepts de CSP (voir section 2.3) et Object-Z (section 2.2.3). L'idée de cette approche est d'identifier une classe Object-Z à un processus CSP. Les opérations d'une classe Object-Z peuvent être contrôlées par une expression de processus de type CSP. Les propriétés de vivacité et de sûreté peuvent toutes deux être vérifiées en CSP-OZ. Le raffinement est supporté grâce à des règles de simulation.

3.4.1 Syntaxe

Le langage CSP-OZ est une extension du langage CSP_Z, un dialecte CSP utilisant la syntaxe Z (voir section 2.2.2) pour décrire les expressions de processus et les données, qui intègre en outre des descriptions orientées objet issues du langage Object-Z. Nous décrivons dans un premier temps CSP_Z, puis le langage CSP-OZ.

CSP_Z La syntaxe et la sémantique de CSP_Z sont détaillées dans [Fis00]. Le langage CSP_Z est une extension du langage Z qui permet de décrire des processus CSP avec une syntaxe proche de Z. Une spécification Z est une séquence de paragraphes (ou déclarations) regroupés dans des schémas (voir section 2.2.2). CSP_Z étend la syntaxe de Z par l'ajout de deux nouveaux types de paragraphe, les canaux et les processus :

$$\mathit{Paragraph}_{CSPZ} \quad ::= \quad \mathit{Paragraph}_Z \mid \mathit{Channel} \mid \mathit{Process}$$

Le paragraphe *Channel* permet de définir les canaux de communication des processus et d'introduire les événements possibles. Un canal est spécifié par :

$$\mathit{Channel} \quad ::= \quad \text{chan } \mathit{Name} \; [\, : \mathit{Expr}]$$

où *Expr* est une expression Z qui déclare les types des paramètres d'entrée et de sortie.

Le paragraphe *Process* permet de spécifier les processus en les associant à des expressions du langage *Expr* :

$$\mathit{Process} \quad ::= \quad \mathit{DefProc} = \mathit{Expr}$$

où *DefProc* permet d'identifier le processus spécifié.

Ces déclarations sont appliquées sur des constructeurs Z standard, mais elles nécessitent également une extension de la grammaire des expressions pour décrire les processus. En particulier, *ProcExpr* est le sous-ensemble des expressions *Expr* qui est utilisé pour définir les expressions de processus. Les processus de base sont *STOP* (le processus de blocage), *SKIP* (le processus qui ne fait rien) et *DIVER* (le processus qui diverge). Les expressions *ProcExpr* utilisent la plupart des opérateurs CSP : préfixe ($\rightarrow$), choix externe ($\Box$) et interne ($\sqcap$), composition parallèle ($||$) et entrelacement ($|||$). La quantification des opérateurs binaires est permise.

Par exemple, un processus de la forme :

$$\mathit{Alternative} = \mathit{On} \rightarrow (\mathit{DIVER} \,\Box\, \mathit{STOP})$$

où *On* est défini par :

$$\texttt{chan } \mathit{On}$$

représente un processus qui exécute tout d'abord l'événement *On* puis se comporte comme *DIVER* ou *STOP*.

CSP-OZ Le langage CSP-OZ étend le langage CSP_Z afin de rajouter la notion de classe dans la syntaxe. La grammaire CSP-OZ est de la forme :

$$\mathit{Paragraph}_O \quad ::= \quad \mathit{Paragraph}_C \mid \mathit{Class}_O \mid \mathit{Class}_C$$

où $Paragraph_C$ est la grammaire du langage CSP_Z, $Class_C$ une classe CSP-OZ et $Class_O$ une classe Object-Z. De manière informelle, la classe CSP-OZ est la version orientée objet d'un processus, tandis que la classe Object-Z est utilisée pour décrire les aspects données d'un système. De plus, les classes Object-Z du langage CSP-OZ jouent un rôle dans l'instanciation des objets en CSP-OZ, puisque les classes peuvent hériter ou être des instances de classes Object-Z.

Dans la syntaxe de CSP-OZ, les classes Object-Z ($Class_O$) encapsulent les données suivantes :

- une liste de visibilité : les items visibles depuis l'extérieur de la classe,
- une liste des classes dont la classe hérite,
- une liste de définitions locales,
- un schéma décrivant l'état de l'objet,
- un schéma initial spécifiant l'état initial,
- une liste des opérations qui peuvent modifier l'état de l'objet.

Une telle classe ne peut hériter que d'une autre classe Object-Z.

Les classes CSP-OZ ($Class_C$) sont similaires aux classes Object-Z, mais elles ne contiennent pas de liste de visibilité. En revanche, une classe CSP-OZ contient :

- une interface,
- une liste de processus CSP_Z.

L'interface d'une classe CSP-OZ permet de décrire tous les liens de communications de la classe, comme les méthodes de la classe (dans une classe Object-Z, elles sont contenues dans la liste de visibilité) mais aussi les méthodes des autres objets que la classe peut utiliser. Elle contient donc l'alphabet qu'une classe CSP-OZ peut utiliser pour communiquer avec les autres classes. La liste de processus CSP_Z constitue une des particularités de ce langage, puisque la classe CSP-OZ intègre une partie CSP dans sa description. Elle permet de contraindre l'ordre d'exécution des méthodes de la classe.

Pour relier les opérations d'une classe Object-Z aux événements de processus d'une classe CSP-OZ, il existe en CSP-OZ plusieurs mots-clés. Si *Op* désigne une opération, alors :

- `enable_`*Op* définit la garde de l'opération,
- `effect_`*Op* est l'effet de l'opération *Op*,
- et `com_`*Op* est utilisé pour définir à la fois la garde et l'effet de *Op*.

Il est alors possible d'exprimer les caractéristiques d'une opération (garde, effet) en fonction des événements d'un processus CSP_Z.

3.4.2 Spécification

Si on reprend l'exemple de la figure 3.1, deux classes sont définies dans cette approche pour représenter le système : une classe Object-Z et une classe CSP-OZ. Un schéma d'état permet de définir par ailleurs les ensembles abstraits utilisés par l'ensemble des classes du système. Dans cet exemple, on a besoin uniquement d'un type pour les produits : $[PRODUCTS]$.

La classe Object-Z permet de spécifier les types de données et les opérations du système. On considère que le système est représenté par une seule classe qui reçoit les commandes depuis l'extérieur. La classe définie représente les produits. La spécification, qui est proche des machines B définies dans les approches précédentes, est présentée dans la figure 3.5.

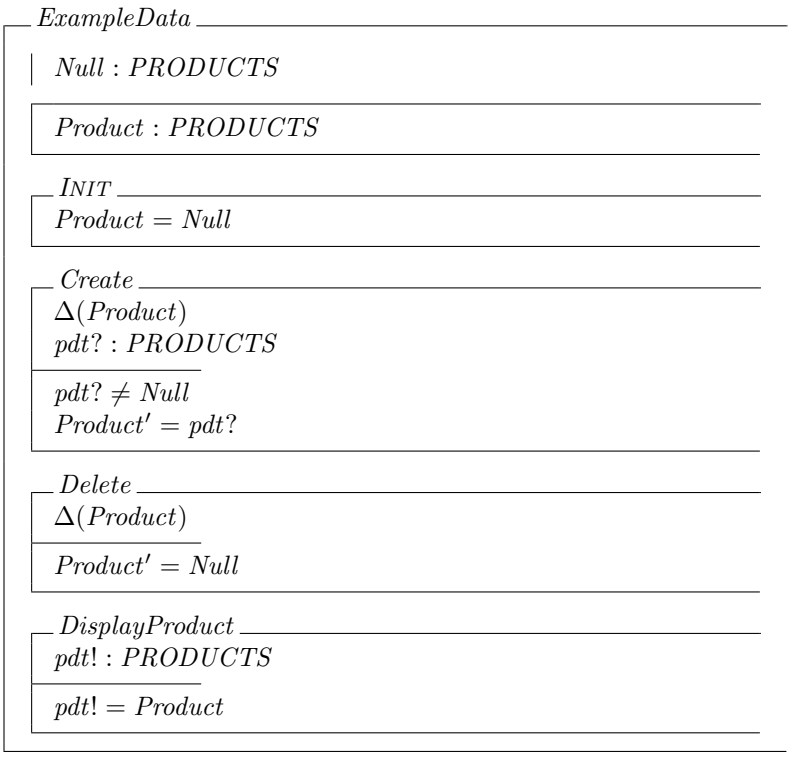

FIG. 3.5 – Spécification CSP-OZ : classe *ExempleData*

Dans cette classe, la variable d'état *Product* représente le produit existant, ou alors elle a pour valeur la constante *Null* lorsqu'il n'existe aucun produit. Ainsi, *Product* et *Null* sont de type *PRODUCTS*. Trois schémas d'opération sont définis : *Create*, *Delete* et *DisplayProduct*. La notation Δ est utilisée, le cas échéant, pour indiquer quelles sont les variables d'état qui sont modifiées lors de l'exécution de l'opération. Dans notre exemple, seule la variable *Product* est concernée dans les opérations *Create* et *Delete*. L'opération *Create* a un paramètre d'entrée *pdt ?*, dont le type est *PRODUCTS*. L'opération *DisplayProduct* a un paramètre de sortie *pdt !*.

Les classes CSP-OZ ont la particularité de pouvoir spécifier des processus avec le langage $\mathrm{CSP_Z}$. La classe *ExampleProc* définit le processus principal qui modélise le comportement du système. Les opérations *Create*, *Delete* et *DisplayProduct*, ainsi que les canaux de communication *On* et *Off*, sont déclarés dans l'interface de cette classe, afin qu'ils puissent être utilisés par le processus *Main* :

ExempleProc

`method` $Create[pdt? : PRODUCTS]$
`method` $Delete$
`method` $DisplayProduct[pdt! : PRODUCTS]$
`chan` On
`chan` Off

$Main = On \rightarrow Active$
$Active = (ProductCycles \;|||\; Requests) \Box (Off \rightarrow Main)$
$Requests = (DisplayProduct!pdt \rightarrow Requests) \Box SKIP$
$ProductCycles = Create?pdt \rightarrow Delete \rightarrow Active$

Une méthode comme *Create* est une opération qui doit être implémentée par la classe elle-même ou bien par une des classes qui héritent d'elle. Un canal de communication comme *On* est une opération qui doit être implémentée par d'autres classes, mais qui peut être utilisée dans la classe.

La classe *ExempleProc*, si elle est considérée indépendamment des autres classes, permet de contraindre l'ordonnancement des méthodes définies dans son interface, mais ne fournit pas d'implémentation. Une dernière classe Object-Z appelée *Example* est alors définie pour implémenter les méthodes de la classe *ExampleProc* :

Example

`method` $Create[pdt? : PRODUCTS]$
`method` $Delete$
`method` $DisplayProduct[pdt! : PRODUCTS]$
`chan` On
`chan` Off
`inherit` $ExempleData, ExempleProc$

`com_`$Create = Create$
`com_`$Delete = Delete$
`com_`$DisplayProduct = DisplayProduct$

Cette classe hérite (mot-clé `inherit`) à la fois des classes *ExampleData* et *ExampleProc*. Cette spécification signifie que les événements *Create*, *Delete* et *Display-*

Product de la classe CSP-OZ *ExampleProc* ont les mêmes gardes et les mêmes effets que les opérations homonymes, définies dans la classe *ExampleData*.

3.4.3 Vérification et raffinement

Un éditeur graphique ainsi qu'un *type-checker* ont été implémentés pour CSP-OZ [vG99]. Les spécifications Object-Z et CSP sont faciles à traduire en CSP-OZ. En revanche, comme CSP-OZ introduit de nouveaux constructeurs dans la syntaxe, quelques modifications doivent être réalisées sur la spécification CSP-OZ, avant de pouvoir utiliser des *model-checkers*, comme FDR [For97] pour CSP et Moby/OZ [Cor06] pour Object-Z.

Le raffinement CSP-OZ est basé sur les relations de raffinement définies en Z et en CSP. Fischer a montré que si des classes Object-Z sont raffinées pour les données selon les règles définies dans [Fis00], alors les processus associés sont raffinés au sens CSP. Ainsi, les deux notions de raffinement sont cohérentes.

3.4.4 Adéquation avec les SI

CSP-OZ est un langage formel de spécification qui permet de décrire des systèmes à l'aide d'une combinaison des formalismes de CSP et Object-Z. Un langage intermédiaire, CSP_Z, permet de décrire des propriétés dynamiques à l'aide de processus utilisant à la fois les principaux opérateurs CSP et une syntaxe Z. La principale difficulté concerne l'utilisation de ce langage. Dans la pratique, un nouveau langage requiert une nouvelle méthodologie pour spécifier les systèmes, de nouveaux outils d'assistance et une adaptation de l'utilisateur aux concepts proposés par la nouvelle approche. D'un autre côté, CSP-OZ permet de spécifier à la fois des propriétés statiques et dynamiques du système. Le même langage regroupe en effet deux formalismes complémentaires, inspirés de CSP et Object-Z. Ainsi, une classe CSP-OZ est une classe Object-Z associée à un processus CSP_Z.

Même si la partie CSP n'est pas limitée comme dans les approches précédentes, nous avons déjà discuté des faiblesses de CSP concernant la spécification des SI. Dans CSP-OZ, le processus contrôle principalement les interactions de la classe avec les autres classes CSP-OZ. Toutefois, il est possible de spécifier le comportement entre plusieurs objets de la même classe en utilisant l'opérateur d'entrelacement quantifié. Par exemple, si une classe paramétrée $Product(pId)$ est définie pour représenter l'ensemble des produits, alors les contraintes qui doivent être vérifiées par chaque entité de ce type sont spécifiées par un processus CSP_Z dans cette classe, tandis que le comportement entre les différentes entités est spécifié dans une autre classe, dans laquelle les méthodes de $Product(pId)$ sont visibles. L'expression de processus est alors de la forme suivante :

$$Main = (\ |||\ pId : PRODUCTS \bullet Product(pId)\ )\ \ ||_{\{m_1,...,m_k\}}\ \ P$$

où $PRODUCTS$ désigne l'ensemble des produits, P est une autre expression de processus, et $m_1, ..., m_k$ est l'ensemble des méthodes sur lesquelles les processus se synchronisent.

Object-Z semble être un langage formel adapté à la spécification du modèle de données de SI dans un style orienté objet. En Object-Z, les attributs d'une classe ne peuvent être modifiés que par les méthodes indiquées. La liste de

visibilité est par conséquent une description importante des classes Object-Z, puisqu'elle détermine quelles sont les méthodes accessibles par les autres classes.

L'approche CSP-OZ se distingue des autres combinaisons qui identifient une opération basée sur les états avec des événements, en permettant deux types d'identification dans le langage :

- avec un seul événement par opération, *i.e.,* chaque opération est considérée comme un canal de communication (de manière analogue à CSP || B, section 3.3, par exemple) ;
- avec deux événements par opération, *i.e.,* chaque opération est associée à deux événements : un pour la lecture des paramètres d'entrée, l'autre pour l'écriture des paramètres de sortie.

Cette dernière option peut être intéressante pour traiter le calcul des sorties d'un SI ou bien pour gérer les messages d'erreur lorsqu'une opération n'est pas dans les bonnes préconditions.

3.5 Circus

Circus [WC02] appartient au deuxième groupe de combinaisons (création d'un nouveau langage). Il s'agit d'un langage formel basé sur Z (section 2.2.2) CSP (section 2.3) et Action Systems (voir section 2.2.1), qui intègre aussi le calcul de raffinement de Back [BvW98]. La sémantique s'inspire de la théorie unifiée de programmation [HJ98]. L'idée de Circus est de distinguer les transitions d'états des communications définies dans le système d'actions qui représente le comportement du système. Les propriétés de vivacité peuvent être vérifiées sur le système d'actions, alors que les propriétés de sûreté sont prouvées sur les schémas d'état. Circus fournit aussi une relation de raffinement unifiée. La notion de raffinement est en effet complexe dans le cadre d'une méthode intégrée, car les raffinements au sens Z et CSP n'utilisent pas les mêmes modèles sémantiques. Pour pallier ce manque, Circus s'appuie sur la théorie unifiée de programmation, qui permet d'interpréter les deux raffinements dans un modèle unique.

3.5.1 Syntaxe

Une spécification Circus est structurée en termes de processus. Chaque processus est représenté par un état, décrit par un schéma Z, et par un comportement, décrit par un système d'actions. Des schémas Z classiques sont utilisés pour représenter les états, les initialisations et les opérations. Les canaux de communication, similaires à ceux de CSP, sont déclarés par la clause **channel**. Les processus sont spécifiés de la manière suivante :

process *Name* $\triangleq$ *ProcExpr*

où *Name* est le nom du processus et *ProcExpr* est une expression de processus du langage Circus. Les expressions de processus sont définies en Circus par :

begin *Declaration* • *Action* **end**

où *Declaration* désigne un ensemble de paragraphes Z standard ou de déclarations de processus.

Les actions en Circus sont décrites par des schémas Z, des commandes gardées et des opérateurs CSP. Les actions de base sont : *Stop* (l'action qui arrête l'exécution du processus), *Skip* (l'action qui ne fait rien) et *Chaos* (l'action qui diverge). Les actions gardées sont de la forme : *Predicate&Action*. Les opérateurs sur les actions sont : les choix interne ($\sqcap$) et externe ($\Box$), la synchronisation paramétrée ($|\ [\Delta]\ |$, similaire à $||$, mais synchronisée uniquement sur les actions de l'ensemble Δ), l'entrelacement ($|||$), la restriction ($\setminus$), la séquence ($;$), et les communications (de la forme $c\,?x \rightarrow A$ ou $c\,!x \rightarrow A$). Circus permet enfin d'utiliser les μ-expressions de Z pour définir des actions de manière récursive :

$$\mu\, N \bullet A(N)$$

où N est un identifiant et $A(N)$ est une expression d'action qui dépend de N. Cette expression représente le plus petit point fixe tel que : $N = A(N)$.

Les processus de la forme *ProcExpr* peuvent être composés entre eux avec les opérateurs suivants : les choix interne ($\sqcap$) et externe ($\Box$), la synchronisation paramétrée ($|\ [\Delta]\ |$), l'entrelacement ($|||$), la restriction ($\setminus$) et la séquence ($;$). Les processus peuvent également être indexés avec l'opérateur $\odot$ comme suit. Le processus $i : T \odot P$, où T est un ensemble et P est une expression de processus, se comporte comme P, mais agit sur différents canaux ; chaque occurrence d'un canal de communication c est alors indexée par $i \in T$, autrement dit, elle est remplacée par c_i.

3.5.2 Spécification

3.5.2.1 Déclaration

Notre exemple de référence peut être spécifié en Circus de la manière suivante. Le type des produits est défini par : $[PRODUCTS]$. Le type *STATES* est un ensemble énuméré qui contient les trois états du système :

$$STATES = Stop \mid Active \mid Created$$

Un canal de communication appelé *In* permet de déclarer le nouveau produit à créer. Le canal *Out* est utilisé pour retourner le produit courant.

channel $In, Out : PRODUCTS$

Une constante *Null* est définie par :

$$\mid\ Null : PRODUCTS$$

Le processus principal est appelé *Machine* :

process $Machine \triangleq$
begin .../* schémas Z et système d'actions indiqués ci-dessous */

Les deux premiers schémas Z, *State* et *StateInit*, définissent respectivement l'espace d'états et l'initialisation du système. Ils sont similaires aux schémas d'état définis pour CSP-OZ (section 3.4). Toutefois, une variable d'état supplémentaire est spécifiée ici. Ainsi, la variable *state* représente l'état courant du système.

State

$state : STATES$
$product : PRODUCTS$

StateInit

$State$

$state' = Stop$
$product' = Null$

Les schémas suivants définissent les opérations du système : *On*, *Off*, *Create*, *Delete* et *DisplayProduct*.

On

$\Delta State$

$state = Stop$
$state' = Active$
$product' = product$

Off

$\Delta State$

$state = Active$
$state' = Stop$
$product' = product$

Create

$\Delta State$
$pdt? : PRODUCTS$

$state = Active \land pdt? \neq Null$
$state' = Created$
$product' = pdt?$

Delete

$\Delta State$

$state = Created$
$state' = Active$
$product' = Null$

DisplayProduct

$pdt! : PRODUCTS$

$pdt! = product$

Une action intermédiaire, *ProductCycles*, représente le cycle de vie de chaque produit :

$$\begin{array}{l} ProductCycles \triangleq \\ \mu\, X \bullet In?pdt : PRODUCTS \ \rightarrow \ ((pdt \neq Null \ \&\ Create\,;Delete) \\ \qquad \Box(pdt = Null \ \&\ Skip))\,;X \end{array}$$

L'action *ProductCycles* reçoit tout d'abord un produit *pdt* en entrée par le canal *In*. Si *pdt* est différent de *Null*, alors les opérations Z *Create* et *Delete* sont exécutées consécutivement. Sinon, rien ne se passe et le processus est itéré. Cette première action intermédiaire permet d'éviter un blocage au cas où une valeur indésirable serait donnée pour le paramètre d'entrée *pdt*. Une autre action, *Requests*, permet de spécifier que l'opération Z *DisplayProduct* peut être exécutée un nombre arbitraire fini de fois :

$$\begin{array}{l} Requests \triangleq \\ \mu\, Y \bullet (DisplayProduct\,;Out!pdt \ \rightarrow \ Y)\Box Skip \end{array}$$

Le résultat de l'opération *DisplayProduct* est communiqué par le canal *Out*.

3.5.2.2 Action principale

L'action principale spécifie le comportement global du système. Cette action utilise tous les paragraphes Z présentés dans la section 3.5.2.1.

$$StateInit\,;(\mu\, V \bullet (On\,;((ProductCycles \ |||\ Requests)\Box(Off\,;V))))$$

Ce système d'actions a le comportement suivant. Le schéma d'initialisation *StateInit* est exécuté en premier. Ensuite, une μ-expression définit l'action récursivement. Après avoir exécuté l'action *On*, il existe deux alternatives : soit le processus exécute *ProductCycles*, dont les actions sont entrelacées avec les actions de *Requests*, soit il exécute l'action *Off* et le système est prêt à être activé de nouveau.

3.5.3 Vérification et raffinement

La sémantique de Circus est basée sur la théorie unifiée de programmation [HJ98]. Dans leur unification, Hoare et al. s'appuient sur la théorie des relations pour représenter les spécifications et les implémentations. Ces travaux ont pour but d'apporter une base commune à toutes les branches de la programmation quels que soient les paradigmes utilisés. Par exemple, la séquence est représentée par une composition relationnelle et le non-déterminisme par une disjonction. Des concepts comme la correction ou le raffinement d'une spécification sont interprétés comme des inclusions de relations. Cette base permet ainsi de raisonner sur une représentation des langages avec les lois du calcul relationnel.

Dans [WC02], Woodcock et al. utilisent cette théorie pour définir la sémantique de Circus, décrite en Z. Un analyseur syntaxique existe pour les spécifications Circus. Un *model-checker* basé sur FDR [For97], un *model-checker* du langage CSP, et sur ProofPower, un prouveur de Z, est actuellement en cours de développement [CSW05].

Contrairement aux autres approches présentées dans ce chapitre, Circus propose une relation de raffinement qui intègre les différents aspects du langage : les actions, les processus et les données. Woodcock et al. proposent dans [SWC02, CSW02] des règles de raffinement pour décomposer une spécification en plusieurs sous-composants et pour vérifier ensuite l'ensemble de ces composants. Une stratégie a été définie pour dériver une spécification abstraite en un système distribué. Tout processus Circus est composé d'une action principale et d'un espace d'états. Dans la sémantique unifiée de Circus, un processus P est raffiné par un processus Q s'il existe une relation de simulation entre les espaces d'états et si l'action principale de P est raffinée par celle de Q.

3.5.4 Adéquation avec les SI

Le langage Circus a été créé dans le but d'unifier les calculs de raffinement de Z, CSP et des Action Systems. À l'instar de CSP-OZ (section 3.4), une nouvelle syntaxe et une nouvelle sémantique ont été définies. Pour les mêmes raisons, l'application de Circus requiert une nouvelle façon d'appréhender la spécification des systèmes. En particulier, comme la sémantique est unifiée, de nouvelles techniques de vérification et de nouveaux outils doivent être créés pour analyser les spécifications Circus.

Comme la définition d'une théorie unifiée pour le raffinement est la motivation première de Circus, le langage dissocie les transitions d'états des systèmes d'actions afin d'améliorer les techniques de raffinement. Par conséquent, les propriétés basées sur les états peuvent être vérifiées séparément. En Circus, les systèmes d'actions permettent essentiellement de décrire les propriétés d'ordonnancement des événements, car les transitions d'états sont encapsulées dans les schémas Z. Pour éviter les blocages, il faut alors analyser les gardes des actions et vérifier qu'elles sont cohérentes avec les préconditions des schémas Z correspondants.

Pour permettre le raffinement d'un processus en plusieurs sous-processus en parallèle, l'expressivité du langage Circus pour les systèmes d'actions est limitée. En particulier, les actions entrelacées d'un même processus peuvent partager le même espace d'états, mais elles ne peuvent pas modifier les mêmes variables d'état. Par conséquent, la description de plusieurs entités du même type qui seraient en concurrence entre elles est difficile à prendre en compte avec l'opérateur d'entrelacement quantifié sur les actions. En revanche, l'opérateur d'entrelacement sur les processus, couplé avec l'opérateur $\odot$, peut être utilisé pour représenter le comportement d'un type d'entité. Par exemple, si le processus *Product* définit le comportement d'un produit, alors l'expression $|||\ pdt : PRODUCTS \odot Product\lfloor pdt \rfloor$ représente l'entrelacement de plusieurs instances de *Product*, indexées par *pdt*. De plus, une extension orientée objet de Circus, appelée OhCircus [CSW05], a été définie. Cependant, quelques restrictions demeurent, en particulier sur les appels de méthodes, afin de conserver la propriété de monotonicité du raffinement.

En conclusion, le principal bénéfice du langage Circus est la définition d'une relation de raffinement basée sur une sémantique unifiée. Ainsi, Circus est la seule approche de cet état de l'art à proposer un raffinement pour des combinaisons de spécifications. La stratégie de raffinement consiste en des décompositions successives des processus. Le raffinement de Circus inclut à la fois les raffinements de processus, d'actions et de données. En particulier, si un processus est

décomposé en plusieurs sous-processus en paralèlle, et si un schéma d'état doit être utilisé par l'ensemble de ces sous-processus, alors le schéma est divisé et des canaux de communications sont introduits au cours du raffinement afin de partager les différentes variables d'état du schéma initial. Une telle stratégie peut s'avérer intéressante dans le cadre des SI pour modéliser le caractère concurrentiel ou bien l'implémentation de systèmes distribués.

3.6 PLTL et B événementiel

Cette approche [Dar02] appartient au troisième groupe de combinaisons (vérification de propriétés). Pour des raisons pratiques, nous désignerons cette approche par PLTL-Event B dans la suite de ce chapitre. Les systèmes sont spécifiés en B événementiel (section 2.2.5), qui est une extension basée sur les événements du langage B (section 2.2.4). L'idée est d'ajouter des propriétés de la logique temporelle PLTL [Pnu81] à la spécification B. Les techniques de preuve utilisées en B sont couplées avec des techniques de *model-checking* pour vérifier ces propriétés sur le modèle. Les propriétés de sûreté sont prouvées de manière usuelle en B. L'approche fournit aussi des règles de reformulation des propriétés PLTL qui sont cohérentes avec le raffinement défini en B événementiel.

3.6.1 Syntaxe

Deux langages existants sont utilisés : B événementiel et PLTL.

Systèmes d'événements en B. Un système d'événements est décrit en B par une machine comprenant les clauses suivantes :

- CONSTRAINTS : prédicats de la machine,
- SETS : ensembles abstraits et énumérés,
- INVARIANT : propriétés d'invariance du système,
- INITIALISATION : initialisation du système,
- EVENTS : description des événements du système.

Les propriétés dynamiques peuvent être définies en B sous la forme d'invariants dynamiques et de modalités. Un invariant dynamique est de la forme suivante :

$$DYNAMICS\ \mathcal{P}(V, V')$$

où $\mathcal{P}(V, V')$ est un prédicat qui permet de caractériser comment les variables V sont autorisées à évoluer vers V' lors de l'appel d'un événement du système. Une modalité B est de la forme :

SELECT P Leadsto Q [WHILE $e_1, ..., e_n$]
INVARIANT J
VARIANT V
END

ou bien de la forme :

SELECT P Until Q [WHILE $e_1, ..., e_n$]
INVARIANT J
VARIANT V
END

La modalité "P Leadsto Q" signifie que si P est vraie, alors les séquences d'événements indiquées dans la clause WHILE conduisent forcément à Q. La modalité "P Until Q" signifie que "P Leadsto Q" est satisfaite et que P est vraie tant que Q ne l'est pas.

PLTL. La logique temporelle PLTL est un formalisme mieux adapté que le B événementiel pour exprimer des propriétés temporelles. Les opérateurs temporels PLTL utilisés dans cette approche sont : $\bigcirc$ (état suivant), $\Diamond$ (fatalement), $\Box$ (toujours), Θ (état précédent), $\mathcal{U}$ (jusqu'à), $\mathcal{S}$ (depuis) et $\mathcal{W}$ (à moins que). Une sémantique opérationnelle de PLTL est définie dans [Dar02].

3.6.2 Spécification

Notre exemple de référence est spécifié en B événementiel de la manière suivante :

EVENT SYSTEM *Example*
SETS $PRODUCTS$; $EXState = \{Inactive, Active, Created\}$
CONSTANTS *Null*
PROPERTIES $Null \in PRODUCTS$
VARIABLES *Product*, *Output*, *EX*
INVARIANT $Product \in PRODUCTS \wedge Output \in PRODUCTS$
  $\wedge\ EX \in EXState$
INITIALISATION
  $Product := Null \parallel Output := Null \parallel EX := Inactive$
EVENTS
  On =
  SELECT $EX = Inactive$
  THEN $EX := Active$
  END ;

  Off =
  SELECT $EX = Active$
  THEN $EX := Inactive$
  END ;

  Create =
  SELECT $EX = Active$
  THEN
    ANY *xx* WHERE $xx \in PRODUCTS \wedge xx \neq Null$
    THEN
      $EX := Created \parallel$
      $Product := xx$
    END
  END ;

  Delete =
  SELECT $EX = Created$
  THEN
    $EX := Active \parallel$

Product := *Null*
END ;

DisplayProduct =
SELECT *EX* ∈ {*Active*, *Created*}
THEN
Output := *Product*
END
END

Contrairement à l'approche B classique, il n'est pas possible d'exprimer précisément quel produit est effectivement créé, car les événements n'ont ni entrée, ni sortie. En B événementiel, les systèmes d'événements sont en effet considérés comme "fermés" : ainsi, les entrées et les sorties du système logiciel à modéliser sont représentées comme des variables d'état internes au système d'événements B. À travers les événements **Create** et **Delete**, l'utilisateur ne peut qu'"observer" les changements d'états de la variable *Product*. On ne peut donc pas spécifier quels produits il souhaite ajouter. Cependant, la garde de **Create** garantit qu'un produit doit être supprimé avant qu'un nouveau produit ne soit créé. De manière analogue, la variable d'état *Output* représente la sortie de l'événement **DisplayProduct**.

L'objectif de cette approche est d'utiliser PLTL pour exprimer des propriétés temporelles sur ce système d'événements B. Par exemple, le seul état possible après *Inactive* est *Active* :

$$\Box(EX = Inactive \Rightarrow \bigcirc(EX = Active))$$

3.6.3 Vérification et raffinement

Comme les propriétés temporelles spécifiées avec la logique PLTL sont préservées par le raffinement défini en B événementiel, il est alors possible de reformuler ces propriétés à l'aide des nouvelles variables introduites au cours du raffinement des systèmes d'événements B. Dans sa thèse [Dar02], Darlot présente des règles de reformulation afin de raffiner les propriétés en logique temporelle PLTL. Il définit également une méthode qui combine à la fois preuve et *model-checking* pour raffiner des systèmes développés avec son approche.

La figure 3.6 est un résumé de la méthode proposée pour vérifier des propriétés PLTL sur des systèmes d'événements B. La méthode de vérification consiste en trois étapes :

1. Dans un premier temps, la propriété temporelle $P1$ est définie en logique PLTL sur le système abstrait $TS1$. Cette propriété est vérifiée par *model-checking*. Comme le système est abstrait, l'espace d'états à explorer est réduit et il y a peu de risque d'explosion combinatoire.
2. Ensuite, le système abstrait $TS1$ est raffiné en $TS2$. Cette étape est justifiée par preuve (technique usuelle en B) ou par un algorithme de vérification (sur les LTS). Darlot a montré que la propriété $P1$ est préservée sur le système raffiné $TS2$.
3. La troisième étape consiste à reformuler la propriété temporelle $P1$ en une propriété $P2$, en utilisant les règles de reformulation définies par Darlot dans [Dar02]. Chacune de ces règles est associée à des obligations de preuve

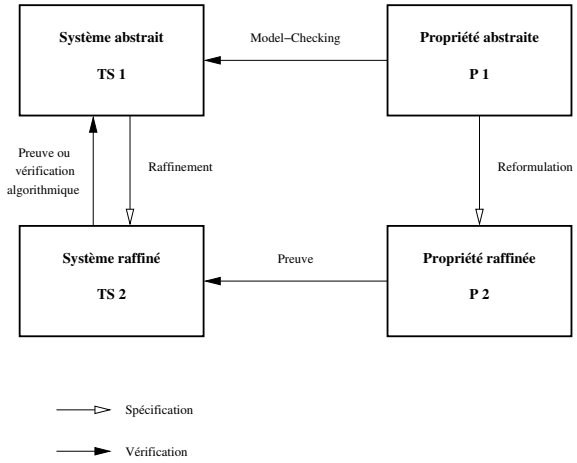

FIG. 3.6 – Vérification de propriétés PLTL sur des systèmes d'événements B

qui permettent d'assurer que la nouvelle propriété soit vérifiée. Ainsi, il est possible de prouver que $P2$ est satisfaite par $TS2$.

3.6.4 Adéquation avec les SI

Le B événementiel est un langage basé sur les états qui utilise des opérations gardées afin de représenter des systèmes d'actions. L'utilisation de gardes n'est pas vraiment adaptée dans le cadre de la spécification de SI. En effet, dans un SI, on s'attend à une réponse du système pour chaque appel d'opération ; il peut s'agir par exemple du résultat d'une requête ou bien d'un message d'erreur. Or les opérations gardées ne peuvent être exécutées que lorsque leur garde est satisfaite. Par conséquent, il est difficile de prendre en compte la gestion des réponses du système.

En B événementiel, les systèmes sont "fermés" et les événements (contrairement aux "opérations" basées sur les états du langage B) n'ont ni paramètre d'entrée, ni paramètre de sortie. La spécification de SI avec des modèles fermés semble difficile, puisqu'un SI peut impliquer de nombreux utilisateurs avec un grand nombre d'entrées ou des systèmes de gestion de bases de données distribués. D'un autre côté, le raffinement défini en B événementiel permet de définir de nouveaux événements et de décomposer un système d'événements en systèmes concurrents. De plus, le raffinement permet également de retrouver à la fin du cycle de développement le système logiciel en le distinguant de son environnement. C'est la raison pour laquelle les futurs paramètres d'entrée et de sortie sont représentés par des variables d'état internes lors des premières spécifications abstraites du système. La spécification de SI en B événementiel semble donc possible, mais demande un effort important pour tenir compte des nombreuses entrées et sorties potentielles du système.

Concernant l'exemple de référence, la création de plusieurs produits peut aisément être prise en compte. Dans ce cas, l'état *Created* n'est plus considéré dans l'ensemble énuméré $EXState$: $EXState = \{Inactive, Active\}$. La variable d'état *Product* est définie comme un sous-ensemble de $PRODUCTS$. Les évé-

nements **Create** et **Delete** sont alors spécifiés de la manière suivante :

Create =
SELECT $EX = Active \wedge Product \neq PRODUCTS$
THEN
  ANY xx WHERE $xx \in PRODUCTS - Product \wedge xx \neq Null$
  THEN
    $Product := Product \cup \{xx\}$
  END
END ;

Delete =
SELECT $EX = Active \wedge Product \neq \varnothing$
THEN
  ANY xx WHERE $xx \in Product$
  THEN
    $Product := Product - \{xx\}$
  END
END ;

Les nouvelles gardes permettent de considérer la création de plusieurs produits par le système. En particulier, la garde sur la variable locale xx empêche la création d'un produit existant par l'événement **Create**. De même, la garde sur xx de l'événement **Delete** permet d'éviter la suppression d'un produit qui n'existe pas.

En conclusion, l'approche PLTL-Event B a pour objectif de vérifier, sur des systèmes d'événements B, des propriétés temporelles décrites dans la logique PLTL. Les techniques de preuve B sont couplées avec des techniques de vérification. L'utilisation d'une logique temporelle comme PLTL peut s'avérer fort intéressante pour vérifier les propriétés dynamiques des SI, comme par exemple, des contraintes d'intégrité dynamiques.

3.7 Comparaison des approches

Dans les sections précédentes, nous avons présenté les différents couplages en les classant selon leurs objectifs. Ainsi, nous avons considéré trois groupes : spécification en plusieurs parties, création d'un nouveau langage et vérification de propriétés. Dans cette section, nous nous concentrons sur la manière dont les combinaisons sont définies. Nous avons identifié trois niveaux de combinaison entre les spécifications, selon leur sémantique sous-jacente :

1. **Unification :** Ce niveau de combinaison est le plus fort, puisqu'il implique les définitions d'une nouvelle syntaxe et d'une nouvelle sémantique. Le nouveau langage emprunte à d'autres langages les opérateurs qui semblent les plus utiles et la sémantique est unifiée. Cette approche a pour avantage de bien intégrer certaines notions comme le raffinement ou la vérification, mais elle a pour défaut l'impossibilité de réutiliser des spécifications ou des outils existants. Deux exemples présentés dans ce chapitre appartiennent à cette catégorie : CSP-OZ et Circus.
2. **Traduction :** Ce niveau de combinaison consiste à définir le sous-ensemble d'un langage dans la sémantique de l'autre. On peut le considérer comme

une combinaison “verticale” de langages de spécification. Ce niveau permet de conserver certaines des notations existantes, mais le langage dont la sémantique est redéfinie perd souvent de sa richesse, à cause des restrictions du langage cible. Ce type de combinaisons est parfois utilisé pour vérifier une propriété dynamique sur un modèle de la statique. Dans ce chapitre, deux exemples appartiennent à ce niveau de couplage : csp2B et PLTL-Event B.

3. **Juxtaposition :** Ce niveau de combinaison permet de considérer plusieurs vues du même système. On peut le définir comme une combinaison “horizontale” de langages de spécification. Le lien entre les sémantiques est généralement une identification d'une structure d'un langage avec une structure de l'autre. Par conséquent, il est souvent difficile d'analyser le modèle dans son ensemble. Ce niveau de couplage implique des problèmes comme la redondance ou l'incohérence de spécifications. Seul CSP || B appartient à cette catégorie de combinaisons parmi les approches présentées dans ce chapitre.

La complémentarité entre les descriptions des deux parties couplées dans la combinaison est un critère important. Si les deux langages sont orthogonaux en style de spécification, alors le modèle sera difficile à comprendre ou à vérifier dans son ensemble. Si les deux langages ont la même capacité d'expression, alors les spécifications risquent d'être redondantes ou incohérentes. D'un autre côté, des erreurs de spécification peuvent être détectées en décrivant deux fois les mêmes propriétés, mais dans des langages distincts, et en prouvant que les deux modèles sont cohérents.

Lorsque le sous-ensemble d'un langage est défini dans la sémantique de l'autre, une partie du pouvoir d'expression du premier langage peut être perdue par traduction. Par exemple, il est plus difficile de définir des propriétés d'ordonnancement en B qu'en CSP. La traduction de CSP vers B de l'approche csp2B a donc pour conséquence de perdre ce type d'information. Quand une simple identification de structure est réalisée comme dans les combinaisons de type juxtaposition, la vérification demande plus d'analyse afin de considérer le modèle dans son ensemble et pour éviter les problèmes de contradiction entre les spécifications.

3.7.1 Synthèse des approches présentées

Les tableaux 3.1 et 3.2 résument les caractéristiques des différentes approches présentées dans ce chapitre. Pour chaque combinaison, nous indiquons :

- son nom ;
- la section dans laquelle elle a été présentée ;
- les langages qu'elle réutilise ou dont elle est inspirée ;
- les raisons de sa création ;
- la sémantique sur laquelle elle est fondée ;
- les techniques de vérification qu'elle propose ou qu'elle utilise, s'il en existe. L'expression “m.c.” est utilisée comme abréviation pour *model-checking*.
- les techniques de raffinement dont elle dispose, s'il en existe ;
- les outils qu'elle utilise, s'il en existe ;
- notre évaluation de sa lisibilité : le symbole + signifie qu'une spécification décrite dans cette approche est très lisible, alors que − signifie que la

TAB. 3.1 – Comparaison des combinaisons - partie 1

Approche	csp2B [But99]	CSP || B [ST02]	CSP-OZ [Fis00]
Section	3.2	3.3	3.4
Langages	CSP B	CSP B	CSP Object-Z
But	outil de CSP vers B	processus CSP = contrôleur des opérations B	nouveau langage
Sémantique	opérationnelle avec LTS	dénotationnelle échecs stables échecs-divergences	opérationnelle et dénotationnelle
Vérification	m.c. en CSP, preuve en B	m.c. en CSP, preuve en B	*model-checking*
Raffinement	raffinement B	non	raffinement de données
Outils	FDR, outils B, csp2B	FDR, outils B	*type-checker*, m.c. à développer
Lisibilité	+	+	−
Adéquation SI	−	−	+
Classification : but	spécification en plusieurs parties	spécification en plusieurs parties	création d'un nouveau langage
Classification : sémantique	traduction	juxtaposition	unification

compréhension du modèle est difficile ;
- notre évaluation de son adéquation avec les SI : le symbole + signifie que les principales propriétés des SI peuvent être décrites par l'approche, alors que − signifie que la spécification des SI est difficile avec cette approche ;
- la classification selon son but (voir section 3.1) ;
- la classification selon sa sémantique (voir section 3.7).

3.7.2 Discussion

Pour chaque approche, il existe au moins un aspect intéressant concernant la spécification des SI. CSP-OZ est un langage formel orienté objet qui permet de contraindre les méthodes d'une classe par des expressions de processus. De plus, une méthode de classe peut être identifiée dans cette approche avec un ou deux événements. Dans le deuxième cas, il est notamment possible de distinguer exécution de la méthode et gestion des erreurs.

Circus est la seule approche de cet état de l'art qui définisse une relation de raffinement intègrant à la fois raffinement de données et raffinement de processus. En effet, le raffinement proposé dans les autres approches, s'il existe, n'implique qu'une partie de la spécification et le reste du modèle reste cohérent par monotonicité. Plus généralement, les langages CSP-OZ et Circus

TAB. 3.2 – Comparaison des combinaisons - partie 2

Approche	Circus [WC02]	PLTL-Event B [Dar02]
Section	3.5	3.6
Langages	CSP Z	PLTL Event B
But	unifier théorie	propriétés PLTL sur systèmes d'événements B
Sémantique	dénotationnelle	opérationnelle avec LTS
Vérification	*model-checking*	preuve en B, m.c. pour PLTL
Raffinement	raffinement unifié	raffinement B événementiel
Outils	m.c. à développer	outils B, *model-checker*
Lisibilité	−	+
Adéquation SI	+	−
Classification : but	création d'un nouveau langage	vérification de propriétés
Classification : sémantique	unification	traduction

sont intéressants d'un point de vue SI, car les propriétés dynamiques et statiques sont spécifiées en même temps dans le même modèle. La principale faiblesse de ces combinaisons concerne le haut niveau d'expertise réclamé par ces langages pour les comprendre et les appliquer. D'un autre côté, ils peuvent être utilisés pour spécifier et analyser des systèmes très complexes.

csp2B est un outil efficace pour traduire des descriptions CSP en des spécifications B. Il permet ainsi d'éviter d'introduire des contradictions entre les parties CSP et B, puisque la spécification B est obtenue à partir de celle en CSP. De plus, les spécifications générées par traduction satisfont les propriétés CSP. La principale faiblesse de csp2B pour spécifier des SI vient de l'utilisation restreinte des opérateurs d'entrelacement et de composition parallèle.

Dans l'approche PLTL-Event B, la logique temporelle est utilisée pour vérifier des propriétés dynamiques sur des systèmes d'événements B. L'application de la logique temporelle est particulièrement intéressante dans le cadre des SI. En effet, on pourrait vérifier des contraintes d'intégrité dynamiques sur les données qui sont très difficiles à exprimer dans un langage basé sur les états comme Z ou B. Cependant, l'utilisation du B événementiel ne semble pas approprié pour décrire des SI. Il faudrait en effet représenter le comportement de tous les utilisateurs puisque les systèmes sont fermés dans cette approche.

CSP || B propose une façon originale de décomposer les spécifications basées sur les états en des descriptions événementielles. Chaque machine B est contrôlée par un processus CSP appelé contrôleur. Les différents contrôleurs sont composés en parallèle afin de communiquer entre eux. Le principal défaut de cette approche concerne la faible capacité d'expression des contrôleurs CSP. En particulier, les entrelacements quantifiés ne sont pas autorisés, alors qu'ils permettraient de considérer plusieurs entités du même type.

3.7.3 Autres approches de combinaisons

Il existe d'autres approches de combinaisons qui n'ont pas été détaillées, car elles sont assez proches des méthodes présentées dans ce chapitre. Nous en citons quelques-unes et nous les classons selon leur sémantique.

Abstract State Machines [Gur95]. Les *Abstract State Machines* (ASM), appelées aussi *Evolving Algebras* (EA), ont pour but de relier les spécifications de type algébrique et leur modèle sémantique. L'approche consiste à construire des machines (appelées aussi "e-algèbres") qui représentent les systèmes de manière à ce que la correction des spécifications soit établie par de simples observations et vérifications. Des outils permettent en outre de simuler les machines obtenues.

Les ASM sont une variante de la logique du premier ordre avec égalité. Les structures algébriques "classiques" sont généralement définies par la syntaxe du langage et par une algèbre qui modélise ce langage. Une signature est définie par la donnée des symboles de langage. Elle peut comprendre, suivant la logique considérée, des noms de domaines, des noms de relations ou bien des noms de fonctions. Une algèbre A de signature S est un ensemble non vide X associé à une interprétation γ des symboles de S dans X.

Une e-algèbre permet en outre de changer la sémantique d'une fonction ou d'une relation par l'intermédiaire de règles sur les symboles du langage. Plusieurs opérateurs sont utilisés pour définir ces règles, comme la séquence, les boucles conditionnelles ou la récursivité. Elles peuvent enfin être gardées. À chaque étape

de l'exécution de l'ASM, les gardes sont réévaluées afin d'exécuter les règles et de modifier la sémantique des symboles de relations et de fonctions concernées.

Cette approche se distingue un peu des autres exemples de cet état de l'art, car elle concerne les spécifications de type algébrique et il ne s'agit pas vraiment d'une combinaison de spécifications formelles. Elle permet toutefois d'intégrer les aspects dynamiques en modifiant la sémantique. Pour cette raison, cette approche appartient au groupe de combinaisons **unification**.

CSP et Z [BDW99]. Cette approche propose une méthode de comparaison des langages CSP et Z, en identifiant un processus CSP avec un type de donnée abstrait de Z. Pour ce faire, Bolton définit une sémantique du comportement pour les types abstraits de données. Les opérations d'un type de donnée sont ainsi représentées dans le modèle sémantique comme des événements. L'approche propose également des relations de simulation sur les données qui sont valides et complètes par rapport aux relations de raffinement CSP.

La démarche adoptée ici est similaire à celle des travaux de combinaison entre CSP et Object-Z. Un premier exemple a été donné avec CSP-OZ dans la section 3.4. Un autre exemple, décrit dans [SD01], permet aussi d'identifier un processus CSP avec une classe Object-Z. La différence vient en fait des structures comparées : Bolton considère les types de données abstraits de Z, tandis que Fischer et Derrick utilisent les classes Object-Z. Toutefois, Bolton ne définit pas un nouveau langage, mais propose une nouvelle sémantique pour représenter le comportement des types de données Z en fonction des processus CSP. De ce point de vue, cet exemple ressemble plutôt aux approches comme csp2B qui utilisent la sémantique d'un langage pour compléter celle de l'autre. Ce couplage appartient donc au groupe de combinaisons **juxtaposition**.

Z + Petri Nets [PJ03]. Les réseaux de Petri [Pet81] sont des graphes bipartis composés de places et de transitions. Ce langage formel permet de décrire de manière graphique, mais rigoureuse, des systèmes concurrents. Dans cette approche, de nouveaux réseaux, appelés des réseaux d'activation concurrente, sont définis comme des réseaux de Petri classiques dans lesquels deux types de places sont distingués : les places classiques et les *ZPlaces*. Pour représenter le comportement du système, les réseaux de Petri associent aux places des jetons. Les transitions permettent de déterminer sous quelles conditions les jetons du système peuvent passer d'une place à une autre.

Dans l'approche Z + Petri Nets, le système est représenté d'une part par une spécification Z classique et d'autre part par un réseau d'activation concurrente dans lequel les ZPlaces sont associées à des opérations de la spécification Z. Quand une ZPlace contient un jeton, alors l'opération Z correspondante est activée et peut être exécutée.

Cette approche conserve le double point de vue entre les deux parties de la modélisation. Le seul lien entre spécification Z et réseaux de Petri est l'association d'une opération Z à une ZPlace. Par conséquent, l'utilisation des outils existants est possible dans chacune des deux parties. Pour éviter des redondances, les opérations Z ne spécifient aucune précondition concernant une variable représentant l'état global du système et les réseaux de Petri utilisés sont les plus simples et ne permettent pas, comme les réseaux de Petri de haut niveau ou les réseaux colorés, de spécifier des contraintes sur les données.

Les défauts de Z + Petri Nets sont, d'une part, que la séparation entre les deux parties de la spécification ne permet de vérifier le système dans son ensemble et que, d'autre part, l'analyse des réseaux de Petri devient difficile dès que le système est complexe.

Cet exemple est assez proche de la méthode avec CSP et Z, où les deux parties de la spécification ne sont liées que par l'identification d'un type de données abstrait Z avec un processus CSP. Z + Petri Nets appartient au groupe de combinaisons **juxtaposition**.

ZCCS [GS97]. Le langage CCS ne prévoit pas une syntaxe et une sémantique précises concernant le passage des valeurs en paramètre des agents, utilisés en CCS pour décrire le comportement d'un système. Cette approche propose d'utiliser la notation Z et une sémantique opérationnelle pour compléter CCS [Mil89].

Une spécification ZCCS est donc une spécification CCS dans laquelle les agents font appel à des données décrites par des schémas Z. La partie Z de la spécification permet de définir les ensembles abstraits, les variables et les constantes ainsi que les axiomes vérifiés par ces données qui seront utilisées dans la seconde partie de la spécification. Cette dernière est une séquence de déclarations d'agents, comme dans une description CCS standard, qui permet de modéliser le comportement du système. Les paramètres des agents sont exprimés avec la syntaxe Z pour décrire d'une part les prédicats et les types qu'ils doivent vérifier et d'autre part les effets attendus.

L'intégration est ici plus facile à réaliser que dans les approches avec CSP, car la sémantique associée à CCS est opérationnelle. La principale difficulté des approches CSP avec Z ou Object-Z réside dans la diversité des sémantiques utilisées pour les deux langages à intégrer. CSP utilise en effet des modèles de la sémantique dénotationnelle.

Par sa définition complète de la sémantique, ce travail s'apparente à celui de CSP-OZ ou Circus. Cette combinaison appartient au groupe **unification**.

Chapitre 4

Langage EB^3

> "L'entité ou l'être de la chose."
>
> — René Descartes

Dans ce chapitre, nous présentons le contexte dans lequel s'insèrent nos travaux de recherche. Le langage formel appelé EB^3 [FSD03] fournit une approche de spécification basée sur les traces d'événements, contrairement aux langages basés sur les transitions d'états qui ont été utilisés auparavant dans le cadre des spécifications formelles des SI [FFL05]. Le langage EB^3 est présenté dans la section 4.1. Dans le cadre de nos travaux, nous nous sommes plus particulièrement intéressé à la modélisation des données du SI en EB^3. Cette partie est représentée en EB^3 par des fonctions définies sur les traces valides du système. Les fonctions qui permettent d'évaluer les valeurs d'attributs sont appelées des définitions d'attributs ; elles sont présentées dans la section 4.2. Enfin, nous concluons ce chapitre avec une discussion sur le langage EB^3 et ses applications dans la section 4.3.

4.1 Spécification en EB^3

EB^3 est un langage formel inspiré de la méthode JSD [Jac83] et du concept de boîte noire de Cleanroom [PTLP99]. Il a été défini par Marc Frappier et Richard St-Denis [FSD03]. Une *boîte noire* est une fonction qui, à une séquence d'événements en entrée, associe une sortie. En EB^3, une algèbre de processus inspirée de CSP [Hoa85], CCS [Mil89] et LOTOS [BB87], est utilisée pour spécifier les entités du SI comme des boîtes noires. Dans ce langage, les termes *type d'entité* et *entité* sont employés pour désigner une classe et un objet, respectivement. EB^3 fournit à la fois une notation formelle et un processus de développement afin de décrire une spécification précise et complète du comportement des entrées-sorties d'un SI. Le cadre d'application du langage EB^3 est le projet APIS [FFLR02], qui a pour objectif de développer un environnement et des outils pour générer automatiquement des SI à partir de spécifications formelles décrites en EB^3. L'idée du projet est de libérer le concepteur de SI des détails d'implémentation pour qu'il se consacre aux phases d'analyse et de spécification. Le projet APIS sera détaillé dans le chapitre 9.

Une spécification EB^3 comprend cinq parties : i) un diagramme représentant les types d'entités et associations du SI (appelé diagramme ER dans la suite) ;

ii) une expression de processus décrivant le comportement du SI ; iii) un ensemble de règles d'entrée-sortie indiquant comment calculer les sorties du SI ; iv) un ensemble de définitions d'attributs permettant d'évaluer les attributs du système ; v) une description des spécifications d'interfaces graphiques (ou GUI, c'est-à-dire *graphical user interface*) du SI. Les parties i), ii) et iii) sont détaillées dans cette section. Les définitions d'attributs sont présentées dans la section 4.2. Enfin, la partie sur les descriptions GUI n'est pas présentée dans cette thèse, car elle ne dépend pas de nos travaux et n'a pas d'effets non plus dessus. La génération d'interfaces est discutée dans [Ter05].

Un exemple est introduit dans la section 4.1.1 afin d'illustrer les différentes parties d'une spécification EB^3. Un exemple de système de gestion de commandes est fourni dans [GFSD06]. Une spécification EB^3 est de la forme suivante. Tout d'abord, une représentation graphique, appelée diagramme ER, décrit les types d'entité et les associations du système, ainsi que leurs actions et attributs respectifs. Ce diagramme, basé sur les concepts du modèle entité-relation (ER) [Che76, EN04], utilise des notations à la UML [Obj06]. Ensuite, une expression de processus, appelée *main*, caractérise les traces valides d'événements à l'entrée du système. Une *trace* est, par définition, une séquence d'événements. L'algèbre de processus du langage EB^3 est présentée dans la section 4.1.2. Afin de représenter les sorties du SI, des règles d'entrée-sortie associent une sortie à chaque événement valide à l'entrée du système. En particulier, des fonctions récursives, définies sur les traces valides de *main*, déterminent les valeurs d'attributs des types d'entité ou associations. Les règles d'entrée-sortie sont présentées dans la section 4.1.3, tandis que les définitions d'attributs le sont dans la section 4.2.

La sémantique d'une spécification EB^3 est donnée par une relation R définie sur $\mathcal{T}(main)\times O$, où $\mathcal{T}(main)$ dénote les traces acceptées par *main* et O est l'ensemble des événements en sortie. La trace courante du système, dénotée par *trace*, est la liste finie des événements en entrée acceptés et exécutés par le système. L'expression $t :: \sigma$ représente l'insertion de l'événement σ à la fin de la trace t, alors que la notation "[ ]" représente la trace vide. Le comportement d'un système spécifié en EB^3 est défini par :

trace := [ ] ;
faire toujours
  recevoir l'événement σ ;
  si *main* peut accepter *trace* :: σ **alors**
    trace := *trace* :: σ ;
    envoyer un événement de sortie o tel que $(trace, o) \in R$;
  sinon
    envoyer un message d'erreur ;

La syntaxe complète et la sémantique d'EB^3 se trouvent dans [FSD03].

4.1.1 Exemple

Pour illustrer, on considère un exemple de gestion de bibliothèque. Le système doit gérer des emprunts de livres par des membres. Les exigences sont les suivantes :

1. Un livre est acquis (action Acquire) par la bibliothèque. Il peut être supprimé des références (Discard), mais seulement s'il n'est plus emprunté. Le

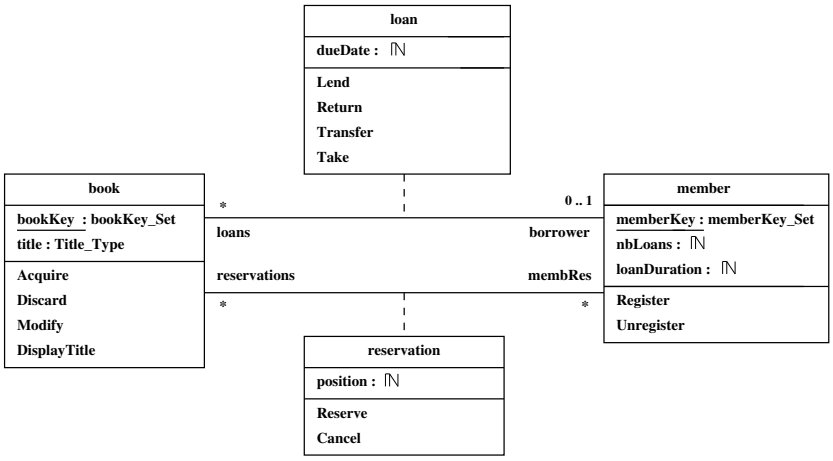

FIG. 4.1 – Diagramme EB^3 de la bibliothèque

titre du livre peut être modifié à tout moment (Modify). Il est également possible d'afficher le titre d'un livre (DisplayTitle).

2. Un membre doit s'inscrire à la bibliothèque (Register) pour pouvoir emprunter un livre. Il ne peut résilier son adhésion (Unregister) que lorsqu'il a rendu tous ses prêts.
3. Un membre peut emprunter un livre (Lend) si ce dernier est disponible. L'attribut *nbLoans* représente le nombre de prêts d'un membre. La date d'échéance d'un prêt est représentée par *dueDate*. Tout membre doit rendre un livre emprunté (Return) avant la date d'échéance prévue.
4. Un même livre ne peut être emprunté que par un seul membre à la fois.
5. Un membre peut réserver un livre (Reserve) si ce dernier est déjà emprunté par un autre membre. Plusieurs membres peuvent réserver le même livre. L'attribut *position* indique la position d'un membre dans la liste de réservation d'un livre.
6. Dès qu'il est retourné, un livre qui a été réservé peut être récupéré (Take) par le membre dont la position est égale à 1 dans la liste de réservation. Tant qu'un membre n'a pas récupéré un livre, il peut à tout moment annuler (Cancel) sa réservation.
7. Un membre peut transférer (Transfer) un de ses prêts à un autre membre, si le livre en question n'a pas été réservé.
8. La durée des prêts dépend du type de prêts accordé par la bibliothèque : permanent ou classique. Un prêt de type permanent dure un an, alors qu'un prêt classique dure le temps prévu pour le membre en question (représenté par l'attribut *loanDuration*).

La figure 4.1 représente le diagramme EB^3 de cet exemple. Les signatures des actions EB^3 sont les suivantes :

$\mathsf{Acquire}(bId : bookKey\_Set, bTitle : Title\_Type^{\perp}) : void$
$\mathsf{Discard}(bId : bookKey\_Set) : void$

Modify(bId : $bookKey\_Set$, $newTitle$: $Title\_Type^{\perp}$) : $void$
DisplayTitle(bId : $bookKey\_Set$) : ($title$: $Title\_Type^{\perp}$)
Register(mId : $memberKey\_Set$, lD : NAT) : $void$
Unregister(mId : $memberKey\_Set$) : $void$
Lend(bId : $bookKey\_Set$, mId : $memberKey\_Set$, $type$: $Loan\_Type$) : $void$
Return(bId : $bookKey\_Set$) : $void$
Transfer(bId : $bookKey\_Set$, mId : $memberKey\_Set$, $type$: $Loan\_Type$) : $void$
Reserve(bId : $bookKey\_Set$, mId : $memberKey\_Set$) : $void$
Take(bId : $bookKey\_Set$, mId : $memberKey\_Set$, $type$: $Loan\_Type$) : $void$
Cancel(bId : $bookKey\_Set$, mId : $memberKey\_Set$) : $void$

Le type *void* est utilisé pour représenter une action sans sortie. Rappelons que toute action en EB^3 renvoie implicitement un paramètre de sortie pour indiquer si l'action est valide ("*ok*") ou pas ("*error*"). Lorsqu'un paramètre peut prendre la valeur **NULL**, qui correspond à une valeur indéfinie, alors le type est décoré par l'exposant "$\perp$".

4.1.2 Traces valides d'un SI

Un SI est vu en EB^3 comme une boîte noire. L'algèbre de processus d'EB^3 permet de décrire les traces valides d'événements à l'entrée du système.

4.1.2.1 Syntaxe

Un événement σ est l'instance d'une action et de ses éventuels paramètres d'entrée ; il est considéré comme une expression de processus élémentaire. Le symbole "_" est utilisé pour indiquer qu'un paramètre peut prendre toute valeur bien typée. Une expression de processus EB^3 est formée à partir des expressions de processus élémentaires et des opérateurs suivants : séquence (dénotée par " **.** "), choix (" | "), fermeture de Kleene (" * "), entrelacement (" ||| "), composition parallèle (" || ", avec une synchronisation des actions partagées comme en CSP), garde (" $\Longrightarrow$ "), appel de processus et quantification du choix (" | $x : T : ...$ ") et de l'entrelacement (" $|||x : T : ...$ "). Bien qu'elles soient assez proches, la notation EB^3 pour les expressions de processus se distingue de celle du langage CSP [Hoa85]. En EB^3, il est possible, par exemple, d'utiliser une variable d'état, la trace du système, dans les prédicats des gardes des actions. D'autre part, le langage EB^3 définit un unique opérateur de concaténation, comme dans les expressions régulières, plutôt que d'utiliser deux opérateurs (préfixe et composition séquentielle) en CSP.

4.1.2.2 Expression de processus

Le processus principal du SI s'appelle *main*. Il fait appel à des sous-processus définis pour chaque type d'entité et association du diagramme ER. Le processus associé à un type d'entité (respectivement à une association) décrit le cycle de vie d'une entité (respectivement d'une instance d'association) et les contraintes d'ordonnancement entre les actions. Par exemple, le processus associé au type d'entité *book* est le suivant :

$book(bId : bookKey\_Set) =$
  Acquire(bId, _)**.**

$$
\begin{array}{l}
\quad (\\
\quad\quad (\mid mId : memberKey\_Set : loan(mId, bId))^* \\
\quad \| \\
\quad\quad loanCycleBook(bId)^* \\
\quad \| \\
\quad\quad (\|\|\| mId : memberKey\_Set : reservation(mId, bId)^*) \\
\quad \|\|\| \\
\quad\quad \mathsf{Modify}(bId, \_)^* \\
\quad \|\|\| \\
\quad\quad \mathsf{DisplayTitle}(bId)^* \\
\quad). \\
\quad \mathsf{Discard}(bId)
\end{array}
$$

où *loan* désigne le processus associé à l'association *loan* et *reservation* celui de l'association *reservation*. Le processus *book* décrit le cycle de vie d'une entité de livre *bId*. L'entité *bId* est tout d'abord créée par l'action Acquire. Ensuite, le livre *bId* peut être emprunté par un seul membre *mId* à la fois, d'où le choix quantifié " $\mid mId : memberKey\_Set : ...$". Le processus *book* fait alors appel au sous-processus *loan*, qui implique notamment les actions Lend, Return, Transfer et Take. La fermeture de Kleene sur le choix quantifié de *loan* signifie qu'un nombre fini arbitraire de prêts peut être réalisé sur le livre *bId*. Le sous-processus *loanCycleBook* est un processus qui permet de contrôler le comportement décrit dans le processus *loan*. La description complète de ce processus est fournie dans l'annexe C.

Le livre *bId* peut être réservé par un ou plusieurs membres, d'où l'entrelacement quantifié " $|||mId : memberKey\_Set : ...$". Le processus *book* fait appel au sous-processus *reservation*, qui implique les actions Reserve, Take et Cancel. L'opérateur d'entrelacement permet aux actions Modify et DisplayTitle d'être intercalées à tout moment entre les différentes actions prévues dans les processus *loan* et *reservation*. Enfin, le livre *bId* est supprimé par Discard.

D'un autre côté, le processus associé au type d'entité *member* est de la même forme que *book* :

$$
\begin{array}{l}
member(mId : memberKey\_Set) = \\
\quad \mathsf{Register}(mId, \_). \\
\quad (\\
\quad\quad (\|\|\| bId : bookKey\_Set : loan(mId, bId)^*) \\
\quad\quad \| \\
\quad\quad (\|\|\| bId : bookKey\_Set : reservation(mId, bId)^*) \\
\quad). \\
\quad \mathsf{Unregister}(mId)
\end{array}
$$

À la différence de *book*, un membre peut emprunter plusieurs livres. C'est pourquoi le processus *loan* est précédé par un entrelacement quantifié.

Les interactions entre les différents types d'entité et associations s'expriment sous la forme d'une composition parallèle en EB^3. Dans le cas de l'exemple de la bibliothèque, le processus *main* est de la forme suivante :

$$
\begin{array}{l}
main = \\
\quad\quad (\|\|\| bId : bookKey\_Set : book(bId)^*) \\
\quad \| \\
\quad\quad (\|\|\| mId : memberKey\_Set : member(mId)^*)
\end{array}
$$

Ainsi, ce processus met en parallèle les différentes entités de type *book* et celles de type *member*.

4.1.2.3 Patrons de processus

Nous présentons maintenant plusieurs patrons de processus caractéristiques des SI, qui ont été définis dans [FSD03] afin de faciliter la spécification des expressions de processus. Dans le cadre de cette thèse, les patrons décrits ci-dessous serviront à définir des patrons de raffinement pour notre approche de combinaison des langages EB^3 et B. Le processus principal du système est de la forme suivante :

$$main = E_1 \parallel ... \parallel E_i \parallel ... \parallel E_n$$

où chaque E_i est de la forme $|||\ k_i : K\_Set_i : e_i(k_i)^*$. Les $e_i(k_i)$ sont les processus des différents types d'entité du SI.

(a) Patron de base pour les types d'entité et associations. Le cycle de vie des entités d'un type d'entité est souvent de la forme "production, modification, consommation". Autrement dit, une entité est tout d'abord produite, elle peut ensuite être modifiée à plusieurs reprises par une ou plusieurs actions, et elle est consommée à la fin de son cycle. Le patron appelé **producer-modifier-consumer** décrit le comportement type d'un type d'entité `e`, dont les actions sont soit des producteurs $(P_1, ..., P_l)$, soit des modificateurs $(M_1, ..., M_n)$, soit des consommateurs $(C_1, ..., C_m)$:

$$\begin{array}{l} e(k : K\_Set) = \\ \quad P_1(k,\_) \mid ... \mid P_l(k,\_)\textbf{.} \\ \quad (\\ \quad\quad (\ M_1(k,\_) \mid ... \mid M_n(k,\_) \ )^* \\ \quad \parallel \\ \quad\quad AP_1 \ \parallel \ ... \ \parallel \ AP_r \\ \quad ||| \\ \quad\quad Rq_1^* \ ||| \ ... \ ||| \ Rq_p^* \\ \quad)\textbf{.} \\ \quad C_1(k,\_) \mid ... \mid C_m(k,\_) \end{array}$$

où k est une clé qui est composée d'un ou plusieurs attributs de e et K_Set le type de k. $Rq_1, ..., Rq_p$ sont des actions EB^3 qui ont pour effet de retourner des valeurs d'attributs. Les expressions $AP_1, ..., AP_r$ correspondent aux éventuels appels de processus définis pour les associations. Le nombre r dépend du nombre d'associations auxquelles l'entité e participe. En particulier, il n'y a aucun appel de processus (*i.e.*, $r = 0$) lorsque l'entité ne participe à aucune association. Chaque expression AP_i, où i varie de 1 à r, est à remplacer par un des appels de processus indiqués dans le paragraphe (b) sur les entités participant à des associations binaires, suivant la cardinalité de l'association. Par exemple, le processus du type d'entité *book* décrit dans la section 4.1.2.2 est une instanciation de ce patron. Notons que nous n'avons pas défini comme dans [BGL03] un processus d'instanciation en EB^3 de manière formelle. La formalisation de l'instanciation des patrons de processus fait partie de nos perspectives de travail.

Comme pour les types d'entité, le processus qui décrit le cycle de vie d'une association binaire a est de la forme "production, modification, consommation".

Dans ce cas, les expressions $AP_1, ..., AP_r$ n'apparaissent pas dans l'instanciation du patron **producer-modifier-consumer**. Par exemple, dans le processus de l'association *loan*, les actions Lend et Return sont respectivement un producteur et un consommateur de prêt.

(b) Patrons pour les entités participant à des associations binaires. Nous considérons les associations binaires de cardinalité $1 : N$, $1 : 1$ et $M : N$. Les patrons pour les autres types d'association sont décrits dans [FSD03]. Pour les traces EB^3, le processus d'une association est un sous-processus d'un ou plusieurs types d'entité. En effet, les actions d'une association concernent aussi bien l'association que les types d'entité qui participent à cette association. Par conséquent, les processus des types d'entité font appel aux processus des associations auxquelles ils participent. Chaque appel est représenté par une expression AP_i dans le patron **producer-modifier-consumer**.

Soient $e1$ et $e2$ les types d'entité qui participent à l'association binaire a. Soient k_1 et k_2 leurs clés respectives. Soient AP_{i1} et AP_{i2} les expressions dans les processus de e_1 et e_2 qui correspondent aux appels du processus de a. La forme des appels de processus AP_{i1} et AP_{i2} dépend de la cardinalité de l'association a. Dans le cas d'une association $1 : N$, les expressions AP_{i1} et AP_{i2} sont respectivement de la forme (AP-Multi) et (AP-One) :

$$||| \; k2 : K\_Set2 : a(k1, k2)^* \qquad \text{(AP-Multi)}$$

$$(\; | \; k1 : K\_Set1 : a(k1, k2) \;)^* \qquad \text{(AP-One)}$$

En effet, dans le processus e_1, le sous-processus a est appelé dans un entrelacement quantifié. Par conséquent, une entité k_1 peut être associée à plusieurs entités k_2. Dans le processus e_2, le processus a est appelé dans un choix quantifié. Par conséquent, k_2 est associé à une seule entité k_1 à la fois. Par exemple, dans le processus *book* de la section 4.1.2, le processus de l'association *loan* est appelé avec un choix quantifié. Comme *loan* est une association $1 : N$, *book* est en effet une instanciation de e_2 avec AP_{i2} de la forme (AP-One).

Dans le cas d'une association $1 : 1$, les types d'entité qui participent à l'association sont alors tous les deux définis avec des appels de processus de la forme (AP-One). Dans le cas d'une association $M : N$, les deux appels de processus sont de la forme (AP-Multi).

4.1.3 Règles d'entrée-sortie

En EB^3, la sortie d'un SI correspond à la réponse du système après avoir exécuté l'événement en entrée σ. Il existe trois possibilités :

- σ n'est pas un événement valide ; autrement dit, la trace courante du système concaténée à σ n'est pas une trace valide du SI. Dans ce cas, un message d'erreur est retourné et l'événement σ n'est pas exécuté.
- σ est un événement valide, mais il n'y a pas de sortie prévue dans la signature de l'action correspondante. Dans ce cas, le message de retour indique que l'événement a bien été accepté et exécuté.
- σ est un événement valide et une sortie est prévue dans la signature. Dans ce cas, les règles d'entrée-sortie indiquent comment calculer les paramètres de sortie. Ils sont évalués en EB^3 à l'aide de fonctions récursives.

Un message de notification est retourné en même temps que les résultats de l'évaluation.

La règle d'entrée-sortie suivante est définie pour DisplayTitle :

RULE R
INPUT DisplayTitle(bId)
OUTPUT $title(t, bId)$
END ;

Quand l'action DisplayTitle est valide, alors la fonction récursive *title* est appelée pour évaluer l'attribut *title* en fonction de la trace courante du SI. Il s'agit de l'unique règle d'entrée-sortie de l'exemple, puisque les autres actions ne prévoient pas de sortie.

4.2 Définitions d'attributs

Les expressions de processus (section 4.1.2) et les règles d'entrée-sortie (section 4.1.3) sont les parties d'une spécification EB^3 qui permettent de représenter le comportement du système d'un point de vue entrées et sorties. Le modèle de données du SI est spécifié à l'aide du diagramme ER et d'un ensemble de fonctions récursives définies sur les traces valides du système. Une première définition de ces fonctions a été donnée dans [FSD03], mais la syntaxe n'avait pas été fixée. Une de nos contributions a été la définition, avec précision, du langage de définition des attributs. Ces travaux préliminaires étaient nécessaires pour définir des règles de traduction vers d'autres langages, comme B ou Java/SQL. De plus, ils ont permis de compléter la définition du langage EB^3.

4.2.1 Langage

Une *définition d'attribut* est une fonction récursive sur les traces valides du système qui est totale et définie dans un style fonctionnel. Elle retourne les valeurs d'attributs qui correspondent à l'état du SI, une fois que ce dernier a exécuté la séquence d'événements de la trace donnée en paramètre. On distingue deux types d'attributs suivant qu'il s'agit d'une clé ou non.

Conventions

Dans les définitions suivantes, les termes fonctionnels sont distingués des termes conditionnels. Un *terme fonctionnel* est un terme composé de constantes, de variables et de fonctions d'autres termes fonctionnels. Les types de données sur lesquels les constantes et variables sont définies peuvent être des ensembles abstraits ou énumérés, des types de base comme $\mathbb{N}, \mathbb{Z}, \ldots$, des produits cartésiens de types de données ou des sous-ensembles finis de types de données. Un *terme conditionnel* est de la forme **if** *pred* **then** w_1 **else** w_2 **end**, où *pred* est un prédicat, w_2 est soit un terme conditionnel, soit un terme fonctionnel, et w_1 est un terme fonctionnel. L'expression $var(e)$ désigne les variables libres d'une expression e. Un terme de base t est une expression sans aucune variable libre ; par conséquent, $var(t) = \varnothing$.

4.2.1.1 Définition de clé

De manière classique, une clé est définie en EB^3 pour identifier les instances des types d'entité ou associations. La fonction récursive associée à une clé de type d'entité ou d'association, appelée *définition de clé*, retourne l'ensemble des valeurs de clé de ce type d'entité ou de cette association une fois que le système a exécuté la trace donnée en paramètre.

Soit e un type d'entité dont la clé est composée des attributs $k_1, \ldots, k_m$. La définition de clé de e est une fonction totale $eKey$ de type :

$$eKey \ : \ \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(T_1 \times \cdots \times T_m)$$

où $T_1, \ldots, T_m$ sont les types respectifs des attributs clé $k_1, \ldots, k_m$ et l'expression $\mathbb{F}(S)$ désigne l'ensemble des sous-ensembles finis de l'ensemble S. Par exemple, le type de la définition de clé du type d'entité *book* est :

$$bookKey \ : \ \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(bookKey\_Set)$$

En EB^3, les fonctions récursives sont toujours définies dans un style à la CAML [CM98], avec un filtrage sur le dernier événement de la trace. La définition de clé d'un type d'entité e a la forme suivante :

$$\begin{array}{l} eKey\ (s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(T_1 \times \cdots \times T_m) \ \triangleq \\ \quad \textbf{match}\ last(s)\ \textbf{with} \\ \qquad \begin{array}{lcl} \bot & : & u_0, \\ a_1(\overrightarrow{p_1}) & : & u_1, \\ \ldots & & \\ a_n(\overrightarrow{p_n}) & : & u_n, \\ \_ & : & eKey(front(s))\,; \end{array} \end{array} \tag{4.1}$$

Les expressions $\bot : u_0$, $a_1(\overrightarrow{p_1}) : u_1$, ..., $a_n(\overrightarrow{p_n}) : u_n$ et $\_ \ : \ eKey(front(s))$ sont appelées des *clauses d'entrée*. L'expression $last(s)$ représente le dernier événement de la trace s, tandis que $front(s)$ est la trace s, dont on a supprimé le dernier élément. Les fonctions *last* et *front* retournent la valeur "$\bot$" lorsque s est une trace vide.

Dans une clause d'entrée, l'expression $a_i(\overrightarrow{p_i})$ représente un *motif* possible pour l'instance de $last(s)$ qui est filtrée. La notation a_i désigne un nom d'action et $\overrightarrow{p_i}$ est une liste de paramètres, dont les éléments sont des variables, des constantes, ou le symbole "_", dans le cas où le paramètre en question peut prendre n'importe quelle valeur bien typée. Les expressions $u_0, \ldots, u_n$ sont uniquement des termes fonctionnels. Pour chaque clause, on a : $var(u_i) \subseteq var(\overrightarrow{p_i})$.

Dans une définition de clé, le terme fonctionnel d'une clause d'entrée $a_i(\overrightarrow{p_i}) : u_i$ est une expression ensembliste de la forme suivante :

- $\varnothing$, dans le cas où il n'y a pas d'entité,
- S, qui représente l'ensemble des entités existantes,
- $eKey(front(s)) \cup S$, qui représente l'ajout d'un ensemble de nouvelles entités,
- $eKey(front(s)) - S$, qui représente la suppression d'un ensemble d'entités.

L'ensemble S est composé d'éléments de deux types :

- c, une constante de type $T_1 \times \cdots \times T_m$,
- v, une variable de $var(\overrightarrow{p_i})$.

Par exemple, la clé du type d'entité *book* est définie par la fonction suivante :

$$
\begin{array}{l}
bookKey(s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(bookKey\_Set) \quad \triangleq \\
\quad \textbf{match } last(s) \textbf{ with} \\
\qquad \bot : \varnothing, \\
\qquad \mathsf{Acquire}(bId, \_) : bookKey(front(s)) \cup \{bId\}, \\
\qquad \mathsf{Discard}(mId) : bookKey(front(s)) - \{bId\}, \\
\qquad \_ : bookKey(front(s))\,;
\end{array}
$$

La fonction *bookKey* a un unique paramètre d'entrée $s \in \mathcal{T}(\mathsf{main})$, où $\mathcal{T}(\mathsf{main})$ représente l'ensemble des traces acceptées par le processus *main*, et elle retourne l'ensemble des valeurs de clé de *book*. Informellement, cela signifie que si la trace est vide ($last(s) = \bot$), alors : $bookKey = \varnothing$. Si le dernier événement de la trace est Acquire, alors *bId* est ajouté à l'ensemble *bookKey* calculé récursivement sur $front(s)$. De manière analogue, *bId* est supprimé de $bookKey(front(s))$ lorsque le dernier événement de la trace est Discard. Enfin, si $last(s)$ ne correspond à aucun de ces cas, alors la fonction *bookKey* est appelée récursivement sur $front(s)$, par la dernière clause avec le symbole "_". L'exécution d'une telle fonction est détaillée dans la section 4.2.2.

Soit *asc* une association l-aire entre l types d'entité $e_1, \ldots, e_l$. Par définition, la clé d'une association *asc* est formée avec les clés des types d'entité correspondants. Pour chaque $j \in \{1, 2, \ldots, l\}$, on note $k^j_1, \ldots, k^j_{m_j}$ les attributs clé du type d'entité e_j. La clé de *asc* est alors définie par l'ensemble de tous les attributs clé des types d'entité $e_1, \ldots, e_l$:

$$asc \;:\; \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(T_{k^1_1} \times \cdots \times T_{k^1_{m_1}} \times \cdots \times T_{k^l_1} \times \cdots \times T_{k^l_{m_l}})$$

Par exemple, la clé de l'association *reservation* est de type :

$$reservation \;:\; \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(bookKey\_Set \times memberKey\_Set)$$

Lorsque la cardinalité de l'association est 0..1 ou 1, la clé de l'association est alors plus simple. Les algorithmes de normalisation peuvent être trouvés dans [EN04]. Par exemple, la clé de l'association *loan* est de type :

$$loan \;:\; \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(bookKey\_Set)$$

parce que le type d'entité *member* a une multiplicité de 0..1 et que le rôle *borrower* représente l'unique emprunteur d'un livre.

4.2.1.2 Définition d'attribut non clé

La fonction récursive associée à un attribut non clé, appelée *définition d'attribut non clé*, retourne la valeur de l'attribut pour la clé et la trace données en paramètre. La fonction d'un attribut non clé b_i d'un type d'entité ou d'une association est de type :

$$b_i \;:\; \mathcal{T}(\mathsf{main}) \times T_1 \times \cdots \times T_m \rightarrow T_i$$

où $T_1, \ldots, T_m$ sont les types des attributs clé du type d'entité ou de l'association et le codomaine T_i est le type de l'attribut non clé b_i. Le codomaine d'une définition d'attribut non clé peut inclure $\bot$, afin de représenter le fait qu'un

attribut n'est pas défini. Par exemple, le type de la fonction associée à l'attribut *nbLoans* est :

$$nbLoans \ : \ \mathcal{T}(\mathsf{main}) \times memberKey\_Set \rightarrow \mathbb{N}$$

Dans ce cas, la fonction est totale, car *nbLoans* est défini pour tous les membres de la bibliothèque.

La définition d'un attribut non clé b_i a la forme suivante :

$$\begin{array}{l} b_i \ (s : \mathcal{T}(\mathsf{main}), \overrightarrow{k} : T_1 \times \cdots \times T_m) : T_i \ \triangleq \\ \quad \textbf{match} \ last(s) \ \textbf{with} \\ \qquad \begin{array}{lcl} \bot & : & u_0, \\ a_1(\overrightarrow{p_1}) & : & u_1, \\ \cdots & & \\ a_n(\overrightarrow{p_n}) & : & u_n, \\ \_ & : & b_j(front(s), \overrightarrow{k}) ; \end{array} \end{array} \tag{4.2}$$

où b_j est un attribut (j peut être égal à i). L'expression $\overrightarrow{k}$ représente la liste des attributs clé. Les expressions $u_0, \ldots, u_n$ sont soit des termes fonctionnels, soit des termes conditionnels. Pour chaque clause d'entrée, on a : $var(u_j) \subseteq var(\overrightarrow{p_j}) \cup var(\overrightarrow{k})$.

Pour les définitions d'attributs non clé, un terme fonctionnel de l'expression u_j prend l'une des formes suivantes :

- $\bot$, soit une valeur indéfinie pour l'attribut,
- c, une constante de type T_i, qui représente une valeur d'attribut,
- v, une variable de $var(\overrightarrow{p_j}) \cup var(\overrightarrow{k})$,
- $g(\overrightarrow{t})$, où chaque $t_i \in \overrightarrow{t}$ est un terme fonctionnel et g est soit un attribut, soit un opérateur sur un ou plusieurs types d'attribut.

La dernière expression, $g(\overrightarrow{t})$, permet de définir des appels récursifs de b_i ou des appels d'autres définitions d'attributs. Une référence à une clé *eKey* ou à un attribut b dans une clause d'entrée est toujours de la forme $eKey(front(s))$ ou $b(front(s), ...)$. De plus, les opérateurs utilisés doivent respecter les types des expressions u_j.

Par exemple, l'attribut non clé *nbLoans* est défini par la fonction suivante :

$$\begin{array}{l} nbLoans(s : \mathcal{T}(\mathsf{main}), mId : memberKey\_Set) : \mathbb{N} \ \triangleq \\ \textbf{match} \ last(s) \ \textbf{with} \\ \quad \bot \ : \ \bot, \\ \quad \mathsf{Register}(mId, \_) \ : \ 0, \\ \quad \mathsf{Lend}(\_, mId, \_) \ : \ 1 + nbLoans(front(s), mId), \\ \quad \mathsf{Return}(bId) \ : \ \textbf{if} \ mId = borrower(front(s), bId) \\ \qquad\qquad\qquad \textbf{then} \ nbLoans(front(s), mId) - 1 \ \textbf{end}, \\ \quad \mathsf{Transfer}(bId, mId', \_) \ : \ \textbf{if} \ mId = mId' \\ \qquad\qquad\qquad\qquad\qquad \textbf{then} \ nbLoans(front(s), mId) + 1 \\ \qquad\qquad\qquad\qquad\qquad \textbf{else if} \ mId = borrower(front(s), bId) \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{then} \ nbLoans(front(s), mId) - 1 \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{end} \\ \qquad\qquad\qquad\qquad\qquad \textbf{end}, \\ \quad \mathsf{Take}(\_, mId, \_) \ : \ 1 + nbLoans(front(s), mId), \\ \quad \mathsf{Unregister}(mId) \ : \ \bot, \\ \quad \_ \ : \ nbLoans(front(s), mId) ; \end{array}$$

Cette définition utilise deux termes conditionnels, dans les clauses Return et Transfer. Si les principes d'exécution des définitions d'attributs sont détaillés dans la section 4.2.2, nous présentons maintenant une première explication, informelle, de l'évaluation de ces termes conditionnels. La fonction *nbLoans* a pour paramètre d'entrée un membre *mId*. Le motif Return(bId) ne permet pas de caractériser le membre concerné par cet événement. Dans ce cas, le prédicat du terme conditionnel indique quel est le membre dont le nombre de prêts diminue d'une unité : il s'agit de l'emprunteur de *bId*. De même, le motif Transfer($bId, mId', \_$) n'est pas suffisant pour déterminer le ou les membres *mId* concernés par cet événement. En fait, deux membres distincts sont caractérisés par les prédicats du terme conditionnel :

- le membre qui fait le transfert : $borrower(front(s), bId)$,
- et le membre qui profite du transfert : mId'.

L'exécution d'un événement de la forme Transfer($bId, mId', \_$) a pour effet de décrémenter d'une unité le nombre de prêts (*nbLoans*) du membre qui fait le transfert et d'incrémenter d'une unité le nombre de prêts du membre mId'.

Les termes conditionnels sont également utilisés lorsque la valeur d'un ou plusieurs paramètres détermine la valeur de l'attribut. Par exemple, l'attribut *dueDate* de l'association *loan* permet de représenter la date de retour d'un prêt. On suppose que la date de retour dépend du type de prêt : a) s'il est de type "*Classic*", alors le prêt dure le délai prévu pour le membre, qui est indiqué par l'attribut *loanDuration* du type d'entité *member* ; b) s'il est de type "*Permanent*", alors le prêt dure une année. Ce type de prêt est précisé comme paramètre des actions Lend, Transfer et Take. La définition d'attribut de *dueDate* peut être spécifiée en EB^3 par la fonction suivante :

$$
\begin{array}{l}
dueDate(s : \mathcal{T}(\mathsf{main}), bId : bookKey\_Set) : DATE \triangleq \\
\textbf{match } last(s) \textbf{ with} \\
\quad \bot : \bot, \\
\quad \mathsf{Lend}(bId, \_, Permanent) : CurrentDate{+}365, \\
\quad \mathsf{Lend}(bId, mId, Classic) : CurrentDate{+}loanDuration(front(s), mId), \\
\quad \mathsf{Return}(bId) : \bot, \\
\quad \mathsf{Transfer}(bId, mId, type) : \textbf{if } type = Permanent \\
\qquad\qquad\qquad\qquad\quad \textbf{then } CurrentDate{+}365 \\
\qquad\qquad\qquad\qquad\quad \textbf{else } CurrentDate \\
\qquad\qquad\qquad\qquad\qquad\quad {+}loanDuration(front(s), mId) \\
\qquad\qquad\qquad\qquad\quad \textbf{end}, \\
\quad \mathsf{Take}(bId, \_, Permanent) : CurrentDate{+}365, \\
\quad \mathsf{Take}(bId, mId, Classic) : CurrentDate{+}loanDuration(front(s), mId), \\
\quad \_ : dueDate(front(s), bId)\,;
\end{array}
$$

Dans l'exemple ci-dessus, on remarque qu'il existe plusieurs façons de spécifier les hypothèses sur les paramètres d'action comme dans le cas du type de prêt. D'un côté, on peut spécifier autant de clauses d'entrée qu'il y a de valeurs pour le paramètre comme, par exemple, pour les actions Lend et Take. Dans ce cas, l'analyse des clauses d'entrée se fait dans l'ordre qu'elles sont indiquées (voir section 4.2.2). D'un autre côté, il est possible d'utiliser des termes conditionnels pour considérer les différentes valeurs possibles du paramètre d'action. Cette option est illustrée par la clause Transfer. Pour faciliter l'analyse des clauses d'entrée lors de l'évaluation des définitions d'attributs, on se ramènera de préférence à la deuxième option avec les termes conditionnels. Le passage

FIG. 4.2 – Rôles d'une association de type $M : N$

FIG. 4.3 – Rôles d'une association de type $1 : N$

d'une écriture à l'autre se fait aisément. Par exemple, dans le cas de la clause Take, il suffit de déterminer les différentes valeurs du troisième paramètre et de les retranscrire dans un terme conditionnel :

$$\begin{aligned}\mathsf{Take}(bId, mId, type) \ : \ & \textbf{if } type = Permanent \\ & \textbf{then } CurrentDate{+}365 \\ & \textbf{else if } type = Classic \\ & \quad \textbf{then } CurrentDate \\ & \qquad\quad {+}loanDuration(front(s), mId) \\ & \quad \textbf{else } dueDate(front(s), bId) \\ & \quad \textbf{end} \\ & \textbf{end},\end{aligned}$$

Pour le nom du paramètre, on utilise celui indiqué dans la signature de l'action.

En résumé, les termes conditionnels sont principalement utilisés dans trois cas :

- lorsque le filtrage n'est pas suffisant pour lier les paramètres d'entrée de la fonction récursive aux valeurs effectives des motifs (exemple : clause Return dans *nbLoans*),
- lorsque plusieurs valeurs de clé sont concernées par le même motif (exemple : Transfer dans *nbLoans*),
- lorsque l'évaluation d'un attribut dépend de la valeur d'un ou plusieurs paramètres d'une action (exemple : Transfer dans *dueDate*).

4.2.1.3 Définition des rôles dans les associations

Un type d'entité qui participe à une association joue un *rôle* particulier dans l'association. En EB^3, la définition d'un rôle est obligatoire lorsque ce dernier est utilisé dans les définitions d'attributs pour caractériser certaines valeurs dans les prédicats des termes conditionnels. Cela permet de donner une sémantique précise au rôle. La définition des rôles dépend de la cardinalité des associations.

Dans une association de cardinalité $M : N$, plusieurs entités du même type d'entité peuvent participer à l'association. La clé de cette dernière est composée des attributs clé de ces types d'entité. Soit *asc* une association binaire de cardinalité $M : N$ entre les types d'entité e_1 et e_2. La définition de clé de l'association

asc est du type suivant :

$$asc \ : \ \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(\, T_{k_1^1} \times \cdots \times T_{k_{m_1}^1} \times T_{k_1^2} \times \cdots \times T_{k_{m_2}^2}\,)$$

La figure 4.2 représente les rôles de r_1 et r_2 dans l'association *asc*. Le rôle r_1 de e_2 dans *asc* est défini en EB^3 de la manière suivante :

$$r_1(k_1^1, ..., k_{m_1}^1) = \{(k_1^2, ..., k_{m_2}^2) \mid (k_1^1, ..., k_{m_1}^1, k_1^2, ..., k_{m_2}^2) \in asc\}$$

De même, le rôle r_2 de e_1 dans *asc* est défini par :

$$r_2(k_1^2, ..., k_{m_2}^2) = \{(k_1^1, ..., k_{m_1}^1) \mid (k_1^1, ..., k_{m_1}^1, k_1^2, ..., k_{m_2}^2) \in asc\}$$

Ces définitions sont retrouvées à partir du diagramme EB^3 du système. Par exemple, le rôle *membRes* dans l'association *reservation* est défini par :

$$membRes(bId) = \{mId \mid (bId, mId) \in reservation\}$$

Dans une association de cardinalité $1 : N$, l'entité du côté 1 de l'association est unique pour chaque entité du côté N. La clé d'une telle association n'est composée que des attributs clé du type d'entité du côté N. Par conséquent, pour l'association de type $1 : N$ décrite dans la figure 4.3, le rôle r_2 du type d'entité situé du côté 1 de l'association est défini en EB^3 par une fonction récursive de la forme suivante :

$$r_2 \ : \ \mathcal{T}(\mathsf{main}) \times T_{k_1^2} \times \cdots \times T_{k_{m_2}^2} \ \rightarrow \ T_{k_1^1} \times \cdots \times T_{k_{m_1}^1}$$

où $T_{k_1^2}, ..., T_{k_{m_2}^2}$ sont les types des attributs clé de e_2 et $T_{k_1^1}, ..., T_{k_{m_1}^1}$ les types des attributs clé de e_1. Comme pour les définitions d'attributs, la fonction r_2 est totale et définie dans un style fonctionnel. Par exemple, le rôle *borrower* dans l'association *loan* est spécifié par la fonction suivante :

$$\begin{array}{l} borrower(bId : bookKey\_Set) : memberKey\_Set \ \triangleq \\ \textbf{match } last(s) \textbf{ with} \\ \quad \bot \ : \ \bot, \\ \quad \mathsf{Lend}(bId, mId, \_) \ : \ mId, \\ \quad \mathsf{Return}(bId) \ : \ \bot, \\ \quad \mathsf{Transfer}(bId, mId, \_) \ : \ mId, \\ \quad \mathsf{Take}(bId, mId, \_) \ : \ mId, \\ \quad \_ \ : \ borrower(front(s), bId)\,; \end{array}$$

4.2.2 Exécution des définitions d'attributs

Les définitions d'attributs EB^3 décrivent la dynamique des données d'un SI. Quand une définition d'attribut est exécutée, alors toutes les clauses d'entrée de la fonction sont évaluées comme des filtres sur le dernier événement de la trace ; la première clause satisfaite est celle qui est exécutée. Par conséquent, l'ordre des clauses a une importance. Le filtrage s'effectue toujours sur le dernier événement de la trace $s \in \mathcal{T}(\mathsf{main})$.

Soit $b(s, v_1, \ldots, v_n)$ la fonction à évaluer et ρ la substitution $\overrightarrow{k} := v_1, \ldots, v_n$. Chaque clause d'entrée $a_i(\overrightarrow{p_i}) : u_i$ génère une condition de filtrage de la forme :

$$\exists\, (var(\overrightarrow{p_i}) - \overrightarrow{k}\,) \ \bullet \ last(s) = a_i(\overrightarrow{p_i})\ \rho \qquad (4.3)$$

où le membre droit de l'équation désigne l'application de la substitution ρ sur la clause d'entrée $a_i(\overrightarrow{p_i})$. Une telle condition est satisfaite si l'action est la même et si les paramètres de $last(s)$ correspondent aux valeurs des variables $\overrightarrow{k}$ dans $\overrightarrow{p_i}$. Dans ce cas, l'expression u_i associée est exécutée pour évaluer la valeur d'attribut. Toute définition d'attribut inclut toujours les clauses "$\bot$" et "$\_$". La première correspond à une initialisation, puisque le dernier élément d'une trace vide est indéfini : $last([\,]) = \bot$. Le motif "$\_$" coïncide avec tout événement ; cette clause permet donc d'appeler récursivement la fonction, mais avec la trace s amputée du dernier élément (dénotée par $front(s)$).

Un attribut b est affecté par un événement $\sigma = a(\sigma_1, \ldots, \sigma_m)$ s'il existe une clause d'entrée $a(\overrightarrow{p}) : u$ dans la définition de b. Plusieurs clauses peuvent contenir la même action a :

$$\begin{array}{l} b\ (\overrightarrow{k}) : T \triangleq \\ \quad \ldots \\ \quad a(\overrightarrow{p_1}) \ : \ u_1, \\ \quad \ldots \\ \quad a(\overrightarrow{p_n}) \ : \ u_n\,; \\ \quad \ldots \end{array} \tag{4.4}$$

Dans ce cas, la première clause qui génère une condition de filtrage qui est satisfaite sera exécutée.

Pour chaque clause d'entrée de la forme $a(\overrightarrow{p}) : u$, le but est d'identifier les variables libres de $\overrightarrow{p}$ avec les paramètres actuels $\sigma_1, \ldots, \sigma_m$ de σ. La condition de filtrage (4.3) revient à la conjonction d'équations suivante :

$$\exists\,(var(\overrightarrow{p}) - \overrightarrow{k}) \bullet \bigwedge_j p_j = \sigma_j$$

où p_j est le j-ème paramètre formel de la clause $a(\overrightarrow{p})$ et $j \geq 1$. Pour évaluer chaque égalité, trois cas sont possibles :

1. si p_j est le symbole "$\_$", alors l'égalité est vraie ;
2. si p_j est un terme de base, alors l'égalité $p_j = \sigma_j$ est évaluée en calculant la valeur de p_j et en la comparant à σ_j ;
3. si p_j est une variable, alors p_j est évaluée à σ_j et l'égalité est supposée satisfaite. Si cette variable v a plusieurs occurrences dans $\overrightarrow{p}$, alors il faut vérifier aussi que les valeurs correspondantes sont les mêmes ; soit J_v la liste des indices où v apparaît dans $\overrightarrow{p}$:

$$\bigwedge_{j_1, j_2 \in J_v} \sigma_{j_1} = \sigma_{j_2}$$

Lorsqu'une condition de filtrage est satisfaite pour une clause de la forme $a(\overrightarrow{p}) : u$, alors une affectation de valeur a été déterminée pour chaque variable dans $var(\overrightarrow{p})$. On note par θ l'ensemble de ces affectations ; θ peut être considéré comme une substitution.

Par exemple, la définition de l'attribut *title* est :

$$
\begin{array}{l}
title(s : \mathcal{T}(\mathsf{main}), bId : bookKey\_Set) : Title\_Type \triangleq \\
\quad \textbf{match } last(s) \textbf{ with} \\
\qquad \begin{array}{lcll}
\bot & : & \bot, & \text{(I1)} \\
\mathsf{Acquire}(bId, bTitle) & : & bTitle, & \text{(I2)} \\
\mathsf{Discard}(bId) & : & \bot, & \text{(I3)} \\
\mathsf{Modify}(bId, nTitle) & : & nTitle, & \text{(I4)} \\
\_ & : & title(front(s), bId)\,; & \text{(I5)}
\end{array}
\end{array}
$$

Dans ce cas, on peut calculer les traces suivantes :

$$
\begin{array}{l}
title([\,], b_1) \overset{\text{(I1)}}{=} \bot \\
title([\mathsf{Register}(m_1)], b_1) \overset{\text{(I5)}}{=} title([\,], b_1) \overset{\text{(I1)}}{=} \bot \\
title([\mathsf{Acquire}(b_1, t_1), \mathsf{Register}(m_1), \mathsf{Modify}(b_1, t_2)], b_1) \overset{\text{(I4)}}{=} t_2
\end{array}
$$

Dans le premier cas, le résultat est directement obtenu par la clause d'entrée (I1), car $last([\,]) = \bot$. Dans le second cas, on utilise d'abord (I5) car il n'existe pas de clause avec Register, et puis on obtient le résultat en appliquant (I1). Dans le dernier cas, la valeur est obtenue directement avec (I4) et on a :

$$\theta = \{bId := b_1, nTitle := t_2\}$$

Les définitions d'attributs EB^3 ont les propriétés suivantes.

Terminaison

Les expressions u_i des clauses d'entrée peuvent contenir des appels récursifs. Cependant, la taille d'une trace valide est finie et décroît à chaque appel récursif. De plus, une clause d'entrée correspondant à une trace vide est définie par défaut dans toute définition d'attribut. Par conséquent, le calcul des valeurs d'attributs termine pour toute trace valide.

Condition de cohérence

Les définitions d'attributs doivent respecter la condition suivante. Quand un attribut non clé b retourne une valeur autre que $\bot$ pour une certaine valeur de clé $\overrightarrow{k}$, alors la définition de clé doit contenir $\overrightarrow{k}$:

$$\forall\, s, \overrightarrow{k} \;\bullet\; b(s, \overrightarrow{k}) \neq \bot \;\Rightarrow\; (\overrightarrow{k}) \in \kappa(s)$$

où κ est la définition de clé correspondante. Ainsi, les entités concernées par des valeurs d'attributs différentes de **NULL** doivent effectivement exister dans la base de données.

4.3 Conclusion

Dans ce chapitre, nous avons commencé par présenter le langage EB^3, tel qu'il a été défini par Frappier et St-Denis dans [FSD03]. Puis, nous avons introduit le langage de définition des attributs, que nous avons défini dans le cadre de la thèse.

EB^3 se distingue des autres approches de spécification des SI par sa sémantique basée sur les traces d'événements. Cette particularité constitue un atout pour modéliser le comportement des systèmes et, en particulier, pour spécifier les contraintes d'ordonnancement des actions ou bien les interactions entre les différentes entités du système. La forme des définitions d'attributs est une conséquence de cette modélisation. Ainsi, les fonctions récursives permettent d'associer à la trace courante du système les valeurs d'attributs qui sont valides pour cet état. La spécification de type fonctionnel permet de décrire, pour chaque attribut, quels sont les effets des différentes actions.

Cette forme de modélisation a des conséquences sur la spécification et l'implémentation des transactions du SI. En effet, la description des traces valides du SI est séparée de la définition des attributs. Par conséquent, la description d'une transaction se retrouve dispersée dans plusieurs parties d'une spécification EB^3. Par exemple, l'action Acquire est définie dans les expressions de processus comme le producteur du type d'entité *book*. D'autre part, il existe plusieurs occurrences de cette action dans les clauses d'entrée des définitions d'attributs : on la retrouve dans les fonctions *bookKey* et *title*. Enfin, l'action *Acquire* n'est pas utilisée dans les règles d'entrée-sortie, car elle ne renvoie aucun paramètre de sortie à l'exception de *res*.

Dans les chapitres qui suivent, nous présentons des techniques pour regrouper ces descriptions dans une spécification basée sur les transitions d'états. L'idée de cette approche consiste à proposer plusieurs vues du même modèle afin de prendre en compte le maximum d'exigences. D'autre part, les techniques que nous avons développées permettent également de générer automatiquement des transactions de bases de données relationnelles qui correspondent aux définitions d'attributs EB^3.

Deuxième partie

Intégration d'EB^3 et B

Chapitre 5

Vers une combinaison des approches EB^3 et B

> "Le raisonnement est l'art de comparer les vérités connues pour en composer d'autres vérités qu'on ignorait et que cet art nous fait découvrir."
>
> — Jean-Jacques Rousseau

Dans cette deuxième partie de la thèse, nous présentons nos contributions concernant l'intégration des langages EB^3 et B, afin de définir une nouvelle méthode de spécification formelle des SI appelée EB^4. Dans ce court chapitre d'introduction, nous rappelons les principales motivations de notre travail de recherche et nous discutons des raisons qui nous ont poussé à intégrer les langages EB^3 et B. Le chapitre est organisé de la manière suivante. Dans la section 5.1, nous présentons les idées que nous avons cherchées à concrétiser pendant la thèse. En particulier, nous dressons un bilan de l'état de l'art et nous discutons des hypothèses et conclusions que nous en avons tirées. Dans la section 5.2, nous présentons une première expérience de combinaison d'EB^3 et B qui nous a servi de support pour la définition de la méthode EB^4. Enfin, nous décrivons dans la section 5.3 l'organisation des chapitres suivants consacrés à l'intégration d'EB^3 et B.

5.1 Motivations et rappel du problème

L'expérience montre que les méthodes de conception actuelles des SI n'intègrent pas la modélisation du comportement de manière formelle. Notre objectif est de définir une méthode de spécification des SI qui soit formelle et adaptée pour prendre en compte le comportement du système. S'il existe des approches de spécification formelle dans le domaine des SI, les aspects dynamiques ne sont pas considérés au premier plan mais sont plutôt intégrés lors des phases ultérieures du développement (par exemple, avec des règles actives : voir chapitre 1).

Cette approche ne nous semble pas intéressante dans le cas de SI dont les transactions sont considérées comme critiques (par exemple, des transactions sur des comptes bancaires ou des mises à jour de données confidentielles). L'intérêt des méthodes formelles est la possibilité de vérifier des propriétés sur le modèle.

Si les aspects dynamiques ne sont pas pris en compte dès les premières étapes du développement, le SI risque de ne pas correspondre aux attentes du client.

En utilisant des méthodes de spécifications formelles, il est possible de faire des vérifications, de corriger au plus vite les erreurs du modèle et de le modifier afin de le rendre compatible avec les exigences du client. Le comportement des SI a donc tout intérêt à être modélisé de manière formelle. Toutefois, il est difficile avec les approches formelles de représenter à la fois les aspects statiques et dynamiques d'un système. D'une part, un langage basé sur les états comme Z [Spi92], Object-Z [Smi00] ou B [Abr96], permet de bien caractériser les structures de données et les effets des opérations sur les états, ainsi que les propriétés d'invariance sur les transitions d'états. D'autre part, un langage basé sur les événements, comme CSP [Hoa85], CCS [Mil89] ou EB^3 [FSD03], met en avant les comportements possibles d'un système, comme les propriétés de vivacité ou d'ordonnancement. La complémentarité des informations et des propriétés modélisées par ces deux types de modélisation nous a donc incité à étudier les méthodes existantes qui combinent plusieurs approches de spécification formelle.

En analysant l'état de l'art du chapitre 3, on se rend compte tout d'abord que les combinaisons de spécifications formelles constituent un axe de recherche important dans le domaine des méthodes formelles, avec la définition d'un grand nombre de nouvelles méthodes. Il n'existe pas de combinaison qui soit clairement destinée au domaine des SI. En particulier, les langages de processus comme CSP ont été définis de manière à prendre en compte tous les comportements possibles des systèmes modélisés. Dans les SI, on s'intéresse plutôt aux comportements admissibles et à la gestion des messages d'erreur en cas de demande erronée. L'une des forces d'EB^3 est la simplicité de son algèbre de processus pour décrire le comportement des événements à l'entrée du système. Contrairement à d'autres langages comme CSP, les expressions de processus EB^3 n'ont pas besoin de modéliser les éventuels blocages ou divergences du SI. En effet, la sémantique d'une spécification EB^3 prévoit que tout nouvel événement à l'entrée du SI doit être analysé afin de vérifier qu'il respecte les traces valides du système.

Concernant le niveau d'intégration entre les langages formels, plusieurs pistes sont possibles. Notre principal argument contre la création d'un nouveau langage comme Circus est la difficulté d'adaptation et de récupération des travaux existants. De plus, un tel langage peut paraître difficile à appréhender. Les approches de type juxtaposition demandent un effort d'analyse important afin de définir des liens suffisants entre les deux parties de la spécification. L'utilisation d'une traduction sémantique d'un langage vers un autre, comme dans csp2B, semble être une approche plus efficace pour vérifier des propriétés.

5.2 Preuve de propriétés EB^3 sur des spécifications B

Une première expérience de combinaison d'EB^3 et B, qui a été proposée par Marc Frappier et Régine Laleau en 2003, est présentée dans cette section. Ce travail exploratoire nous a servi de support pour la définition d'EB^4. Les propriétés d'ordonnancement des opérations d'une machine B sont décrites en EB^3. L'idée est de prouver que la machine B est un raffinement au sens B d'un modèle B des traces d'expressions de processus EB^3. Si c'est le cas, alors les

propriétés d'ordonnancement sont satisfaites par les opérations B. Le principe de l'approche est détaillé dans [FL03].

5.2.1 Exemple

Pour illustrer cette expérience, on reprend l'exemple de référence du chapitre 3 (voir figure 3.1). La spécification B est de la forme suivante :

MACHINE *Product*
SETS $PRODUCTS$; $STATES = \{Inactive, Active, Created\}$
CONSTANTS *Null*
PROPERTIES $Null \in PRODUCTS$
VARIABLES *Products*, *State*
INVARIANT $Products \in PRODUCTS \wedge State \in STATES$
INITIALISATION
  $Products := Null \parallel State := Inactive$
OPERATIONS
  $res \longleftarrow$ **On** =
  IF $State = Inactive$
  THEN $State := Active \parallel res :=$ "*ok*"
  ELSE $res :=$ "*error*"
  END ;

  $res \longleftarrow$ **Off** =
  IF $State = Active$
  THEN $State := Inactive \parallel res :=$ "*ok*"
  ELSE $res :=$ "*error*"
  END ;

  $res \longleftarrow$ **Create**(pdt) =
  PRE $pdt \in PRODUCTS$
  THEN
    IF $pdt \neq Null \wedge State = Active$
    THEN
      $Products := pdt \parallel State := Created \parallel$
      $res :=$ "*ok*"
    ELSE $res :=$ "*error*"
    END
  END ;

  $res \longleftarrow$ **Delete** =
  IF $State = Created$
  THEN
    $Products := Null \parallel State := Active \parallel$
    $res :=$ "*ok*"
  ELSE $res :=$ "*error*"
  END ;

  $pdt, res \longleftarrow$ **DisplayProduct** =
  IF $State \in \{Active, Created\}$

```
    THEN
        pdt := Products ||res := "ok"
    ELSE
        pdt := Products ||res := "error"
    END
END
```

Les opérations sont ici définies sous la forme IF THEN ELSE END afin de représenter les transitions déterministes du système et les préconditions sont utilisées uniquement comme contraintes de typage sur les paramètres d'entrée. Toutes les opérations ont au moins un paramètre de sortie, *res*, qui indique si l'opération a été correctement exécutée ("*ok*") ou pas ("*error*"). Ainsi, chaque opération de cette machine est disponible en permanence et elle s'exécute avec succès lorsque le prédicat de la partie IF est satisfait. Sinon, un message d'erreur est retourné. La machine contient deux variables d'état, *Products* and *State*, et cinq opérations : **On**, **Off**, **Create**, **Delete** et **DisplayProduct**.

En EB^3, le comportement des entrées/sorties du système est spécifié de la manière suivante. La signature des événements du système est :

```
On( ) : void
Off( ) : void
Create(pid : PRODUCTS) : void
Delete( ) : void
DisplayProduct( ) : (pdt : PRODUCTS)
```

Les actions EB^3 correspondent aux opérations homonymes de la machine B. Rappelons que le paramètre de sortie *res* n'est pas indiqué dans la signature, car il est représenté par la sémantique de la spécification EB^3 (voir section 4.1). Les actions On, Off et Delete n'ont ni entrée, ni sortie (*void*).

Concrètement, les entrées du système sont les événements reçus par le système. Dans notre exemple, un seul type d'entité est considéré : *Product*. Le processus suivant décrit le cycle de vie de toute entité *pid* de ce type :

```
Product(pid : PRODUCTS) =
    Create(pid).
    Delete
```

Ce processus utilise l'opérateur de séquence : une entité *pid* est créée par l'action Create(*pid*), puis détruite par l'action Delete.

Le comportement global du système est défini par le processus *main* :

```
main =
(
    On.
    (
        ( | pid : PRODUCTS : Product(pid) )*
    |||
        DisplayProduct*
    ).
    Off
)*
```

Ce processus décrit l'ensemble des traces valides d'événements à l'entrée du système. Il permet de représenter le LTS de l'exemple décrit dans la figure 3.1, à la page 39. D'autre part, l'action DisplayProduct peut être exécutée à tout moment, même si le produit n'existe pas. On rappelle que la notation " * " (appelée fermeture de Kleene) représente un nombre arbitraire fini d'exécutions du processus sur lequel elle est appliquée.

Le produit courant est calculé par la fonction récursive suivante :

$CurrentProduct(s : \mathcal{T}(\mathsf{main})) : PRODUCTS \triangleq$
match $last(s)$ **with**
$\bot : \bot,$
$\mathsf{Create}(pid) : pid,$
$\mathsf{Delete} : \bot,$
$\_ : CurrentProduct(front(s))\,;$

La règle d'entrée-sortie associée est la suivante :

RULE R1
INPUT DisplayProduct
OUTPUT *CurrentProduct*(*trace*)
END

où *trace* représente la trace courante du système. Ainsi, à chaque fois que DisplayProduct est considéré comme un nouvel événement valide à l'entrée du système, alors la fonction *CurrentProduct* est appelée pour déterminer quel est le produit courant (la valeur "⊥", qui correspond à **NULL**, est renvoyée le cas échéant).

5.2.2 Vérification de propriétés EB^3

La vérification de propriétés d'ordonnancement décrites en EB^3 par une machine M décrite en B est prouvée en traduisant tout d'abord l'expression de processus EB^3 en une machine B, puis en prouvant que cette dernière est raffinée, au sens B, par la machine M. La traduction en B de l'expression de processus EB^3 de l'exemple est de la forme suivante :

MACHINE *TranslationEB*3
SEES ...
VARIABLES t
INVARIANT $t \in \tau(main)$
INITIALISATION $t := []$
OPERATIONS
$res \longleftarrow$ **On** $=$
IF $t \leftarrow On \in \tau(main)$
THEN $t := t \leftarrow On \parallel res :=$ "ok"
ELSE $res :=$ "$error$"
END ;
...
END

Cette spécification permet d'exprimer en B la sémantique opérationnelle d'EB^3. Pour simplifier les notations, le même identifiant est utilisé pour représenter

l'opération B et l'action EB^3 correspondante. Par exemple, la notation **On** représente une opération B, tandis que *On* désigne l'action EB^3 associée.

La variable d'état t représente la trace courante du système. L'ensemble des traces valides est représenté par $\tau(main)$. L'invariant de la machine assure que la trace courante est toujours valide. Chaque opération correspond à une action de l'expression de processus EB^3. La clause SEES indique que les définitions de l'ensemble $\tau(main)$ et de la fonction *CurrentProduct* se trouvent dans une autre machine B. L'expression "$t \leftarrow u$" désigne l'ajout de l'élément u en queue de la séquence t. Comme toutes les opérations suivent le même patron, seule la traduction de l'opération **On** est indiquée ici. Pour chaque opération *op*, le principe de traduction est le suivant. Le prédicat de la partie IF de l'opération vérifie que $t \leftarrow op$ est une trace valide du système. Si c'est le cas, alors *op* est rajoutée en queue de la trace t et un message "*ok*" est retourné. Sinon, un message d'erreur est envoyé.

La vérification des propriétés d'ordonnancement consiste à prouver que la machine *Product* raffine la machine *TranslationEB3* dans la sémantique de B. La principale difficulté vient de la définition de l'invariant de collage entre les espaces d'états abstrait et concret. La variable d'état concrète *Products* représente le produit existant. Ce dernier est représenté dans la partie EB^3 par la sortie de la fonction récursive *CurrentProduct* lorsque cette dernière est appliquée à la trace courante t. Par conséquent, l'invariant de collage entre *Products* et t est :

$$Products \quad = \quad CurrentProduct(t)$$

En analysant le LTS de l'exemple (voir figure 3.1), les invariants de collage suivants sont déterminés pour les variables *State* et t :

$$\begin{aligned} State = Inactive &\Leftrightarrow t = [] \vee last(t) = Off \\ State = Active &\Leftrightarrow last(t) \in \{On, Delete, DisplayProduct\} \\ State = Created &\Leftrightarrow last(t) \in \{Create, DisplayProduct\} \end{aligned}$$

5.2.3 Discussion

Cette expérience de vérification de propriété d'ordonnancement sur une spécification B comporte plusieurs limites. Tout d'abord, la preuve de raffinement est difficile, car elle fait appel à l'expertise du concepteur, et il n'existe pas d'outils d'assistance ou de techniques de réutilisation spécifiques à cette approche. D'autre part, cette approche adopte une stratégie qui est opposée au développement classique d'un logiciel en B. En effet, dans l'approche ci-dessus, le raffinement (la spécification B sur laquelle est prouvée la propriété d'ordonnancement) est définie préalablement à la spécification abstraite (le modèle B de la spécification EB^3 qui décrit la propriété d'ordonnancement). Dans la méthode B, on commence par définir une spécification abstraite, avant de la raffiner. Par conséquent, une spécification B risque d'être mal adaptée pour la considérer comme un raffinement. Enfin, l'approche proposée est restreinte à la vérification. Notre objectif concerne plutôt la définition d'une méthode globale de spécification des SI.

Cette première ébauche nous conforte dans le choix des langages EB^3 et B. L'utilisation d'EB^3 pour décrire des propriétés d'ordonnancement est intéressante, car le langage B n'est pas doté d'opérateurs pour décrire une séquence

valide d'opérations. Une propriété d'ordonnancement est par conséquent représentée en B par la définition de préconditions et de substitutions adéquates dans les opérations concernées. Par exemple, la propriété "a, suivi d'un nombre fini de b, suivi de c" est spécifiée en EB^3 simplement par :

$$a \;.\; b^* \;.\; c$$

En B, il faut spécifier les opérations a, b et c de manière à vérifier que :
- il existe un état dans lequel l'opération a pourra être exécutée,
- l'état après avoir exécuté a correspond à un état dans lequel il est possible d'exécuter b ou c,
- chaque état après avoir exécuté b correspond à un état dans lequel b ou c peut être exécutée,
- il existe un état après avoir exécuté a ou b dans lequel c sera exécutée.

Parfois, l'ajout de variables supplémentaires qui ne représentent aucun attribut du système est nécessaire. Par exemple, dans la section 5.2, on remarque que la spécification B définit une variable d'état *State* pour représenter et distinguer les différents états du LTS de l'exemple.

Contrairement à CSP, EB^3 a été créé dans le but de spécifier des SI. La spécification des entrées et sorties du système est divisée en deux parties. Les expressions de processus représentent les traces valides des événements à l'entrée du système, alors que les sorties sont calculées à partir des traces valides. En particulier, les expressions de processus EB^3 sont similaires aux expressions régulières, avec notamment l'utilisation de l'opérateur de séquence et la fermeture de Kleene. Par exemple, l'opérateur d'entrelacement quantifié permettrait de considérer plusieurs produits en même temps dans l'exemple traité dans la section 5.2.2.

L'utilisation d'opérations B avec des substitutions de la forme IF THEN ELSE END est radicalement différente des styles de spécification de csp2B et CSP || B. Les préconditions sont uniquement utilisées comme contraintes de typage sur les paramètres d'entrée, tandis que les substitutions IF THEN ELSE END distinguent les appels valides et invalides des opérations. De plus, les opérations de la forme IF THEN ELSE END garantissent que le prédicat de la partie IF ne peut être ni affaibli, ni renforcé au cours du raffinement B. Par conséquent, la preuve du raffinement entre les modèles EB^3 et B implique que les LTS des actions EB^3 et des opérations B sont exactement équivalents. Cette première tentative d'intégration d'EB^3 et B se rapproche plus des travaux sur PLTL-Event B que de l'approche csp2B. La vérification des propriétés EB^3 consiste en une preuve de raffinement et non en une traduction.

5.3 De EB^3-B vers EB^4

L'approche présentée dans la section 5.2 constitue en fait le point de départ de nos travaux de recherche. Notre contribution a été de définir des techniques de synthèse de spécifications et de programmes afin de compléter les langages EB^3 et B et d'en déduire une méthode de spécification des SI. Plusieurs problèmes se sont posés pour prendre en compte le maximum d'exigences liées aux SI. Tout d'abord, dans quel ordre doit-on utiliser les langages EB^3 et B pour spécifier un nouveau SI ? En général, lors de la phase d'analyse, on commence par décrire les

exigences sous forme de fonctionnalités du système. Les propriétés sur le comportement sont indépendantes de la manière dont l'état du SI est représenté. Nous avons donc choisi de commencer la spécification en EB^3. Un autre problème est lié aux propriétés qu'on veut exprimer. Selon le type de propriétés, il faut déterminer dans quel langage (EB^3 ou B) elles doivent être spécifiées. La méthode doit donc préciser à chaque étape les éléments à spécifier et les propriétés à vérifier : par exemple, contraintes d'intégrité statiques en B, propriétés d'ordonnancement en EB^3. Enfin, il faut assurer la cohérence de l'ensemble du modèle. Il faut en particulier vérifier que les propriétés exprimées en EB^3 sont respectées par la spécification B, et réciproquement.

Une des principales difficultés de l'approche présentée dans la section 5.2 concerne la génération de la spécification B sur laquelle sont prouvées les propriétés EB^3 ; a priori, elle est supposée être définie indépendamment de la spécification EB^3. Or une telle description peut être difficile à déterminer. En effet, un raffinement consiste généralement à faire des choix d'implémentation. Si on considère la spécification EB^3 comme le modèle abstrait du SI et la spécification B comme son raffinement, alors la trace courante du système représentée par l'unique variable d'état t doit être raffinée en plusieurs variables d'état qui représentent les différents attributs du SI. La difficulté consiste à déterminer des variables pertinentes pour représenter le modèle de données.

Dans le cadre d'EB^4, la spécification B est générée à partir d'une partie de la spécification EB^3. Nous rappelons que les valeurs d'attributs sont calculées en EB^3 à l'aide de fonctions récursives sur les traces. Par conséquent, les définitions d'attributs de la spécification EB^3 décrivent le modèle de données. Cependant, il est difficile de vérifier des propriétés de sûreté sur ce type de fonctions. Afin de pouvoir vérifier des propriétés statiques sur le modèle de données, nous avons défini des règles de traduction qui permettent de générer systématiquement une représentation en B des définitions d'attributs EB^3. Le chapitre 6 présente la traduction en B du modèle de données EB^3. Cette étape est cruciale dans l'approche EB^4, car elle permet de passer d'une représentation à une autre.

Le chapitre 7 décrit ensuite les principales étapes de la méthode EB^4. Enfin, le chapitre 8 conclut cette partie sur l'intégration d'EB^3 et B avec les analyses et perspectives concernant l'approche EB^4.

Chapitre 6

Traduction vers B

> "Je peux comprendre et parler plus de 6 milliards de langages et dialectes."
>
> — Z-6PO dans Star Wars

Dans ce chapitre, nous présentons les algorithmes que nous avons développés pour traduire les définitions d'attributs EB^3 en B. La section 6.1 présente le principe général de la traduction. Puis, la génération de l'espace d'états et des invariants de la spécification B est détaillée dans la section 6.2. Dans la section 6.3, nous montrons comment générer les substitutions des opérations B. L'exemple de la bibliothèque est traité dans la section 6.4. Enfin, la section 6.5 conclut ce chapitre par une analyse et une discussion de la méthode.

6.1 Algorithme général

Notre objectif est de générer des spécifications B qui correspondent aux définitions d'attributs EB^3. En EB^3, une fonction est définie pour chaque attribut ; elle décrit les effets de chaque action sur l'attribut considéré. En B, la spécification est en quelque sorte "orthogonale", car les attributs sont définis comme des variables d'état et une opération B est définie pour chaque action. Chaque opération décrit ses effets sur les différents attributs du système. Les modifications sur les variables sont spécifiées à l'aide de substitutions. Pour mémoire, les principaux opérateurs du langage B sont présentés dans l'annexe B.

Lors de la phase de traduction, une opération B est définie pour chaque action EB^3 et, pour chaque opération, nos algorithmes permettent de générer les substitutions qui correspondent aux effets de l'action correspondante dans les définitions d'attributs EB^3. Les substitutions d'initialisation du système sont également générées. Cependant, il n'est pas possible de déterminer les préconditions des opérations en tenant compte uniquement des définitions d'attributs. Par conséquent, les opérations B obtenues ne sont pas complètes et ne définissent que des préconditions de typage sur les paramètres d'entrée. Elles seront utilisées dans l'approche EB^4 comme des squelettes de spécification à compléter.

L'algorithme général de traduction en B des définitions d'attributs consiste en quatre étapes :

1. générer la partie statique de la machine B à partir du diagramme ER et de la signature des actions EB^3,
2. générer les substitutions pour l'initialisation,
3. générer la signature et la précondition de typage de chaque opération B à partir de la signature des actions EB^3,
4. générer les substitutions des opérations.

La première étape, décrite dans la section 6.2, permet de définir les variables d'état qui représentent les attributs du SI. La section 6.3 présente les étapes suivantes. L'étape 2 est détaillée dans la section 6.3.1, tandis que les étapes 3 et 4 sont décrites dans la section 6.3.2. Dans la section 6.4, nous présentons quelques opérations B générées pour l'exemple de la bibliothèque afin d'illustrer les algorithmes décrits dans ce chapitre. Enfin, nous discutons des forces et des faiblesses de nos algorithmes dans la section 6.5.

6.2 Partie statique

En pratique, la spécification B qui est générée utilise les mêmes identifiants (noms de variables, de paramètres, de fonctions, etc ...) que dans la spécification EB^3. Afin de distinguer syntaxiquement les identifiants de la spécification EB^3 de ceux de la spécification B, nous notons par id_B l'identifiant B qui correspond à l'identifiant id_F en EB^3. Ainsi,

- la variable d'état b_B correspond à l'attribut b_F,
- le type T_B en B correspond au type T_F de la spécification EB^3,
- l'opération $\mathtt{a}_B$ est la traduction B de l'action $\mathtt{a}_F$,
- chaque paramètre formel q_{i_F} de l'action $\mathtt{a}_F$ est associé à son paramètre formel homologue q_{i_B} dans l'opération $\mathtt{a}_B$.

La partie statique de la spécification B est obtenue par traduction à partir du diagramme ER de la spécification EB^3. Cette traduction s'inspire de la formalisation en B des diagrammes de classes UML ou OMT [MS99, Ngu98, Mam02]. Pour simplifier les spécifications présentées dans cette thèse, nous regroupons toutes les opérations B dans la même machine B. Cependant, cette mise en forme n'a aucune influence sur les algorithmes décrits dans cette section, qui pourraient très bien être appliqués sur un ensemble de machines B composées entre elles par des liens d'inclusion incrémentale (clauses INCLUDES, SEES et USES de la méthode B).

Chaque fonction récursive k_F qui définit une clé en EB^3 est traduite en une variable d'état k_B en B. L'invariant pour k_B est une inclusion de la forme $k_B \subseteq T_B$, où T_B représente l'ensemble de toutes les valeurs possibles de la clé k_F. Par exemple, dans la spécification EB^3 de la bibliothèque, la définition de clé du type d'entité *book* est la fonction $bookKey_F$. Cette fonction est traduite en B en une variable d'état $bookKey_B$. On note $bookKey\_Set_B$ l'ensemble de toutes les valeurs possibles de $bookKey_F$. Par conséquent, la propriété d'invariance que doit respecter la variable $bookKey_B$ est :

$$bookKey_B \subseteq bookKey\_Set_B$$

Chaque attribut non clé b_F du diagramme ER est traduit en B par une variable d'état b_B. Soit k_B la variable B qui correspond à la clé de l'association ou du type d'entité dans lequel l'attribut b_F est défini et soit T_B l'ensemble de

toutes les valeurs possibles de l'attribut b_F. L'invariant généré pour b_B est de la forme $b_B \in k_B \rightarrow T_B$ ou $b_B \in k_B \nrightarrow T_B$, suivant que l'attribut b_F admette la valeur **NULL** (dans ce cas, la fonction est partielle : "$\nrightarrow$") ou pas (la fonction est totale : "$\rightarrow$"). Par exemple, l'attribut $title_F$ peut prendre la valeur **NULL** d'après la signature de l'action Modify_F. Par conséquent, la variable $title_B$ vérifie la propriété d'invariance suivante :

$$title_B \in bookKey_B \nrightarrow Title\_Type_B$$

Dans la suite, nous considérons uniquement des associations binaires, car une association n-aire peut toujours être décomposée en $n-1$ associations binaires si des contraintes d'intégrité supplémentaires sont ajoutées. Chaque association asc_F est traduite en une variable d'état asc_B qui est une relation entre les variables d'état qui représentent les clés des deux types d'entité qui participent à cette association. Selon les cardinalités de l'association asc_F, la variable asc_B, ou bien son inverse, est définie comme une fonction partielle, une fonction injective, etc ... Par exemple, une association de type *,* est représentée en B par une relation ("$\leftrightarrow$"), alors qu'une association *,0..1 est traduite en une fonction partielle. L'ensemble des règles de traduction pour les associations est détaillé dans [Ngu98]. Les rôles d'une association sont traduits par des définitions en B (dans la clause DEFINITIONS).

Concernant l'exemple de la bibliothèque, la spécification B générée par nos algorithmes pour la partie statique est :

MACHINE *B_Library*
SEES ... /* machine B qui définit les opérations sur le type DATE */
SETS
  $memberKey\_Set$; $bookKey\_Set$; $Title\_Type$;
  $Loan\_Type = \{Permanent, Classic\}$
VARIABLES
  $memberKey$, $nbLoans$, $loanDuration$, $bookKey$, $title$, $loan$,
    $dueDate$, $reservation$, $position$
INVARIANT
  $memberKey \subseteq memberKey\_Set \ \wedge \ nbLoans \in memberKey \rightarrow NAT \ \wedge$
  $loanDuration \in memberKey \rightarrow NAT \ \wedge \ bookKey \subseteq bookKey\_Set \ \wedge$
  $title \in bookKey \nrightarrow Title\_Type \ \wedge \ loan \in bookKey \nrightarrow memberKey \ \wedge$
  $dueDate \in loan \rightarrow DATE \ \wedge \ reservation \in bookKey \leftrightarrow memberKey \ \wedge$
  $position \in reservation \rightarrow NAT$
DEFINITION
  $borrower(x) \triangleq loan(x)$;
  $membRes(x) \triangleq reservation[\{x\}]$

Dans l'invariant, NAT est un type prédéfini du langage B qui représente l'ensemble des entiers naturels.

6.3 Génération des substitutions B

Dans cette section, nous présentons les algorithmes utilisés pour générer les substitutions B. Dans la section 6.3.1, nous considérons les substitutions d'initialisation de la spécification B. Ensuite, nous présentons dans la section 6.3.2

la partie la plus intéressante de nos algorithmes, qui concerne les substitutions des opérations B.

6.3.1 Substitutions pour l'initialisation

Pour chaque définition d'attribut, il existe une clause d'entrée de la forme $\perp : u$, qui est parfois définie par défaut. Cette clause correspond à la valeur initiale de l'attribut en question. En effet, lorsque $last(s) = \perp$, alors la trace s du système est vide et ce cas représente le SI à l'état initial.

Soit $b_{1_B}, \ldots, b_{m_B}$ l'ensemble des définitions d'attributs d'une spécification EB^3. Pour chaque b_{i_B}, une substitution S_i est générée pour l'initialisation. L'initialisation de la machine B est donnée par :

INITIALISATION
$S_1 \;||\; S_2 \;||\; \ldots \;||\; S_m$

Le symbole " || " signifie en B que les substitutions $S_1, ..., S_m$ sont exécutées en parallèle. Autrement dit, l'ordre d'exécution des substitutions n'a pas d'importance à ce niveau d'abstraction. La génération des S_i dépend du type de définition d'attribut. Dans le cas de l'exemple de la bibliothèque, l'initialisation obtenue est :

INITIALISATION
memberKey, *nbLoans*, *loanDuration*, *bookKey*, *title*, *loan*, *dueDate*, *reservation*, *position* := $\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing$

6.3.1.1 Définition de clé

Soient k_F une définition de clé et T_{k_F} le type des valeurs de clé. Nous considérons l'expression u de la clause d'entrée $\perp : u$ de k_F. Pour une définition de clé, u est nécessairement un terme fonctionnel. La valeur la plus courante pour u est $\varnothing$; dans ce cas, cela signifie qu'il n'y a aucune entité dans l'association ou le type d'entité identifié par k. La substitution générée pour l'initialisation est alors : $k_B := \varnothing$. L'autre expression possible pour u est un ensemble énuméré de la forme $\{c_{1_F}, \ldots, c_{n_F}\}$. Ce cas correspond à la création de n valeurs de clé pour k_F. La substitution B qui est générée alors est : $k_B := \{c_{1_B}, \ldots, c_{n_B}\}$, où les éléments $c_{1_B}, \ldots, c_{n_B}$ sont des constantes de type T_{k_B} qui doivent être définies dans la spécification B.

6.3.1.2 Définition d'attribut non clé

Soit b_F une définition d'attribut non clé dont la clé est k_F et dont les valeurs sont de type T_{b_F}. Dans ce cas, b_B est une fonction de k_B vers T_{b_B}. Nous considérons l'expression u de la clause d'entrée $\perp : u$ de la fonction b_F. La valeur la plus courante pour u est $\perp$; dans ce cas, aucune valeur n'est définie pour b_F et la substitution générée pour l'initialisation est : $b_B := \varnothing$. L'autre expression possible pour u est un terme conditionnel de la forme suivante :

if $k_F = c_{1_F}$ **then** d_{1_F}
else if $k_F = c_{2_F}$ **then** d_{2_F}
else ...
end
end,

où $c_{1_F}, \ldots, c_{n_F}$ sont des valeurs de clé de k_F et $d_{1_F}, \ldots, d_{n_F}$ sont des constantes de type T_{b_F}. D'après la condition de cohérence des définitions d'attributs EB^3 (voir section 4.2.2), les valeurs $c_{1_F}, \ldots, c_{n_F}$ doivent exister dans la BD pour être utilisées dans la clause d'initialisation ci-dessus. Par conséquent, $c_{1_F}, \ldots, c_{n_F}$ apparaissent dans une ou plusieurs clauses d'entrée de la forme $\bot : \{...\}$ des définitions de clé. En B, la notation "$a \mapsto b$" représente un élément d'une relation. La substitution d'initialisation générée pour b_F est de la forme suivante : $b_B := \{c_{1_B} \mapsto d_{1_B}, \ldots, c_{n_B} \mapsto d_{n_B}\}$, où $c_{1_B}, \ldots, c_{n_B}$ sont des constantes qui ont été déjà définies par l'initialisation des définitions de clé et $d_{1_B}, \ldots, d_{n_B}$ sont des constantes de type T_{b_B} à définir dans la machine B.

6.3.2 Génération des opérations

Pour chaque action $\mathtt{a}_F$ définie en EB^3, une opération a_B est générée en B. Ainsi, une action EB^3 dont la signature est :

$$\mathtt{a}_F(q_{1_F} : T_{1_F}, \ldots, q_{n_F} : T_{n_F}) : (q_{n+1_F} : T_{n+1_F}, \ldots, q_{m_F} : T_{m_F})$$

est traduite en B comme une opération de signature :

$$q_{n+1_B}, \ldots, q_{m_B}, res_B \longleftarrow \mathtt{a}_B(q_{1_B}, \ldots, q_{n_B}) \triangleq$$
PRE $q_{1_B} \in T_{1_B} \ \wedge \ \ldots \ \wedge \ q_{n_B} \in T_{n_F}$
THEN ...
END ;

D'après la sémantique opérationnelle d'une spécification EB^3 (voir section 4.1), lorsqu'un événement d'une action EB^3 est généré par l'utilisateur, le système retourne toujours une sortie qui permet d'indiquer si l'événement en question est valide ou pas. Le paramètre de sortie res_B de l'opération B permet de représenter cette sortie du système. Dans la suite, nous supposons pour simplifier que res_B ne peut prendre que deux valeurs : "*ok*" ou "*error*". Dans la pratique, le message res_B est un peu plus explicite et indique notamment quel type d'erreur a été commis. Les préconditions de l'opération B sont uniquement des contraintes de typage qui sont déduites de la signature de l'action EB^3 correspondante.

6.3.2.1 Algorithme général

L'algorithme général que nous avons défini pour générer les substitutions d'opérations est présenté dans la figure 6.1. Dans cette section, nous donnons un aperçu des principales étapes de l'algorithme. Pour définir les substitutions de a_B, nous devons analyser l'ensemble des clauses d'entrée des définitions d'attributs EB^3 afin de déterminer :

- quels attributs sont affectés par l'exécution de $\mathtt{a}_F$;
- quels sont les effets de $\mathtt{a}_F$ sur ces attributs.

Une définition d'attribut b_F peut être affectée par l'action $\mathtt{a}_F$ si elle comporte au moins une clause d'entrée de la forme $\mathtt{a}_F(\overrightarrow{p_I}) : u$. On note $B(\mathtt{a}_F)$ l'ensemble des définitions d'attributs qui satisfont ce critère. Pour éviter des confusions avec les paramètres formels de l'action $\mathtt{a}_F$, chaque paramètre p d'une clause d'entrée est maintenant dénoté par p_I. Pour chaque b_F dans $B(\mathtt{a}_F)$, plusieurs clauses d'entrée de la forme $\mathtt{a}_F(\overrightarrow{p_{j_I}}) : u_j$ peuvent exister pour la même action $\mathtt{a}_F$. La liste de ces clauses est notée $IC(b_F, \mathtt{a}_F)$. Comme l'ordre des clauses d'entrée a

pour chaque action a_F de la spécification EB^3
déterminer l'ensemble $B(\mathsf{a}_F)$ des attributs qui sont affectés par a_F
pour chaque attribut b_F de $B(\mathsf{a}_F)$
déterminer l'ensemble $IC(b_F, \mathsf{a}_F)$ des clauses d'entrée
de la forme $\mathsf{a}_F(\overrightarrow{p_I}) : u$
pour chaque clause $\mathsf{a}_F(\overrightarrow{p_I}) : u$ de $IC(b_F, \mathsf{a}_F)$
identifier les variables libres de $\overrightarrow{p_I}$ avec les paramètres
formels de a_F
identifier les hypothèses sous lesquelles u est valide
générer l'arbre de décision pour b_F
générer la substitution pour b_B
générer la substitution pour l'opération a_B

FIG. 6.1 – Algorithme de génération des substitutions d'opérations

une importance lors de l'évaluation des attributs, l'analyse de la liste $IC(b_F, \mathsf{a}_F)$ est réalisée dans l'ordre de déclaration des clauses dans leur définition d'attribut.

Supposons que $B(\mathsf{a}_F)$ soit de la forme suivante : $B(\mathsf{a}_F) = \{b_{1_F}, \ldots, b_{m_F}\}$. Une fois qu'une substitution S_{b_j} a été générée pour chaque définition d'attribut b_{j_F} de l'ensemble $B(\mathsf{a}_F)$, alors l'opération a_B générée est de la forme suivante :

$q_{n+1_B}, \ldots, q_{m_B}, res_B \longleftarrow \mathbf{a}_B(q_{1_B}, \ldots, q_{n_B}) \triangleq$
PRE $q_{1_B} \in T_{1_B} \ \wedge \ \ldots \ \wedge \ q_{n_B} \in T_{n_F}$
THEN
IF $CS \wedge CD$ THEN
$S_{b_1} \ || \ \ldots \ || \ S_{b_m} \ || \ res_B := \text{“}ok\text{”}$
ELSE
$res_B := \text{“}error\text{”}$
END
END ;

Rappelons que la substitution de la forme IF THEN ELSE END des opérations B générées permet de prendre en compte le paramètre de réponse implicite des actions EB^3. Cette réponse est représentée par le paramètre res_B. Les conditions décrites dans la clause IF doivent caractériser les états du système où un nouvel événement de l'action a_F est considéré comme valide ; autrement dit, le nouvel événement permet de respecter les traces valides du SI décrites par l'expression de processus *main* (voir section 4.1.2). Le prédicat IF est principalement composé de deux conditions :

1. la condition requise pour préserver l'invariant de la machine B. Cette condition est notée CS dans la suite.
2. la condition requise pour imposer les contraintes d'ordonnancement décrites dans le processus *main*. Cette dernière est notée par CD.

Nous avons proposé, dans le cadre d'EB^4, des techniques pour déterminer les conditions CS et CD. Elles seront présentées dans le chapitre 7.

Par exemple, l'opération B générée pour Acquire est la suivante :

$res \longleftarrow \mathbf{Acquire}(bId, bTitle) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ bTitle \in Title\_Type$

THEN
  IF $CS1 \wedge CD1$ THEN
    $bookKey := bookKey \cup \{bId\}$ ||
    $title := title \lhd\!+ \{bId \mapsto bTitle\}$ ||
    $res :=$ "ok"
  ELSE
    $res :=$ "$error$"
  END
END ;

Dans les définitions d'attributs, on ne trouve que deux occurrences de l'action Acquire dans les clauses d'entrée, dans les fonctions *bookKey* et *title* (ces définitions d'attributs sont détaillées respectivement dans les sections 4.2.1.1 et 4.2.2). C'est pourquoi on trouve dans la définition de l'opération des substitutions sur les variables B correspondantes. Les conditions $CS1$ et $CD1$ seront définies ultérieurement, dans les étapes suivantes de la méthode EB^4.

6.3.2.2 Analyse des clauses d'entrée

Lorsque l'expression u_j d'une clause d'entrée de la liste $IC(b_F, \mathsf{a}_F)$ est un terme fonctionnel, alors sa traduction en B est simple. Si l'expression u_j est un terme conditionnel, alors il faut analyser les différentes conditions spécifiées dans les prédicats des parties **if**. Intuitivement, le but de l'analyse est de retrouver quelles sont les valeurs de clé concernées par l'exécution de l'action a_F. Ainsi, lorsqu'un nouvel événement de l'action a_F est reçu par le système, on cherche à déterminer les valeurs $\{\overrightarrow{v}\}$ telles que : $b_F(t :: \mathsf{a}_F, \overrightarrow{v}) \neq b_F(t, \overrightarrow{v})$. D'autre part, les prédicats **if** des termes conditionnels peuvent aussi inclure des contraintes qui ne portent pas sur la clé de l'attribut, mais sur les valeurs des paramètres du nouvel événement. Ces contraintes sont appelées dans la suite des *hypothèses*.

Nous utilisons un arbre binaire appelé *arbre de décision* pour analyser les prédicats des termes conditionnels. Par exemple, le terme associé à la clause $\mathsf{Transfer}_F(bId_I, mId'_I, \_)$ dans la définition de $nbLoans_F$ comprend deux expressions **if then else end** imbriquées :

$nbLoans_F(s : \mathcal{T}(\mathsf{main}), mId : memberKey\_Set_F) : \mathbb{N} \triangleq$
match $last(s)$ **with**
  $\vdots$
  $\mathsf{Transfer}_F(bId_I, mId'_I, \_)$: **if** $mId = mId'_I$
    then $nbLoans_F(front(s), mId) + 1$
    else if $mId = borrower_F(front(s), bId_I)$
      then $nbLoans_F(front(s), mId) - 1$
      end
    end,
  $\vdots$

Le filtrage ne permet pas de déterminer la valeur de la variable clé mId de $nbLoans_F$. Les conditions dans les parties **if** caractérisent deux valeurs possibles pour mId. L'expression $nbLoans_F(front(s), mId)+1$ est associée à la valeur de clé $mId = mId'_I$, alors que l'expression $nbLoans_F(front(s), mId)-1$ correspond à la valeur déterminée par le prédicat $mId = borrower_F(front(s), bId_I)$. Ainsi,

le membre qui reçoit le transfert ($mId = mId'_I$) voit son nombre de prêts augmenté d'une unité, alors que l'emprunteur du livre bId_I avant le transfert ($mId = borrower_F(front(s), bId_I)$) voit son nombre de prêts diminué d'une unité. La construction et l'analyse des arbres de décision sont détaillées dans le paragraphe "Arbres de décision" ci-dessous.

L'algorithme consiste essentiellement à identifier les valeurs de clé affectées par chaque action et les hypothèses sous lesquelles l'action peut être exécutée. Afin de considérer tous les cas possibles, l'analyse de chaque clause d'entrée $\mathsf{a}_F(\overrightarrow{p_{j_I}}) : u_j$ de la liste $IC(b_F, a_F)$ génère :

- une substitution θ_{u_j} qui relie chaque paramètre actuel de la clause $\mathsf{a}_F(\overrightarrow{p_{j_I}})$ à un paramètre formel de l'action a_F de la spécification EB^3,
- une conjonction d'hypothèses H_{u_j} sous laquelle u_j est valide.

Les paragraphes suivants décrivent comment sont calculés θ_{u_j} et H_{u_j}.

Filtrage et hypothèses. Soit $\mathsf{a}_F(\overrightarrow{p_{j_I}})$ une clause d'entrée de $IC(b_F, \mathsf{a}_F)$. L'analyse décrite dans ce paragraphe est commune à toutes les clauses d'entrée, que le terme associé soit fonctionnel ou conditionnel. L'objectif est d'identifier les paramètres actuels de la clause $\mathsf{a}_F(p_{1_I}, \ldots, p_{m_I})$ avec les paramètres formels de l'action $\mathsf{a}_F(p_{1_F}, \ldots, p_{m_F})$, dans le but de retrouver pour quelles valeurs de $\overrightarrow{p_F}$ l'action a_F peut être exécutée :

$$\exists\ var(p_{1_I}, \ldots, p_{m_I}) \bullet \bigwedge_l p_{l_I} = p_{l_F}$$

On peut alors déterminer les variables du terme associé u_j en fonction des paramètres formels de a_F. Au début de l'analyse, le prédicat H_{u_j} est *true* et la substitution θ_{u_j} est vide. Ensuite, pour chaque paramètre p_{l_I} de la clause d'entrée, il y a trois cas à considérer :

1. Si p_{l_I} est le symbole "_", alors le terme u_j est valide pour chaque valeur possible du paramètre p_{l_F}. θ_{u_j} et H_{u_j} restent inchangés.
2. Si p_{l_I} est une constante c, alors le terme u_j est valide uniquement lorsque $p_{l_F} = c$. La condition $p_{l_F} = c$ est alors ajoutée en conjonction à l'hypothèse H_{u_j}. La substitution θ_{u_j} reste inchangée.
3. Si p_{l_I} est une variable v, alors le paramètre p_{l_I} est associé au paramètre formel p_{l_F} et la substitution $p_{l_I} := p_{l_F}$ est ajoutée dans θ_{u_j}. De plus, si cette variable v apparaît à plusieurs reprises dans l'ensemble des paramètres $p_{1_I}, \ldots, p_{m_I}$, alors il faut vérifier que les paramètres correspondants dans $p_{1_F}, \ldots, p_{m_F}$ sont les mêmes. Formellement, soit J_v la liste des indices des paramètres $p_{1_I}, \ldots, p_{m_I}$ où v apparaît, alors pour chaque $i \in J_v$, la substitution $p_{l_I} := p_{l_F}$ doit appartenir à θ_{u_j} et la condition $\bigwedge_{i_1, i_2 \in J_v}(p_{i_{1_F}} = p_{i_{2_F}})$ est ajoutée dans H_{u_j}.

Par exemple, prenons le cas de la clause $\mathsf{Transfer}_F(bId_I, mId'_I, \_)$ dans la définition de $nbLoans_F$. Par l'analyse décrite ci-dessus, on obtient :

$$\begin{aligned} \theta_{u_j} &= \{bId_I := bId_F, mId'_I := mId_F\} \\ H_{u_j} &= true \end{aligned}$$

S'il existe des hypothèses sur les paramètres, comme dans le cas des clauses d'entrée Lend_F dans la définition d'attribut $dueDate_F$:

$$
\begin{array}{l}
dueDate_F(s : \mathcal{T}(\mathsf{main}), bId : bookKey\_Set_F) : DATE \triangleq \\
\textbf{match } last(s) \textbf{ with} \\
\qquad \vdots \\
\quad \mathsf{Lend}_F(bId_I, \_, Permanent) \ : \ CurrentDate_F{+}365, \\
\quad \mathsf{Lend}_F(bId_I, mId_I, Classic) \ : \ CurrentDate_F \\
\qquad\qquad\qquad\qquad +loanDuration_F(front(s), mId_I), \\
\qquad \vdots
\end{array}
$$

alors les résultats de l'analyse sont respectivement pour la première et pour la deuxième clause :

$$
\begin{array}{rcl}
\theta_{u_1} & = & \{bId_I := bId_F\} \\
H_{u_1} & = & (type_F = Permanent)
\end{array}
$$

$$
\begin{array}{rcl}
\theta_{u_2} & = & \{bId_I := bId_F, mId_I := mId_F\} \\
H_{u_2} & = & (type_F = Classic)
\end{array}
$$

En résumé, les substitutions θ_{u_j} permettent de faire le lien entre les variables utilisées dans la clause d'entrée et les paramètres formels de l'action a_F. Comme les signatures (à l'exception du paramètre res_B) de a_B et a_F sont les mêmes, chaque paramètre de l'action EB[3] est identifié au paramètre correspondant dans l'opération B. Par conséquent, nous considérons dans la suite que, par extension, θ_{u_j} fait le lien entre les variables utilisées dans les clauses d'entrée et les paramètres formels des opérations B à générer.

Arbres de décision. La principale difficulté dans le processus de traduction concerne l'analyse des termes conditionnels. En effet, pour une même clause d'entrée, plusieurs valeurs de clé peuvent être affectées et les prédicats **if** peuvent aussi inclure des hypothèses sur les paramètres de l'événement. Afin d'analyser les différentes conditions des termes conditionnels, nous utilisons des arbres binaires, appelés *arbres de décision*. Dans la suite, l'application de la substitution θ au terme u est notée par l'expression "$u\ \theta$" ; elle est définie comme le remplacement, pour chaque expression de la forme $x := y$ dans θ, de toutes les occurrences de x dans u par y.

Rappelons que seules des définitions d'attributs non clé peuvent inclure des termes conditionnels (voir section 4.2.1). Soit b_F une définition d'attribut non clé de $B(\mathsf{a}_F)$. Pour chaque clause d'entrée $\mathsf{a}_F(\overrightarrow{p_{jI}}) : u_j$ de la liste $IC(b_F, \mathsf{a}_F)$, on calcule tout d'abord θ_{u_j} et H_{u_j}. Ensuite, l'arbre de décision associé à b_F est construit en deux temps.

Dans un premier temps, la racine de l'arbre est b_F et les feuilles correspondent aux différentes clauses d'entrée $a_F(\overrightarrow{p_{jI}}) : u_j$ de $IC(b_F, \mathsf{a}_F)$. Pour conserver la forme binaire de l'arbre, la construction est réalisée de la manière suivante. Le nombre n de niveaux dans l'arbre est le nombre de clauses d'entrée dans $IC(b_F, \mathsf{a}_F)$. Le fils gauche de la racine est $u_1\ \theta_{u_1}$ et la branche correspondante est étiquetée par H_{u_1}. Le fils droit de la racine est soit le sous-arbre généré au niveau suivant, soit l'appel récursif $b_F(front(s), \overrightarrow{k_F})$ s'il n'y a qu'une seule clause d'entrée. La branche vers le fils droit est étiquetée par $\neg H_{u_1}$. Les sous-arbres

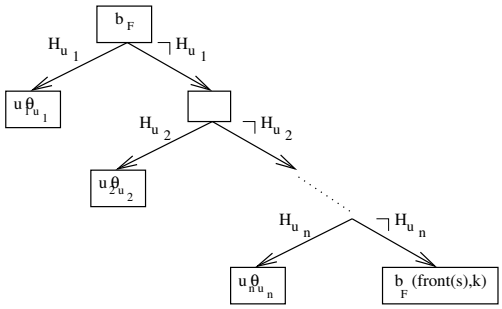

FIG. 6.2 – Première étape de la construction de l'arbre de décision de b_F

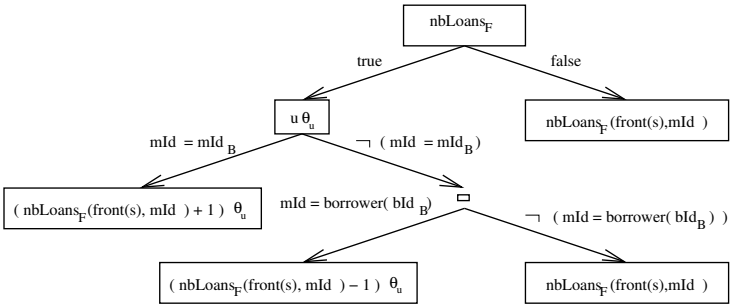

FIG. 6.3 – Arbre de décision généré pour la clause $\mathsf{Transfer}_F$ de $nbLoans_F$

sont générés de la manière suivante. Au niveau $j-1 \neq n$, le fils gauche est $u_j\ \theta_{u_j}$ avec une branche étiquetée par H_{u_j} et le fils droit est le sous-arbre de niveau j avec une branche étiquetée par $\neg H_{u_j}$. Au niveau n, le sous-arbre est simplement la feuille $b_F(front(s), \overrightarrow{k_F})$. Cette construction est illustrée par la figure 6.2.

Dans un deuxième temps, chaque feuille de la forme $u_j\ \theta_{u_j}$, où u_j est un terme conditionnel, est remplacée par un sous-arbre dont les feuilles ne contiennent que des termes fonctionnels. Le sous-arbre de décision généré pour un terme conditionnel u_j est de la forme suivante. La racine est $u_j\ \theta_{u_j}$. Les fils de chaque nœud sont respectivement les parties **then** et **else** du terme conditionnel considéré, avec comme étiquettes respectives sur les branches les conditions c et $\neg c$, où c est la condition de la partie **if**. Le processus est réitéré sur les parties **then** tant que toutes les feuilles de l'arbre ne sont pas des termes fonctionnels. Par exemple, dans le cas de la clause $\mathsf{Transfer}_F$ de la définition d'attribut $nbLoans_F$, on obtient l'arbre de décision décrit dans la figure 6.3.

6.3.2.3 Génération des substitutions

Pour chaque définition d'attribut b_F de $B(\mathsf{a}_F)$, l'arbre de décision de b_F est analysé afin de générer une *formule de substitution* qui représente l'ensemble des substitutions possibles sur la variable correspondante b_B. Dans les définitions suivantes, θ_{u_j} non seulement relie les variables libres d'une clause d'entrée aux paramètres formels d'une opération B, mais transforme aussi chaque occurrence

de l'appel $b_F(front(s), \overrightarrow{p_F})$ en une expression B de la forme $b_B(\overrightarrow{p_B})$. En B, un tuple "$u \mapsto v$" d'une relation peut également être noté par "(u, v)". Dans la suite, nous utilisons indifféremment ces deux notations qui sont équivalentes.

Définitions de clé. Soit b_F une définition de clé de $B(\mathsf{a}_F)$. Par définition, les expressions u_j des clauses d'entrée de la liste $IC(b_F, \mathsf{a}_F)$ sont toutes des termes fonctionnels. Pour générer la substitution b_B, il y a deux cas à considérer :

1. b_F est la clé d'un type d'entité ou d'une association $M : N$,
2. b_F est la clé d'une association de type $1 : N$.

Dans le premier cas, la substitution b_B générée pour l'opération a_B est directement définie par :

$$b_B := u_j\ \theta_{u_j}$$

Par exemple, la substitution générée pour la clause $\mathsf{Acquire}_F$ dans la définition de clé $bookKey_F$ du type d'entité *book* (voir section 4.2.1.1) est :

$$bookKey_B := (bookKey_F(front(s)) \cup \{bId_I\})\theta_u$$

où $\theta_u = \{bId_I := bId_B, bookKey_F(front(s)) := bookKey_B\}$. Ensuite, par application de θ_u, on obtient la substitution B suivante :

$$bookKey_B := bookKey_B \cup \{bId_B\}$$

Dans le deuxième cas, la cardinalité de l'association identifiée par b_F est de type $1 : N$. Le lien entre la définition de clé en EB^3 et la variable B de l'association est alors plus difficile à établir, car les représentations en EB^3 et en B de ce type d'associations sont de natures différentes. Par exemple, la définition de clé de *loan* est de type :

$$loan_F\ :\ \mathcal{T}(\mathsf{main}) \rightarrow \mathbb{F}(bookKey\_Set_F)$$

alors que la variable B correspondante est de type :

$$loan_B \in bookKey_B \nrightarrow memberKey_B$$

En EB^3, la fonction $loan_F$ renvoie uniquement comme valeurs de clé les livres empruntés. Or, pour certaines substitutions sur $loan_B$, il faut donner explicitement le couple (bId, mId), car la variable $loan_B$ est une fonction partielle des livres vers les membres. Il faut alors retrouver dans la spécification EB^3 les emprunteurs associés à ces livres ; ils sont déterminés grâce au rôle $borrower_F$. Contrairement à la modélisation B, où un rôle est décrit comme une abréviation de la variable qui représente son association, la définition d'un rôle en EB^3 est complémentaire des définitions d'attributs et obligatoire (voir section 4.2.1.3). La définition de clé de l'association *loan* est :

$$\begin{array}{l} loan(s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(bookKey\_Set) \ \triangleq \\ \textbf{match}\ last(s)\ \textbf{with} \\ \quad \mathsf{Lend}(bId, \_, \_)\ :\ loan(front(s)) \cup \{bId\}, \\ \quad \mathsf{Return}(bId)\ :\ loan(front(s)) - \{bId\}, \\ \quad \mathsf{Take}(bId, \_, \_)\ :\ loan(front(s)) \cup \{bId\}, \\ \quad \_\ :\ loan(front(s))\,; \end{array}$$

La définition du rôle *borrower* a été présentée dans la section 4.2.1.3.

Pour générer la substitution $loan_B$, il y a deux sous-cas possibles :
- soit a_F est une clause d'entrée dans $loan_F$ et dans $borrower_F$,
- soit a_F est une clause d'entrée uniquement dans $borrower_F$.

Dans le premier sous-cas, la substitution $loan_B$ est alors de la forme :

$$loan_B := u_j^* \ \theta_{u_j}$$

où u_j^* est défini comme u_j, mais chaque occurrence de bId_I y est remplacée par :
- le couple (bId_I, m), où m est le membre associé à bId_I par la fonction $borrower_F$, si u_j est de la forme $loan_F(front(s)) \cup S$;
- le couple $(bId_I, borrower_F(front(s), bId_I))$, si le terme u_j est de la forme $loan_F(front(s)) - S$.

Par exemple, l'action Lend_F apparaît à la fois dans les clauses d'entrée de $loan_F$ et $borrower_F$. Par conséquent, la substitution sur la variable $loan_B$ générée pour l'opération $\mathbf{Lend}_B$ est :

$$loan_B := loan_B \cup \{(bId_B, mId_B)\}$$

où mId_B est déduit du terme fonctionnel associé à la clause Lend_F dans la définition de $borrower_F$. L'action Return_F appartient aussi aux clauses d'entrée de $loan_F$ et $borrower_F$, mais u_j est de la forme $loan_F(front(s)) - S$. La substitution générée pour $\mathbf{Return}_B$ est la suivante :

$$loan_B := loan_B - \{(bId_B, borrower_F(front(s), bId_I))\}$$

Dans le second sous-cas, où la clause a_F n'apparaît que dans $borrower_F$, la substitution est de la forme suivante :

$$loan_B := loan_B \Leftarrow \{(bId_I, m) \ \theta_{u_j}\}$$

où m est le membre associé à bId_I par $borrower_F$. Par exemple, l'action $\mathsf{Transfer}_F$ ne fait pas partie des clauses d'entrée de $loan_F$. La fonction $borrower_F$ permet alors de retrouver quel est le nouvel emprunteur du livre bId après avoir exécuté l'opération $\mathbf{Lend}_B$. La substitution générée est :

$$loan_B := loan_B \Leftarrow \{(bId_B, mId_B)\}$$

Définition d'attribut non clé. Si b_F est un attribut non clé, alors il peut contenir des termes conditionnels. Les chemins intéressants dans l'arbre de décision associé à b_F sont ceux qui partent de la racine et qui conduisent à une feuille qui ne contient pas un simple appel récursif de b_F. Soit $u_{j,i}$ le terme fonctionnel de la i-ème feuille $u_{j,i} \ \theta_{u_j}$ du sous-arbre de décision correspondant à la j-ème clause d'entrée de la liste $IC(b_F, \mathsf{a}_F)$. On note par $\Phi_{u_{j,i}}$ la conjonction des conditions ϕ qui apparaissent sur les étiquettes des branches du chemin qui mène au terme fonctionnel $u_{j,i}$. La formule de substitution pour la variable b_B est de la forme :

$$b_B := (A \lhd b_B) \Leftarrow R$$

où l'ensemble A représente les tuples à supprimer et R, ceux qui sont à insérer ou à mettre à jour. Formellement, A est défini, pour chaque terme fonctionnel $u_{j,i}$ des feuilles de l'arbre de décision tel que $u_{j,i} = \bot$, par :

$$A = \{c \mid \exists \ \overrightarrow{k} \cdot \bigvee_j \left(c = (\overrightarrow{k})\theta_{u_j} \ \wedge \ \bigvee_i \Phi_{u_{j,i}} \theta_{u_j} \right)\}$$

et R est défini, pour chaque terme fonctionnel $u_{j,i}$ des feuilles de l'arbre de décision tel que $u_{j,i} \neq \bot$, par :

$$R = \{(c,d) \mid \exists \overrightarrow{k} \cdot \bigvee_j \left(c = (\overrightarrow{k})\theta_{u_j} \quad \wedge \quad \bigvee_i \left(\Phi_{u_{j,i}} \wedge d = u_{j,i}\right) \theta_{u_j} \right)\}$$

Les principaux opérateurs du langage B sont présentés dans l'annexe B.

Nous expliquons maintenant ce que représentent les ensembles A et R. La disjonction sur j représente les différentes clauses d'entrées avec la même action dans la définition d'attribut b_F. Ainsi, chaque branche j dans A et dans R correspond à une clause d'entrée. Lorsque $u_{j,i}$ est égal à $\bot$, alors l'attribut b_F devient indéfini pour la valeur de clé correspondante, qui est déterminée par θ_{u_j} et $\Phi_{u_{j,i}}$. Chaque branche i de la disjonction dans l'ensemble A représente une de ces valeurs de clé. Par exemple, l'action $\mathsf{Discard}_F(bId_F)$ affecte la valeur $\bot$ à l'attribut $title_F$ pour le type d'entité bId. D'après la formule ci-dessus, on en déduit la substitution suivante : $title_B := A \lhd title_B$, où A est défini par :

$$A = \{c \mid \exists\, bId_I \cdot c = bId_I\ \theta_u \wedge true\ \theta_u\}$$

Dans cet exemple, bId_I est directement lié par le filtrage, car θ_u contient la substitution $bId_I := bId_B$. On en déduit alors que : $A = \{bId_B\}$. La substitution générée pour la variable $title_B$ dans l'opération $\mathbf{Discard}_B$ est donc : $title_B := \{bId_B\} \lhd title_B$.

R est l'ensemble des tuples qui affectent de nouvelles valeurs à l'attribut b_B. La première disjonction sur j représente les différentes clauses d'entrée avec la même action dans la définition d'attribut b_F. Considérons maintenant la j-ème clause d'entrée. Si u_j est un terme fonctionnel, alors le tuple correspondant est (c,d), où c est la valeur de clé déterminée par θ_{u_j} et d est la valeur d'attribut $u_j\theta_{u_j}$. Si u_j est un terme conditionnel, alors plusieurs valeurs de clé sont concernées ; elles sont caractérisées par θ_{u_j} et $\Phi_{u_{j,i}}$. Chaque branche i représente alors une valeur de clé et sa valeur d'attribut associée $u_{j,i}\theta_{u_j}$. Par exemple, le terme conditionnel pour la clause $\mathsf{Transfer}_F$ dans la définition d'attribut $nbLoans_F$ implique deux entités de $member$: mId'_I et $borrower_F(front(s), bId_I)$. La formule de substitution générée alors est : $nbLoans_B := nbLoans_B \lhd\!\!+ R$, où R est défini par :

$$\begin{array}{rcl} R & = & \{(c,d) \mid \exists\, mId \cdot (c = mId)\ \theta_u \wedge \\ & & ((mId = mId'_I \wedge d = nbLoans_F(front(s), mId){+}1)\ \theta_u) \vee \\ & & ((mId \neq mId'_I \wedge mId = borrower_F(front(s), bId_I) \\ & & \wedge d = nbLoans_F(front(s), mId){-}1)\ \theta_u)\} \end{array}$$

On remarque que mId reste ici une variable libre, même après avoir pris en compte le filtrage : $\theta_u = \{bId_I := bId_B, mId'_I := mId_B\}$.

6.3.2.4 Réécriture des formules de substitution

Les substitutions générées par notre algorithme sont syntaxiquement correctes en B, mais peuvent parfois être difficiles à prouver. Nous avons défini quelques règles de réécriture afin de transformer les substitutions obtenues. En particulier, des substitutions de la forme IF THEN ELSE END sont extraites de

la formule générale définie dans la section 6.3.2.3 lorsque cette dernière implique des hypothèses sur les paramètres d'entrée de l'opération B.

Règle 1 (Application des substitutions θ_{u_j}) *Soit E un des ensembles A ou R dans la formule de substitution. Pour chaque j, la substitution θ_{u_j} est appliquée à l'expression correspondante $expr_j = expr\ \theta_{u_j}$ dans E de la manière suivante : pour chaque substitution $v := q$ dans θ_{u_j}, où v est une variable de E, alors chaque occurrence de v dans $expr_j$ est remplacée par q et la notation "θ_{u_j}" est supprimée de l'expression $expr_j$. Si v est une des variables de $\overrightarrow{k}$, alors l'expression "$\exists\, v$" est supprimée de l'expression $expr_j$.*

Les quantifications existentielles dans les ensembles A et R de la formule de substitution peuvent être simplifiées en utilisant la règle suivante. Nous supposons que c est de la forme $c = (c_1, \ldots, c_m)$.

Règle 2 (Règle d'application "one-point") *Soit E un des ensembles A ou R. Pour chaque condition $c_l = w$, où w est une variable liée de E ou un paramètre formel de l'action $\mathtt{a}_F$, alors :*

- *chaque occurrence de w dans E est remplacée par c_l,*
- *la condition $c_l = w$ est supprimée de E,*
- *si w est une des variables de $\overrightarrow{k}$, alors l'expression $\exists\, w$ est supprimée de E.*

Après l'application des règles (1) et (2), les ensembles A et R de la formule de substitution sont maintenant de la forme $\{e \mid \bigvee_{i,j} expr_{i,j}\}$. Il existe principalement deux formes de condition dans les expressions $expr_{j,i}$: i) des conditions qui permettent de déterminer les valeurs des tuples concernés par les substitutions et ii) des hypothèses sur les paramètres d'entrée de l'opération sous lesquelles les substitutions sont valides. Les conditions de la forme i) ont une incidence sur les variables et les paramètres affectés par les substitutions, tandis que les conditions de la forme ii) indiquent si une substitution peut être exécutée ou non. La règle suivante permet de réécrire la formule de substitution en exprimant les hypothèses de la forme ii) dans les clauses IF de substitutions de la forme IF THEN ELSE END.

Règle 3 (Détection des hypothèses) *Soit E de la forme A ou R. Pour chaque j, pour chaque i, chaque condition de la forme $p_F = w$ ou $p_F \in T$ dans $expr_{i,j}$ génère une expression de la forme IF $cond$ THEN S_1 ELSE S_2 END, où $cond$ est la condition $p_F = w$ ou $p_F \in T$, S_1 est l'ensemble E restreint aux expressions $expr_{j,i}$ pour lesquelles la condition $cond$ est satisfaite, et S_2 est l'ensemble E restreint aux expressions $expr_{j,i}$ pour lesquelles la condition $cond$ est fausse. Chaque occurrence de $cond$ et $\neg cond$ est supprimée de S_1 et S_2.*

L'application de la règle (3) est simplifiée par le fait que les conditions $\Phi_{u_{j,i}}$ ont été déduites à partir d'un arbre binaire. Par conséquent, il existe une condition négative de la forme $\neg cond$ pour toute condition $cond$ apparaissant sur une étiquette des branches de l'arbre.

Pour illustrer l'application de ces règles, considérons par exemple l'ensemble R obtenu pour l'attribut $nbLoans_B$ dans la section 6.3.2.3 :

$$R \quad = \quad \{(c, d) \mid \exists\, mId \cdot (c = mId)\ \theta_u \wedge$$

$$((mId = mId'_I \wedge d = nbLoans_F(front(s), mId)+1)\ \theta_u) \vee$$
$$((mId \neq mId'_I \wedge mId = borrower_F(front(s), bId_I)$$
$$\wedge d = nbLoans_F(front(s), mId)-1)\ \theta_u)\}$$

Dans un premier temps, on applique la règle (1), avec comme substitution $\theta_u = \{bId_I := bId_B, mId'_I := mId_B\}$:

$$\begin{aligned} R \quad = \quad & \{(c, d) \mid \exists\, mId \cdot c = mId \wedge (mId = mId_B \wedge d = nbLoans_B(mId)+1) \vee \\ & (mId \neq mId_B \wedge mId = borrower_B(bId_B) \wedge d = nbLoans_B(mId)-1)\} \end{aligned}$$

On élimine ensuite la quantification existentielle en appliquant la règle (2) :

$$\begin{aligned} R \quad = \quad & \{(c, d) \mid (c = mId_B \wedge d = nbLoans_B(c)+1) \vee \\ & (c \neq mId_B \wedge c = borrower_B(bId_B) \wedge d = nbLoans_B(c)-1)\} \end{aligned}$$

À cette étape, on ne peut plus appliquer de règle, car les conditions ne sont pas des hypothèses sur les paramètres formels de l'opération. Toutefois, on pourrait simplifier l'ensemble R en un ensemble énuméré si la condition $mId_B \neq borrower_B(bId_B)$ était satisfaite :

$$\begin{aligned} R \quad = \quad & \{(mId_B \mapsto nbLoans_B(mId_B)+1), \\ & (borrower_B(bId) \mapsto nbLoans_B(borrower_B(bId_B))-1)\} \end{aligned}$$

Cependant, cette condition ne peut pas être déduite des définitions d'attributs et nous laissons l'ensemble R sous sa forme en extension. Dans le chapitre 7, nous verrons que la condition $mId_B \neq borrower_B(bId_B)$ peut être déduite des substitutions si on impose que l'ensemble soit énuméré.

Un autre exemple intéressant concerne l'attribut *dueDate* dans l'association *loan* (voir section 4.2.1.2). Dans ce cas, les conditions $\Phi_{u_{j,i}}$ générées lors de la construction de l'arbre de décision des clauses d'entrée associées à l'action $\mathsf{Transfer}_F$ incluent des hypothèses sur les paramètres formels de l'action. Il en est de même pour les clauses associées aux actions Lend_F et Take_F. Par exemple, la formule de substitution générée pour $\mathsf{Transfer}_F$ est :

$$dueDate_B := dueDate_B \lhd\!\!+ R$$

où R est défini par :

$$\begin{aligned} R \quad = \quad & \{(c, d) \mid \exists (bId, mId) \cdot (c = (bId, mId)\ \theta_{u_1} \ \wedge \\ & ((type_F = Permanent \ \wedge \ d = CurrentDate_F+365)\ \theta_{u_1})) \vee \\ & (c = (bId, mId)\ \theta_{u_2} \ \wedge \ ((type_F \neq Permanent \ \wedge \ type_F = Classic \wedge \\ & d = CurrentDate_F+loanDuration_F(front(s), mId_I))\ \theta_{u_2}))\} \end{aligned}$$

où $\theta_{u_1} = \theta_{u_2} = \{bId_I := bId_B, mId_I := mId_B\}$. En appliquant les règles (1) et (2), on obtient :

$$\begin{aligned} R \quad = \quad & \{(c, d) \mid (c = (bId_B, mId_B) \ \wedge \\ & (type_F = Permanent \ \wedge \ d = CurrentDate_B+365)) \vee \\ & (c = (bId_B, mId_B) \ \wedge \ (type_F \neq Permanent \ \wedge \ type_F = Classic \wedge \\ & d = CurrentDate_B+loanDuration_B(mId_B)))\} \end{aligned}$$

La règle (3) peut maintenant être appliquée puisqu'il existe des conditions sur le paramètre formel $type_F$ de l'action $\mathsf{Transfer}_F$. La première condition $type_F = Permanent$ génère l'expression suivante :

$$
\begin{array}{rcl}
R & = & \text{IF } type_B = Permanent \text{ THEN} \\
 & & \{(c,d) \mid c = (bId_B, mId_B) \ \wedge \ d = CurrentDate_B{+}365\} \\
 & & \text{ELSE} \\
 & & \{(c,d) \mid c = (bId_B, mId_B) \ \wedge \\
 & & type_F = Classic \wedge d = CurrentDate_B{+}loanDuration_B(mId_B)\} \\
 & & \text{END}
\end{array}
$$

La règle (3) est de nouveau appliquée sur la condition $type_F = Classic$ dans la partie ELSE et on obtient :

$$
\begin{array}{rcl}
R & = & \text{IF } type_B = Permanent \text{ THEN} \\
 & & \{(c,d) \mid c = (bId_B, mId_B) \ \wedge \ d = CurrentDate_B{+}365\} \\
 & & \text{ELSE IF } type_F = Classic \text{ THEN} \\
 & & \{(c,d) \mid c = (bId_B, mId_B) \\
 & & \ \wedge \ d = CurrentDate_B{+}loanDuration_B(mId_B)\} \\
 & & \text{END} \\
 & & \text{END}
\end{array}
$$

Dans la pratique, cette étape n'est pas nécessaire, car le paramètre $type_B$ ne peut prendre que deux valeurs, mais il est parfois difficile de détecter ce type de situation de manière systématique.

Lorsque la règle (3) ne peut plus être appliquée sur A ou sur R, on fait une analyse des expressions IF THEN ELSE END pour en déduire la substitution finale. Dans l'exemple précédent, les hypothèses n'impliquent que l'ensemble R ; par conséquent, l'expression IF THEN ELSE END peut porter sur l'ensemble de la substitution et on obtient finalement la substitution suivante :

$$
\begin{array}{l}
\text{IF } type_B = Permanent \\
\text{THEN } dueDate_B := dueDate_B \\
\quad \lhd\!\!+ \{(bId_B, mId_B) \mapsto CurrentDate_B{+}365\} \\
\text{ELSE } dueDate_B := dueDate_B \\
\quad \lhd\!\!+ \{(bId_B, mId_B) \mapsto CurrentDate{+}loanDuration(mId)\} \\
\text{END}
\end{array}
$$

6.4 Exemple de la bibliothèque

Dans cette section, nous présentons les principales opérations B de l'exemple de la bibliothèque, dont la génération des substitutions a été discutée dans les sections précédentes. La spécification complète, obtenue en appliquant les algorithmes décrits dans le présent chapitre, est détaillée dans l'annexe C.

L'exemple de l'opération **Acquire** a été présenté dans la section 6.3.2.1. La génération des substitutions de cette opération ne pose pas de problème particulier, puisque les conditions de filtrage déterminent directement les valeurs de clé affectées par l'exécution de Acquire. Il en est de même pour la génération de l'opération **Discard** :

$res \longleftarrow$ **Discard**$(bId) \triangleq$
PRE $bId \in bookKey\_Set$
THEN
IF $CS2 \wedge CD2$ THEN
$bookKey := bookKey - \{bId\}$ ||
$title := \{bId\} \lhd title$ ||
$res :=$ "ok"
ELSE
$res :=$ "$error$"
END
END ;

Dans le cas de l'opération **Lend**, l'analyse des définitions d'attributs requiert la construction d'un arbre de décision pour la fonction *dueDate* (présentée dans la section 4.2.1.2) afin d'identifier les hypothèses sur le paramètre *type* :

$res \longleftarrow$ **Lend**$(bId, mId, type) \triangleq$
PRE $bId \in bookKey\_Set \wedge mId \in memberKey\_Set$
$\wedge\ type \in Loan\_Type$
THEN
IF $CS6 \wedge CD6$ THEN
$nbLoans := nbLoans \Leftarrow \{mId \mapsto 1+nbLoans(mId)\}$ ||
$loan := loan \cup \{(bId, mId)\}$ ||
IF $type = Permanent$
THEN $dueDate := dueDate$
$\Leftarrow \{(bId, borrower(bId)) \mapsto CurrentDate+365\}$
ELSE $dueDate := dueDate$
$\Leftarrow \{(bId, mId) \mapsto CurrentDate+loanDuration(mId)\}$
END ||
$res :=$ "ok"
ELSE
$res :=$ "$error$"
END
END ;

Cet exemple est intéressant, car il regroupe plusieurs étapes non triviales de l'algorithme. En particulier, on y retrouve la génération de substitutions associées à des définitions de clé d'associations de type 1 : N (substitution sur *loan*, étape décrite dans la section 6.3.2.3, paragraphe sur les définitions de clé), ainsi que la génération et la réécriture de substitutions associées à des définitions d'attributs non clé incluant des hypothèses (substitution sur *dueDate*, étapes décrites dans les sections 6.3.2.3 et 6.3.2.4). L'opération **Transfer** inclut en outre des substitutions générées à partir de termes conditionnels comprenant à la fois des hypothèses sur les paramètres d'entrée de l'opération (comme pour la substitution *dueDate* dans **Lend**) et des conditions permettant de déterminer les valeurs de clé affectées par l'exécution de l'opération (substitutions *nbLoans* et *dueDate*) :

$res \longleftarrow$ **Transfer**$(bId, mId, type) \triangleq$
PRE $bId \in bookKey\_Set \wedge mId \in memberKey\_Set$
$\wedge\ type \in Loan\_Type$

THEN
  IF $CS8 \wedge CD8$ THEN
    $nbLoans := nbLoans \Leftarrow \{(c, d) \mid (c = mId \wedge d = nbLoans(c)+1) \vee$
          $(c \neq mId \wedge c = borrower(bId) \wedge d = nbLoans(c)-1)\}$
    $loan := loan \Leftarrow \{(bId, mId)\}$ ||
    IF $type = Permanent$
    THEN $dueDate := dueDate$
        $\Leftarrow \{(bId, borrower(bId)) \mapsto CurrentDate+365\}$
    ELSE $dueDate := dueDate$
        $\Leftarrow \{(bId, borrower(bId)) \mapsto CurrentDate+loanDuration(mId)\}$
    END ||
    $res :=$ "ok"
  ELSE
    $res :=$ "$error$"
  END
END ;

6.5 Analyse et discussion

Dans ce chapitre, un algorithme de génération de substitutions B, qui correspondent à la dynamique des données décrite dans les définitions d'attributs EB^3, a été présenté. L'objectif est de pouvoir créer un lien entre une spécification EB^3 d'un SI et un modèle B de ce même système. Pour définir ces règles de traduction, nous nous sommes en partie inspiré de la formalisation en B des diagrammes semi-formels comme UML ou OMT, notamment pour générer le modèle de données du diagramme ER de la spécification EB^3. Nous avons défini en outre des règles pour représenter en B les effets des actions sur les attributs du système. À notre connaissance, ce passage d'une vue fonctionnelle du modèle de données d'un SI à une vue par modèle basé sur les états constitue une contribution originale de notre thèse.

Optimisations

L'une des principales difficultés de la traduction vers B concerne la génération de substitutions lorsque les définitions d'attributs incluent des termes conditionnels. Les formules de substitution définies dans la section 6.3.2.3 constituent à la fois une force et une faiblesse de notre algorithme. En effet, la forme compacte et le caractère général des ensembles A et R permettent de générer directement et simplement une substitution pour n'importe quel arbre de décision.

Toutefois, la substitution est difficile à analyser avec les outils actuels de la méthode B. En effet, les règles d'inférence utilisées dans les preuves ne comprennent pas beaucoup de règles de réécriture et de simplification sur des ensembles en extension contenant des disjonctions comme dans nos formules de substitution. Nous avons donc défini des règles pour simplifier les substitutions générées. Parmi nos perspectives de travail, nous envisageons l'implémentation d'un interpréteur des règles de réécriture que nous avons définies afin de réécrire automatiquement les substitutions B.

En attendant, nous avons défini dans [GFL04] un algorithme de traduction directe vers des substitutions ne comprenant aucune disjonction, en insistant

sur l'analyse des arbres de décision. Le principe est simple : chaque arbre est analysé à deux reprises ; la première passe permet de détecter les hypothèses sur les paramètres d'entrée et d'identifier la hiérarchie des expressions IF THEN ELSE END imbriquées dans les termes conditionnels, alors que la deuxième passe permet d'identifier les valeurs de clé concernées par les substitutions de chaque feuille de l'arbre.

Correction de la traduction

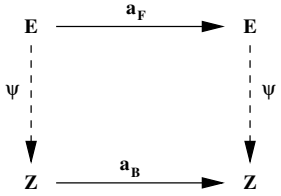

FIG. 6.4 – Correction de la traduction en B

Un aspect important concernant l'algorithme est la preuve de sa correction. Le schéma de la preuve est représenté par la figure 6.4. Comme une preuve similaire sera présentée dans le chapitre 11, nous donnons simplement dans cette section une explication intuitive de la preuve. Pour chaque action $\mathtt{a}_F$ de la spécification EB^3, une opération $\mathbf{a}_B$ est générée en B. Informellement, l'idée est de montrer que, pour chaque action $\mathtt{a}_F$, il existe un morphisme Ψ du modèle EB^3 vers le modèle B, tel que le diagramme de la figure 6.4 soit commutatif. Autrement dit, l'état obtenu en exécutant l'action $\mathtt{a}_F$, suivie de Ψ, est le même que si on exécute Ψ, suivi de l'opération $\mathbf{a}_B$.

Pour l'instant, nous avons prouvé à la main que les substitutions générées correspondaient bien aux effets décrits dans les définitions d'attributs EB^3. La preuve ne comporte pas de difficulté majeure. Comme le nombre de substitutions dans une opération B dépend du nombre d'occurrences de l'action correspondante dans les clauses d'entrée de l'ensemble des définitions d'attributs, la preuve dépend en fait de la forme et du contenu des définitions d'attributs EB^3. Il existe principalement deux formes de substitutions B générées : les substitutions associées à des définitions de clé et celles associées à des définitions d'attributs non clé. Dans le cas des attributs non clé, la forme compacte des formules de substitution facilite la tâche puisqu'elle prend en compte tous les cas possibles de la forme des expressions des termes conditionnels. En outre, le fait d'utiliser l'opérateur de surcharge ("$<+$") dans les substitutions associées à des termes fonctionnels, autres que le symbole "$\perp$", permet de prendre en compte à la fois les insertions et les mises à jour de tuples. Parmi les travaux futurs, nous souhaitons prouver mécaniquement la correction en utilisant un outil de preuve.

Chapitre 7

Méthode EB^4

> "Toute connaissance rationnelle ou bien est matérielle et se rapporte à quelque objet, ou bien est formelle et ne s'occupe que de la forme de l'entendement et de la raison en eux-mêmes et des règles universelles de la pensée en général sans acception d'objets."
>
> — Emmanuel Kant

Nous présentons dans ce chapitre l'approche EB^4, que nous avons développée afin d'exploiter à la fois les avantages des langages EB^3 et B dans une méthode intégrée de spécification formelle dédiée aux SI. Dans la section 7.1, nous décrivons les principales étapes de la méthode EB^4 pour concevoir un SI. Ensuite, nous discutons dans la section 7.2 des techniques utilisées pour décrire et vérifier des propriétés statiques du système à modéliser. Dans la section 7.3, nous montrons comment prendre en compte les propriétés dynamiques. Enfin, la section 7.4 conclut ce chapitre avec une discussion sur l'application de la méthode.

7.1 Description générale d'EB^4

Notre objectif est de bénéficier à la fois des avantages d'EB^3 et B. L'idée est d'utiliser EB^3 pour spécifier le comportement du SI et le langage B pour les propriétés statiques. Ainsi, il est possible de considérer deux vues orthogonales du même système. En analysant l'état de l'art (voir chapitre 3), nous avons remarqué que la principale difficulté liée à l'intégration de plusieurs paradigmes de spécification concernait la manière dont chaque représentation du système était reliée aux autres. Dans l'approche EB^4, nous avons choisi d'adopter une intégration "verticale". Autrement dit, une spécification dans un langage est traduite dans l'autre langage.

Le principal défaut de ce type de combinaison de spécifications formelles est la perte possible d'information ou de propriétés exprimées dans une spécification lorsque cette dernière est traduite dans l'autre langage. Pour y remédier, nous avons repris le principe de raffinement utilisé dans l'expérience EB^3-B (voir section 5.2) et nous couplons la traduction d'EB^3 vers B avec une preuve de raffinement. Autrement dit, une spécification complète en EB^3 du SI est raffinée en une spécification B. Cependant, un tel raffinement ne peut pas être obtenu directement, car de nombreux choix de traduction sont possibles à ce niveau

d'abstraction. Plusieurs étapes ont donc été définies dans l'approche EB^4 pour simplifier le processus de raffinement.

7.1.1 Description des principales étapes

La figure 7.1 est un résumé des six principales étapes :

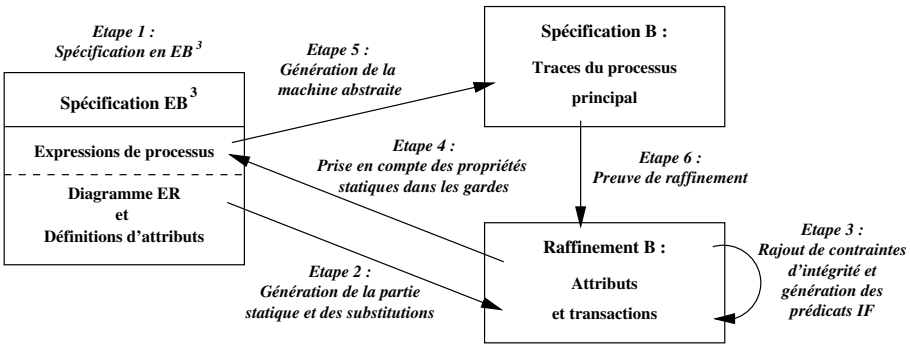

FIG. 7.1 – Résumé de l'approche EB^4

- *Étape 1 :* Le SI est dans un premier temps spécifié en EB^3. L'objectif est de caractériser le comportement du SI. Les différentes parties d'une spécification EB^3 ont été présentées dans la section 4.1.
- *Étape 2 :* Le diagramme ER et les définitions d'attributs sont traduits en B. Les règles de traduction ont été détaillées dans le chapitre 6. Cette partie de la spécification EB^3 permet de modéliser l'espace d'états ainsi que les effets des transactions du système. La traduction B obtenue est incomplète, car elle ne prend pas en compte les propriétés dynamiques des expressions de processus EB^3. Les opérations B ne permettent pas non plus de respecter les propriétés d'invariance générées automatiquement à partir du diagramme ER, car il manque des préconditions. Cependant, cette description B va nous servir de squelette de raffinement pour les étapes ultérieures. En particulier, elle nous fournit les variables d'état et les propriétés d'invariance pertinentes pour le raffinement.
- *Étape 3 :* Nous bénéficions dans cette étape de la complémentarité entre EB^3 et B. L'étape 3 est composée de deux sous-étapes. La première sous-étape consiste à rajouter dans la clause INVARIANT de la description B obtenue à l'issue de l'étape 2, des contraintes d'intégrité statiques supplémentaires qui ne sont pas explicites dans le diagramme ER. Les contraintes d'intégrité statiques sont des propriétés de sûreté que les types d'entité et associations du SI doivent toujours satisfaire. Rappelons que toute opération B doit préserver les propriétés d'invariance de la spécification. La seconde sous-étape consiste à générer des conditions pour les parties IF de chaque opération B, de manière à préserver les prédicats de la clause INVARIANT. Ces conditions, notées *CS* dans la section 6.3.2.1, ne sont pas faciles à déterminer. Nous discutons plus en détail de la génération de ces conditions dans la section 7.2.2. Cette sous-étape permet également de détecter les éventuelles incohérences dans les propriétés statiques du

SI. En particulier, si les contraintes d'intégrité statiques sont trop fortes, alors les conditions *CS* de la partie IF seront équivalentes à *false*. Il faut alors éventuellement retourner à l'étape 1 pour modifier la spécification EB^3.
- *Étape 4 :* Les conditions *CS* générées dans l'étape 3 peuvent se représenter comme des gardes sur les actions dans les expressions de processus EB^3. Cette traduction est réalisée dans l'étape 4 ; elle est présentée dans la section 7.2.2.4.
- *Étape 5 :* Les traces du processus EB^3 principal sont représentées par une machine abstraite B. Les principes de cette traduction ont été décrits dans [FL03] et dans la section 5.2.
- *Étape 6 :* Cette dernière étape consiste à compléter la description B obtenue à l'étape 3 de manière à respecter les propriétés dynamiques des expressions de processus EB^3. Dans ce but, on utilise la relation de raffinement B pour montrer que la description B de l'étape 3 est un raffinement de la spécification B de l'étape 5. Les seules parties qui manquent dans les opérations du raffinement sont les conditions des clauses IF, que nous avions notées par *CD* dans la section 6.3.2.1, et qui représentent le comportement qui est décrit par les contraintes d'ordonnancement dans la spécification EB^3. Cette étape est présentée dans la section 7.3.

7.1.2 Mise en œuvre

L'étape 1 est une étape importante puisqu'il s'agit de la spécification initiale du SI. Des exemples de spécification EB^3 ont été donnés dans la section 4.1 pour la bibliothèque. Cette partie ne peut évidemment pas être automatisée. En revanche, plusieurs patrons caractéristiques des SI ont été définis dans [FSD03] par Frappier et St-Denis afin de faciliter la spécification des processus. Nous en avons donné quelques exemples dans la section 4.1.2.3. Les étapes 3, 4, 5 et 6 permettent de vérifier que l'ensemble de la spécification est cohérent. En particulier,
- les propriétés statiques sont rendues explicites grâce au modèle B (étapes 3 et 4),
- les propriétés dynamiques sont respectées par le modèle B (étapes 5 et 6).

D'un point de vue pratique, certaines étapes de la méthode EB^4 sont automatisables et des règles systématiques ont été définies pour les étapes 2, 4 et 5. Une intervention humaine reste nécessaire dans les étapes 3 et 6. L'étape 3 consiste à compléter la spécification B obtenue à l'étape 2 afin de spécifier et de vérifier des contraintes d'intégrité statiques. Nous détaillons cette partie dans la section 7.2. L'étape 6 permet de prouver que le modèle B respecte les propriétés dynamiques de la spécification EB^3 ; elle est présentée dans la section 7.3.

7.2 Propriétés statiques du SI

L'étape 2 permet d'obtenir une modélisation en B de l'espace d'états et des effets des transactions du système. La traduction en B des définitions d'attributs a été présentée dans le chapitre 6. Nous rappelons que la précondition d'une opération est la condition qui doit être satisfaite avant son exécution afin d'assurer que l'opération s'exécute correctement. Contrairement aux autres méthodes

de dérivation de spécifications B à partir de diagrammes semi-formels [MS99, Mam02], où les préconditions des opérations et les invariants sont générés automatiquement à partir des diagrammes de classes, la traduction présentée dans le chapitre 6 ne permet pas de générer les préconditions que les variables d'état doivent vérifier afin que l'exécution de l'opération préserve les invariants de la machine. Dans une substitution de la forme IF c THEN S_1 ELSE S_2 END, la condition c du IF est la précondition (au sens de Hoare) de la substitution S_1.

7.2.1 Prise en compte des gardes des actions EB^3

Avant d'aborder le problème de l'identification des conditions *CS* des opérations de la description B obtenue à la fin de l'étape 2, nous devons dans un premier temps compléter la traduction des définitions d'attributs afin de prendre en compte les gardes des actions dans les expressions de processus EB^3. En effet, la garde d'une action EB^3 est un prédicat, impliquant des définitions d'attributs, qui doit être satisfait avant d'exécuter l'action en question. Elle permet essentiellement de restreindre les conditions d'exécution d'une action afin d'éviter que cette dernière n'entraîne la violation de contraintes d'intégrité statiques sur les données. À la fin de l'étape 2, toute garde d'une action EB^3 est traduite en une condition *CS* dans l'opération correspondante en B.

Par exemple, supposons que la garde de l'action Transfer dans l'expression de processus de l'association *loan* soit :

$$membRes(trace, bId) = \varnothing \\ \implies \mathsf{Transfer}(bId, mId, \_)$$

où le rôle *membRes* a été défini dans la section 4.2.1.3 et l'action Transfer a pour effet de transférer le prêt d'un membre à un autre membre de la bibliothèque. La garde permet de vérifier alors que l'ensemble des membres qui ont réservé le livre faisant l'objet du transfert est vide. La garde est traduite en B par :

$$membRes(bId) = \varnothing$$

Pour ce faire, on reprend les bijections entre les identifiants EB^3 et B des définitions d'attributs qui ont été déterminées lors de la traduction de l'étape 2.

Les prédicats utilisés dans les gardes EB^3 utilisent une logique à trois valeurs, comme en SQL. Par exemple, la garde de l'action Reserve est la suivante : $mId \neq borrower(trace, bId)$. Le prédicat est équivalent à *false* dans cette logique, si $borrower(trace, bId)$ n'est pas défini. Lorsque la garde implique une fonction partielle ou une relation, un prédicat supplémentaire doit être défini lors de la traduction en B, afin de prendre en compte ce cas particulier de la logique. Le nouveau prédicat spécifie alors que la fonction (ou la relation) est bien appelée dans son domaine de définition. Par exemple, dans le cas de *borrower*, on sait d'après la spécification B que : $borrower(x) == loan(x)$ et $loan \in bookKey \nrightarrow memberKey$. Par conséquent, $borrower(trace, bId)$ est bien défini, si le prédicat suivant est vérifié : $bId \in dom(loan)$. La traduction de la garde de Reserve donne le résultat qui suit :

$$mId \neq borrower(bId) \wedge bId \in dom(loan)$$

7.2.2 Identification des conditions *CS*

Le but de l'étape 3 est de trouver les conditions *CS* de chaque opération de manière à préserver les invariants de la spécification B. Le problème consiste essentiellement à calculer les plus faibles préconditions telles que les invariants soient satisfaits. Les travaux de recherche sur l'identification systématique de préconditions ne sont pas nombreux dans le domaine des SI. Dans [Led98], une approche a été proposée dans le cadre des SI avec le prouveur Z/EVES [Saa97]. La description du système est formalisée en Z. L'approche permet de valider des préconditions proposées par les concepteurs du SI, mais une intervention humaine reste requise.

Dans le langage B, la sémantique des substitutions est définie par leur plus faible précondition [Abr96]. Soit S une substitution et soit R un prédicat, alors l'expression $[S]R$ représente la plus faible précondition telle que la substitution S est assurée d'établir le prédicat R. Autrement dit, toute condition P telle que $P \Rightarrow [S]R$ est une précondition de S. Dans la méthode B, une obligation de preuve est associée à chaque opération afin de prouver qu'elle préserve bien les propriétés d'invariance ; le lemme à vérifier est : $INV \wedge P \Rightarrow [S]INV$, où INV est l'invariant de la spécification, P représente les conditions de la clause PRE et S est la substitution de l'opération.

Plusieurs approches ont été explorées afin de déterminer les conditions *CS* de chaque opération. Idéalement, il faudrait définir un outil capable de réécrire l'obligation de preuve à vérifier, afin de calculer la condition P en fonction des substitutions S. Un tel outil utilise des techniques de résolution d'équations logiques, comme le font la plupart des outils appelés "*logic solvers*". Comme l'outil de support de la méthode B que nous utilisons, l'Atelier B [Cle06], intègre un *logic solver* interne, nous avons proposé une première technique de résolution dans [GFL05b]. Elle consiste simplement à utiliser l'Atelier B pour décharger automatiquement les obligations de preuve et pour en déduire à partir des sous-buts non prouvés les conditions *CS* de chaque opération. Par la forme très spécifique des opérations et des invariants générés, cette approche permet de retrouver les conditions *CS* dans la plupart des cas. Malheureusement, l'Atelier B peut parfois appliquer des règles d'inférence sans le signaler à l'utilisateur et le sous-but non prouvé correspond alors à une prémisse de cette règle et non à la condition *CS* recherchée. Pour ces raisons, nous avons finalement décidé de définir nos propres règles de réécriture pour calculer les conditions *CS*.

7.2.2.1 Règles de calcul

Bien que les contextes d'application soient différents, ce travail a été réalisé en collaboration avec Amel Mammar, qui avait besoin, de son côté, de générer des préconditions pour assurer la correction de la composition de plusieurs opérations B, obtenues à partir de diagrammes UML dans le cadre de l'approche UML-B-SQL. Dans [MGL06], nous avons défini des règles de réécriture et de simplification des prédicats de la forme $[S]INV$ qui sont systématiques, car la forme générale des propriétés d'invariance INV est restreinte à un sous-ensemble d'expressions caractéristiques du domaine des SI.

Dans le contexte de nos travaux, les règles sont appliquées de la manière suivante. Soit **a** une opération B, dont la substitution dans la partie THEN est S, et soit INV l'invariant de la spécification B. La condition *CS* de **a** est égale à

$S[INV]$. Pour calculer cette plus faible précondition, il est possible de prendre en compte les prédicats de INV afin de réécrire et simplifier l'expression $S[INV]$. Le principe est le suivant :

1. Pour définir les effets de la substitution S sur les prédicats de l'invariant, la substitution S est simplement appliquée sur les prédicats de INV ; autrement dit, chaque variable dans INV est remplacée par sa nouvelle valeur indiquée dans la substitution S.
2. Les prédicats obtenus sont ensuite normalisés afin de préparer l'étape suivante.
3. Les prédicats normalisés sont réécrits grâce à un ensemble de règles de simplification et on obtient ainsi la condition CS de l'opération **a**.

La première étape ne présente pas vraiment de difficulté. Par exemple, si on considère l'exemple de l'opération **Discard**, les substitutions suivantes :

$$bookKey := bookKey - \{bId\} \quad || \\ title := \{bId\} \lhd title$$

sont appliquées aux prédicats de l'invariant généré à l'étape 2 (voir section 6.2). En particulier, on obtient les prédicats suivants :

$$... \ \wedge \\ bookKey - \{bId\} \subseteq bookKey\_Set \ \wedge \\ \{bId\} \lhd title \in (bookKey - \{bId\}) \rightarrow\!\!\!\!\!\!\rightarrow Title\_Type \ \wedge \\ loan \in (bookKey - \{bId\}) \rightarrow\!\!\!\!\!\!\rightarrow memberKey \ \wedge \\ reservation \in (bookKey - \{bId\}) \leftrightarrow memberKey \ \wedge \\ ...$$

Les autres prédicats de l'invariant ne sont pas affectés par l'application des substitutions de l'opération **Discard**.

La normalisation permet de mettre en évidence les hypothèses de l'obligation de preuve, à savoir $P \wedge INV$, où P représente les contraintes de typage indiquées dans la clause PRE de l'opération **a**. Par exemple, le tableau 7.1 présente quelques règles de normalisation des prédicats B. La fonction *NormalForm*, qui calcule la forme normale d'un prédicat, y est définie pour des expressions de base (première partie du tableau), pour des fonctions (seconde partie), pour des opérateurs relationnels (troisième partie) et pour des connecteurs logiques (dernière partie). Les expressions de la forme val_i représentent des valeurs, f des fonctions, a et b des ensembles, E_i des expressions relationnelles et P_i des prédicats. La notation *ident* représente un identifiant et $P_{x/y}$ est le résultat obtenu lorsque chaque occurrence de la variable x est remplacée par y dans le prédicat P.

Les règles de normalisation sont appliquées à la fois sur $S[INV]$ et sur $P \wedge INV$. À chaque fois qu'une expression normalisée de $S[INV]$ se retrouve dans les expressions normalisées des prédicats de $P \wedge INV$, alors elle peut être éliminée, car elle n'a pas besoin de nouvelles préconditions pour être prouvée. Les conditions de $S[INV]$ qui ne peuvent pas être éliminées correspondent donc à la condition CS. Dans le cas de l'opération **Discard**, le seul prédicat qui pose problème est : $loan \in (bookKey - \{bId\}) \rightarrow\!\!\!\!\!\!\rightarrow memberKey$. En effet, par l'application de notre ensemble de règles, il reste un seul prédicat qui ne peut pas être prouvé : $bId \notin dom(loan)$; il correspond à la condition CS de l'opération **Discard**.

Prédicat	*NormalForm*
$x \in \{val\}$	$x = val$
$x \in ident$	$x \in ident$
$x \in dom(\{val_1 \mapsto val_2\})$	$x = val_1$
$x \in dom(ident)$	$x \in dom(ident)$
$f \in ident_1 \nrightarrow ident_2$	$f \in ident_1 \nrightarrow ident_2$
$f \in a \rightarrow b$	$NormalForm(f \in a \nrightarrow b) \wedge NormalForm(dom(f) = a)$
$f \in a \rightarrowtail b$	$NormalForm(f \in a \nrightarrow b) \wedge NormalForm(f^{-1} \in b \nrightarrow a)$
$f \in a \rightarrowtail b$	$NormalForm(f \in a \nrightarrow b) \wedge NormalForm(f^{-1} \in b \nrightarrow a) \wedge$ $NormalForm(dom(f) = a)$
$f \in a \twoheadrightarrow b$	$NormalForm(f \in a \nrightarrow b) \wedge NormalForm(ran(f) = b)$
$f \in a \twoheadrightarrow b$	$NormalForm(f \in a \nrightarrow b) \wedge NormalForm(ran(f) = b) \wedge$ $NormalForm(dom(f) = a)$
$f \in a \rightarrowtail\!\!\!\!\twoheadrightarrow b$	$NormalForm(f \in a \nrightarrow b) \wedge NormalForm(f^{-1} \in b \nrightarrow a) \wedge$ $NormalForm(ran(f) = b) \wedge NormalForm(dom(f) = a)$
$f \in (a - \{x\}) \nrightarrow b$	$NormalForm(f \in a \nrightarrow b) \wedge NormalForm(x \notin dom(f))$
$f \in a \nrightarrow (b - \{y\})$	$NormalForm(f \in a \nrightarrow b) \wedge NormalForm(y \notin ran(f))$
$\{x\} \lhd f \in (a - \{x\}) \nrightarrow b$	$NormalForm(f \in a \nrightarrow b)$
$E_1 = \varnothing$	$\forall x.(NormalForm(x \in E_1) \Rightarrow x \in \varnothing)$
$E_1 = E_2$	$NormalForm(E_1 \subseteq E_2) \wedge NormalForm(E_2 \subseteq E_1)$
$E_1 \subseteq E_2$	$\forall x.(NormalForm(x \in E_1) \Rightarrow NormalForm(x \in E_2))$
$x \in dom(E_1 - E_2)$	$NormalForm(x \in dom(E_1)) \wedge$ $\exists y.(x \mapsto y \in NormalForm(E_1) \wedge$ $x \mapsto y \notin NormalForm(E_2))$
$x \in dom(E_1 \cup E_2)$	$NormalForm(x \in dom(E_1)) \vee$ $NormalForm(x \in dom(E_2))$
$x \in (E_1 - E_2)$	$NormalForm(x \in E_1) \wedge NormalForm(x \notin E_2)$
$x \in (E_1 \cup E_2)$	$NormalForm(x \in E_1) \vee NormalForm(x \in E_2)$
$(P_1 \vee P_2) \wedge P_3$	$(P_1 \wedge P_3) \vee (P_2 \wedge P_3)$
$P_1 \wedge (P_2 \vee P_3)$	$(P_1 \wedge P_2) \vee (P_1 \wedge P_3)$
$\forall x.(P_1 \vee P_2 \Rightarrow P_3)$	$\forall x.(P_1 \Rightarrow P_3) \wedge \forall x.(P_2 \Rightarrow P_3)$
$\forall x.(P_1 \Rightarrow P_2 \vee P_3)$	$\forall x.(P_1 \Rightarrow P_2) \vee \forall x.(P_1 \Rightarrow P_3)$
$\forall x.(P_1 \Rightarrow P_2 \wedge P_3)$	$\forall x.(P_1 \Rightarrow P_2) \wedge \forall x.(P_1 \Rightarrow P_3)$
$\forall x.(P_1 \wedge (x = val) \Rightarrow P_2)$	$(P_1 \Rightarrow P_2)_{x/val}$
$P_1 \Rightarrow P_2$	$NormalForm(P_1) \Rightarrow NormalForm(P_2)$

TAB. 7.1 – Formes normales de prédicats B

Prédicat	Simplification
$(P_1 \Rightarrow P) \wedge (P_2 \Rightarrow P)$	$((P_1 \vee P_2) \Rightarrow P)$
$(P_1 \Rightarrow P) \wedge (\neg P_1 \Rightarrow P)$	P
$\forall x.(P \Rightarrow x \in \varnothing)$	$\neg P$ si x n'apparaît pas dans P

TAB. 7.2 – Règles de simplification

Généralement, la seconde étape n'est pas suffisante pour déterminer une condition *CS* qui soit concise. La dernière étape consiste à simplifier les prédicats obtenus. Le tableau 7.2 présente quelques exemples de règles de simplification. Grâce à ces règles, les expressions de prédicats qui ont été développées pour les besoins de la normalisation peuvent être de nouveau simplifiées. En particulier, certaines quantifications peuvent être éliminées.

7.2.2.2 Exemple de la bibliothèque

Dans cette section, nous présentons quelques exemples de conditions *CS* qui sont déduites grâce à nos règles de calcul, pour les opérations B de la bibliothèque (voir section 6.4). Les conditions *CS* pour les autres opérations se trouvent dans l'annexe C. Nous reprenons les mêmes numéros de conditions *CS* :

- Dans le cas de l'opération **Acquire**, la condition $CS1$ est tout simplement *true*, puisque tous les prédicats de $[S]INV$ se retrouvent dans les hypothèses.
- Dans la section précédente, nous avons présenté la condition $CS2$ générée pour l'opération **Discard** : $bId \notin dom(loan)$.
- Nous rappelons que la condition $CS8$ de l'opération **Transfer** inclut le prédicat $membRes(bId) = \varnothing$, afin de prendre en compte la garde de l'action correspondante dans le processus EB^3 (voir section 7.2.1). Les conditions générées pour $CS8$ sont :
  $$bId \in bookKey \wedge bId \in dom(loan) \wedge$$
  $$nbLoans(borrower(bId)) \geq 1 \wedge mId \in memberKey$$
  On remarque que la condition $nbLoans(borrower(bId)) \geq 1$ permet de préserver l'invariant suivant : $nbLoans \in memberKey \rightarrow NAT$.

7.2.2.3 Spécification et vérification de propriétés de sûreté

Certaines propriétés sont naturellement plus faciles à exprimer sur une spécification basée sur les états que sur une spécification basée sur les événements. C'est le cas pour toutes les contraintes d'intégrité statiques, qui sont très nombreuses en SI. Le processus est alors le suivant :

1. Les nouvelles propriétés statiques à vérifier sont spécifiées en B dans la clause INVARIANT.
2. Les conditions *CS* des opérations sont déterminées grâce aux règles de la section 7.2.2.1. L'objectif de ces règles est d'une part de compléter les opérations B obtenues en appliquant les algorithmes du chapitre 6, afin de préserver l'invariant de la spécification. D'autre part, elles permettent aussi de générer les conditions *CS* pour toute propriété d'invariance spécifiée directement par le concepteur du système.
3. Les conditions *CS* de chaque opération sont analysées. Si la conjonction des conditions d'une opération est équivalente à *false*, alors les propriétés d'invariance sont trop fortes et l'opération en question ne pourra pas être exécutée. Dans ce cas, il faut retourner à l'étape 1 afin de modifier le modèle ou bien être moins exigent au niveau des propriétés à vérifier.

Par exemple, on peut exiger que le nombre de prêts d'un membre soit inférieur à un nombre maximum. Soit *maxNbLoans* la constante qui représente ce nombre. Une telle contrainte est spécifiée en B par :

$$\forall\, mId \in memberKey \bullet nbLoans(mId) \leq maxNbLoans$$

La condition *CS* calculée qu'il faut rajouter dans l'opération **Lend** est alors :

$$nbLoans(mId) < maxNbLoans$$

7.2.2.4 Génération des gardes

Les conditions *CS* qui ont été générées à l'étape 3 doivent aussi apparaître comme des gardes des actions EB^3 correspondantes dans le processus, afin de préserver l'équivalence entre les deux représentations. Cette modification est réalisée dans l'étape 4 de la méthode EB^4. On réutilise les mêmes règles de traduction que dans la section 7.2.1, mais dans l'autre sens, afin de retrouver les définitions d'attributs qui correspondent aux variables B utilisées dans la propriété à traduire. Par exemple, la garde générée pour l'action Lend est :

$$nbLoans(trace, mId) < maxNbLoans$$

7.2.3 Discussion sur les gardes

Les règles présentées dans la section 7.2.2.1 peuvent être utilisées pour déterminer les gardes des actions. L'idée est la suivante. Lors de l'étape 1 de la méthode EB^4, aucune garde n'est spécifiée dans les expressions de processus EB^3. Ensuite, lorsque les étapes 2 et 3 sont appliquées, certaines gardes seront identifiées grâce au calcul des conditions *CS*. L'étape 4 permettra alors de générer ces gardes dans les expressions de processus.

Grâce au modèle B, il est également possible de vérifier la cohérence des gardes des actions dans les expressions de processus EB^3. D'après la section 7.2.1, toute garde d'une action est traduite en une condition dans la clause IF de l'opération B correspondante. Cette traduction permet d'une part de détecter si les gardes d'une même action apparaissent à divers endroits de la spécification EB^3. D'autre part, les gardes EB^3 peuvent être en contradiction avec les conditions *CS* de l'opération B. Dans ce cas, il faut agir soit sur les substitutions de l'opération, soit sur les propriétés d'invariance, afin de rendre l'ensemble cohérent.

Dans la section 6.3.2.3, nous avons remarqué que les formules de substitutions utilisaient des ensembles de valeurs caractérisées par des prédicats sous la forme de disjonction. Par la forme particulière des arbres de décision, les conditions de chaque disjonction sont toujours de la forme $\neg P \wedge Q$, où P est un prédicat vérifié par les valeurs de clé d'un fils gauche de l'arbre et Q est le prédicat qui correspond au fils gauche suivant. En plus des règles de réécriture décrites dans la section 6.3.2.4, nous aurions souhaité pouvoir ajouter une règle supplémentaire pour transformer ces ensembles en extension en des ensembles énumérés de valeurs. Par exemple, dans le cas de la substitution *nbLoans* de l'opération **Transfer** (voir section 6.3.2.4), on souhaiterait pouvoir remplacer l'ensemble suivant :

$$\begin{aligned} R \quad = \quad & \{(c, d) \mid (c = mId_B \wedge d = nbLoans_B(c)+1) \vee \\ & (c \neq mId_B \wedge c = borrower_B(bId_B) \wedge d = nbLoans_B(c)-1)\} \end{aligned}$$

par :

$$\begin{array}{rcl} R & = & \{(mId_B \mapsto nbLoans_B(mId_B)+1), \\ & & (borrower_B(bId) \mapsto nbLoans_B(borrower_B(bId_B))-1)\} \end{array}$$

Cependant, cette réécriture n'est possible que si $mId_B \neq borrower_B(bId_B)$ est vrai. Dans notre cas, cette condition est une garde de l'action Transfer. Par conséquent, la condition est vérifiée et l'ensemble R peut être simplifié dans la substitution de la variable *nbLoans*. Afin d'identifier et de générer automatiquement ce type de garde, nous utilisons l'heuristique suivante : l'ensemble R est spécifié sous sa forme énumérée dans l'opération **Transfer**. En appliquant les règles de la section 7.2.2.1, il est possible de générer la garde de l'action EB^3 qui permet de justifier la réécriture de R en un ensemble énuméré.

7.3 Propriétés dynamiques

À l'issue de l'étape 4 de la méthode EB^4, la substitution d'initialisation et chaque opération B préservent bien les invariants du système, mais la spécification n'est pas encore complète, car elle ne reflète pas les propriétés d'ordonnancement exprimées dans les expressions de processus EB^3. En particulier, il manque les conditions *CD* des opérations B pour représenter exactement le comportement décrit dans la spécification EB^3.

Pour illustrer ce point, considérons par exemple les opérations **Acquire** et **Discard**, dans lesquelles nous avons remplacé les conditions *CS* par leur valeur respective :

$res \longleftarrow$ **Acquire**$(bId, bTitle) \triangleq$
  PRE $bId \in bookKey\_Set \ \wedge \ bTitle \in Title\_Type$
  THEN
    IF $true \wedge CD1$ THEN
      $bookKey := bookKey \cup \{bId\}$ ||
      $title := title \Leftarrow \{bId \mapsto bTitle\}$ ||
      $res :=$ "*ok*"
    ELSE
      $res :=$ "*error*"
    END
  END ;

$res \longleftarrow$ **Discard**$(bId) \triangleq$
  PRE $bId \in bookKey\_Set$
  THEN
    IF $bId \notin dom(loan) \wedge CD2$ THEN
      $bookKey := bookKey - \{bId\}$ ||
      $title := \{bId\} \lhd title$ ||
      $res :=$ "*ok*"
    ELSE
      $res :=$ "*error*"
    END
  END ;

Si on fait abstraction des conditions CD, les obligations de preuve associées à ces deux opérations sont déchargées automatiquement par l'Atelier B. Par conséquent, les opérations préservent bien les invariants de la spécification B. Cependant, les conditions de déclenchement de ces opérations ne sont pas suffisantes pour satisfaire les exigences du client.

Par exemple, un événement de la forme **Acquire**$(bId, bTitle)$ ne doit pas être exécuté si le livre identifié par bId existe déjà dans la bibliothèque. De même, l'exécution de **Discard**(bId) n'a pas de sens si le livre bId n'existe pas. Or ces deux exemples d'exécution sont possibles, même si les conditions CD ne sont pas définies. En EB^3, ces contraintes sont exprimées dans les expressions de processus par le fait que Acquire est la première action possible sur un livre et **Discard**(bId) est nécessairement précédé d'une occurrence de **Acquire**$(bId, bTitle)$. Pour déterminer les conditions CD, on va se servir des preuves de raffinement.

7.3.1 Raffinement

L'étape 6 de la méthode EB^4 reprend le principe de la preuve de propriétés EB^3 sur une spécification B, décrite dans la section 5.2. Elle consiste à prouver que le modèle B obtenu après plusieurs itérations des étapes 1, 2 et 3 est bien un raffinement du modèle B des expressions de processus EB^3. Si tel est le cas, alors la spécification B de l'étape 3 respecte les propriétés dynamiques de la spécification EB^3.

À l'étape 5, on génère un modèle B des expressions de processus EB^3. Ce modèle est décrit dans [FL03] et dans la section 5.2.2. En particulier, chaque action EB^3 est traduite en une opération B de la forme suivante :

$$
\begin{array}{l}
res \longleftarrow \textbf{Operation}(parametres) \triangleq \\
\quad \text{PRE} \quad Typ \\
\quad \text{THEN} \\
\quad\quad \text{IF } t \leftarrow \textbf{Operation}(parametres) \in \mathcal{T}(main) \\
\quad\quad \text{THEN } t := t \leftarrow \textbf{Operation}(parametres) \quad || \quad res := \text{“}ok\text{”} \\
\quad\quad \text{ELSE } res := \text{“}error\text{”} \\
\quad\quad \text{END} \\
\quad \text{END}
\end{array}
$$

où Typ représente les contraintes de typage sur les paramètres d'entrée, et res est un paramètre de sortie qui permet de vérifier que l'événement de cette opération est valide (“ok”) ou non (“$error$”). La variable t représente en B la trace courante du système et $\mathcal{T}(main)$ est l'ensemble des traces valides du SI.

Dans l'étape 6, il faut montrer que cette opération est raffinée au sens B par l'opération homonyme de la spécification B obtenue à la fin de l'étape 3. Nous rappelons qu'une opération de cette spécification est de la forme suivante :

$$
\begin{array}{l}
res \longleftarrow \textbf{Operation}(parametres) \triangleq \\
\quad \text{PRE} \quad Typ' \\
\quad \text{THEN} \\
\quad\quad \text{IF } CS \wedge CD \\
\quad\quad \text{THEN } Subst \quad || \quad res := \text{“}ok\text{”} \\
\quad\quad \text{ELSE } res := \text{“}error\text{”} \\
\quad\quad \text{END}
\end{array}
$$

END

où Typ' représente les contraintes de typage, CS les contraintes de sûreté générées à l'étape 3, et $Subst$ les substitutions B générées à l'étape 2.

La preuve de raffinement de l'étape 6 sert à retrouver les conditions CD que les variables doivent vérifier afin que l'opération respecte les propriétés dynamiques de la spécification EB^3. En faisant une analyse par cas sur les valeurs possibles du paramètre *res*, nous en déduisons que l'obligation de preuve associée au raffinement de cette opération est satisfaite si et seulement si les trois conditions suivantes sont vérifiées :

$$\begin{array}{ll} \text{(OP1)} & Typ \;\Rightarrow\; Typ' \\ \text{(OP2)} & CS \wedge CD \;\Rightarrow\; [Subst][t := t \leftarrow \mathbf{Operation}(parametres)]J \\ \text{(OP3)} & CS \wedge CD \;\Leftrightarrow\; t \leftarrow \mathbf{Operation}(parametres) \in \mathcal{T}(main) \end{array}$$

où le prédicat J est l'invariant de collage entre l'unique variable abstraite t et les variables de la spécification B obtenue à l'étape 2. Comme ces dernières sont générées automatiquement, l'invariant de collage est facile à déterminer. Par convention, l'expression V_e représente dans la suite la variable d'état qui correspond à la définition d'attribut e. Pour chaque définition de clé $eKey$, l'invariant J inclut un prédicat de la forme $V_{eKey} = eKey(t)$. Par exemple, on a l'invariant $bookKey_B = bookKey_F(t)$ pour la clé de *book*. Pour chaque définition d'attribut non clé $b(\overrightarrow{k})$, l'invariant J inclut un prédicat de la forme $V_b = \lambda\, z \cdot (z \in V_{\overrightarrow{k}} \mid b(t, z))$. Par exemple, on a l'invariant suivant pour l'attribut *title* :

$$title_B = \lambda\, z \cdot (z \in bookKey_F \mid title_F(t, z))$$

L'obligation de preuve (OP1) est triviale, car les conditions Typ et Typ' sont équivalentes. L'obligation de preuve (OP2) permet de prouver que les effets de $Subst$ ne sont pas en contradiction avec les effets des actions EB^3. Par définition des règles de traduction des définitions d'attributs, les substitutions $Subst$ sont équivalentes aux définitions d'attributs EB^3. Par conséquent, la preuve de (OP2) est immédiate. Le problème concerne la preuve de l'obligation (OP3) qui requiert une certaine créativité. Pour ce faire, il faut comparer les transitions d'états de l'opération et de son raffinement et vérifier que les conditions de déclenchement des transitions sont les mêmes. On connaît celles d'EB^3, il faut alors trouver celles de B (CD) qui sont équivalentes.

Par exemple, le processus associé au type d'entité *book* est de la forme suivante (voir section 4.1.2) :

$book(bId : bookKey\_Set) =$
  $\mathsf{Acquire}(bId, \_)$.
  (
    $(\mid mId : memberKey\_Set : loan(mId, bId)\;)^*$
  ||
    $(\;|||mId : memberKey\_Set : reservation(mId, bId)^*)$
  ||
    ...
).
  $\mathsf{Discard}(bId)$

En analysant cette expression de processus, on en déduit que *bId* n'appartient pas encore à la clé de *book* avant l'exécution de l'action Acquire. Par conséquent, la condition $CD1$ de l'opération **Acquire** est de la forme $bId \notin bookKey$. D'autre part, l'action Discard est exécutée en fin de cycle d'une entité de livre, par conséquent, le livre *bId* doit exister avant l'exécution de Discard. La condition $CD2$ est de la forme : $bId \in bookKey$.

Toutefois, la détermination de ces conditions n'est pas immédiate et requiert une analyse du LTS associé au processus *book*.

7.3.2 Définition de patrons de raffinement

Afin d'aider le concepteur à trouver les conditions CD des opérations B, nous proposons des conditions qui fonctionnent dans le cas des patrons de processus présentés dans la section 4.1.2.3. Par exemple, le processus du type d'entité *book* décrit ci-dessus est une instanciation du patron (a) décrit dans la section 4.1.2.3. Or un processus de la forme producteur-modificateur-consommateur implique des conditions CD très précises pour chaque type d'action dans le processus. Dans les paragraphes suivants, nous présentons les conditions CD que nous avons identifiées et prouvées pour deux patrons caractéristiques des expressions de processus EB^3.

Pour les besoins de compréhension des sections suivantes, rappelons le sens des termes "producteur", "modificateur" et "consommateur" utilisés dans les patrons de processus. Un producteur d'une entité ou d'un lien d'association identifié par $\overrightarrow{k}$, dans le processus d'un type d'entité ou d'une association dont la définition de clé est *eKey*, est une action qui a pour effet de rajouter $\overrightarrow{k}$ dans l'ensemble des valeurs de clé existantes ; autrement dit, après l'exécution du producteur, la propriété suivante est satisfaite : $\overrightarrow{k} \in eKey(t)$, où t est la trace courante du système. Un modificateur d'une entité ou d'un lien d'association identifié par $\overrightarrow{k}$, dans le processus d'un type d'entité ou d'une association, est une action qui a pour effet de modifier les valeurs d'attributs associées à $\overrightarrow{k}$. Un consommateur d'une entité ou d'un lien d'association identifié par $\overrightarrow{k}$, dans le processus d'un type d'entité ou d'une association dont la définition de clé est *eKey*, est une action qui a pour effet de supprimer $\overrightarrow{k}$ dans l'ensemble des valeurs de clé existantes ; autrement dit, après l'exécution du consommateur, la propriété suivante est satisfaite : $\overrightarrow{k} \notin eKey(t)$, où t est la trace courante du système.

Dans la suite, on reprend les mêmes notations et les mêmes classes de patrons de processus décrites dans la section 4.1.2.3.

7.3.2.1 Patron de base pour les types d'entité et associations

Les opérations de type producteur doivent vérifier que l'entité k n'existe pas avant l'exécution de l'opération. Par conséquent, CD est de la forme : $k \notin V_k$, où V_k est la variable d'état qui représente la clé du type d'entité. Par exemple, la condition $CD1$ de l'opération **Acquire** de la bibliothèque est : $bId \notin bookKey$. Les opérations de types modificateur ou consommateur doivent vérifier avec CD que l'entité k existe : $k \in V_k$. Par exemple, la condition $CD2$ de l'opération **Discard** est : $bId \in bookKey$.

7.3.2.2 Patron pour les entités participant à des associations binaires

Les opérations de type producteur d'une association doivent vérifier que les entités $k1$ et $k2$ existent et que le couple $(k1, k2)$ n'existe pas encore dans l'association. La condition CD est de la forme suivante : $k1 \in V_{k1} \wedge k2 \in V_{k2} \wedge CC$, où V_a est une relation de type $V_{k1} \leftrightarrow V_{k2}$. La forme de V_a et la condition CC dépendent de la cardinalité de l'association.

Si a est une association $1 : N$, alors V_a est de la forme $V_a \in V_{k1} \nrightarrow V_{k2}$ et le processus contient un choix quantifié sur $k1$ et un entrelacement quantifié sur $k2$. Dans ce cas, la condition CC est de la forme $k1 \notin dom(V_a)$. Par exemple, la condition $CD6$ de l'opération **Lend** est : $bId \in bookKey \wedge mId \in memberKey \wedge bId \notin dom(loan)$.

Si a est une association $1 : 1$, alors V_a est de la forme $V_a \in V_{k1} \rightarrowtail V_{k2}$ et le processus contient un choix quantifié sur $k1$ et sur $k2$. Dans ce cas, la condition CC est : $k1 \notin dom(V_a) \wedge k2 \notin ran(V_a)$, car le couple $(k1, k2)$ doit être unique.

Si a est une association $M : N$, alors V_a est de la forme $V_a \in V_{k1} \leftrightarrow V_{k2}$ et le processus contient un entrelacement quantifié sur $k1$ et sur $k2$. Dans ce cas, la condition CC est : $(k1, k2) \notin V_a$, car il peut y avoir d'autres tuples incluant des $k1$ ou des $k2$, mais pas le couple $(k1, k2)$.

Les opérations de types modificateur et consommateur d'une association doivent vérifier que les entités $k1$ et $k2$ existent et que le couple $(k1, k2)$ existe dans l'association. Par conséquent, CD est de la forme : $k1 \in V_{k1} \wedge k2 \in V_{k2} \wedge (k1, k2) \in V_a$.

7.3.2.3 Analyse de la preuve

Nous présentons maintenant un aperçu de l'analyse à faire pour prouver l'obligation de preuve (OP3) pour chaque patron de raffinement. Pour illustrer, nous considérons le cas d'une action de type producteur dans une association de cardinalité $1 : N$. Les preuves pour les autres cas suivent le même raisonnement.

Soit P_j un producteur d'une association $1 : N$ entre deux types d'entité $e1$ et $e2$. Les expressions AP_{i1} et AP_{i2} sont respectivement de la forme (AP-Multi) et (AP-One), comme décrites dans la section 4.1.2.3. Pour prouver la condition suivante :

$$CS \wedge CD \;\Leftrightarrow\; t \leftarrow \textbf{Operation}(parametres) \in \mathcal{T}(main)$$

nous devons comparer les conditions de déclenchement des transitions de l'action P_j dans le LTS qui représente les transitions du processus EB^3 avec les conditions de déclenchement des transitions de l'opération B qui correspond à P_j. Si les quatre premières étapes de la méthode EB^4 ont été réalisées correctement, la condition CS est exactement équivalente aux gardes de l'action P_j dans l'expression de processus.

Le raisonnement suivant est commun à tous les cas de processus. Si on analyse le patron de processus défini dans la section 4.1.2.3, paragraphe (a), on remarque que les actions de type producteur pour une association sont précédées par :

- un producteur de l'entité identifiée par $k1$ (dans le processus du type d'entité $e1$),
- un producteur de l'entité identifiée par $k2$ (dans le processus du type d'entité $e2$).

Par conséquent, nous pouvons identifier la forme de la trace t afin qu'elle respecte ces deux conditions. Soit T_{k1} l'ensemble des traces valides de la forme $t1 :: P_{k1} :: t2$, où P_{k1} est un producteur de l'entité identifiée par $k1$, $t1$ est une trace arbitraire, et $t2$ est une trace qui ne contient aucune action de type consommateur pour $k1$. De manière analogue, T_{k2} est défini comme l'ensemble des traces valides de la forme $t1 :: P_{k2} :: t3$, où P_{k2} est un producteur de l'entité identifiée par $k2$, et $t3$ est une trace qui ne contient aucune action de type consommateur pour $k2$. Dans ce cas, la trace t que nous analysons vérifie le prédicat suivant : $t \in T_{k1} \cap T_{k2}$. Par définition de P_{k1} et P_{k2}, ce prédicat est équivalent à : $k1 \in e1(t) \wedge k2 \in e2(t)$, où $e1$ et $e2$ sont respectivement les définitions de clé des types d'entité $e1$ et $e2$. Nous notons ce prédicat par (CE1).

Si on analyse le patron de processus défini dans la section 4.1.2.3, paragraphe (b), nous déduisons du cycle de vie d'une association que :

- soit un producteur de l'entité $(k1, k2)$ n'a jamais été exécuté dans le processus de l'association depuis l'initialisation du système,
- soit un nombre fini de producteurs de $(k1, k2)$ ont déjà été exécutés (par la fermeture de Kleene sur l'association a), mais les effets de chacun d'eux ont été annulés par la suite par un consommateur de $(k1, k2)$.

Ainsi, nous pouvons déterminer la forme de la trace t afin qu'elle respecte ces contraintes d'ordonnancement et nous en déduisons qu'elle vérifie la propriété suivante : $(k1, k2) \notin a(t)$, où a est la définition de clé de l'association a. Ce prédicat est noté par (CE2).

Dans la cas particulier de l'association $1 : N$, on déduit de l'analyse des quantifications de $k1$ et $k2$ que la trace t satisfait le prédicat suivant, que nous notons par (CE3) : $k1 \notin \{k \in K\_Set1 \mid \exists\, k2 \in K\_Set2 \bullet (k, k2) \in a(t)\}$.

L'invariant de collage J permet de relier la variable d'état t aux variables d'état qui représentent les différents attributs dans la spécification B obtenue à l'étape 3. En particulier, nous avons : $e1(t) = V_{k1}$, $e2(t) = V_{k2}$ et $a(t) = V_a$. Le prédicat (CE1) est donc équivalent à : $k1 \in V_{k1} \wedge k2 \in V_{k2}$. De même, on en déduit que (CE2) est équivalent à : $(k1, k2) \notin V_a$, et (CE3) à : $k1 \notin dom(V_a)$. Par conséquent, la conjonction (CE1) $\wedge$ (CE2) $\wedge$ (CE3) est équivalente à la condition CD que nous avons proposée dans le patron de raffinement.

7.3.2.4 Exemple de la bibliothèque

Dans cette section, nous présentons quelques exemples de conditions CD, qui sont déduites de nos patrons de raffinement, pour les opérations de l'exemple de la bibliothèque :

- Dans le cas de l'opération **Acquire**, la condition $CD1$ est $bId \notin bookKey$, car elle correspond à un producteur de bId dans le processus du type d'entité $book$.
- La condition $CD2$ pour l'opération **Discard** est $bId \in bookKey$, car **Discard** est un consommateur de bId dans le type d'entité $book$.
- La condition $CD8$ de l'opération **Transfer** est plus complexe à déterminer, car il s'agit à la fois d'un producteur et d'un consommateur de prêts :
  $$bId \in bookKey \wedge mId \in memberKey \wedge$$
  $$bId \in dom(loan)$$
- La condition $CD10$ de l'opération **Take** est la suivante :
  $$bId \in bookKey \wedge mId \in memberKey \wedge$$

$$bId \notin dom(loan) \wedge (bId, mId) \in reservation$$

Cet exemple est intéressant, car **Take** est à la fois un producteur de prêt (d'où la condition $bId \notin dom(loan)$) et un consommateur de réservation (d'où la condition $(bId, mId) \in reservation$).

Les autres conditions se trouvent dans l'annexe C.

7.4 Conclusions

Dans ce chapitre, nous avons présenté EB^4, une nouvelle méthode qui permet de spécifier des SI avec les langages EB^3 et B. Dès l'origine du projet, nous avons souhaité favoriser l'utilisation des spécifications formelles. Les notations formelles permettent, d'une part, de modéliser avec précision le SI et, donnent l'avantage, d'autre part, de pouvoir raisonner sur le modèle obtenu afin de faire de la preuve ou du *model-checking*. La méthode EB^4 a été définie dans le but d'offrir cette possibilité aussi bien sur une spécification EB^3 du système que sur une spécification B. Nous avons également cherché à minimiser les efforts que représentent les vérifications liées à la preuve d'équivalence entre les parties EB^3 et B. En particulier, dans la section 7.3.2, nous avons proposé des patrons de raffinement pour déterminer la valeur des conditions *CD* des opérations.

Les étapes décrites dans la section 7.1 permettent d'obtenir à la fin du processus une spécification qui satisfait à la fois les propriétés des parties EB^3 et B. Plusieurs options existent pour implémenter le système. Si on considère la spécification B obtenue avec la méthode EB^4, alors il est possible d'utiliser les techniques de raffinement de la méthode B. Pour le domaine particulier des SI, des règles de raffinement ont été développées dans le cadre de l'approche UML-B-SQL [Mam02] (voir section 1.4.1) afin d'obtenir des transactions Java/SQL à partir d'une spécification B. Dans le cadre de l'approche EB^4, ces règles peuvent être utilisées afin d'implémenter les transactions du SI.

Une alternative est de générer directement des SI à partir de la spécification EB^3. C'est l'objectif du projet APIS. Dans le cadre de cette thèse, nous avons développé des techniques de synthèse de transactions Java/SQL à partir des définitions d'attributs EB^3 ; le projet APIS et notre contribution seront présentés plus en détail dans le chapitre 9.

Enfin, l'implémentation des outils de support de la méthode EB^4 fait partie des objectifs à court terme. Tous les algorithmes ont été déjà développés et, de plus, un outil de traduction des définitions d'attributs EB^3 en Java/SQL (voir chapitre 11) a démontré la validité du principe d'analyse des définitions d'attributs dans l'algorithme de traduction.

Chapitre 8

Synthèse de l'approche EB^4

> "Si vous avez compris tout ce que je viens de dire, c'est que je me suis mal exprimé."
>
> — Alan Greenspan

Dans cette partie de la thèse, nous avons présenté EB^4, une méthode formelle qui permet de spécifier des SI. Dans le chapitre 5, nous avons présenté les raisons pour lesquelles nous avons travaillé sur cette approche de combinaison. Puis, nous avons décrit dans le chapitre 6 notre algorithme de traduction des définitions d'attributs EB^3 en des spécifications B. Dans le chapitre 7, nous avons présenté le processus des différentes étapes d'EB^4. Dans ce chapitre, nous faisons la synthèse de nos travaux autour d'EB^4. Dans la section 8.1, nous présentons nos contributions concernant la méthode EB^4. Ensuite, dans la section 8.2, nous discutons des forces et des limites d'EB^4 et nous comparons notre proposition avec les approches présentées dans le chapitre 3 sur l'état de l'art. Enfin, nous décrivons dans la section 8.3 les perspectives de nos travaux de thèse sur EB^4.

8.1 Contributions

Notre objectif est de modéliser de manière formelle les SI. Les méthodes de conception actuelles proposent essentiellement des superpositions relativement complexes de vues complémentaires des SI (comme par exemple, modèles des données, des flux de données et des transactions) avec pour objectif d'obtenir à la fin une spécification cohérente. Pour assurer la cohérence de toutes ces vues complémentaires, les méthodes reposent essentiellement sur les capacités d'analyse et de synthèse du concepteur, puisque ces méthodes qui sont semi-formelles permettent uniquement une vérification syntaxique. En outre, les quelques approches formelles qui sont présentées dans la littérature pour spécifier les SI ne représentent pas bien les aspects à la fois statiques et dynamiques des systèmes.

Dans l'approche que nous proposons, l'idée est de coupler deux langages formels qui sont complémentaires dans la spécification des propriétés dynamiques et statiques des SI. L'objectif est de bénéficier à la fois des avantages des langages formels de spécification EB^3 et B. Nous utilisons le langage EB^3 pour spécifier le comportement du système, puis nous utilisons B pour prendre en compte les propriétés statiques. La méthode EB^4 prévoit ensuite plusieurs itérations de

spécification et de preuve afin de rendre le modèle du SI cohérent. Pour définir la méthode EB^4, nous avons travaillé sur les éléments suivants :

1. une étude approfondie des approches de combinaison de spécifications formelles de type transitions d'états avec des spécifications formelles de type événementiel (chapitre 3),
2. la définition d'un algorithme de traduction en B de la partie des spécifications EB^3 qui représente l'espace d'états et les effets des transactions du SI (chapitre 6). L'algorithme a été testé sur plusieurs exemples, et notamment sur le cas d'étude de la bibliothèque, et nous avons prouvé à la main sa correction.
3. la définition de règles de réécriture et de simplification de prédicats B afin de déterminer les conditions des opérations B qui sont suffisantes pour décharger les obligations de preuve liées à la préservation des invariants (section 7.2.2.1, travail réalisé en collaboration avec Amel Mammar),
4. la définition de patrons de raffinement afin de faciliter la preuve de raffinement de la spécification EB^3 en une spécification B (section 7.3.2),
5. la définition d'un processus méthodologique afin de mettre en œuvre les différentes étapes de la méthode EB^4 (section 7.1).

Ces contributions ont fait l'objet de plusieurs articles publiés ou en cours de soumission. L'état de l'art sur les combinaisons de spécifications formelles est le sujet d'un article en cours de révision pour la revue IST. L'algorithme de traduction en B des définitions d'attributs EB^3 a été présenté à la conférence internationale IFM 2005 [GFL05b]. L'approche EB^4 et les règles de calcul des conditions *CS* ont été présentées à INFORSID 2006 [Ger06, MGL06]. Les patrons de raffinement seront présentés à la conférence B 2007 [GFL07].

8.2 Discussion

La méthode EB^4 répond en grande partie aux questions que nous nous étions posées à l'origine du projet (voir section 5.3). Le premier problème concernait l'ordre d'utilisation des langages EB^3 et B. Alors que les propriétés dynamiques d'un SI ne sont généralement considérées qu'une fois que les structures du système ont été déjà définies, l'approche EB^4 permet de considérer deux vues orthogonales et complémentaires du modèle du SI. En effet, un des principaux objectifs d'EB^4 est de développer des SI en considérant dès le travail d'analyse les problèmes liés à la modélisation des propriétés dynamiques. Une analyse entre deux modèles d'un même système permet de corriger rapidement les erreurs et d'éviter ainsi des délais ou des coûts supplémentaires liés à une prise en compte tardive de certaines exigences ou à la modification d'erreurs dans l'implémentation qui auraient pu être détectées dans les phases de développement antérieures.

Un autre problème soulevé au début de la thèse consistait à déterminer dans quel langage les propriétés du SI devaient être spécifiées. Le langage EB^3 met en avant le comportement du système, comme les propriétés de vivacité ou les contraintes d'ordonnancement. Le langage B permet, d'une part, de bien représenter les structures de données de la BD et les effets des opérations sur les états, et d'autre part, de spécifier des propriétés d'invariance sur les états.

TAB. 8.1 – Comparaison d'EB^4 avec les autres approches

Approche	EB^4
Langages	EB^3 B
But	raffinement d'une spécification EB^3 en B
Sémantique	opérationnelle
Vérification	à définir en EB^3 preuve en B
Raffinement	raffinement B
Outils	outils EB^3 outils B
Lisibilité	+
Adéquation SI	+
Classification : but	vérification de propriétés
Classification : sémantique	traduction + juxtaposition

Enfin, un dernier problème concernait la cohérence des spécifications. Afin de rendre l'ensemble du modèle cohérent, la spécification EB^3 est raffinée en B. La méthode EB^4 fournit des techniques pour faciliter ce processus qui serait difficile à réaliser directement par le concepteur. Tout d'abord, un squelette du raffinement B est généré grâce à des règles systématiques. Ainsi, des variables d'état pertinentes sont générées pour représenter les différents attributs du SI. La cohérence de la spécification B ainsi obtenue est assurée par le calcul systématique de conditions dans les clauses IF des opérations qui sont suffisantes pour préserver les propriétés d'invariance. Enfin, des patrons de raffinement ont été proposés afin d'aider le concepteur à trouver des conditions de déclenchement d'opération qui respectent les propriétés d'ordonnancement exprimées dans la spécification EB^3.

8.2.1 Comparaison avec les autres approches de combinaison

Le tableau 8.1 réutilise les mêmes critères de comparaison que dans la section 3.7.1. EB^4 s'inspire des combinaisons de type traduction (voir section 3.7) et vérification de propriétés (section 3.1), mais elle peut aussi être classée parmi les combinaisons de type juxtaposition. En EB^4, la spécification B satisfait les propriétés d'ordonnancement décrites en EB^3, car on prouve qu'il s'agit d'un raffinement, au sens B, de la spécification EB^3. Afin d'aider le concepteur à trouver le raffinement B, on utilise des techniques de traduction pour générer une partie significative de la description B. Par conséquent, EB^4 utilise à la fois une traduction comme dans csp2B et une preuve de propriété comme dans PLTL-Event B.

Pour éviter les problèmes de réutilisation et de lisibilité discutés dans la

section 5.1, EB^4 n'a pas été défini comme un nouveau langage à l'instar de Circus et CSP-OZ, mais comme une méthode qui permet de bénéficier à la fois des avantages d'EB^3 et B. Les approches CSP || B et csp2B consistent à partager la spécification en deux parties distinctes qui sont dépendantes l'une de l'autre : le contrôle des opérations est décrit dans un processus CSP, tandis que les données sont spécifiées dans une description B. En EB^4, les spécifications EB^3 et B sont complètes en elles-mêmes. De plus, EB^3 permet de considérer des choix et des entrelacements quantifiés qui sont très utiles pour représenter les cycles de vie de nombreuses entités de types d'entité et associations d'un SI.

La méthode EB^4 permet d'obtenir à la fin une spécification B qui satisfait à la fois les propriétés des parties EB^3 et B. Pour implémenter le système, on peut raffiner la spécification B obtenue à la fin de l'étape 6, notamment en utilisant les techniques de raffinement développées au sein du CEDRIC pour l'approche UML-B-SQL [Mam02]. Si on s'inspire de la méthode CSP || B, une autre piste serait possible pour intégrer EB^3 et B. Cette approche, que nous appelons EB^3 || B, est définie comme EB^4, à l'exception des étapes 5 et 6 qui ne sont pas prises en compte. Ensuite, les techniques de raffinement présentées dans [Mam02] sont utilisées pour obtenir des transactions Java/SQL qui implémentent les opérations de la spécification B. À l'exécution, on utilise l'interpréteur des expressions de processus EB^3, appelé EB^{3}PAI, pour contrôler les transactions obtenues à partir des opérations B.

Comme dans CSP || B, l'idée de EB^3 || B est d'utiliser EB^3 pour le contrôle et B pour le modèle de données. Comme les opérations B ne définissent pas des conditions *CD* comme dans EB^4, l'interpréteur EB^{3}PAI permet de contrôler l'exécution des transactions qui implémentent ces opérations. Finalement, nous n'avons pas développé cette piste, car l'approche EB^3 || B nous semblait trop dépendante de l'utilisation de l'interpréteur EB^{3}PAI. Dans le cas de la méthode EB^4, on garde la possibilité d'utiliser l'interpréteur EB^{3}PAI, tout en obtenant des opérations B qui prennent en compte les conditions *CD*. De plus, nous pouvons obtenir l'implémentation à partir de la spécification B ou bien à partir de celle en EB^3, comme nous en discuterons dans la troisième partie de cette thèse.

Comme il n'existait pas dans la littérature de combinaisons qui se prêtaient bien à la spécification des SI, la méthode EB^4 est par définition l'approche qui s'adapte le mieux aux SI selon nos critères de comparaison. Évidemment, la méthode que nous proposons n'a pas que des avantages. La section suivante présente les inconvénients et les améliorations à apporter.

8.2.2 Limites

Nous pouvons classer les limites d'EB^4 en deux catégories : i) les inconvénients liés à la nature même de la combinaison et des langages utilisés et ii) les défauts de jeunesse, qui seront atténués par des travaux ultérieurs.

8.2.2.1 Inconvénients

Les inconvénients de la catégorie i) sont des problèmes liés à l'intégration des langages EB^3 et B dans EB^4 :

- le besoin de connaissance en méthodes formelles pour les utilisateurs d'EB^4,
- la difficulté de passer de B vers EB^3,
- l'absence de logique temporelle en EB^3.

Le premier problème est probablement le plus critique. En effet, des techniques formelles comme la preuve demandent une haute expertise et une grande expérience en la matière. Comme nous l'avons mentionné dans le chapitre 1, il existe peu de langages formels qui soient utilisés dans le domaine d'application des SI. Cependant, certains langages formels comme B ou EB^3 se prêtent bien à la spécification des SI. Nous n'avons pas souhaité créer, comme dans l'approche Circus, un nouveau langage qui intègrerait à la fois EB^3 et B, car il existe déjà assez de langages formels pour satisfaire les besoins en spécification des concepteurs de SI. De plus, il est très difficile dans ce domaine de faire accepter l'idée-même d'utiliser un langage formel. Notre point de vue est plus pragmatique et cette opinion s'est considérablement renforcée au cours de notre thèse, en particulier, lorsque nous avons eu nos premières discussions avec des membres de la communauté des SI.

L'objectif initial d'EB^4 était de proposer un processus de développement ou une méthodologie autour des langages EB^3 et B, mais en minimisant les interactions possibles entre les experts en SI et les experts en méthodes formelles. L'aspect formel présente bien évidemment un avantage pour assurer la correction du système final, mais les efforts que demande l'utilisation de notations et de techniques formelles ne seront acceptés que si :

1. il existe une volonté très nette de la part du client de choisir cette option ;
2. le processus peut être automatisé au maximum, et même dans sa globalité si c'est possible ;
3. il existe des moyens de représentation plus standard (comme des diagrammes UML) pour communiquer ou discuter de la spécification formelle.

Dans le cas d'EB^4, nous avons cherché, d'une part, à définir des règles de traduction ou de calcul pour automatiser la majorité des étapes, et d'autre part, à minimiser la difficulté des étapes qui requièrent une intervention humaine. Cependant, l'étape 6 ne pourra pas être automatisée dans tous les cas ; la question qui se pose alors est de savoir si l'effort que représentent ces preuves de raffinement sera plus coûteux que le risque encouru en l'absence de preuve. Si la condition 1) est vérifiée, la réponse sera probablement réfléchie et la question aura au moins eu le mérite d'être posée. Cependant, on ne peut pas assurer que les preuves seront effectivement réalisées. L'autre limite de l'approche EB^4 concerne le point 3), mais l'implémentation d'outils graphiques devrait permettre de résoudre en partie ce problème. L'idée est de proposer une représentation graphique qui permettrait d'animer et de valider les spécifications EB^3 avec le client.

Le deuxième problème concerne la traduction d'EB^3 vers B. EB^4 repose en effet sur un raffinement de la spécification EB^3 en une spécification B. Par conséquent, le passage d'une spécification à l'autre semble être unidirectionnel : de EB^3 vers B. Dans la pratique, la méthode consiste plutôt à faire des itérations sur les premières étapes et, par conséquent, la spécification EB^3 est adaptée en fonction des résultats des étapes ultérieures. En particulier, les gardes sont difficiles à déterminer dès la première tentative de spécification en EB^3. Elles peuvent être déduites en fonction des propriétés statiques vérifiées par le modèle B.

Enfin, le langage EB^3 n'est pas doté d'une logique temporelle. Par conséquent, certaines contraintes dynamiques ne peuvent pas s'exprimer dans ce langage. Par exemple, une contrainte temporelle du type "éventuellement" ou "pour toujours" requiert l'utilisation d'une logique temporelle pour être prise en compte. L'extension d'EB^3 avec des opérateurs de logique temporelle fait partie de nos

perspectives de travail, mais il faudra également développer des techniques de vérification. Ce point est discuté dans la section suivante.

8.2.2.2 Améliorations possibles

Les limites de la catégorie ii) sont essentiellement des problèmes que nous avions identifiés au début de notre thèse [Ger04], mais que nous n'avons pas encore eu l'occasion de résoudre :

- la difficulté de spécifier des systèmes complexes en EB^3,
- l'absence de techniques de vérification en EB^3,
- l'absence d'outils graphiques pour spécifier et animer des systèmes avec EB^4.

Les deux derniers points sont liés aux problèmes évoqués dans la section 8.2.2.1 ; ils permettront de faciliter l'utilisation des langages formels dans la méthode EB^4.

La spécification en EB^3 de systèmes complexes est actuellement difficile. Deux problèmes se posent. D'une part, il est difficile de tenir compte de nombreuses propriétés sur de nombreux événements dès la première transcription de la spécification. Une solution consiste à modéliser le système progressivement en ajustant de manière presque empirique le modèle. Cette approche n'est toutefois pas acceptable pour une méthode formelle. L'autre problème concerne le rajout d'un type d'entité supplémentaire sur une spécification existante. Il est alors difficile de prouver la cohérence du nouvel ensemble. Pour concevoir des systèmes complexes, il est très utile de pouvoir rajouter des détails tout en s'assurant que la cohérence est maintenue. Le raffinement est un atout supplémentaire pour les méthodes formelles qui en sont dotées. Par conséquent, nous envisageons comme perspective la définition d'une nouvelle relation de raffinement pour EB^3 qui permettra de détailler, étape après étape, une spécification abstraite.

Un autre problème concernant EB^3 est le manque de techniques et d'outils de vérification. S'il est possible, grâce à la technique de raffinement décrite dans la section 7.3.1, de prouver des propriétés d'ordonnancement décrites en EB^3 sur une spécification B, nous ne disposons pas encore de techniques pour faire du *model-checking* ou de la preuve en EB^3. Comme EB^3 est un langage formel proche de CSP, nous devrions toutefois être en mesure de définir des techniques de *model-checking* et de développer, à l'instar d'un outil comme FDR, un *model-checker* pour des expressions de processus EB^3.

Pour rendre la méthode EB^4 plus facile à utiliser, nous devons non seulement développer des outils d'assistance pour supporter les différentes étapes de l'approche, mais aussi créer des outils graphiques qui sont largement utilisés dans l'industrie pour modéliser et représenter des SI. Parmi nos perspectives de travail, nous souhaitons définir des LTS étendus, que nous appelons diagrammes étendus de transitions d'états (*extended state transition diagrams* en anglais, soit le sigle ESTD), qui représenteraient de manière compacte les quantifications des opérateurs EB^3 de choix et d'entrelacement. Ainsi, les ESTD seraient utilisés comme support pour développer des outils graphiques comme des éditeurs ou des animateurs de spécifications EB^3.

8.3 Perspectives

D'un point de vue théorique, tout est en place pour mettre en œuvre la méthode EB^4. L'implémentation des outils de support de la méthode EB^4 n'est pas un problème en soi, car tous les algorithmes sont déjà développés. Pour répondre aux besoins de vérification des propriétés dynamiques, nous comptons développer, en outre, des techniques de *model-checking* sur EB^3. Une des perspectives discutées dans la section 8.2 qui nous semble particulièrement intéressante concerne la définition d'une nouvelle relation de raffinement en EB^3. Le système serait décrit de manière très abstraite. Ensuite, par raffinements successifs, de nouveaux types d'entité et associations, et de nouveaux événements seraient introduits progressivement afin de spécifier tous les éléments composant le SI. Grâce à cette nouvelle notion de raffinement en EB^3, on assurerait ainsi que le système est globalement cohérent et qu'il vérifie les propriétés désirées. L'intérêt d'une telle approche est de pouvoir se concentrer sur un petit nombre de propriétés à la fois. Ensuite, les différentes techniques présentées dans cette thèse peuvent être utilisées pour spécifier et vérifier des propriétés statiques.

En définissant la méthode EB^4, nous avons choisi de garder une certaine liberté au niveau de l'implémentation. En effet, cette dernière peut être obtenue soit par raffinement à partir de la spécification B, soit par synthèse automatique à partir de la spécification EB^3. La raison est la suivante : notre objectif initial était de développer EB^4 indépendamment de toute autre approche ou projet. Toutefois, nous nous sommes rendu compte que le fait d'utiliser autant de techniques formelles avait tendance à rebuter les professionnels des SI qui pourraient être amenés à utiliser une méthode comme EB^4. Pour ces raisons, nous avons développé EB^4 avec deux objectifs distincts :

- comme une méthode à part entière pour concevoir des SI,
- comme un outil de vérification d'appoint des propriétés statiques dans le cadre du projet APIS.

Dans cette partie de la thèse, nous avons essentiellement présenté les contributions liées au premier objectif. Nous comptons bien sûr poursuivre le développement d'EB^4 comme méthode, notamment au travers de la définition d'outils graphiques ou pédagogiques. Ces travaux permettront de poser des problèmes concrets liés à la diffusion et à une meilleure compréhension des méthodes formelles dans le domaine des SI. Il s'agit en effet du seul moyen dont nous disposons aujourd'hui pour faire accepter de telles approches dans ce domaine.

L'autre objectif consiste à utiliser EB^4 comme un outil complémentaire d'EB^3 et de son environnement d'applications, APIS. L'idée est d'utiliser la spécification B de l'étape 6 pour des fins de vérification et de documentation, mais la spécification principale reste la description en EB^3. Comme les algorithmes de traduction de définitions d'attributs en B présentés dans le chapitre 6 semblaient très proches des techniques de synthèse de programmes, nous avons également travaillé, dans le cadre de notre thèse, sur la génération automatique de transactions BD relationnelles qui correspondent aux définitions d'attributs EB^3. Ces contributions sont décrites dans les chapitres suivants.

Troisième partie

Synthèse de transactions

Chapitre 9

Génération de transactions BD relationnelles

"Le doute est le sel de l'esprit."

— Alain

Dans ce chapitre, nous présentons les principaux algorithmes de traduction des définitions d'attributs EB^3 en transactions Java/SQL. La section 9.1 introduit le projet APIS qui constitue la motivation première de ce travail. Puis, l'algorithme principal est détaillé dans la section 9.2. Enfin, la section 9.3 conclut ce chapitre en présentant les optimisations possibles du processus de synthèse.

9.1 Projet APIS

Le projet APIS (qui est le sigle de "Automatic Production of Information Systems") [FFLR02] a pour objectif de générer des SI à partir de spécifications formelles décrites en EB^3. L'idée est de libérer le concepteur de SI des détails d'implémentation pour qu'il se consacre aux phases d'analyse et de spécification. Plutôt que d'utiliser des techniques de raffinement comme dans les approches basées sur les transitions d'états [Edm95, Mam02], les outils d'APIS sont basés principalement sur des techniques de synthèse automatique.

9.1.1 Composantes du projet APIS

La figure 9.1 présente les différentes composantes d'APIS. Du point de vue du concepteur, la spécification en EB^3 d'un SI est décomposée en cinq parties (voir chapitre 4) : a) une description des spécifications des GUI (sigle en anglais de *graphical user interface*) ; b) un diagramme ER ; c) un ensemble de définitions d'attributs ; d) un ensemble de règles d'entrée-sortie (notées règles E/S dans la figure) ; e) des expressions de processus. Dans le projet APIS, l'environnement d'exécution du SI est construit autour d'un moteur d'analyse des expressions de processus qui vérifie que tout nouvel événement du système respecte le comportement décrit dans la spécification. Pour mettre à jour ou pour interroger le système, l'utilisateur génère un événement à travers une interface Web. L'événement est ensuite analysé par EB^{3}PAI [FF02, Fra06], l'interpréteur

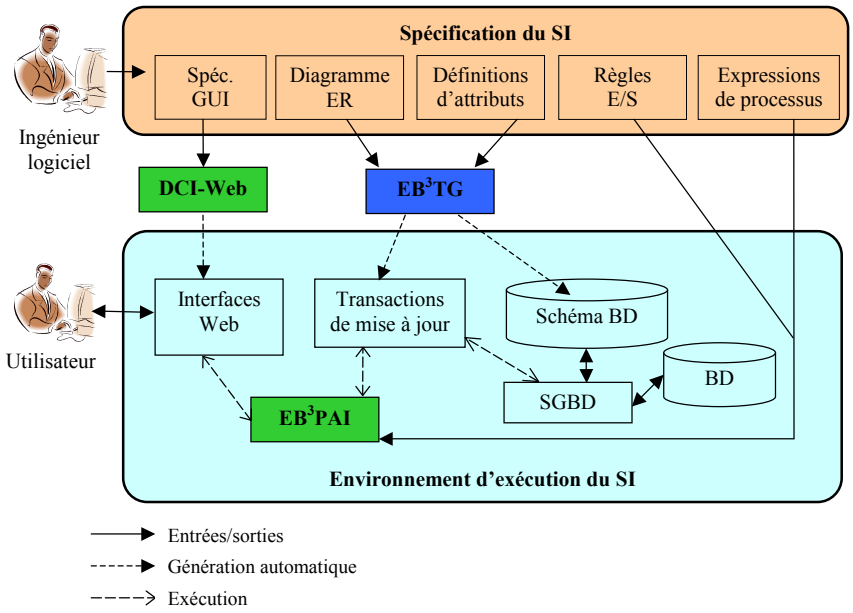

FIG. 9.1 – Composantes du projet APIS

des expressions de processus EB^3. Si le nouvel événement est considéré comme valide, alors il est exécuté par l'intermédiaire des transactions de mise à jour de la BD ; sinon, un message d'erreur est envoyé à l'utilisateur.

Les outils DCI-Web et EB^{3}TG sont utilisés à l'initialisation pour fournir à l'environnement d'exécution les interfaces Web et les transactions de mise à jour dont le moteur d'analyse EB^{3}PAI a besoin pour interagir avec l'utilisateur et avec la BD. L'outil appelé DCI-Web permet de générer des interfaces Web [Ter05] à partir de la description des GUI. Notre contribution a été de définir des algorithmes qui permettent de générer automatiquement des programmes Java qui exécutent des transactions de BD relationnelles correspondant à la spécification des attributs du SI en EB^3. Ces travaux ont permis d'implémenter l'outil EB^{3}TG, qui fait le lien entre l'interprétation des expressions de processus EB^3 et la gestion effective de la BD du SI. Les transactions générées par EB^{3}TG peuvent ainsi être utilisées en combinaison avec l'outil EB^{3}PAI afin de mettre à jour ou interroger la BD lorsque les événements correspondants sont considérés comme valides par l'interprétation des expressions de processus EB^3. EB^{3}TG est compatible avec différents systèmes de gestion de BD (SGBD).

9.1.2 Exemple d'application d'EB^{3}TG

Pour illustrer l'intérêt de nos algorithmes, nous présentons dans la figure 9.2 un extrait de code Java qui a été automatiquement généré par l'outil à partir de la spécification EB^3 pour les transactions Acquire et Discard lorsque le SGBD est Oracle. Les actions Acquire et Discard n'apparaissent que dans les fonctions *bookKey* et *title*. Pour générer ces transactions, nous devons réaliser des analyses similaires à celles présentées dans le chapitre 6 pour la traduction

```
public static void Acquire(int bId,String bTitle){
    try{
        int var0 = connection.createStatement().executeUpdate("UPDATE book SET "+
            "title = '"+bTitle+"' "+
            "WHERE bookKey = "+ bId +" ");
        if( var0==0 ){
            connection.createStatement().executeUpdate("INSERT INTO book ( bookKey,title) "+
                " VALUES ( "+ bId +",'"+bTitle+"')");
        }
        connection.commit();
    } catch ( Exception e ) {
        connection.rollback();
        System.err.println(e.getMessage());
    } finally {
        connection.closeAllStatements();
    }
}

public static void Discard(int bId){
    try {
        connection.createStatement().executeUpdate("DELETE FROM book "+
            "WHERE bookKey = "+ bId +" ");
        connection.commit();
    } catch ( Exception e ) {
        connection.rollback();
        System.err.println(e.getMessage());
    } finally {
        connection.closeAllStatements();
    }
}
```

FIG. 9.2 – Exemple de code Java généré par l'outil EB^{3}TG

en B. Les algorithmes de génération des transactions DB relationnelles sont détaillés dans les sections suivantes. Pour accéder au SGBD et pour intégrer des énoncés SQL dans le code Java, les programmes utilisent la technologie "Java Database Connectivity" (JDBC). JDBC fournit en effet une bibliothèque de méthodes qui gèrent l'ouverture et la fermeture de connexion au SGBD, l'envoi de code SQL vers la BD et le traitement des résultats retournés par la BD. Par exemple, la transaction générée pour Discard consiste essentiellement en un énoncé **DELETE**, car l'effet sur l'attribut *bookKey* d'un événement de la forme Discard(bId) est spécifié par l'expression : $bookKey(front(s)) - \{bId\}$. Dans la transaction de Discard, on a alors une instruction de la forme suivante :

```
connection.createStatement().executeUpdate(...);
```

où l'objet `connection` représente une connexion au SGBD. Ce code permet au programme Java de s'interfacer avec la BD afin d'exécuter l'énoncé SQL. Si l'instruction **DELETE** a été exécutée avec succès, alors une validation de la transaction ("commit") est envoyée à la BD par l'instruction suivante : `connection.commit()`. Sinon, une annulation de la transaction ("rollback") est transmise par l'instruction suivante : `connection.rollback()`. À la fin de la transaction, la connexion est fermée avec la méthode `closeAllStatements()`. La transaction générée pour Acquire suit le même principe, mais l'exécution des énoncés SQL est plus complexe. Les algorithmes de génération et les exemples seront détaillés dans les sections suivantes.

L'implémentation de transactions complexes sans outil de synthèse peut être une source facile d'erreurs, en particulier lorsque les insertions et les mises à jour

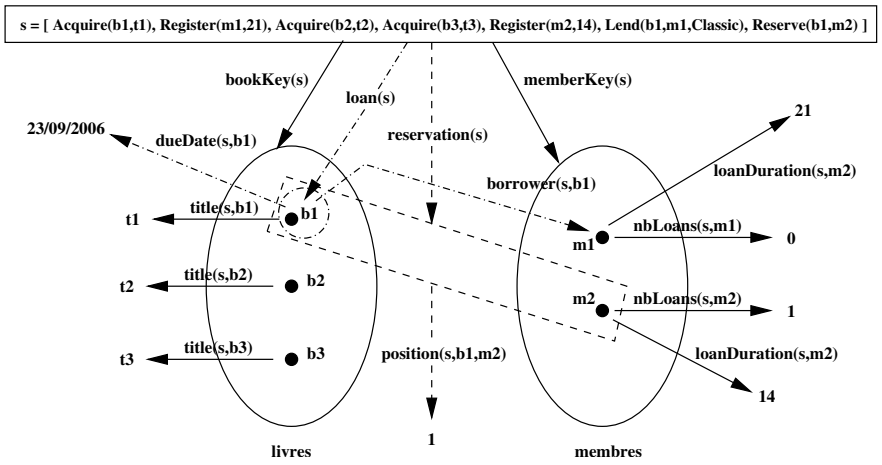

FIG. 9.3 – Modèle de données représenté par les définitions d'attributs EB^3

impliquent de nombreux tuples. En générant automatiquement les transactions Java/SQL à partir des définitions d'attributs EB^3, EB^{3}TG permet ainsi d'éviter des erreurs de codage et le concepteur peut alors concentrer ses efforts sur les phases d'analyse et de spécification des SI.

9.2 Algorithmes de génération des transactions

Dans une spécification EB^3, il existe une seule variable d'état, la trace courante du système. La figure 9.3 montre comment le modèle de données du SI est représenté par les définitions d'attributs à partir de la trace courante. Considérons par exemple la trace suivante :

$$s = [\mathsf{Acquire}(b1, t1), \mathsf{Register}(m1, 21), \mathsf{Acquire}(b2, t2), \mathsf{Acquire}(b3, t3), \\ \mathsf{Register}(m2, 14), \mathsf{Lend}(b1, m1, \mathit{Classic}), \mathsf{Reserve}(b1, m2)]$$

Les définitions de clé caractérisent l'ensemble des valeurs de clé existantes pour chaque type d'entité et association. Par exemple, la fonction *bookKey* associe la trace s à l'ensemble $\{b1, b2, b3\}$, qui représente l'ensemble des livres existants. De manière analogue, l'ensemble des membres existants est $\{m1, m2\}$ dans la fonction *memberKey* et l'ensemble des réservations inclut seulement le couple $(b1, m2)$ dans la fonction *reservation*, puisqu'un seul événement Reserve apparaît dans la trace s. Les valeurs des attributs sont calculées grâce aux définitions d'attributs non clé. Par exemple, le titre du livre $b2$ est retrouvé par le calcul de $\mathit{title}(s, b2)$.

Cette représentation découle de la sémantique par trace adoptée en EB^3. Cependant, l'utilisation d'une trace n'est pas appropriée dans une implémentation du SI, parce que la trace d'un système croît indéfiniment. Nous avons donc choisi de stocker chaque attribut dans une BD relationnelle conventionnelle, qui respecte le diagramme ER. Il nous faut alors générer des transactions qui mettent à jour cette BD suivant les spécifications des définitions d'attributs. La section 9.2.1 présente l'algorithme général de synthèse des transactions.

9.2.1 Algorithme général

Plusieurs analyses sont requises pour générer des transactions de BD relationnelles à partir des définitions d'attributs. Comme dans le cadre de la traduction vers B (voir chapitre 6), il faut analyser les clauses d'entrée des définitions d'attributs, et le cas échéant, construire et analyser des arbres de décision pour les termes conditionnels.

Les clauses d'entrée des définitions d'attributs sont analysées afin de déterminer :

- quels sont les attributs qui sont affectés par l'exécution d'une action a ;
- et quels sont les effets de cette action a sur les différents attributs.

En particulier, il faut déterminer quels sont les tuples à supprimer, à insérer et à mettre à jour dans les tables de la BD. Soit b une définition d'attribut. Soit t la trace courante du système. L'objectif d'une telle analyse est de déterminer, lorsqu'un événement de a est reçu par le système, quels sont les tuples $\{\overrightarrow{v}\}$ tels que :

$$b(t :: a, \overrightarrow{v}) \neq b(t, \overrightarrow{v})$$

Pour définir la transaction correspondant à l'action a, nous générons, pour chaque table affectée par a, les énoncés SQL qui correspondent aux effets de l'action, sous la forme d'énoncés **DELETE**, **UPDATE** et **INSERT**, que nous appelons par la suite des énoncés DUI. Une des difficultés de la synthèse de transactions concerne le niveau d'abstraction des spécifications EB^3. Par analogie aux substitutions parallèles du langage B, les définitions d'attributs font abstraction des ordonnancements des énoncés DUI. Par exemple, une substitution de la forme $x, y := y, x$ peut être implémentée par une séquence d'instructions de la forme suivante : $temp := x\,;\, x := y\,;\, y := temp$. Dans le cadre de la synthèse de transactions, nous avons choisi de générer automatiquement des variables et des tables temporaires afin de stocker les tuples à supprimer, à mettre à jour ou à insérer qui sont caractérisés par des termes conditionnels. Cette phase de l'algorithme est détaillée dans le chapitre 10.

L'algorithme général est le suivant :

```
(1)   créer les tables de la BD
(2)   initialiser la BD
(3)   pour chaque action a de la spécification EB3
(4)       analyser les clauses d'entrée pour a
(5)       définir une transaction pour a
(6)           générer les définitions SQL des variables et des tables
(7)           temporaires
(8)           pour chaque table T affectée par a
(9)               générer les énoncés DUI
(10)          générer un commit et un rollback
```

Les étapes (1) et (2) sont détaillées dans les sections 9.2.2 et 9.2.3, respectivement. L'analyse des clauses d'entrée (ligne (4)) est brièvement rappelée et adaptée à la synthèse de transactions dans la section 9.2.4. La définition des variables et des tables temporaires (ligne (6)) est discutée dans le chapitre 10. Enfin, la génération des transactions (lignes (5)-(10)) est présentée dans la section 9.2.5.

9.2.2 Création des tables de la BD

Les algorithmes standard de [EN04] sont utilisés pour définir les tables relationnelles à partir du diagramme ER de la spécification EB^3. La signature des actions permet de déterminer les attributs qui peuvent prendre la valeur **NULL**. Si le type d'un paramètre d'entrée est décoré par le symbole "$\perp$", alors l'attribut correspondant accepte la valeur **NULL**. Par exemple, les définitions de tables pour les types d'entité *book* et *member* sont respectivement :

```
CREATE TABLE book (
    bookKey int PRIMARY KEY,
    title varchar(20)
);
CREATE TABLE member (
    memberKey int PRIMARY KEY,
    nbLoans int NOT NULL,
    loanDuration int NOT NULL
);
```

9.2.3 Initialisation

L'initialisation de la BD est déterminée lors de l'analyse des clauses d'entrée ; il s'agit en effet d'un cas particulier de clause d'entrée dans les définitions d'attributs. Pour chaque définition d'attribut, il existe une clause de la forme "$\perp : u$". Cette dernière correspond au cas initial, puisque la trace t du système est vide. La valeur la plus courante pour l'expression u est "$\varnothing$" pour une définition de clé et "$\perp$" pour une définition d'attribut non clé. Dans ce cas, la table correspondante ne contient aucun tuple à l'initialisation.

9.2.4 Analyse des clauses d'entrée

Les résultats de l'analyse des clauses d'entrée (ligne (4) de l'algorithme général) sont les suivants :

1. les attributs affectés par l'action a, dénotés par $Att(\mathsf{a})$,
2. les tables affectées par a, notées par $\mathbb{T}(\mathsf{a})$,
3. pour chaque table T de $\mathbb{T}(\mathsf{a})$,
   - les valeurs de clé des tuples à supprimer de la table T, notées par $K_{Delete}(T, \mathsf{a})$,
   - les valeurs de clé des tuples à insérer ou à mettre à jour dans T, notées par $K_{Change}(T, \mathsf{a})$.

Dans la suite de cette thèse, les expressions de la forme $eKey(front(s))$ et $b(front(s), k_1, ..., k_m)$ sont simplifiées respectivement par $eKey()$ et $b(k_1, ..., k_m)$, car les appels de définitions d'attributs dans les clauses d'entrée portent toujours sur $front(s)$.

9.2.4.1 Algorithme

Pour calculer $\mathbb{T}(\mathsf{a})$, il faut déterminer, pour chaque attribut b de $Att(\mathsf{a})$, la table de b, notée par $table(b)$, grâce aux définitions de tables générées dans la

section 9.2.2. Alors, l'ensemble $\mathbb{T}(\mathsf{a})$ est défini par :

$$\mathbb{T}(\mathsf{a}) = \{table(b) \mid b \in Att(\mathsf{a})\}$$

Soit T une table de $\mathbb{T}(\mathsf{a})$. Afin de calculer $K_{Delete}(T, \mathsf{a})$ et $K_{Change}(T, \mathsf{a})$, on considère, pour chaque attribut b de $Att(\mathsf{a})$ tel que la table de b soit T, l'ensemble (qu'on appelle $K_D(b)$ dans la suite) des valeurs de clé des tuples à supprimer d'après la définition de b, et l'ensemble (noté $K_{IU}(b)$) des valeurs de clé des tuples à insérer ou à mettre à jour d'après la définition de b. Formellement, il faut déterminer, pour chaque attribut b de $Att(\mathsf{a})$, les ensembles $K_D(b)$ et $K_{IU}(b)$ tels que :

$$\begin{aligned} K_{Delete}(T, \mathsf{a}) &= \cup_{\{b \in Att(\mathsf{a}) \wedge table(b) = T\}} K_D(b) \\ K_{Change}(T, \mathsf{a}) &= \cup_{\{b \in Att(\mathsf{a}) \wedge table(b) = T\}} K_{IU}(b) \end{aligned}$$

Les ensembles $K_D(b)$ et $K_{IU}(b)$ sont déterminés par analyse des clauses d'entrée. L'algorithme est le suivant :

déterminer $Att(\mathsf{a})$
pour chaque attribut b de $Att(\mathsf{a})$
  si b est défini par un terme conditionnel
    générer un arbre de décision pour b
    déterminer les requêtes SQL d'interrogation
    définir les variables et les tables temporaires
  déterminer $table(b)$, $K_{IU}(b)$, $K_D(b)$
calculer $\mathbb{T}(\mathsf{a})$
pour chaque T dans $\mathbb{T}(\mathsf{a})$
  calculer $K_{Delete}(T, \mathsf{a})$,$K_{Change}(T, \mathsf{a})$

Pour retrouver dans la BD les tuples caractérisés en EB^3 par les prédicats des termes conditionnels, des requêtes SQL d'interrogation sont générées. Ces requêtes sont définies par nos algorithmes sous la forme d'énoncés **SELECT**. La génération des énoncés SQL est détaillée dans le chapitre 10. Les autres étapes de l'algorithme sont présentées dans les sections suivantes.

9.2.4.2 Calcul de $Att(\mathsf{a})$

L'ensemble $Att(\mathsf{a})$ est défini comme $B(\mathsf{a})$ dans le cas de la traduction vers B (voir section 6.3.2). Un attribut b est affecté par l'action a s'il existe au moins une clause d'entrée de la forme "$a(\overrightarrow{p}) : u$" dans la définition de b. Par exemple, l'action Transfer apparaît dans les définitions d'attributs de *nbLoans*, *dueDate* et *borrower* ; par conséquent, nous avons :

$$Att(\mathsf{Transfer}) = \{nbLoans, dueDate, borrower\}$$

9.2.4.3 Calcul de $K_D(b)$ et $K_{IU}(b)$

Les ensembles $K_D(b)$ et $K_{IU}(b)$ sont tout d'abord initialisés à "$\varnothing$". Nous rappelons que, pour chaque attribut b de $Att(\mathsf{a})$, plusieurs clauses d'entrée de la forme "$a(\overrightarrow{p_j}) : u_j$" peuvent être définies pour la même action a. Ces clauses sont alors analysées dans l'ordre de leur déclaration dans la définition d'attribut correspondante. Comme dans la section 6.3.2.2, une substitution θ_{u_j} est définie

pour chaque clause d'entrée ; elle permet de relier les paramètres actuels de la clause aux paramètres formels de l'action.

Les ensembles $K_D(b)$ sont calculés uniquement pour les définitions de clé. Si b est une définition de clé, alors l'expression u_j est un terme fonctionnel et une valeur v a été déterminée par analyse du filtrage. Si le terme u_j contient le symbole "−", alors la valeur v est ajoutée à l'ensemble $K_D(b)$; sinon, elle est ajoutée à l'ensemble $K_{IU}(b)$. Par exemple, la clause d'entrée de Discard dans la fonction *bookKey* est associée à l'expression suivante : $bookKey() - \{bId\}$. Par conséquent, on a : $K_D(bookKey) = \{bId\}$.

Pour calculer les ensembles $K_{IU}(b)$, nous considérons à la fois les définitions de clé et les définitions d'attributs non clé. Si l'expression u_j de la clause d'entrée est un terme fonctionnel, alors une valeur de clé v a été entièrement déterminée. D'un autre côté, si u_j est un terme conditionnel, alors il faut analyser les différentes conditions des prédicats dans les parties **if**. Les variables dans $\overrightarrow{k} \cap var(\overrightarrow{p_j})$ sont déterminées par le filtrage θ_{u_j}, alors que les variables dans $\overrightarrow{k} - var(\overrightarrow{p_j})$ sont déterminées par les conditions du terme conditionnel u_j. Comme pour la traduction vers B, nous utilisons des arbres de décision afin d'identifier et d'analyser les valeurs de clé caractérisées par les prédicats des parties **if** des termes conditionnels. Ces arbres ont été définis dans la section 6.3.2.2. Les feuilles de l'arbre de décision sont des termes fonctionnels définis dans les clauses **then** de l'expression u_j et ses branches sont étiquetées par les prédicats décrits dans les clauses **if**. En analysant l'arbre de décision, il est possible de déterminer un ensemble de tuples $KV_{j,i}$ pour chaque feuille $ft_{j,i}$ de l'arbre. Chaque élément de $KV_{j,i}$ est ajouté à $K_{IU}(b)$. Par exemple, dans le cas de l'action Transfer pour la fonction *nbLoans*, on obtient : $K_{IU}(nbLoans) = \{mId', borrower(bId)\}$.

9.2.4.4 Définition de variables et de tables temporaires

Les valeurs de clé à insérer, à mettre à jour ou à supprimer sont caractérisées en EB^3 par des prédicats du premier ordre dans les termes conditionnels. Pour retrouver les tuples correspondants dans les tables de la BD, des énoncés de type **SELECT** sont définis dans les transactions Java/SQL qui sont générées par nos algorithmes. Une variable temporaire est définie au niveau du langage hôte (en l'occurrence Java) lorsqu'un seul enregistrement de la table est concerné, tandis qu'une table temporaire est créée quand plusieurs enregistrements sont en jeu. Ces définitions permettent de manipuler les données en question dans la transaction à générer. La définition des variables et des tables temporaires est couplée avec l'analyse des arbres de décision, décrite dans les paragraphes précédents. La génération des énoncés **SELECT** qui correspondent aux valeurs caractérisées par les prédicats des termes conditionnels est présentée dans le chapitre 10.

Pour les besoins de cette thèse, un pseudo-code de haut niveau est utilisé pour décrire les transactions générées par nos algorithmes. Dans la pratique, ce pseudo-code est traduit en Java. Par exemple, pour caractériser l'ensemble des livres empruntés par le membre identifié par *mId*, la table suivante est générée :

```
CREATE TEMPORARY TABLE
    TAB (bookKey int PRIMARY KEY) ;
INSERT INTO TAB
```

SELECT loan.bookKey
FROM loan
WHERE loan.borrower = #mId ;

Par convention, une variable utilisée dans un énoncé SQL est préfixée par le symbole "#", afin de la distinguer des noms d'attributs.

Les variables et les tables temporaires sont définies au début de chaque transaction, parce que les prédicats des termes conditionnels impliquent toujours les valeurs d'attributs qui sont valables avant l'exécution de la transaction. Par conséquent, nous avons défini les variables et les tables de telle sorte qu'elles soient indépendantes de l'ordonnancement des énoncés DUI. D'autre part, nous utilisons des tables temporaires, plutôt que des vues, afin d'éviter qu'un énoncé DUI ultérieur de la transaction ne modifie la valeur des ensembles de tuples à traiter.

9.2.5 Définition des transactions

Pour définir les transactions Java/SQL, tous les énoncés DUI sont groupés par table. Grâce à l'analyse des clauses d'entrée, les énoncés de type **DELETE** ont été distingués des autres énoncés SQL. Les enregistrements à supprimer de la table T sont indiqués dans l'ensemble $K_{Delete}(T, \mathsf{a})$. Ces énoncés sont groupés au début de la liste d'instructions de chaque table. En effet, la syntaxe des définitions d'attributs EB^3 n'autorise pas, dans les définitions de clé, des expressions de la forme $eKey() \cup \{...\} - \{...\}$, qui pourraient impliquer un ordre entre les énoncés de type **DELETE** et les autres. En fait, le problème ne se pose pas, car de telles expressions peuvent toujours être réécrites en des expressions équivalentes de la forme $eKey() - \{...\} \cup \{...\}$.

L'analyse des clauses d'entrée ne permet pas de distinguer les tuples à insérer de ceux à mettre à jour. En effet, une expression de la forme "$eKey() \cup \{k\}$" correspond soit à une insertion, soit à une mise à jour, suivant que les deux ensembles sont disjoints ou pas. La modélisation en EB^3 ne permet donc pas de faire cette distinction, à cause du niveau d'abstraction et de l'utilisation de la théorie des ensembles. Ces types d'énoncés DUI impliquent tous les deux l'utilisation de l'opérateur "$\cup$" dans les définitions d'attributs, car les effets sur la BD sont les mêmes : un tuple identifié par la clé k doit appartenir à la BD après l'exécution de la transaction. Dans la traduction vers B, nous n'avions pas ce problème, car l'opérateur de surcharge ("$<+$") correspond exactement à ce comportement. Dans le cadre de la synthèse de transactions, nous avons choisi de coder les deux interprétations possibles de l'expression avec un énoncé SQL complexe. Ainsi, les expressions avec le symbole "$\cup$" sont implémentées par une mise à jour, suivie d'une insertion.

Soit T une table de $\mathbb{T}(\mathsf{a})$. Pour chaque k dans $K_{Change}(T, \mathsf{a})$, un ensemble $L = \{UPD_1, ..., UPD_p\}$ d'énoncés de type **UPDATE** est généré pour chaque attribut b de T, affecté par l'action a et tel que $k \in K_{IU}(b)$. D'autre part, un énoncé de type **INSERT** est également généré pour k et le même ensemble d'attributs. Ensuite, on considère le premier énoncé de type **UPDATE** (notons-le UPD_1, mais l'ordre n'a pas d'importance ici) et une conditionnelle est définie de manière à déterminer si UPD_1 a effectivement mis à jour la BD. Si c'est le cas, alors les autres énoncés de L sont aussi exécutés. Sinon, l'énoncé de type **INSERT** est exécuté. Le sous-algorithme pour la ligne (9) de l'algorithme

général présenté dans la section 9.2.1 est le suivant :

pour chaque k dans $K_{Delete}(T, \mathsf{a})$
  déterminer and générer les énoncés de type **DELETE** avec k
pour chaque k dans $K_{Change}(T, \mathsf{a})$
  pour chaque attribut b de T dans $Att(\mathsf{a})$
    si k est dans $K_{IU}(b)$
      calculer la valeur $b(k)$
  déterminer l'ensemble L des énoncés de type **UPDATE**
    notés UPD_l, $1 \leq l \leq p$, pour k et les $b(k)$
  déterminer l'énoncé de type **INSERT**, noté *INS*, pour k et les $b(k)$
  générer UPD_1
  générer l'instruction suivante :
    IF SQL%NotFound
    THEN *INS*
    ELSE UPD_2 ; ... UPD_p ;
    END ;

où la variable "SQL%NotFound" contient une valeur retournée par le SGBD qui indique si la mise à jour a effectivement modifié le tuple dans la table.

Par exemple, la transaction générée pour Discard est :

```
TRANSACTION Discard(bId : int)
  DELETE FROM book
  WHERE bookKey = #bId;
  COMMIT;
```

Cette définition est simple, car l'action Discard ne fait que supprimer le tuple identifié par la clé *bId*. En revanche, lorsque l'action implique des insertions ou des mises à jour, alors la transaction générée est plus complexe. Par exemple, on obtient pour Acquire :

```
TRANSACTION
  Acquire(bId :int,bTitle :varchar(20))
  /* mise à jour */
  UPDATE book SET title = #bTitle
  WHERE bookKey = #bId;
  /* conditionnelle */
  IF SQL%NotFound
  THEN
    /* insertion */
    INSERT INTO book(bookKey,title)
    VALUES (#bId,#bTitle);
  END;
  COMMIT;
```

La partie **ELSE** de la conditionnelle n'est pas indiquée ici, car une seule mise à jour a été générée pour Acquire. Cet exemple démontre que la transaction générée a effectivement le comportement attendu, même si sa définition comporte des instructions inutiles. En réalité, nous savons par notre compréhension de la spécification complète de la bibliothèque (notamment, en analysant les expressions de processus EB^3) que l'action Acquire est un producteur de livre. Par conséquent, l'énoncé de type **UPDATE** ne sera jamais exécutée parce que l'entité *mId* ne peut pas exister dans la BD avant l'exécution de son producteur.

9.3 Conclusions et perspectives

Dans ce chapitre, nous avons présenté l'algorithme général que nous avons défini pour générer automatiquement des transactions de BD relationnelles à partir de définitions d'attributs EB^3. Ces transactions ont été prévues pour être utilisées dans le cadre du projet APIS. Ainsi, elles sont appelées par l'outil EB^{3}PAI (voir section 9.1) pour faire des requêtes ou pour mettre à jour la BD, lorsque les événements correspondants sont considérés comme valides par interprétation des expressions de processus EB^3.

Les énoncés SQL générés pour les **UPDATE** et les **INSERT** peuvent parfois être simplifiés en analysant les définitions de clé. Soit k une valeur de clé appartenant à $K_{Change}(T, \mathsf{a})$. Pour chaque attribut non clé b de la table T dans $Att(\mathsf{a})$, si k appartient à $K_{IU}(b)$, alors on recherche dans la définition d'attribut de la clé de T s'il existe une clause d'entrée pour l'action a contenant le symbole "$\cup$". S'il n'en existe pas, alors l'énoncé DUI pour k ne peut être qu'une mise à jour de type **UPDATE**, car une insertion requiert aussi une union dans la définition de clé pour représenter l'ajout de la clé correspondante. Dans les autres cas, il est impossible de distinguer les insertions des mises à jour, sans une analyse complémentaire des expressions de processus EB^3. Parmi nos perspectives de travail, nous comptons apporter quelques améliorations à notre algorithme lors de son intégration avec les autres outils de l'environnement APIS. En particulier, l'interpréteur EB^{3}PAI des expressions de processus EB^3 peut être utilisé pour distinguer les actions de type producteur et modificateur.

Chapitre 10

Patrons SELECT

> "Un génie écrit des programmes que même un idiot peut comprendre, alors qu'un idiot écrit des programmes que même le compilateur ne peut pas comprendre."
>
> — David Harel

Dans le chapitre 9, nous avons montré comment générer des transactions de BD relationnelles qui correspondent aux définitions d'attributs EB^3. Lors de la synthèse des transactions, l'algorithme prévoit notamment d'analyser les prédicats des termes conditionnels des définitions d'attributs pour déterminer les tuples à insérer, à mettre à jour ou à supprimer. Dans ce chapitre, nous montrons comment générer les énoncés **SELECT** qui correspondent à ces prédicats et qui permettent de retrouver les tuples en question. Dans ce but, nous avons défini une bibliothèque de patrons d'énoncés **SELECT** pour les formes les plus typiques des prédicats rencontrés dans les définitions d'attributs. La section 10.1 commence par présenter quelques notations utilisées pour les besoins de ce chapitre. Dans la section 10.2, nous présentons les patrons de base qui sont utilisés par nos algorithmes. Puis, la section 10.3 présente des patrons plus complexes qui permettent de réutiliser les patrons de base. La section 10.4 indique comment générer l'énoncé SQL qui correspond à une conjonction de prédicats. Enfin, la section 10.5 conclut ce chapitre par une discussion sur les techniques de synthèse par patrons présentées dans cette thèse.

10.1 Introduction

Dans ce chapitre, on considère les clauses d'entrée d'une définition d'attribut b qui a la forme suivante :

$$
\begin{array}{l}
b(s : \mathcal{T}(\mathsf{main}), \overrightarrow{k} : \overrightarrow{T}) : T' \triangleq \\
\quad \ldots \\
\quad \mathsf{a}(\overrightarrow{p}) : \ \textbf{if } \mathit{Pred} \textbf{ then } \mathit{FctTerm} \\
\qquad\quad \textbf{else } \mathit{Expression} \textbf{ end} \\
\quad \ldots
\end{array}
$$

où $\overrightarrow{k}$ est la liste des paramètres qui représentent les attributs clé de la table de b, $\overrightarrow{p}$ est la liste des paramètres de l'action a, *Pred* est un prédicat, *FctTerm* est

un terme fonctionnel et *Expression* peut être soit un terme fonctionnel, soit une autre expression **if then else end** de cette forme. Dans le problème qui suit, les valeurs de $\overrightarrow{p}$ sont connues, puisque le filtrage permet de relier ces paramètres aux valeurs indiquées dans l'événement à exécuter, tandis que les variables de $\overrightarrow{k}$ sont les inconnues à déterminer grâce à un énoncé **SELECT** déduit du prédicat *Pred*. Dans la suite, chaque attribut est préfixé par sa table pour éviter des confusions de noms. La forme générale de l'expression *Pred* est une conjonction de prédicats, $P_1 \wedge P_2 \wedge ... \wedge P_n$, où chaque P_i est une instance I de l'un des patrons de prédicats, ou de sa négation, notée alors par $\neg I$.

Le langage de définition d'attribut préconise qu'au moins un des prédicats P_i de la conjonction soit un prédicat positif, où les valeurs de $\overrightarrow{k}$ sont bornées par des valeurs de la BD. Par exemple, la définition d'attribut suivante n'est pas autorisée, parce qu'elle représente une mise à jour de tous les membres dont l'attribut clé *memberKey* a une valeur supérieure à 10, qu'ils appartiennent ou non à la BD :

$$
\begin{array}{l}
nbLoans(s : \mathcal{T}(\mathsf{main}), mId : memberKey\_Set) : \mathbb{N} \triangleq \\
\quad \vdots \\
\quad \mathsf{a}(\overrightarrow{p}) : \ \textbf{if } mId > 10 \textbf{ then } \ldots \textbf{ end} \\
\quad \vdots
\end{array}
$$

La définition suivante, qui restreint *mId* aux tuples existant dans la BD, est plus appropriée :

$$
\begin{array}{l}
\mathsf{a}(\overrightarrow{p}) : \ \textbf{if } mId > 10 \wedge mId \in memberKey(front(s)) \\
\qquad \textbf{then } \ldots \textbf{ end}
\end{array}
$$

Pour les besoins de description de nos patrons, l'expression $attname(v_i, g)$ est définie comme le nom de l'attribut dans la table de g qui correspond à la variable v_i dans la clause d'entrée, $table(g)$ représente la table dans laquelle l'attribut g est stocké et l'expression $T.key(j)$ désigne le j-ème attribut clé de la table T.

10.2 Patrons de base pour prédicats atomiques

Les sections suivantes présentent les énoncés **SELECT** qui correspondent aux formes les plus typiques des prédicats dans les termes conditionnels. Bien que les patrons soient indépendants les uns des autres, nous les décrivons dans un ordre croissant au niveau de la difficulté des énoncés SQL obtenus.

10.2.1 Patron $k = p$

Le prédicat le plus simple est une équation de la forme $k = p$, où $p \in \overrightarrow{p}$ est le paramètre d'un événement. Un tel patron est souvent utilisé lorsque le terme conditionnel représente simultanément plusieurs insertions ou mises à jour. Dans ce cas, k n'est pas déterminé par le filtrage et les prédicats de la partie **if** correspondent aux différentes valeurs possibles pour k. Par exemple, dans la clause d'entrée de *nbLoans* associée à l'action Transfer, un des prédicats est $mId = mId'$, où mId est la valeur à déterminer et mId' est un paramètre de Transfer.

Dans ce prédicat, p fournit immédiatement la valeur de k, par conséquent, on n'a pas besoin de faire une requête à la BD pour extraire la valeur de k. Toutefois, dans le but d'uniformiser les patrons et de simplifier nos algorithmes, chaque patron de prédicat est implémenté par un énoncé **SELECT**. En particulier, une conjonction de prédicats est représentée par un **SELECT** qui est construit à partir des énoncés **SELECT** de chacun des prédicats qui composent cette conjonction (voir section 10.4). Ainsi, la valeur p est représentée par une table temporaire *TempTab*, avec un seul tuple p, et on considère le **SELECT** suivant :

SELECT $TempTab.key(1)$
FROM $TempTab$;

Ce type de requête peut être optimisé ultérieurement si le prédicat en question n'intervient pas dans un autre patron.

10.2.2 Patron $k = g(\overrightarrow{p})$

On considère maintenant des équations de la forme $k = g(p_1, ..., p_l)$, où g est une définition d'attribut. Alors, k a simplement la valeur de l'attribut g dans l'enregistrement de la table identifié par $p_1, ..., p_l$. Par exemple, le prédicat $mId = borrower(bId)$, utilisé dans la clause d'entrée associée à l'action Transfer dans la fonction *nbLoans*, est une instance de ce patron.

L'énoncé **SELECT** correspondant à ce patron est de la forme suivante :

SELECT $projection(g, g)$
FROM $table(g)$
WHERE $restriction(g, \overrightarrow{p})$

où l'expression $projection(a, b)$ représente la projection de l'attribut b sur la table de l'attribut a. Ainsi, l'expression $projection(g, g)$ désigne le nom d'attribut $table(g).g$. Dans la clause **WHERE**, la notation $restriction(g, \overrightarrow{p})$ représente les restrictions sur la table de g par les valeurs $\overrightarrow{p}$; elle est définie par :

/* évaluation de g pour p_1 */
$table(g).attname(p_1, g) = \#p_1$ **AND**
...
/* évaluation of g pour p_l */
$table(g).attname(p_l, g) = \#p_l$;

Pour illustrer ce patron, le **SELECT** généré pour $mId = borrower(bId)$ est :

SELECT $loan.borrower$
FROM $loan$
WHERE $loan.bookKey = \#bId$;

où $table(borrower) = loan$ et $bookKey$ est la clé de la table $loan$.

10.2.3 Patron $k = g(\overrightarrow{k}, \overrightarrow{p})$, avec $k \notin \overrightarrow{k}$

Le patron de la section 10.2.2 peut être généralisé, puisqu'un ou plusieurs attributs clé à déterminer peuvent également apparaître comme des paramètres d'entrée de la définition d'attribut g. Dans ce cas, le prédicat est de la forme $k =$

$g(\overrightarrow{k}, \overrightarrow{p})$. Cette notation est une abstraction pour l'expression $k = g(w_1, ..., w_m)$, où m est le nombre de paramètres d'entrée de g et w_i est soit un attribut clé k_i, soit un paramètre p_j. Ainsi, $k = g(\overrightarrow{p})$ est effectivement un cas particulier de $k = g(\overrightarrow{k}, \overrightarrow{p})$. On note par $k_1, ..., k_n$ les w_i qui sont des attributs clé. On a alors la contrainte suivante : $n \leq m$. Dans ce patron, on considère que k et $k_1, ..., k_n$ sont des variables distinctes. L'énoncé **SELECT** associé à ce patron est de la forme suivante :

SELECT *projection*(g, g), *projection*$(g, \overrightarrow{k})$
FROM *table*(g)
WHERE *restriction*$(g, \overrightarrow{p})$

Puisque $\overrightarrow{k}$ peut être un ensemble d'attributs clé, la définition *projection*$(g, \overrightarrow{k})$ s'applique pour k_j appartenant à $\overrightarrow{k}$:

$$table(g).attname(k_1, g), ..., table(g).attname(k_n, g)$$

Supposons que l'attribut clé du type d'entité *book* soit implémenté par un entier. Comme illustration du patron décrit ci-dessus, on peut considérer les tuples (bId, mId) tels que le livre bId ait le même identifiant que la position du membre mId dans la liste de réservation du livre $b0$. Le prédicat est alors le suivant : $bId = position(b0, mId)$. En appliquant le patron de cette section avec les valeurs $k = bId$, $g = position$, $\overrightarrow{k} = mId$ et $\overrightarrow{p} = b0$, on obtient :

SELECT *reservation.position*,
reservation.memberKey
FROM *reservation*
WHERE *reservation.bookKey* $= \#b0$;

où le couple $(bookKey, memberKey)$ représente la clé de la table *reservation*.

10.2.4 Patron k *op* $g(\overrightarrow{k}, \overrightarrow{p})$, avec $k \in \overrightarrow{k}$

Dans les patrons précédents, k était caractérisé par une équation. Dans cette section, d'autres opérateurs de comparaison entre scalaires sont considérés. Le prédicat est de la forme k *op* $g(w_1, ..., w_m)$, où *op* est un opérateur binaire sur les scalaires tel que $=$, $>$, $\geq$, $<$ ou $\leq$, w_i est soit un attribut clé k_i, soit un paramètre p_j, et l'une des variables k_i, notée par k_j, est égale à k. L'énoncé **SELECT** correspondant est de la forme suivante :

SELECT *projection*(g, g), *projection*$(g, \overrightarrow{k} - k_j)$
FROM *table*(g)
WHERE *restriction*$(g, \overrightarrow{p})$
AND *predicateOp*

L'expression *projection*$(g, \overrightarrow{k} - k_j)$ est définie comme *projection*$(g, \overrightarrow{k})$, à l'exception de la projection sur k_j (c'est-à-dire $table(g).attname(k_j, g)$) qui est omise. La notation *predicateOp* représente le prédicat à satisfaire ; il est de la forme suivante :

$table(g).attname(k_j, g)$ *op* $table(g).g$;

Par exemple, si on considère l'ensemble des livres bId qui ont le même identifiant que leur position dans la liste de réservation du membre $m1$, alors le prédicat est : $bId = position(bId, m1)$. Dans ce cas, op est une égalité. En appliquant le patron avec $k = bId$, $g = position$, $\overrightarrow{k} = bId$ et $\overrightarrow{p} = m1$, on a alors :

SELECT *reservation.position*
FROM *reservation*
WHERE *reservation.memberKey* $= \#m1$
AND *reservation.bookKey* $=$ *reservation.position* ;

L'expression $projection(g, \overrightarrow{k} - k_j)$ n'apparaît pas ici, parce que $\overrightarrow{k}$ et k_j représentent le même attribut clé mId.

10.2.5 Patron $f(\overrightarrow{k^c}, \overrightarrow{k^1}, \overrightarrow{p^1})\ op\ g(\overrightarrow{k^c}, \overrightarrow{k^2}, \overrightarrow{p^2})$

Plus généralement, nous pouvons comparer deux définitions d'attributs f et g, avec un prédicat de la forme :

$$f(\overrightarrow{k^c}, \overrightarrow{k^1}, \overrightarrow{p^1})\ op\ g(\overrightarrow{k^c}, \overrightarrow{k^2}, \overrightarrow{p^2})$$

où $\overrightarrow{k^c}$ est un ensemble de variables qui sont des paramètres d'entrée à la fois dans f et g, et $\overrightarrow{k^c}$, $\overrightarrow{k^1}$ et $\overrightarrow{k^2}$ contiennent des variables distinctes. La notation op représente un opérateur binaire sur les scalaires comme $=$, $\geq$, $>$, $\leq$ ou $<$. Soit n_c le nombre de variables dans $\overrightarrow{k^c}$. L'énoncé **SELECT** est alors de la forme suivante :

SELECT $projection(T_f, \overrightarrow{k^c}), projection(T_f, \overrightarrow{k^1}),$
$projection(T_g, \overrightarrow{k^2})$
FROM $table(f)\ T_f, table(g)\ T_g$
WHERE $restriction(f, \overrightarrow{p^1})$
AND $restriction(g, \overrightarrow{p^2})$
AND $equijoin(f, g)$
AND *predicateOp* ;

L'utilisation des alias T_f et T_g est obligatoire pour ce patron, parce que f et g peuvent représenter le même attribut. Le premier paramètre de *projection* est remplacé par un alias, lorsque des confusions sont possibles. L'expression $equijoin(f, g)$ représente la jointure entre f et g ; elle est définie par :

$T_f.attname(k_1^c, f) = T_g.attname(k_1^c, g)$ **AND**
...
$T_f.attname(k_{n_c}^c, f) = T_g.attname(k_{n_c}^c, g)$

Par exemple, l'action $\mathsf{Cancel}(b1, m0)$ consiste à supprimer la réservation du livre $b1$ par le membre $m0$. Il faut alors déterminer tous les membres mId tels que leur position dans la liste de réservation du livre $b1$ soit supérieure à celle du membre $m0$ afin de décrémenter leur position d'une unité. Le prédicat est le suivant : $position(b1, mId) > position(b1, m0)$. L'opérateur op est alors “>”. En appliquant le patron décrit ci-dessus avec $f = g = position$, $\overrightarrow{k^c}$ est vide, $\overrightarrow{k^1} = mId$, $\overrightarrow{p^1} = b1$, $\overrightarrow{k^2}$ est vide et $\overrightarrow{p^2} = b1, m0$, on obtient :

SELECT $R1.memberKey$
FROM $reservation\ R1, reservation\ R2$
WHERE $R1.bookKey = \#b1$
AND $R2.bookKey = \#b1$
AND $R2.memberKey = \#m0$
AND $R1.position > R2.position$;

Cet exemple montre combien les spécifications EB^3 sont simples et compactes, comparées à leur implémentation en SQL.

10.2.6 Patron $k \in aRole(\overrightarrow{p})$

Un type d'entité qui participe à une association joue un *rôle* particulier dans cette dernière. La définition d'un rôle dépend de la cardinalité des associations (voir section 4.2.1.3). Dans le cas d'une association $M : N$, les rôles sont définis en EB^3 comme des sous-ensembles des instances de l'association. La figure 4.2 représente le diagramme ER d'une association $M : N$ entre les types d'entité e_1 et e_2. Notons par a cette association. Soient $\overrightarrow{k^1} = k_1^1, ..., k_{m_1}^1$ et $\overrightarrow{k^2} = k_1^2, ..., k_{m_2}^2$ les clés respectives de e_1 et e_2. Dans la figure 4.2, le type d'entité e_2 joue le rôle r_1 dans l'association a. Le rôle r_1 est alors défini par :

$$r_1(v_1^1, ..., v_{m_1}^1) = \{(v_1^2, ..., v_{m_2}^2) \mid (v_1^1, ..., v_{m_1}^1, v_1^2, ..., v_{m_2}^2) \in a\}$$

L'énoncé **SELECT** généré pour r_1 est de la forme suivante :

SELECT $projection(T_a, \overrightarrow{k^2})$
FROM $table(a)\ T_a$
WHERE $restriction(T_a, \overrightarrow{k^1})$;

Le rôle r_2 et son implémentation en SQL sont définis de manière similaire. Par exemple, le rôle *membRes* du type d'entité *member* dans *reservation* est défini en EB^3 par :

$$membRes(bId) = \{mId \mid (bId, mId) \in reservation\}$$

L'énoncé **SELECT** pour ce rôle est alors :

SELECT reservation.memberKey
FROM reservation
WHERE reservation.bookKey = #bId ;

Quand un rôle a une cardinalité d'au plus 1, il est alors défini comme une définition d'attribut et le patron approprié est utilisé à la place du patron décrit ci-dessus.

10.2.7 Patron $k \in eKey()$

Une définition de clé *eKey* renvoie un ensemble de valeurs. Il est possible de considérer des prédicats de la forme $k \in eKey()$. Toutefois, comme *eKey* peut contenir un très grand nombre de valeurs, k doit être borné par ailleurs par un autre prédicat. L'énoncé SQL est le suivant :

SELECT $table(eKey).eKey$
FROM $table(eKey)$;

Par exemple, pour le prédicat $mId \in memberKey()$, on obtient :

SELECT *member.memberKey*
FROM *member* ;

10.3 Patrons étendus

Ce patron permet de traiter les cas où un paramètre w_j de l'attribut f ou g est un appel de fonction d dans un des patrons de prédicat définis dans la section 10.2. Comme la technique est similaire pour les deux attributs, on présente le patron étendu pour f uniquement. La fonction est alors de la forme $f(..., w_{j-1}, d(...), w_{j+1}, ...)$. L'énoncé **SELECT** est déterminé de la manière suivante, à partir du patron de base associé au prédicat à étendre :

SELECT *projection*(*pattern*)
FROM *table*(*pattern*), *table*(*d*)
WHERE *restriction*(*pattern*$-w_j$)
  AND *join*(f, d)
  AND *restriction*(*d*)
  AND *predicateOp*(*pattern*)

Les notations *projection*(*pattern*), *predicateOp*(*pattern*) et *table*(*pattern*) représentent respectivement les projections, les prédicats et les tables définis dans le patron de base qui est réutilisé. Dans la clause **WHERE** de l'énoncé, l'expression *restriction*(*pattern*$-w_j$) désigne les restrictions fournies par le patron de base, hormis la restriction sur w_j qui est omise. L'expression *join*(f, d) définit la jointure entre f et d :

$$T_f.attname(d(...), f) = T_d.d$$

Comme exemple, on souhaite retrouver les membres *mId* tels que la propriété suivante soit satisfaite : $nbLoans(mId) = nbLoans(borrower(b0))$. On applique dans un premier temps le patron de base suivant :

$$f(\overrightarrow{k^c}, \overrightarrow{k^1}, \overrightarrow{p^1}) \ op \ g(\overrightarrow{k^c}, \overrightarrow{k^2}, \overrightarrow{p^2})$$

avec $f = nbLoans$, $g = nbLoans$, $\overrightarrow{k^1} = mId$ et $\overrightarrow{k^c}$, $\overrightarrow{k^2}$, $\overrightarrow{p^1}$ et $\overrightarrow{p^2}$ qui sont vides. En appliquant la technique de réutilisation décrite dans cette section, avec pour valeurs $g = nbLoans$ et $d = borrower$, on obtient le **SELECT** suivant :

SELECT $M_1.memberKey$
FROM $member\ M_1, member\ M_2, loan\ L$
WHERE $M_2.memberKey = L.borrower$
  AND $L.bookKey = \#b0$
  AND $M_1.nbLoans = M_2.nbLoans$;

10.4 Conjonction des prédicats

Nous rappelons que la forme générale des prédicats **if** dans les termes conditionnels est une conjonction de prédicats, $P = P_1 \wedge P_2 \wedge ... \wedge P_n$, où chaque P_i est une instance I de l'un des patrons de prédicat ou bien sa négation, notée par

$\neg I$. Un énoncé SQL est généré pour l'ensemble de la conjonction de la manière suivante.

Soit $\overrightarrow{k_i}$ les attributs clé à déterminer par le **SELECT** du prédicat P_i. Soit $\overrightarrow{k_P}$ les attributs clé à déterminer par le prédicat P. Chaque $\overrightarrow{k_i}$ est un sous-ensemble de $\overrightarrow{k_P}$. Par conséquent, $\overrightarrow{k_P}$ est défini par la formule suivante : $\overrightarrow{k_P} = \bigcup \overrightarrow{k_i}$. L'énoncé **SELECT** pour $\overrightarrow{k_P}$ peut être obtenu en faisant l'intersection des énoncés **SELECT** de chaque $\overrightarrow{k_i}$, afin de satisfaire la conjonction. Cependant, chaque **SELECT** doit avoir la même projection, $\overrightarrow{k_P}$, pour que l'intersection porte bien sur toutes les valeurs possibles. Lorsque $\overrightarrow{k_i} = \overrightarrow{k_P}$, le **SELECT** a déjà la projection appropriée. Par contre, si $\overrightarrow{k_i} \subset \overrightarrow{k_P}$, alors les tables des autres énoncés **SELECT** doivent aussi figurer dans la clause **FROM**, afin de prendre en compte toutes les valeurs possibles des attributs de $\overrightarrow{k_P} - \overrightarrow{k_i}$ dans le produit cartésien.

Ensuite, les prédicats positifs (c'est-à-dire de la forme I) sont distingués des négatifs (de la forme $\neg I$). Nous utilisons les patrons **SELECT** définis dans les sections 10.2 et 10.3 pour générer un énoncé **SELECT** pour chaque prédicat positif I. Dans le cas des prédicats négatifs $\neg I$, on génère l'énoncé **SELECT** qui correspond à I.

La dernière étape consiste à définir l'intersection des différents **SELECT**. Soient S_P l'ensemble des prédicats positifs et S_N l'ensemble des prédicats négatifs. Dans la section 10.1, nous avons spécifié qu'il existait au moins un prédicat positif P_i dans la conjonction. Par conséquent, il est possible de choisir un élément de S_P que nous notons par P_a. Pour chaque prédicat P_j dans S_N, une condition de la forme :

$$\overrightarrow{k_P} \textbf{ NOT IN } (/* \text{ énoncé } \textbf{SELECT} \text{ associé à } P_j \ */)$$

est ajoutée dans la clause **WHERE** de l'énoncé **SELECT** associé à P_a.

Considérons par exemple le prédicat suivant :

$$\begin{array}{ll} \text{(p1)} & mId \in memberKey() \wedge \\ \text{(p2)} & position(b1, mId) > position(b1, m0) \wedge \\ \text{(p3)} & \neg(nbLoans(mId) = nbLoans(m2)) \end{array}$$

Pour calculer l'énoncé SQL qui correspond à cette conjonction, on génère tout d'abord le **SELECT** suivant pour le prédicat négatif (p3), en appliquant le patron $f(\overrightarrow{k^c}, \overrightarrow{k^1}, \overrightarrow{p^1})\ op\ g(\overrightarrow{k^c}, \overrightarrow{k^2}, \overrightarrow{p^2})$:

SELECT $M_1.memberKey$
FROM $member\ M_1, member\ M_2$
WHERE $M_2.memberKey = \#m2$
  AND $M_1.nbLoans = M_2.nbLoans$;

Ensuite, le **SELECT** ci-dessus est ajouté dans la clause **WHERE** de l'énoncé **SELECT** associé au prédicat (p1), que nous avons décrit comme exemple du patron $k \in eKey()$ (voir section 10.2.7). Enfin, une intersection est définie entre le résultat précédent et le **SELECT** généré pour (p2), qui a été présenté dans la section 10.2.5. On obtient alors :

SELECT $member.memberKey$
FROM $member$

WHERE *member.memberKey* **NOT IN**
(**SELECT** M_1.*memberKey*
FROM *member* M_1, *member* M_2
WHERE M_2.*memberKey* = #*m*2
AND M_1.*nbLoans* = M_2.*nbLoans*)
INTERSECT
SELECT *R*1.*memberKey*
FROM *reservation* *R*1, *reservation* *R*2
WHERE *R*1.*bookKey* = #*b*1
AND *R*2.*bookKey* = #*b*1
AND *R*2.*memberKey* = #*m*0
AND *R*1.*position* > *R*2.*position* ;

Cet exemple illustre l'un des principaux intérêts de notre bibliothèque de patrons ; la définition d'une telle requête peut être source d'erreurs, alors que nos algorithmes permettent de la générer automatiquement à partir de prédicats décrits dans un langage formel.

10.5 Analyse et discussion

Dans ce chapitre, nous avons présenté un ensemble de patrons et de techniques de réutilisation dans le but de générer des requêtes SQL qui correspondent aux prédicats décrits dans les clauses **if** des termes conditionnels. Comme indiqué dans la section 10.3, chaque patron atomique peut être étendu afin de composer plusieurs fonctions ou définitions d'attributs. Les conjonctions de prédicats ont également été considérées dans la section 10.4. Comme tout prédicat peut être réécrit sous une forme normale disjonctive, il est possible de lui appliquer les patrons décrits dans ce chapitre.

Dans [GFLB05], des patrons supplémentaires ont été définis pour les prédicats de base qui impliquent des ensembles. En effet, un ensemble de valeurs peut parfois être retourné par une définition d'attribut. Dans cette thèse, nous avons présenté uniquement les cas des définitions de clé et des rôles, mais il est également possible de considérer l'image d'une définition d'attribut lorsqu'un ensemble de valeurs est pris en compte pour un ou plusieurs de ses paramètres. Ce cas est illustré par le patron $k \in f[\overrightarrow{k}, \overrightarrow{p}]$, où les symboles "[]" représentent l'image de la fonction f. Les images des définitions d'attributs peuvent aussi être comparées avec les patrons $f[\overrightarrow{k^c}, \overrightarrow{k^1}, \overrightarrow{p^1}] \subseteq f[\overrightarrow{k^c}, \overrightarrow{k^2}, \overrightarrow{p^2}]$ et $f[\overrightarrow{k^c}, \overrightarrow{k^1}, \overrightarrow{p^1}] = f[\overrightarrow{k^c}, \overrightarrow{k^2}, \overrightarrow{p^2}]$. Ces derniers sont implémentés en SQL par une double négation de **SELECT** imbriqués.

Nous avons déjà souligné la simplicité des termes conditionnels, comparée à leur équivalent en SQL. Elle est principalement due au style fonctionnel des définitions d'attributs EB^3. Les compositions de fonctions s'expriment aisément dans le langage et il n'est pas nécessaire de se soucier des problèmes de boucles ou d'itérations à ce niveau d'abstraction. Supposons par exemple que le membre $m0$ ait réservé le livre $b1$. Si le membre $m0$ décide plus tard d'annuler sa réservation (par l'événement Cancel$(b1, m0)$), alors l'attribut *position* de tous les membres, dans la liste de réservation du livre $b1$, dont la position est supérieure à celle de $m0$, doit décrémenter de 1. Cette condition est simplement spécifiée en EB^3 par : $position(b1, mId) > position(b1, m0)$, tandis que le **SELECT** correspondant

contient deux occurrences de la même table et retourne toutes les réservations dont l'attribut *position* doit être décrémenté. Nous pensons que cette simplicité d'expression et la synthèse automatique de transactions ont comme potentiel, une réduction du temps de développement et une diminution du nombre d'erreurs dans les programmes.

Chapitre 11

Mise en œuvre

> "Faites attention aux bogues dans le programme ci-dessous ; j'ai seulement prouvé qu'il était correct, je ne l'ai pas essayé."
>
> — Donald Knuth

Dans ce chapitre, nous discutons de l'application des algorithmes de synthèse de transactions et nous faisons la synthèse des contributions de cette partie de la thèse. Pour commencer, nous présentons dans la section 11.1 un schéma général de la correction des algorithmes présentés dans les chapitres 9 et 10. Dans la section 11.2, nous présentons l'outil qui supporte la synthèse de transactions. Enfin, la section 11.3 conclut ce chapitre avec une synthèse de nos contributions.

11.1 Correction de la synthèse de transactions

Dans cette section, un schéma général de la preuve de correction des algorithmes de synthèse de transactions est présenté. L'idée de la preuve est la suivante. Pour chaque action a de la spécification EB^3, une transaction T_a est générée grâce aux règles de traduction présentées dans les chapitres 9 et 10. Chaque événement valide de a est modélisé comme une transition de E vers E (notée par "$\xrightarrow{a}$"), où E est l'espace d'états du SI dans un modèle des définitions d'attributs EB^3. De manière analogue, la transaction T_a peut être considérée comme une transition de R vers R (notée par "$\xrightarrow{T_a}$"), où R est l'espace d'états du SI dans un modèle de BD relationnelles. Les modèles utilisés pour EB^3 et les BD relationnelles sont définis dans la section 11.1.1.

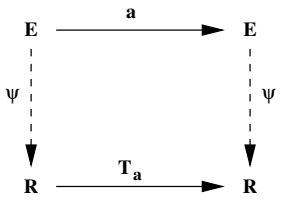

FIG. 11.1 – Correction de la synthèse de transactions

Pour prouver la correction de la traduction, un morphisme Ψ du modèle EB^3 vers le modèle des BD relationnelles doit être défini de sorte que le diagramme de la figure 11.1 soit commutatif. Formellement, pour chaque action a de la spécification EB^3, il faut prouver que, si un événement de a est valide, alors :

- **Validité :** une transition dans le modèle des BD relationnelles associée à une transaction T_a générée à partir de l'action a conduit à un état qui correspond à l'état du modèle EB^3 après avoir exécuté a :

$$\forall\, e \in E,\ \forall\, r, r' \in R,\ \Psi(e) = r \wedge r \xrightarrow{T_a} r'$$
$$\Rightarrow\ \exists\, e' \in E \bullet e \xrightarrow{a} e' \wedge \Psi(e') = r'$$

- **Complétude :** une transition associée à a dans le modèle EB^3 conduit à un état qui correspond à l'état du modèle des BD relationnelles après avoir exécuté T_a :

$$\forall\, e, e' \in E,\ \forall\, r \in R,\ \Psi(e) = r \wedge e \xrightarrow{a} e'$$
$$\Rightarrow\ \exists\, r' \in R \bullet r \xrightarrow{T_a} r' \wedge \Psi(e') = r'$$

La première implication est la plus importante, car elle permet d'assurer que chaque transaction générée avec les règles de traduction a effectivement le même comportement que l'action correspondante en EB^3. La complétude permet de prouver que, pour chaque action EB^3, une transaction avec le même comportement peut être générée.

Pour faire cette preuve, nous posons comme hypothèse que l'interprétation des expressions de processus EB^3 est correcte. En effet, la validité d'un événement dépend de l'interpréteur EB^{3}PAI. La correction des règles de simulation utilisées par EB^{3}PAI a été prouvée dans [Fra06]. Dans la suite, on considère que la transition associée à une action a est effectivement exécutée si et seulement si l'événement de a est considéré comme valide par l'interpréteur.

11.1.1 Modèles utilisés dans la preuve

Dans cette preuve, une spécification EB^3 est considérée comme statique. Autrement dit, elle ne peut pas être modifiée une fois que la BD a été générée. L'intégration de mises à jour dynamiques de la spécification EB^3 fait partie des perspectives de travail. Pour représenter les effets des actions, nous considérons pour le modèle EB^3 l'ensemble des valeurs des définitions de clé et des définitions d'attributs non clé. Soient N_k le nombre de définitions de clé et N_b le nombre de définitions d'attributs non clé. Soit t la trace courante du système. Un état e du modèle E a la forme suivante :

$$e = \{eKey_1(t), ..., eKey_{N_k}(t), \lambda\, \overrightarrow{k_1}.b_1(t, \overrightarrow{k_1}), ..., \lambda\, \overrightarrow{k_{N_b}}.b_{N_b}(t, \overrightarrow{k_{N_b}})\}$$

où $\lambda\, \overrightarrow{k_j}.b_j(t, \overrightarrow{k_j})$ est une λ-expression qui représente la fonction qui associe à chaque variable $\overrightarrow{v}$ de $\overrightarrow{k_j}$ la valeur d'attribut $b_j(t, \overrightarrow{v})$. Par l'utilisation de λ-expressions, toutes les valeurs possibles des définitions d'attributs non clé peuvent ainsi être prises en compte. Pour simplifier les notations, chaque λ-expression $\lambda\, \overrightarrow{k_j}.b_j(t, \overrightarrow{k_j})$ sera notée par $\widetilde{b_j}(t)$ dans la suite.

Pour les BD relationnelles, nous considérons le modèle relationnel classique. Formellement, un état r de BD relationnelle a la forme suivante :

$$r = \{rel_1, ..., rel_p\}$$

où chaque élément rel_j est l'extension d'une relation de la forme :

$$Rel_j(\underline{k_1, k_2, ..., k_{n_j}}, att_{n_j+1}, \ldots, att_{m_j})$$

Par définition, rel_j est un ensemble de m_j-tuples $rel_j = \{t_1, ..., t_{p_j}\}$, où chaque tuple t_l est une liste ordonnée : $t_l =< v_1^l, ..., v_{m_j}^l >$.

Comme indiqué dans la section 9.2.2, le schéma de BD relationnelles est automatiquement généré à partir de la spécification EB^3 en utilisant les algorithmes classiques de [EN04]. La bijection, notée par Φ, entre le diagramme ER et le schéma de BD relationnelles dépend de la forme et des définitions d'attributs de chaque type d'entité et association du système. Pour simplifier, on considère dans la suite que Φ est une bijection simple : une relation est générée pour chaque type d'entité et association. La définition de clé de chaque type d'entité et association est décomposée en plusieurs attributs clé simples qui forment ensemble la clé primaire de la relation correspondante. Les définitions d'attributs non clé associées à chaque type d'entité et association deviennent les attributs de la relation correspondante. Pour considérer des cas complexes avec des types d'entité faibles ou des associations $1 : N$, il suffit de définir une fonction supplémentaire notée *table* qui associe à chaque définition d'attribut, la relation à laquelle l'attribut correspondant appartient. La fonction *table* peut alors être utilisée par Φ pour définir l'intention de chaque relation.

Il faut maintenant définir un morphisme Ψ du modèle EB^3 vers le modèle de BD relationnelles. Les fonctions Φ et Ψ sont étroitement liées entre elles, car Φ permet de déterminer le schéma de BD relationnelles à partir de la spécification EB^3, alors que Ψ associe à chaque état du modèle EB^3, l'état correspondant dans le modèle relationnel. Comme l'ordre des éléments d'un ensemble n'a pas d'importance, chaque état e du modèle EB^3 peut être réécrit, pour les besoins de compréhension de la preuve, en un ensemble dont les définitions d'attributs sont groupées selon les types d'entité et associations auxquels elles appartiennent :

$$\begin{aligned} e = \{ & eKey_1(t), \widetilde{b}_1^1(t), ..., \widetilde{b}_{n_1}^1(t), \\ & ... \\ & eKey_{N_k}(t), \widetilde{b}_1^{N_k}(t), ..., \widetilde{b}_{n_{N_k}}^{N_k}(t)\} \end{aligned}$$

Il est alors possible de calculer $\Psi(e) = \{rel_1, ..., rel_p\}$ en définissant chaque rel_j comme suit. Le schéma Rel_j de la relation rel_j est déduite de Φ. Si la bijection Φ est simple, alors le schéma est défini par :

$$Rel_j(\underline{k_1, ..., k_L}, att_1, \ldots, att_{n_j})$$

tel que $\Phi(eKey_j) = (k_1, ..., k_L)$ et, pour chaque $1 \leq l \leq n_j$, $\Phi(b_l^j) = att_l$. Pour déterminer l'extension de Rel_j, le principe est illustré par la figure 11.2. L'ensemble des valeurs de clé est l'ensemble défini par $eKey_j(t)$. Ensuite, pour chaque tuple $< v_1, ..., v_L >$ appartenant à cet ensemble, les valeurs d'attributs sont fournies respectivement par $b_1^j(t, v_1, ..., v_L)$, ..., $b_{n_j}^j(t, v_1, ..., v_L)$.

11.1.2 Idée de la preuve

La preuve consiste en deux implications. Dans cette thèse, nous présentons uniquement la première implication, qui correspond à la validité.

Hypothèses. On suppose qu'un événement de l'action a a été accepté et que la transition associée à la transaction T_a a été exécutée dans le modèle relationnel. Pour considérer tous les états possibles e, r, r' tels que $r \xrightarrow{T_a} r'$ et $\Psi(e) = r$, il suffit de considérer tous les états possibles e du modèle EB^3, parce

Rel_j

k_1	...	k_L	att_1	...	att_{n_j}
	$\vdots$ $eKey_j(t)$ $\vdots$				
v_1	...	v_L	$b_1^j(t, v_1, ..., v_L)$	...	$b_{n_j}^j(t, v_1, ..., v_L)$
	$\vdots$				

FIG. 11.2 – Extension de la relation rel_j calculée avec Ψ

que r est déterminé à partir de e par Ψ et r' est calculé en exécutant T_a. Le but est de trouver un état e' du modèle EB^3 tel que $e \xrightarrow{a} e'$ et $\Psi(e') = r'$. Pour généraliser le résultat à tous les états possibles de e, la preuve est réalisée par induction structurelle sur la forme des définitions d'attributs. En effet, le cœur de nos algorithmes concerne l'analyse des clauses d'entrée. Par conséquent, un état e du modèle EB^3 dépend des clauses d'entrée des définitions d'attributs.

Cas initial. L'état initial correspond à l'évaluation de toutes les clauses d'entrée de la forme "$\perp : u$" dans l'ensemble des définitions d'attributs. Rappelons que le nombre de définitions d'attributs ne varie pas (voir section 11.1.1). La correction de l'initialisation est directe : la preuve dépend de la forme de u ; ainsi, il faut vérifier pour chaque terme de base et pour chaque constructeur de u que l'initialisation des tables de BD relationnelles est correcte vis-à-vis de l'initialisation spécifiée dans les définitions d'attributs EB^3. Par exemple, considérons le cas simple où une définition de clé *eKey* a une clause d'entrée $\perp : \varnothing$ et chaque définition d'attribut non clé b_j, $1 \leq j \leq n$, qui appartient à la même relation que *eKey*, a une clause d'entrée $\perp : \perp$. Alors, les tables créées par les algorithmes, qui sont vides à l'initialisation, sont conformes à la spécification EB^3.

Pas d'induction. Toute expression qui peut être générée pour la transaction T_a est composée d'un ensemble d'énoncés SQL qui sont déduits à partir des clauses d'entrée qui contiennent une occurrence de l'action a. En particulier, la transaction T_a dépend des nombres respectifs de clauses d'entrée de la forme $\mathsf{a}(\overrightarrow{p}) : u$ dans les définitions de clé et dans les définitions d'attributs non clé. Elle dépend également de la forme des expressions u associées. La preuve est donc réalisée par induction sur les nombres de clauses et sur la forme de u. Il existe principalement deux classes d'expressions pour u : les termes fonctionnels et les termes conditionnels.

Pour illustrer la preuve dans le cas d'un terme fonctionnel, on considère maintenant le cas le plus complexe des expressions de transactions. Quand une définition de clé *eKey* comprend une clause d'entrée de la forme $\mathsf{a}(\overrightarrow{p}) : eKey() \cup \{\overrightarrow{p}\}$, alors la transaction correspondante inclut un énoncé de la forme suivante :

UPDATE T **SET** ...

WHERE ... ;
IF SQL%NotFound
THEN
  INSERT INTO $T(\Phi(eKey), ...)$
  VALUES $(\overrightarrow{p}, ...)$;
END ;
COMMIT ;

où T est la table à laquelle $\Phi(eKey)$ appartient. Le contenu dépend des définitions d'attributs avec au moins une clause d'entrée sur a. Supposons qu'il existe n définitions d'attributs non clé $b_1, ..., b_n$ associées par Φ à la même table que $eKey$ et que chacune comprend une seul clause de la forme $\mathsf{a}(\overrightarrow{p})$: $\perp$. L'énoncé SQL obtenu par traduction est alors :

UPDATE T
  SET $(\Phi(b_1), ..., \Phi(b_n)) =$ (**NULL**,...,**NULL**)
WHERE $\Phi(eKey) = \overrightarrow{p}$;
IF SQL%NotFound
THEN
  INSERT INTO $T(\Phi(eKey), \Phi(b_1), ..., \Phi(b_n))$
  VALUES ($\overrightarrow{p}$,**NULL**,...,**NULL**) ;
END ;
COMMIT ;

Soit e un état arbitraire du modèle EB^3 :

$$e = \{eKey_1(t), \widetilde{b}_1^1(t), ..., \widetilde{b}_{n_1}^1(t),$$
$$...$$
$$eKey_{N_k}(t), \widetilde{b}_1^{N_k}(t), ..., \widetilde{b}_{n_{N_k}}^{N_k}(t)\}$$

L'état r peut être calculé par Ψ : $r = \Psi(e)$. En exécutant l'énoncé SQL ci-dessus, r' est déterminé à partir de r en modifiant l'extension de la relation T. Il y a deux cas possibles. Si un tuple identifié par la clé $\overrightarrow{p}$ existe déjà dans la BD, alors l'énoncé **UPDATE** est exécuté ; sinon, c'est l'énoncé **INSERT**. Dans les deux cas, les effets sont les mêmes : la relation T contient le tuple $<\overrightarrow{p}$,**NULL**,...,**NULL**>.

Quand a est exécutée, alors e' est obtenu à partir de e, en remplaçant $eKey(t)$ par $eKey(t) \cup \{\overrightarrow{p}\}$, et, pour chaque $1 \leq j \leq n$, $b_j(t, \overrightarrow{p})$ est évalué à $\perp$ dans les λ-expressions correspondantes. L'état $\Psi(e')$ est défini comme r, à l'exception de la relation T, où un nouveau tuple est rajouté : $<\overrightarrow{p}$,**NULL**,...,**NULL**>. D'où la preuve de l'implication pour cet énoncé SQL. Plus généralement, il est possible de généraliser le résultat à tout nombre de clauses d'entrée dans la définition en effectuant une preuve par récurrence.

Un dernier aspect important de la preuve concerne les termes conditionnels. Il faut alors prouver, en outre, que les énoncés **SELECT** correspondent bien aux prédicats. Nous considérons l'algèbre relationnelle pour représenter les résultats des requêtes SQL (voir [EN04] pour plus de détails sur cette algèbre couramment utilisée pour les BD relationnelles). Pour chaque patron de prédicat défini dans le chapitre 10, il faut prouver que le **SELECT** généré caractérise et met à jour les mêmes enregistrements que les termes conditionnels dans le modèle EB^3. Par exemple, supposons qu'un terme conditionnel soit de la forme **if** $k = g(\overrightarrow{p})$ **then** *FctTerm* **else** *Expression* **end**. L'énoncé **SELECT** est alors :

SELECT *projection*(g, g)
FROM *table*(g)
WHERE *restriction*$(g, \overrightarrow{p})$

où les expressions *projection*, *table* et *restriction* sont définies dans la section 10.2. Dans l'algèbre relationnelle, ce **SELECT** est représenté par :

$$\pi_g(\sigma_{restriction(g,\overrightarrow{p})}(table(g))$$

où $\sigma_{cond}(rel)$ désigne la sélection dans *rel* de l'ensemble des tuples qui satisfont la propriété *cond* et $\pi_{attr}(select)$ la projection des résultats de *select* sur les attributs *attr*. On peut alors vérifier que cette expression représente les mêmes valeurs de clé que le prédicat correspondant dans le modèle EB^3.

Conclusion. Nous avons prouvé à la main que les transactions générées sont conformes aux effets décrits dans les définitions d'attributs EB^3. La preuve n'est pas difficile, mais comporte de nombreux cas à considérer à cause des différents niveaux d'induction sur les définitions d'attributs, sur le nombre et sur la forme des clauses d'entrée et sur la forme des expressions u associées. L'utilisation d'un outil de preuve pour rassembler et valider les schémas de preuve décrits dans cette section fait partie des perspectives de travail.

11.2 Outil

Les algorithmes présentés dans les chapitres 9 et 10 sont implémentées dans EB^{3}TG, un outil réalisé en collaboration avec Panawé Batanado [Bat05], lors de son stage de maîtrise (équivalent au master M2 en France) à l'Université de Sherbrooke. L'outil permet d'une part de générer un schéma de BD à partir d'une représentation XML du diagramme ER de la spécification EB^3. Il permet d'autre part de vérifier syntaxiquement les définitions d'attributs EB^3 et de générer un ensemble de classes Java qui implémentent les transactions qui correspondent aux actions EB^3.

11.2.1 Description de l'outil EB^{3}TG

L'outil a été implémenté en Java. Le code source comprend 50 classes et 625 méthodes, pour un total de 20000 lignes de code. Les fonctionnalités de l'outil sont présentées dans la figure 11.3. Les fichiers correspondant à l'exemple de la bibliothèque sont fournis dans l'annexe C.

La figure 11.4 représente l'outil EB^{3}TG, qui a été développé sous l'environnement Eclipse. Pour commencer, EB^{3}TG réalise une vérification de la représentation XML du diagramme ER par rapport à la DTD (*Document Type Definition*) du modèle ER ; des messages d'erreur sont retournés en cas de problèmes. L'outil génère ensuite un schéma de BD relationnelles à partir de la représentation XML. Les énoncés SQL sont générés en fonction du SGBD choisi par l'utilisateur. La version courante d'EB^{3}TG supporte Oracle, PostgreSQL et MySQL. Le schéma de BD généré pour l'exemple de la bibliothèque est présenté dans la figure 11.5. Le SGBD choisi dans ce cas est PostgreSQL.

L'outil EB^{3}TG vérifie ensuite que les définitions d'attributs sont cohérentes par rapport au diagramme ER. En cas d'erreur, l'outil génère un message. Enfin,

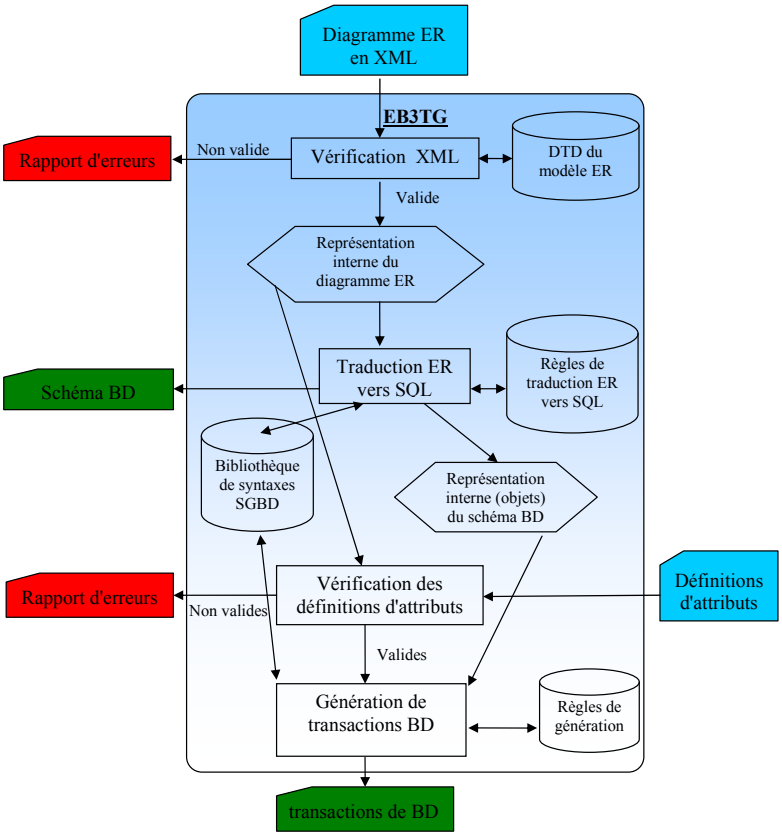

FIG. 11.3 – Architecture fonctionnelle de l'outil EB^{3}TG

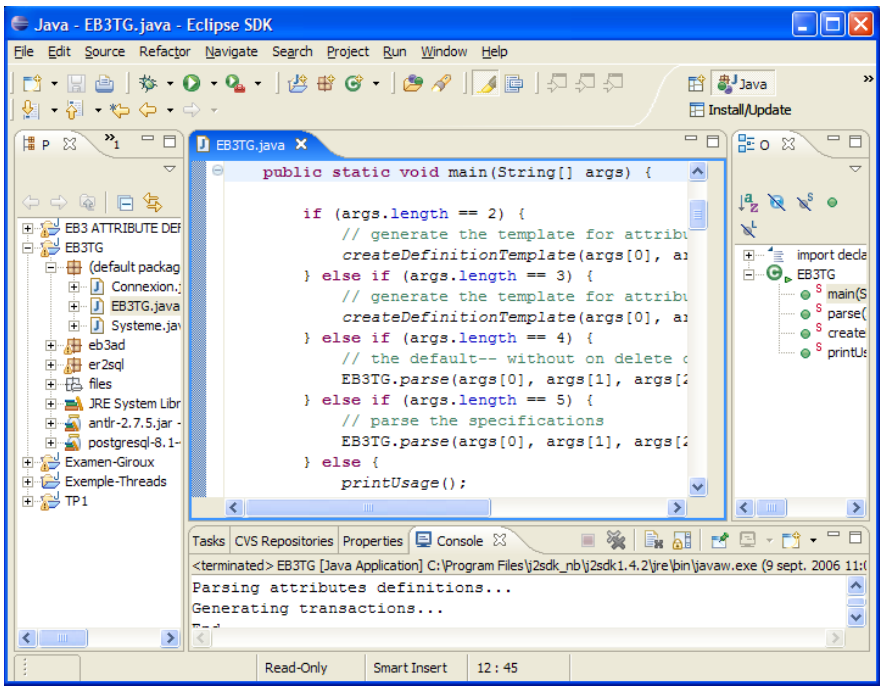

FIG. 11.4 – Outil EB^{3}TG

EB^{3}TG génère les programmes Java qui exécutent des transactions de BD relationnelles qui correspondent aux définitions d'attributs, en fonction du schéma de BD généré. La figure 11.6 montre le résultat d'une exécution de l'outil pour la bibliothèque.

11.2.2 Forme des transactions générées par EB^{3}TG

L'action Transfer(bId, mId) a pour effet de transférer le prêt du livre bId au membre mId. La méthode Java générée par EB^{3}TG pour cette action est décrite dans la figure 11.7.

La technologie JDBC ("*Java DB Connectivity*") offre une interface de programmation permettant de développer des applications en Java capables de se connecter à n'importe quelle BD. Deux classes permettent d'exécuter des requêtes SQL, puis d'en extraire les résultats. Il s'agit des classes *PrepareStatement* et *Statement*. La première a l'avantage d'être plus rapide à l'exécution, parce que les requêtes sont compilées une seule fois et stockées dans un objet. Les méthodes de la forme *setXXX* (où *XXX* représente un type, *String* par exemple) permettent ensuite d'affecter les paramètres de la requête, puis de l'exécuter sans avoir à la recompiler. L'atout principal de la seconde classe est d'être implémentée par la plupart des SGBD. Cette dernière a été utilisée dans EB^{3}TG. La méthode *createStatement*() de l'exemple de la figure 11.7 crée un objet de type *Statement*, dont les méthodes *executeUpdate*(*uneRequete*) et *executeQuery*(*uneAutreRequete*) permettent d'exécuter respectivement des requêtes SQL de mise à jour et d'interrogation.

```
CREATE TABLE book (
bookKey numeric(5,2),
title varchar(20),
CONSTRAINT PKbook PRIMARY KEY(bookKey)
);

CREATE TABLE member (
memberKey numeric(5),
nbLoans numeric(5) NOT NULL,
loanDuration numeric(3) NOT NULL,
CONSTRAINT PKmember PRIMARY KEY(memberKey)
);

CREATE TABLE loan (
borrower numeric(5),
bookKey numeric(5,2),
dueDate date,
CONSTRAINT PKloan PRIMARY KEY(bookKey)
);

CREATE TABLE reservation (
bookKey numeric(5,2),
memberKey numeric(5),
position numeric(5),
CONSTRAINT PKreservation PRIMARY KEY(bookKey,memberKey)
);

ALTER TABLE loan ADD CONSTRAINT FKloan_member FOREIGN KEY (borrower)
REFERENCES member (memberKey) INITIALLY DEFERRED;

ALTER TABLE loan ADD CONSTRAINT FKloan_book FOREIGN KEY (bookKey)
REFERENCES book (bookKey) INITIALLY DEFERRED;

ALTER TABLE reservation ADD CONSTRAINT FKreservation_book FOREIGN KEY (bookKey)
REFERENCES book (bookKey) INITIALLY DEFERRED;

ALTER TABLE reservation ADD CONSTRAINT FKreservation_member FOREIGN KEY (memberKey)
REFERENCES member (memberKey) INITIALLY DEFERRED;
```

FIG. 11.5 – Schéma de BD généré pour la bibliothèque

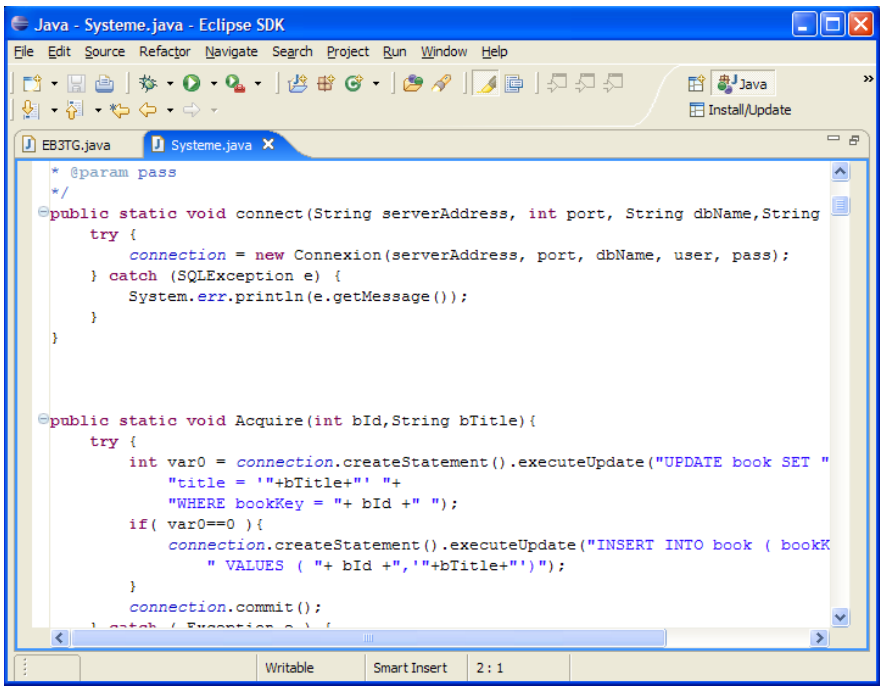

FIG. 11.6 – Exemple d'application d'EB^{3}TG sur la bibliothèque

Les types de données Java équivalents aux types SQL sont utilisés pour stocker les variables temporaires, et la classe *ResultSet*, pour représenter la notion de table temporaire utilisée dans la section 9.2.4.4. Les objets de type *ResultSet* ne sont pas sensibles à des mises à jour ultérieures ; autrement dit, les mises à jour effectuées dans la BD au cours d'une transaction n'affectent pas les résultats des requêtes **SELECT** stockés dans ces objets. Dans l'exemple de Transfer, l'analyse de l'arbre de décision pour l'attribut *nbLoans* (voir figure 6.3) a permis d'identifier deux valeurs possibles pour la mise à jour de cet attribut : le membre qui bénéficie du transfert et celui qui fait le transfert. Ces deux cas sont représentés par les variables *rset*0 et *rset*1 de la figure 11.7.

11.2.3 Conclusion

L'outil présenté permet de générer automatiquement des programmes en Java, qui exécutent des transactions de BD relationnelles, à partir de définitions d'attributs EB^3. La motivation du projet APIS, et de l'outil EB^{3}TG en particulier, est de libérer le programmeur des détails d'implémentation des transactions, pour qu'il se concentre plutôt sur les phases d'analyse et de spécification. L'objectif est maintenant d'intégrer l'outil EB^{3}TG aux autres composantes d'APIS. En particulier, le couplage de l'outil avec l'interpréteur EB^{3}PAI permettra de simplifier les énoncés de type mise à jour ou insertion qu'il n'est pas possible de distinguer en considérant uniquement les définitions d'attributs EB^3.

```
public static void Transfer(int bId,int mId,String typeOfLoan){
  try {
    //create a temporary table eb3Tempmember
    connection.createStatement().executeUpdate("CREATE TABLE eb3Tempmember ( "+
      "memberKey  numeric(5))");
    //Insert parameters in the temporary table eb3Tempmember
    connection.createStatement().executeUpdate("INSERT INTO  eb3Tempmember (memberKey)
      values("+mId+")");
    //end temporary table eb3Tempmember

    ResultSet rset0 = connection.createStatement().executeQuery("SELECT C.memberKey,
      A.nbLoans+1 "+
      "FROM eb3Tempmember C,member A "+
      "WHERE C.memberKey =  "+ mId+ " "+
      "AND A.memberKey = C.memberKey ");
    ResultSet rset1 = connection.createStatement().executeQuery("SELECT G.borrower,
      E.nbLoans-1 "+
      "FROM loan G,member E "+
      "WHERE G.bookKey = "+ bId+ " "+
      "AND G.borrower NOT IN ( "+
      "SELECT C.memberKey "+
      "FROM eb3Tempmember C "+
      "WHERE C.memberKey =  "+ mId+ " ) "+
      "AND E.memberKey = G.borrower ");
    ResultSet rset2 = connection.createStatement().executeQuery("SELECT D.loanDuration "+
      "FROM member D "+
      "WHERE D.memberKey = "+ mId+ " ");
    String var0 = ((rset2.next())?rset2.getInt(1)+"":"null");
    connection.createStatement().executeUpdate("UPDATE loan SET "+
      "borrower = "+mId+" "+
      "WHERE bookKey = "+ bId +" ");
    if ( typeOfLoan=="Permanent" ) {
      connection.createStatement().executeUpdate("UPDATE loan SET "+
      "dueDate = CURRENT_DATE+365 "+
      "WHERE bookKey = "+ bId +" ");
    } else if( typeOfLoan=="Classic" ) {
      connection.createStatement().executeUpdate("UPDATE loan SET "+
      "dueDate = CURRENT_DATE+"+var0+" "+
      "WHERE bookKey = "+ bId +" ");
    }
    while(rset0.next()) {
      connection.createStatement().executeUpdate("UPDATE member SET nbLoans =
        "+rset0.getDouble(2)+ " "+
        "WHERE memberKey = "+ rset0.getDouble(1)+ " ");
    }
    while(rset1.next()) {
      connection.createStatement().executeUpdate("UPDATE member SET nbLoans =
        "+rset1.getDouble(2)+ " "+
        "WHERE memberKey = "+ rset1.getDouble(1)+ " ");
    }
    connection.createStatement().executeUpdate("DROP TABLE eb3Tempmember");
    connection.commit();
  } catch ( Exception e ) {
    try{
      connection.createStatement().executeUpdate("DROP TABLE eb3Tempmember");
      connection.rollback();
    } catch (SQLException s){
       System.err.println(s.getMessage());
    }
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}
```

FIG. 11.7 – Méthode Java générée pour Transfer

11.3 Synthèse de l'approche

Les contributions présentées dans cette partie de la thèse ont deux objectifs distincts et complémentaires. D'un côté, elles permettent d'implémenter les transactions décrites avec la méthode EB^4 (voir chapitre 8). Les algorithmes de synthèse de transactions offrent ainsi une alternative entièrement automatisée au raffinement semi-automatique de la spécification B. D'un autre côté, l'outil EB^{3}TG qui supporte les algorithmes présentés dans les chapitres 9 et 10 complète l'environnement APIS.

11.3.1 Autres travaux

Il existe plusieurs travaux concernant la synthèse d'implémentations relationnelles. Dans la plupart des cas, des techniques de raffinement sont utilisées pour implémenter des spécifications basées sur le paradigme des transitions d'états [Edm95, Mam02]. Cependant, le processus de raffinement n'est pas entièrement automatique. De plus, ces techniques demandent souvent une grande expertise en mathématiques pour prouver chaque étape de raffinement. D'autre part, il existe des travaux sur la synthèse de systèmes par interprétation ou par simulation [GS90, LN99], mais ces outils sont généralement inefficaces pour des SI, qui peuvent impliquer des millions d'entités et de liens d'associations.

Concernant la génération des énoncés **SELECT** à partir des prédicats des termes conditionnels, les travaux les plus proches sont ceux de Hohenstein, avec le langage SQL/EER [HE92]. Ce dernier est un langage formel de requêtes sur un modèle ER étendu appelé EER. Un outil a été implémenté pour traduire automatiquement des requête SQL/EER en des requêtes SQL [Hoh89]. Ses règles de traduction ont été formellement définies. Puisque le langage est basé sur le modèle EER, la forme des requêtes SQL/EER est proche de celle des énoncés SQL. En EB^3, le langage de définition des attributs n'est pas un langage de requêtes, il est basé sur les fonctions récursives qui représentent chacune un attribut. Les fonctions peuvent être composées entre elles pour exprimer aisément des requêtes complexes.

Les outils de traduction de type objet-relationnel, comme Hibernate [JBo06], permettent également de générer une représentation des BD relationnelles, mais ils concernent les BD orientées objet. Ces outils assurent la persistance des objets d'un langage de programmation orienté objet en se servant d'une BD relationnelle. Le langage de requêtes supporté par ce type d'outils est plus proche de SQL que des définitions d'attributs EB^3.

11.3.2 Contributions et perspectives

Notre objectif est de générer des transactions qui correspondent aux spécifications EB^3 et B du modèle développé avec la méthode EB^4. Comme il existe déjà des techniques de raffinement en B pour le domaine des SI, nous nous sommes concentré dans cette thèse sur la synthèse automatique des transactions à partir de la spécification EB^3. Pour définir cette traduction, nous avons travaillé sur les éléments suivants :

1. la définition d'un algorithme de synthèse vers les BD relationnelles à partir du diagramme ER et des définitions d'attributs EB^3 (chapitre 9). L'algo-

rithme a été testé sur le cas d'étude de la bibliothèque, et nous avons prouvé à la main sa correction (section 11.1).

2. la définition de patrons **SELECT** pour générer automatiquement des requêtes SQL à partir des prédicats des termes conditionnels dans les définitions d'attributs EB^3 (chapitre 10),
3. l'implémentation d'un outil de support de ces algorithmes de synthèse (chapitre 11, travail réalisé en collaboration avec Panawé Batanado).

Ces contributions ont fait l'objet de plusieurs articles publiés ou en cours de soumission. L'algorithme de synthèse des transactions a été présenté à la conférence internationale SEFM 2005 [GFL05a]. L'outil EB^{3}TG a été présenté aux conférences AFADL 2006 [GBFL06b] et ICEIS 2006 [GBFL06a]. Les patrons **SELECT** et la preuve de correction font l'objet d'un article en cours de soumission à la revue "Software and Systems Modeling".

Avec l'implémentation d'EB^{3}TG, les principales composantes d'APIS sont désormais en place. Les différents outils doivent maintenant être reliés entre eux. En particulier, l'interpréteur a besoin d'interroger la BD pour évaluer les gardes des actions dans les expressions de processus. De plus, le couplage d'EB^{3}TG avec EB^{3}PAI permettra d'optimiser certains programmes, comme discuté dans la section 11.2.3.

Conclusion générale

Conclusion

> "J'ai toujours souhaité que mon ordinateur soit aussi simple à utiliser qu'un téléphone. Mon souhait est aujourd'hui devenu réalité. Je ne sais plus comment utiliser mon téléphone."
>
> — Bjarne Stroustrup

L'objectif de cette thèse est de profiter des avantages de deux formes de modélisation complémentaires pour représenter les SI qui sont basés sur des SGBD. En particulier, nous avons choisi d'utiliser uniquement des notations et des techniques formelles pour les concevoir, contrairement aux méthodes actuelles qui sont au mieux semi-formelles. La spécification formelle a l'avantage de permettre au concepteur de constater une erreur au plus tôt, notamment grâce aux vérifications mathématiques sur le modèle. Notre motivation a été de coupler deux langages formels, dans le but de développer des SI en considérant, dès le travail d'analyse, les problèmes liés à la modélisation des propriétés dynamiques.

Dans l'approche proposée, deux langages existants ont été réutilisés. D'une part, EB^3 est un langage formel basé sur les traces d'événements, qui a été défini pour la spécification des SI. D'autre part, B est un langage formel basé sur les états, dans lequel les propriétés statiques des SI s'expriment naturellement. Cette thèse a permis la définition d'EB^4, une méthode qui bénéficie à la fois des avantages des langages EB^3 et B pour modéliser les propriétés statiques et dynamiques des SI. Le principe de l'approche consiste à considérer deux descriptions dans des langages de styles différents du même système afin, d'une part, de prendre en compte le maximum d'exigences et, d'autre part, de détecter les éventuelles erreurs dans le modèle.

Synthèse des contributions

La figure 12.1 est un résumé de l'approche EB^4. Dans cette méthode, le SI est tout d'abord spécifié en EB^3. Ensuite, la spécification est raffinée en B. Pour faciliter cette tâche, le modèle de données décrit dans le raffinement est généré par traduction à partir des définitions d'attributs EB^3. De plus, les opérations du raffinement peuvent être complétées grâce à des patrons de raffinement qui ont été identifiés et prouvés pour des expressions de processus typiques en EB^3. Le modèle B permet de spécifier et de vérifier des contraintes d'intégrité statiques qui ne sont pas exprimées dans la spécification initiale en EB^3. Ce va-et-vient préconisé dans la méthode EB^4 entre le modèle statique représenté en B et le modèle dynamique représenté en EB^3 permet de compléter par étapes successives

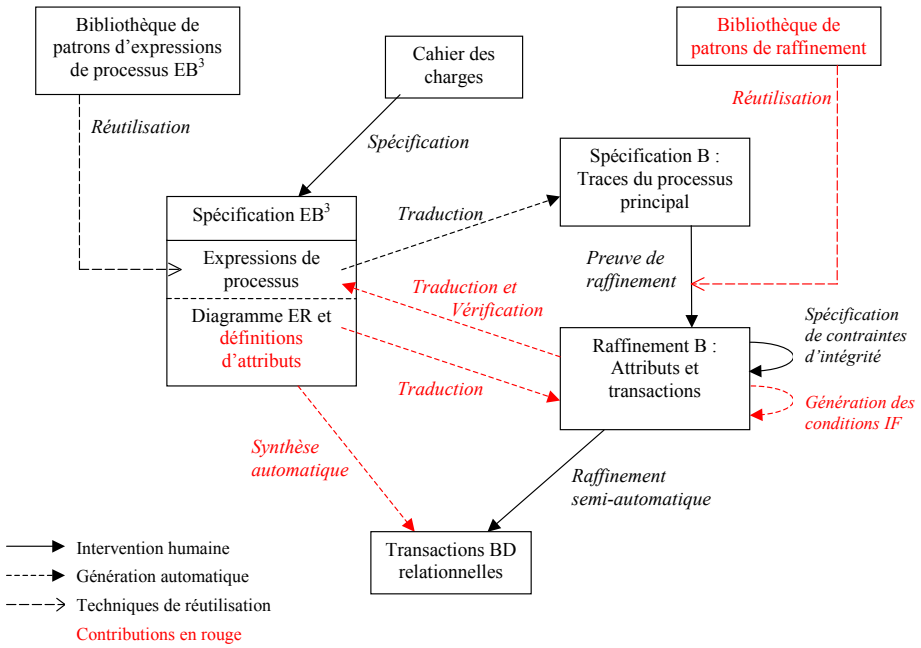

FIG. 12.1 – Vue globale de la méthode EB^4

la modélisation du SI. Lorsque la spécification du système est définie et validée, deux pistes sont alors possibles pour implémenter les transactions. D'un côté, on peut utiliser les techniques classiques de raffinement en B et d'un autre côté, des algorithmes de synthèse ont été proposés dans cette thèse pour générer automatiquement les transactions à partir de la spécification EB^3.

Plusieurs étapes de la méthode EB^4 sont automatisables, afin de faciliter le travail du concepteur et de minimiser ses efforts sur les phases qui ne sont pas critiques. Un algorithme de traduction a été défini afin de générer une représentation en B du modèle de données décrit en EB^3. Des règles de calcul ont été proposées afin de compléter les spécifications B en fonction des propriétés statiques à vérifier sur le système. Des patrons de raffinement ont été identifiés et catalogués afin de simplifier la preuve de raffinement utilisée dans notre méthode pour prouver les propriétés dynamiques décrites en EB^3. Pour préserver la cohérence du modèle, des règles de traduction ont été définies pour identifier les gardes dans les expressions de processus EB^3 à partir des préconditions des opérations B. Enfin, un algorithme de traduction a été proposé et implémenté pour générer automatiquement des transactions de BD relationnelles qui correspondent aux définitions des attributs en EB^3.

L'approche proposée répond en grande partie à nos objectifs initiaux. EB^4 est une méthode formelle, multi-paradigme et dédiée aux SI. Elle reprend l'idée de base, qui consiste à écrire deux fois la spécification d'un système, mais avec des langages de natures différentes, dans le but de prendre en compte le maximum d'exigences. Enfin, EB^4 s'intègre aussi au projet APIS, qui a pour objectif de générer automatiquement des SI à partir de spécifications EB^3. Elle fournit d'une part des techniques de vérification de contraintes d'intégrités statiques grâce à

la spécification B du système. D'autre part, l'outil EB^{3}TG qui implémente les algorithmes de synthèse de transactions complète les composantes de l'environnement d'applications APIS.

Dans [Hoa03], Hoare a soumis l'idée de lancer un grand défi visant à définir dans les trente prochaines années des techniques de synthèse et de vérification qui permettraient de créer un "compilateur de vérification", c'est-à-dire un outil vérifiant la correction d'un programme lors de sa compilation. L'objectif serait d'obtenir un programme certifié directement à partir d'une spécification abstraite. Un tel défi ne nous semble possible que si la spécification est restreinte à un domaine très particulier. Dans notre cas, nous avons pu proposer dans la méthode EB^4 des patrons de raffinement et des techniques de synthèse, car la spécification est dédiée aux SI. Il est alors possible d'obtenir automatiquement des programmes à partir de la spécification. L'approche EB^4 offre aussi l'option de ne pas appliquer la synthèse automatique, et la spécification peut alors être raffinée avec les techniques standard de B. Ainsi, à l'image de certains outils semi-automatiques, EB^4 permet de choisir les étapes qui sont automatisées ou pas. Si un concepteur souhaite obtenir rapidement une implémentation, alors il pourra la générer automatiquement. Si, au contraire, il préfère maîtriser les choix d'implémentation ou effectuer le raffinement pas à pas, la spécification B sera alors utilisée.

Perspectives

Plusieurs étapes de l'approche peuvent être améliorées. Tout d'abord, le langage de définition des attributs (voir section 4.2) peut être étendu pour considérer des opérateurs supplémentaires. Par exemple, certains attributs du SI peuvent être calculés à partir d'autres attributs, comme le pourcentage de commandes livrées ou bien le rapport entre le salaire d'un employé et le salaire le plus élevé. Dans ce cas, il serait inutile de spécifier en EB^3 une nouvelle définition d'attribut sous la forme d'une fonction récursive. Concernant la génération des prédicats **if** des termes conditionnels (voir section 7.2.2), les règles de réécriture et de simplification devront alors être complétées pour les nouveaux opérateurs du langage de définition des attributs.

L'étape la plus critique de la méthode EB^4 concerne la preuve de raffinement entre le modèle B de la spécification EB^3 et la spécification B obtenue par traduction (voir section 7.3). La bibliothèque de patrons de raffinement mérite d'être complétée et automatisée. La formalisation du processus d'instanciation est une autre piste à explorer afin de garantir la bonne application de ces patrons.

Les programmes Java/SQL générés à partir des définitions d'attributs EB^3 introduisent quelques redondances (voir chapitre 11), car toutes les valeurs d'attributs concernées par un programme sont systématiquement stockées au début de la transaction afin d'éviter l'utilisation de valeurs qui ne sont plus valables. Une optimisation des programmes est possible en analysant les dépendances entre les énoncés SQL. D'autre part, les énoncés **SELECT** correspondant aux prédicats des termes conditionnels (chapitre 10) peuvent également être simplifiés par une analyse des requêtes générées. Enfin, l'intégration de l'outil EB^{3}TG dans l'environnement APIS permettra de supprimer des instructions inutiles dans les transactions de type producteur d'entité ou de lien d'association, grâce à l'interprétation des expressions de processus EB^3.

Une perspective à plus long terme concerne la vérification de contraintes dynamiques. Bien que des propriétés dynamiques comme les propriétés d'ordonnancement soient prises en compte avec EB^4, les contraintes d'intégrité dynamiques nécessitent l'utilisation d'une logique temporelle pour les exprimer. Le fait que le salaire d'un employé ne peut qu'augmenter et la propriété qu'un membre de la bibliothèque empruntera un jour un livre sont des exemples de contraintes dynamiques. Il faut définir des techniques de *model-checking* en EB^3 pour les prendre en compte. Dans le cas de CSP, il existe un outil appelé FDR [For97] qui supporte ce type de vérification. Comme EB^3 est un langage proche de CSP, il sera possible de s'en inspirer, mais le nombre potentiellement élévé d'entités et de liens d'associations gérés par les SI peut poser problème dans le cas d'EB^3.

La spécification en EB^3 de grands systèmes est un autre point à étudier. Il est en effet difficile de tenir compte de nombreuses propriétés sur de nombreux événements dès le premier essai de spécification. Une solution consiste à modéliser le système progressivement en ajustant de manière empirique le modèle. Le raffinement a déjà montré qu'il était une approche adaptée pour ce type de problèmes [Abr00]. L'idée est d'introduire étape après étape les principaux événements, tout en respectant les propriétés déjà vérifiées. Il existe en CSP un raffinement basé sur les traces-divergences. Les processus sont raffinés par restriction des traces admissibles, des blocages et des divergences. Cependant, il ne prend pas en compte le renforcement des gardes des actions, la composition d'actions atomiques ou l'ajout de nouveaux événements dans les processus. Il faut donc définir une nouvelle relation de raffinement pour le langage EB^3. L'idée est de commencer par donner des propriétés dynamiques avec une logique temporelle, et de les raffiner ensuite en décrivant les événements qui s'insèrent entre les événements décrits dans les propriétés, jusqu'à obtenir toutes les expressions de processus de la spécification EB^3.

Bibliographie

[ABK+02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella, and A. Tarlecki. CASL : The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2) :153–196, September 2002.

[Abr96] J.R. Abrial. *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr00] J.-R. Abrial. Guidelines to Formal Systems Studies. ClearSy, November 2000.

[AL88] M. Abadi and L. Lamport. *The Existence of Refinement Mappings*. Technical report, Digital Systems Research Center, Palo Alto, California, 1988.

[AM98] J.R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In *B'98 : Recent Advances in the Development and Use of the B Method*, volume 1393 of *LNCS*, pages 83–128, Montpellier, France, 22-24 April 1998. Springer-Verlag.

[AMF00] Y. Amghar, M. Meziane, and A. Flory. Using Business Rules within a Design Process of Active Databases. *International Journal of Database Management*, 11(3) :3–15, July 2000.

[Bac78] R.J. Back. *On the Correctness of Refinement in Program Development*. PhD thesis, University of Helsinki, Finland, 1978.

[Bac88] R.J. Back. A Calculus of Refinements for Program Derivations. *Acta Informatica*, 25(6) :593–624, 1988.

[Bat05] P. Batanado. Synthèse de transactions de base de données relationnelle à partir de définitions d'attributs EB^3. Master's thesis, Université de Sherbrooke, Québec, Canada, 2005.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1) :25–59, 1987.

[BDW99] C. Bolton, J. Davies, and J. Woodcock. On the Refinement and Simulation of Data Types and Processes. In *IFM 99*, pages 273–292, York, United Kingdom, 28-29 June 1999. Springer-Verlag.

[BG92] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language : Design, Semantics, Implementation. *Science of Computer Programming*, 19(2) :87–152, November 1992.

[BGL03] S. Blazy, F. Gervais, and R. Laleau. Reuse of Specification Patterns with the B Method. In *ZB2003 : Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, pages 40–57, Turku, Finland, 4-6 June 2003. Springer-Verlag.

[BKS83] R.J. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *PODC 1983*, pages 131–142, Montréal, Canada, 17-19 August 1983. ACM.

[Boo94] G. Booch. *Object-Oriented Analysis and Design With Applications*. Addison-Wesley, 1994.

[But92] M. Butler. *A CSP Approach to Action Systems*. PhD thesis, Oxford University, United Kingdom, 1992.

[But99] M. Butler. csp2B : A Practical Approach to Combining CSP and B. In *FM'99*, volume 1708 of *LNCS*, pages 490–508, Toulouse, France, 20-24 September 1999. Springer-Verlag.

[BvW98] R.J. Back and J. von Wright. *Refinement Calculus : A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[Che76] P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1) :9–36, March 1976.

[Cle06] Clearsy. Atelier B. http://www.atelierb-societe.com, 2006.

[CM98] G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, Cambridge, 1998.

[Cor06] Correct System Design Group. Moby/OZ. http://csd.informatik.uni-oldenburg.de/~moby/, 2006.

[CSW02] A.L.C. Cavalcanti, A.C.A. Sampaio, and J.C.P. Woodcock. Refinement of Actions in Circus. In *REFINE'2002*, volume 70, pages 492–522, Copenhagen, Denmark, 20-21 July 2002. Electronic Notes in Theoretical Computer Science.

[CSW05] A.L.C. Cavalcanti, A.C.A. Sampaio, and J.C.P. Woodcock. Unifying Classes and Processes. *Software and Systems Modeling*, 4(3) :277–296, 2005.

[Dar02] C. Darlot. *Reformulation et vérification de propriétés temporelles dans le cadre du raffinement de systèmes d'événements*. PhD thesis, Université de Franche-Comté, France, 2002.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DLCP00] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An Overview of RoZ : A Tool for Integrating UML and Z Specifications. In *CAiSE'00*, volume 1789 of *LNCS*, pages 417–430, Stockholm, Sweden, 2000. Springer-Verlag.

[dRE98] W.P. de Roever and K. Engelhardt. *Data Refinement : Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[DW96] J. Davies and J.C.P. Woodcock. *Using Z : Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[Edm95] D. Edmond. Refining Database Systems. In *ZUM'95*, volume 967 of *LNCS*, pages 25–44, Limerick, Ireland, 7-9 September 1995. Springer-Verlag.

[EN04] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 4th edition, 2004.

[ETLF04] N. Evans, H. Treharne, R. Laleau, and M. Frappier. How to Verify Dynamic Properties of Information Systems. In *SEFM 2004*, pages 416–425, Beijing, China, 28-30 September 2004. IEEE Computer Society Press.

[FF02] B. Fraikin and M. Frappier. EB3PAI : An Interpreter for the EB^3 Specification Language. In *ICSSEA 2002*, Paris, France, 3-5 December 2002.

[FFL05] B. Fraikin, M. Frappier, and R. Laleau. State-Based versus Event-Based Specifications for Information Systems : A Comparison of B and EB^3. *Software and Systems Modeling*, 4(3) :236–257, July 2005.

[FFLR02] M. Frappier, B. Fraikin, R. Laleau, and M. Richard. APIS - Automatic Production of Information Systems. In *AAAI Spring Symposium*, pages 17–24, Stanford, USA, 25-27 March 2002. AAAI Press.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data : from CSP-OZ to Java*. PhD thesis, University of Oldenburg, Germany, 2000.

[FL03] M. Frappier and R. Laleau. Proving Event Ordering Properties for Information Systems. In *ZB2003*, volume 2651 of *LNCS*, pages 421–436, Turku, Finland, 4-6 June 2003. Springer-Verlag.

[For97] Formal Systems (Europe) Ltd. Failures-Divergences Refinement : FDR2 User Manual. http://www.formal.demon.co.uk, 1997.

[Fra06] B. Fraikin. *Interprétation efficace d'expression de processus EB^3*. PhD thesis, Université de Sherbrooke, Québec, Canada, 2006.

[FSD03] M. Frappier and R. St-Denis. EB^3 : An Entity-Based Black-Box Specification Method for Information Systems. *Software and Systems Modeling*, 2(2) :134–149, July 2003.

[Gar99] G. Gardarin. *Bases de données*. Eyrolles, 1999.

[GBFL06a] F. Gervais, P. Batanado, M. Frappier, and R. Laleau. EB^{3}TG : A Tool Synthesizing Relational Database Transactions from EB^3 Attribute Definitions. In *ICEIS 2006*, volume Information Systems Analysis and Specification, pages 44–51, Paphos, Cyprus, 24-27 May 2006. INSTICC Press.

[GBFL06b] F. Gervais, P. Batanado, M. Frappier, and R. Laleau. Génération automatique de transactions de base de données relationnelle à partir de définitions d'attributs EB^3. In *AFADL 2006*, pages 25–39, Paris, France, 15-17 March 2006. ENST.

[Geo91] C. George. The RAISE Specification Language : A Tutorial. In *VDM'91*, volume 552 of *LNCS*, pages 238–319, Noordwijkerhout, The Netherlands, 21-25 October 1991. Springer-Verlag.

[Ger04] F. Gervais. *EB^4 : Vers une méthode combinée de spécification formelle des systèmes d'information*. Dissertation for the general examination, Université de Sherbrooke, Québec, Canada, June 2004.

[Ger06] F. Gervais. EB^4 : Vers une méthode de spécification formelle des SI. In *INFORSID 2006*, volume 2, pages 561–576, Hammamet, Tunisia, 1-3 June 2006. INFORSID.

[GFL04] F. Gervais, M. Frappier, and R. Laleau. *Synthesizing B Substitutions for EB^3 Attribute Definitions*. Technical Report 683, CEDRIC, Paris, France, November 2004.

[GFL05a] F. Gervais, M. Frappier, and R. Laleau. Generating Relational Database Transactions from Recursive Functions Defined on EB^3 Traces. In *SEFM 2005*, pages 117–126, Koblenz, Germany, 7-9 September 2005. IEEE Computer Society Press.

[GFL05b] F. Gervais, M. Frappier, and R. Laleau. Synthesizing B Specifications from EB^3 Attribute Definitions. In *IFM 2005*, volume 3771 of *LNCS*, pages 207–226, Eindhoven, The Netherlands, 29 November - 2 December 2005. Springer-Verlag.

[GFL07] F. Gervais, M. Frappier, and R. Laleau. Refinement of EB^3 Process Patterns into B Specifications. In *B 2007*, volume 4355 of *LNCS*, pages 201–215, Besançon, France, 17-19 January 2007. Springer-Verlag.

[GFLB05] F. Gervais, M. Frappier, R. Laleau, and P. Batanado. *EB^3 Attribute Definitions : Formal Language and Application*. Technical Report 700, CEDRIC, Paris, France, February 2005.

[GFSD06] F. Gervais, M. Frappier, and R. St-Denis. *Software Specification Methods*, chapter EB^3, pages 259–274. ISTE, 2006.

[GH93] J. Guttag and J. Horning. *Larch : Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[GS79] C. Gane and T. Sarson. *Structured Systems Analysis : Tools and Techniques*. Prentice-Hall, 1979.

[GS90] H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In *PSTV 1990*, pages 379–394, Ottawa, Ontario, Canada, 12-15 June 1990. North-Holland.

[GS97] A.J. Galloway and W.J. Stoddart. Integrated Formal Methods. In *INFORSID'97*, pages 549–576, Toulouse, France, 11-13 June 1997. INFORSID.

[Gur93] Y. Gurevich. Evolving Algebras : An Attempt to Discover Semantics. In *Current Trends in Theoretical Computer Science*. World Scientific, 1993.

[Gur95] Y. Gurevich. Evolving Algebras 1993 : Lipari Guide. In *Specification and Validation Methods*. Oxford University Press, 1995.

[Har87] D. Harel. Statecharts : A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8 :231–274, 1987.

[HE92] U. Hohenstein and G. Engels. SQL/EER - Syntax and Semantics of an Entity-Relationship-Based Query Language. *Information Systems*, 17(3) :209–242, May 1992.

[HJ98] C.A.R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10) :576–583, 1969.

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

[Hoa94] C.A.R. Hoare. Unified Theories of Programming. Monograph, Oxford University Computing Laboratory, 1994.

[Hoa03] C.A.R. Hoare. The Verifying Compiler : A Grand Challenge for Computing Research. *Journal ACM*, 50(1) :63–69, January 2003.

[Hoh89] U. Hohenstein. Automatic Transformation of an Entity-Relationship Query Language into SQL. In *ER'89*, pages 303–321, Toronto, Canada, 18-20 October 1989. Elsevier.

[Ill91] V. Illingworth. *Dictionnaire d'informatique*. Hermann, 1991.

[ISO97] ISO. *Dictionary of Computer Science : the Standardized Vocabulary*. ISO, AFNOR, 1997.

[Jac83] M. Jackson. *System Development*. Prentice-Hall, 1983.

[Jac94] I. Jacobson. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, 1994.

[JBo06] JBoss Labs. Hibernate. http://www.hibernate.org, 2006.

[Jen96] K. Jensen. *Coloured Petri Nets : Basic Concepts, Analysis Methods and Practical Use*. Springer-Verlag, 1996.

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

[Lal02] R. Laleau. *Conception et développement formels d'applications bases de données*. Habilitation à diriger des recherches, Université d'Évry Val d'Essonne, France, 2002.

[Led98] Y. Ledru. Identifying Pre-conditions with the Z/EVES Theorem Prover. In *ASE 1998*, pages 32–41, Honolulu, Hawaii, USA, 13-16 October 1998. IEEE Computer Society Press.

[LM00] R. Laleau and A. Mammar. An Overview of a Method and its Support Tool for Generating B Specifications from UML Notations. In *ASE 2000*, pages 269–272, Grenoble, France, 11-15 September 2000. IEEE Computer Society Press.

[LN99] M. Leucker and T. Noll. Rapid Prototyping of Specification Language Implementations. In *RSP 1999*, pages 60–65, Clearwater, Florida, USA, 16-18 June 1999. IEEE Computer Society Press.

[Mam02] A. Mammar. *Un environnement formel pour le développement d'applications base de données*. PhD thesis, CNAM, France, 2002.

[MC88] J. Misra and K.M. Chandy. *Parallel Program Design : A Foundation*. Addison-Wesley, 1988.

[MGL06] A. Mammar, F. Gervais, and R. Laleau. Systematic Identification of Preconditions from Set-Based Integrity Constraints. In *INFORSID 2006*, volume 2, pages 595–610, Hammamet, Tunisia, 1-3 June 2006. INFORSID.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil90] R. Milner. *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter Operational and Algebraic Semantics of Concurrent Processes. MIT Press, 1990.

[MM88] D. Marca and C. McGowan. *SADT : Structured Analysis and Design Techniques*. McGraw-Hill, 1988.

[Mor87] J.M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9 :287–306, 1987.

[Mor88] C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 11(4) :517–561, 1988.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[MS99] E. Meyer and J. Souquières. A Systematic Approach to Transform OMT Diagrams to a B Specification. In *FM'99*, volume 1708 of *LNCS*, pages 875–895, Toulouse, France, 20-24 September 1999. Springer-Verlag.

[Ngu98] H.P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, CNAM, France, 1998.

[Obj06] Object Management Group. Unified Modeling Language. http://www.uml.org, 2006.

[Pel86] P. Pellaumail. *La méthode AXIAL*. Éditions d'Organisation, 1986.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[PJ03] F. Peschanski and D. Julien. When Concurrent Control Meets Functional Requirements, or Z + Petri-Nets. In *ZB2003*, volume 2651 of *LNCS*, pages 79–97, Turku, Finland, 4-6 June 2003. Springer-Verlag.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57, Providence, Rhode Island, USA, 31 October - 2 November 1977. IEEE.

[Pnu81] A. Pnueli. The Temporal Semantics of Concurrent Programs. *Theoretical Computer Science*, 13 :45–60, 1981.

[PTLP99] S.J. Prowell, C.J. Trammell, R.C. Linger, and J.H. Poore. *Cleanroom Software Engineering : Technology and Process*. Addison-Wesley, 1999.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[RFB88] C. Rolland, O. Foucaut, and G. Benci. *Conception des systèmes d'information : la méthode REMORA*. Eyrolles, 1988.

[Rob03] P. Robert. *Le Petit Robert*. Dictionnaires Le Robert, 2003.

[Ros97] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[Saa97] M. Saaltink. The Z/EVES System. In *ZUM'97*, volume 1212 of *LNCS*, pages 72–85, Reading, United Kingdom, 3-4 April 1997. Springer-Verlag.

[Sch99] S. Schneider. *Concurrent and Real-time Systems : The CSP Approach*. Wiley, 1999.

[SD01] G. Smith and J. Derrick. Specification, Refinement and Verification of Concurrent Systems - an Integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18 :249–284, May 2001.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[Spi92] J.M. Spivey. *The Z Notation : a Reference Manual*. Prentice-Hall, 1992.

[ST02] S. Schneider and H. Treharne. Communicating B Machines. In *ZB2002*, volume 2272 of *LNCS*, pages 416–435, Grenoble, France, 23-25 January 2002. Springer-Verlag.

[SWC02] A.C.A. Sampaio, J.C.P. Woodcock, and A.L.C. Cavalcanti. Refinement in Circus. In *FME 2002*, volume 2391 of *LNCS*, pages 451–470, Copenhagen, Denmark, 22-24 July 2002. Springer-Verlag.

[Tan92] A. Tanaka. *On Conceptual Design of Active Databases*. PhD thesis, Georgia Institute of Technology, USA, 1992.

[Ter05] J.-G. Terrillon. Description comportementale d'interfaces web. Master's thesis, Université de Sherbrooke, Québec, Canada, 2005.

[TRC83] H. Tardieu, A. Rochfeld, and R. Colleti. *La méthode MERISE : principes et outils*. Éditions d'Organisation, 1983.

[TS99] H. Treharne and S. Schneider. Using a Process Algebra to Control B OPERATIONS. In *IFM'99*, pages 437–457, York, United Kingdom, 28-29 June 1999. Springer-Verlag.

[TS00] H. Treharne and S. Schneider. How to Drive a B Machine. In *ZB2000*, volume 1878 of *LNCS*, pages 188–208, York, United Kingdom, 29 August - 2 September 2000. Springer-Verlag.

[Ull88] J. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988. 2 volumes.

[vG99] J. von Garrel. *Typechecking und transformation von CSP-OZ spezifikationen nach Jass*. Master's thesis, University of Oldenburg, Germany, 1999.

[WC96] J. Widom and S. Ceri. *Active Database Systems*. Morgan Kaufmann, 1996.

[WC02] J.C.P. Woodcock and A.L.C. Cavalcanti. The Semantics of Circus. In *ZB 2002*, volume 2272 of *LNCS*, pages 184–203, Grenoble, France, 23-25 January 2002. Springer-Verlag.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of ACM*, 14(4) :221–227, 1971.

[YC79] E. Yourdon and E. Constantine. *Structured Design*. Prentice-Hall, 1979.

Table des figures

2.1 Exemple de LTS . . . 21
2.2 Classe Queue[T] . . . 28
2.3 Machine abstraite B : *Exemple_Machine* . . . 30
2.4 Raffinement en B : *Exemple_Refinement* . . . 31
2.5 Système d'événements B : *Exemple_System* . . . 33
2.6 Raffinement en B événementiel : *Exemple_Refinement* . . . 34

3.1 LTS de l'exemple . . . 39
3.2 Méthode de raffinement csp2B . . . 46
3.3 Interactions entre machines B et processus CSP . . . 49
3.4 Lien entre la machine B et le processus CSP . . . 50
3.5 Spécification CSP-OZ : classe *ExempleData* . . . 56
3.6 Vérification de propriétés PLTL sur des systèmes d'événements B 67

4.1 Diagramme EB^3 de la bibliothèque . . . 77
4.2 Rôles d'une association de type $M : N$ . . . 87
4.3 Rôles d'une association de type $1 : N$ . . . 87

6.1 Algorithme de génération des substitutions d'opérations . . . 108
6.2 Première étape de la construction de l'arbre de décision de b_F . . 112
6.3 Arbre de décision généré pour la clause $\mathsf{Transfer}_F$ de $nbLoans_F$. 112
6.4 Correction de la traduction en B . . . 121

7.1 Résumé de l'approche EB^4 . . . 124

9.1 Composantes du projet APIS . . . 150
9.2 Exemple de code Java généré par l'outil EB^{3}TG . . . 151
9.3 Modèle de données représenté par les définitions d'attributs EB^3 . 152

11.1 Correction de la synthèse de transactions . . . 171
11.2 Extension de la relation rel_j calculée avec Ψ . . . 174
11.3 Architecture fonctionnelle de l'outil EB^{3}TG . . . 177
11.4 Outil EB^{3}TG . . . 178
11.5 Schéma de BD généré pour la bibliothèque . . . 179
11.6 Exemple d'application d'EB^{3}TG sur la bibliothèque . . . 180
11.7 Méthode Java générée pour Transfer . . . 181

12.1 Vue globale de la méthode EB^4 . . . 188

B.1 Opérateurs relationnels du langage B 208
B.2 Quelques substitutions du langage B 208

C.1 Diagramme EB^3 de la bibliothèque 210

Annexes

Annexe A

Glossaire des notions utilisées

Acteur. Personne ou entité qui prend une part active ou joue un rôle important dans un système [Rob03].

Action. Opération instantanée associée à un événement [RBP+91]. Dans un système de transitions étiquetées, une action est un événement étiqueté [Mil90]. En EB^3, une action constitue une expression élémentaire de l'algèbre de processus (section 1.4.2).

Agent. Dans un système concurrent, entité agissant de manière indépendante, capable de communiquer avec d'autres agents [Mil80].

Association. Dans le modèle entité-association, lien logique entre deux ou plusieurs entités [Gar99].

Base de données. Ensemble de données interrogeables modélisant les objets d'une partie du monde réel et qui sert de support à une application informatique [Gar99].

Cohérence. État de ce qui est relié de manière étroite et sans contradiction [Rob03]. En B, une machine abstraite est dite cohérente lorsque l'initialisation et chaque opération préservent les propriétés d'invariance (section 2.2.4).

Complétude. Propriété d'une spécification dont le modèle respecte toutes les exigences du système à modéliser.

Contrainte d'intégrité. Dans les bases de données, règle sémantique assurant la cohérence des données lors des mises à jour de la base [Gar99].

Correction. Qualité d'un système ou d'un programme qui est conforme à sa représentation [Rob03]. Une preuve de correction est une démonstration formelle qui prouve que la sémantique d'un programme est cohérente vis-à-vis de sa spécification [ISO97]. Dans la logique de Hoare [Hoa69], toute expression d'un langage est associée à une précondition et à une postcondition. Un programme est *totalement correct* vis-à-vis de sa spécification si, à partir de tout état initial vérifiant la précondition, le programme termine et fait passer le système dans un état satisfaisant la postcondition. Si la terminaison n'est pas assurée, la correction est seulement *partielle*.

Entité. Objet du monde réel doué d'une unité matérielle, dont l'existence est indépendante des autres entités [Gar99].

État. Ensemble de valeurs nommées : les noms sont des variables et les valeurs sont exprimées dans des domaines issus des mathématiques (entiers naturels, réels, etc ...) [Mor90].

Événement. Action indivisible atomique pouvant être exécutée par un processus [Sch99]. En EB^3, un événement correspond à l'exécution d'une action (section 1.4.2).

Formel. Dont la précision exclut toute forme d'équivoque [Rob03]. Une notation est dite formelle si elle peut être exprimée mathématiquement [Rob03]. Une spécification est formelle si elle est écrite avec une notation formelle [ISO97]. Un langage est formel s'il comprend des règles de syntaxe et de sémantique explicites et précises [Ill91]. Une méthode est formelle si elle s'appuie sur des techniques et sur des langages formels (section 2.1).

Implémentation. Représentation d'un logiciel qui peut être traduite automatiquement en un programme exécutable sur un ordinateur (section 2.1). En B, dernier niveau de raffinement qui peut être traduit automatiquement vers du code (section 2.2.4).

Méthode. Ensemble de techniques et de notations utilisées selon certains principes ou un certain ordre pour arriver à un but. Par exemple, une méthode de conception [Rob03].

Model-checking. Techniques de vérification qui consistent à parcourir exhaustivement l'espace d'états d'un modèle fini du système à vérifier. Les propriétés à vérifier sont généralement exprimées en logique temporelle. Un contre-exemple est généré lorsque la propriété n'est pas vérifiée par le modèle considéré. Un *model-checker* est un outil qui permet d'appliquer une technique de *model-checking* (section 2.1).

Preuve. Techniques de vérification qui consistent à prouver des propriétés à partir des axiomes du système et d'un ensemble de règles d'inférence. La preuve peut être manuelle ou automatisée (section 2.1).

Processus. Pattern de communication [Hoa85]. Un processus est une entité indépendante qui communique avec d'autres processus [Sch99].

Programme. Ensemble d'instructions, rédigé dans un langage de programmation, permettant à un système informatique d'accomplir une tâche donnée [Rob03]. Unité syntaxique qui respecte les règles d'un langage de programmation et qui est composée de déclarations et d'instructions utilisées pour résoudre une certaine tâche ou un problème [ISO97]. Ensemble d'instructions détaillées écrit dans un langage de programmation dont la forme (la syntaxe) et le sens (la sémantique) sont définis précisément. Les programmes sont faciles à exécuter, mais difficiles à comprendre [Mor90].

Propriété. Ensemble de caractères d'un objet ou d'un système [Rob03].
- Une propriété de *sûreté* est une propriété positive de la forme : "quelque chose de mauvais n'arrive jamais". Par exemple, une propriété d'invariance est une propriété qui est vraie entre chaque exécution d'opération.
- Une propriété de *vivacité* est une propriété de la forme : "quelque chose de bien arrivera nécessairement". Par exemple, une propriété temporelle est une propriété exprimée dans une logique temporelle comme PLTL (voir section 3.6). Une propriété d'ordonnancement est également une propriété de vivacité.

Raffinement. Processus permettant de réexprimer progressivement les idées de haut niveau en des idées de plus bas niveau [Ill91]. Une instruction S' raffine S si S' satisfait toute spécification satisfaite par S [But92]. Un développement de programme implique le raffinement par étapes d'un programme abstrait en un programme exécutable par l'application d'une série de transformations préservant la correction [But92]. Concevoir un programme complexe implique en général l'application d'une méthode de raffinement qui fournit une façon de transformer graduellement un programme abstrait ou une spécification en une implémentation concrète. Le principe d'une telle méthode est que si le programme abstrait initial est correct et que les étapes de transformation préservent la correction, alors l'implémentation sera correcte par construction [dRE98].

Redondance. Caractère de ce qui apporte une information déjà donnée sous une autre forme [Rob03].

Relation. Dans les bases de données, sous-ensemble du produit cartésien entre plusieurs domaines de valeurs [Gar99].

Sémantique. Partie de la définition d'un langage qui concerne la spécification de la signification ou de l'effet d'un texte construit selon les règles de la syntaxe [Ill91].

Simulation. Relation entre deux espaces d'états qui est préservée par chaque opération. Si A est une simulation de C, alors C contient strictement plus d'informations que A. En fait, il est possible de montrer que A est une simulation de C précisément lorsque C est un raffinement de A [BDW99].

Spécification. Document qui décrit la structure et les fonctionnalités d'un système de manière détaillée, afin d'en faciliter la programmation et la maintenance [ISO97]. Énoncé précis et détaillé des résultats qu'on attend d'un système. La spécification peut être écrite dans la langue naturelle ou bien à l'aide d'un langage de spécification [Ill91]. Une spécification décrit ce qu'un système doit faire. Les spécifications sont difficiles à exécuter, mais faciles à comprendre - ou devraient l'être [Mor90]. Une spécification décrit les propriétés observables et le comportement d'un système qui n'existe pas encore dans le monde physique ; et le but est de concevoir et d'implémenter un produit qui a été prévu, en théorie, pour respecter la spécification [Hoa94].

Syntaxe. Règles définissant les séquences de symboles ou de caractères dans un langage [Ill91].

Système d'information. Système informatisé qui rassemble l'ensemble des informations présentes au sein d'une organisation, sa mémoire, et les activités qui permettent de les manipuler [Lal02].

Transaction. Application sur la base de données qui permet d'y faire des interrogations ou des mises à jour et qui vérifie en général les propriétés suivantes :

- une transaction est atomique, car elle est exécutée dans son ensemble ou pas du tout,
- la base de données doit rester cohérente après l'exécution d'une transaction,
- chaque transaction est indépendante et n'interfère pas avec les autres transactions,
- et les modifications d'une transaction qui a été exécutée perdurent même en cas de panne de la base de données.

Ces propriétés sont appelées ACID (pour *Atomicity, Consistency, Isolation, Durability*) [EN04].

Validation. Étape du développement logiciel qui consiste à vérifier que le logiciel est conforme aux exigences. Elle permet de vérifier que le produit répond aux attentes du client (section 2.1).

Vérification. Étape qui consiste à vérifier que le logiciel implémente correctement certaines propriétés. Il existe principalement deux pistes pour vérifier une propriété avec des méthodes formelles : la preuve et le *model-checking* (section 2.1).

Annexe B

Langage B

Dans cette annexe, nous présentons les principaux concepts du langage B qui sont utilisés dans le cadre de cette thèse. Une présentation complète est donnée dans [Abr96]. Le langage B inclut notamment deux sous-langages :

- le langage de la logique du premier ordre, enrichi par le prédicat d'égalité, qui permet l'expression des propriétés statiques du système ;
- le langage de substitutions généralisées, qui permet de décrire le comportement dynamique du système.

Les propriétés que doit satisfaire le système sont exprimées sous forme de prédicats du premier ordre. Ces prédicats utilisent notamment des opérateurs relationnels. Les principaux opérateurs relationnels utilisés dans cette thèse sont présentés dans la figure B.1. Une substitution généralisée est un transformateur de prédicats ; cette notion est analogue à celle de plus faible précondition de Hoare [Hoa69]. Les principales substitutions sont présentées dans la figure B.2.

Nom	Syntaxe	Définition	Précondition
Produit cartésien	$s \times t$	$\{x \mapsto y \mid x \in s \wedge y \in t\}$	
Relation	$s \leftrightarrow t$	$\mathbb{P}(s \times t)$ où $\mathbb{P}(S)$ est l'ensemble des parties de S	
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	$q ; r$	$\{x \mapsto y \mid x \mapsto y \in s \times u \wedge \exists z.(z \in t \wedge x \mapsto z \in q \wedge z \mapsto y \in r)\}$	$q \in s \leftrightarrow t \wedge r \in t \leftrightarrow u$
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	$r \rhd\!\!\!- u$	$\{x \mapsto y \mid (x, y) \in r \wedge y \notin u\}$	$r \in s \leftrightarrow t \wedge u \subseteq t$
Surcharge	$r <\!\!+ r'$	$(dom(r') \lhd\!\!\!- r) \cup r'$	$r \in s \leftrightarrow t \wedge r' \in s \leftrightarrow t$
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	$s \rightarrowtail\!\!\!\!\twoheadrightarrow t$	$s \rightarrowtail t \cap s \twoheadrightarrow t$	

FIG. B.1 – Opérateurs relationnels du langage B

Notation B	Conditions de définition	Signification
skip		Ne rien modifier
$x := E$	x est une variable E est une expression	Substituer E à x
PRE P **THEN** S **END**	P est un prédicat S est une substitution	S'assurer de P et exécuter S
IF P **THEN** S **ELSE** T **END**	P est un prédicat S une substitution	Si P est vrai, exécuter S sinon exécuter T
SELECT P **THEN** S **END**	P est un prédicat S est une substitution	exécuter S si P est vrai
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
$S \mid\mid T$	S et T sont des substitutions	exécuter S et T en même temps
$S \,[]\, T$	S et T sont des substitutions	exécuter S ou T
$S ; T$	S et T sont des substitutions	exécuter S puis T

FIG. B.2 – Quelques substitutions du langage B

Annexe C

Exemple de la bibliothèque

Dans cette annexe, nous traitons un cas d'étude classique en SI : le système de gestion d'une bibliothèque. Il est utilisé dans la thèse pour illustrer les différents concepts. Nous supposons que les bases du langage EB^3 sont des pré-requis (voir chapitre 4 pour plus de détails sur la syntaxe du langage). La section C.1 présente le cas d'étude. La spécification EB^3 est détaillée dans la section C.2. Puis, les résultats obtenus en appliquant l'approche EB^4 sont présentés dans la section C.3. Enfin, les transactions obtenues par synthèse automatique sont fournies dans la section C.4.

C.1 Cas d'étude

Le système doit gérer des emprunts de livres par des membres. Les exigences sont les suivantes :

1. Un livre est acquis (action Acquire) par la bibliothèque. Il peut être supprimé des références (Discard), mais seulement s'il n'est plus emprunté. Le titre du livre peut être modifié à tout moment (Modify). Il est également possible d'afficher le titre d'un livre (DisplayTitle).
2. Un membre doit s'inscrire à la bibliothèque (Register) pour pouvoir emprunter un livre. Il ne peut résilier son adhésion (Unregister) que lorsqu'il a rendu tous ses prêts.
3. Un membre peut emprunter un livre (Lend) si ce dernier est disponible. L'attribut *nbLoans* représente le nombre de prêts d'un membre. La date d'échéance d'un prêt est représentée par *dueDate*. Tout membre doit rendre un livre emprunté (Return) avant la date d'échéance prévue.
4. Un même livre ne peut être emprunté que par un seul membre à la fois.
5. Un membre peut réserver un livre (Reserve) si ce dernier est déjà emprunté par un autre membre. Plusieurs membres peuvent réserver le même livre. L'attribut *position* indique la position d'un membre dans la liste de réservation d'un livre.
6. Dès qu'il est retourné, un livre qui a été réservé peut être récupéré (Take) par le membre dont la position est égale à 1 dans la liste de réservation. Tant qu'un membre n'a pas récupéré un livre, il peut à tout moment annuler (Cancel) sa réservation.

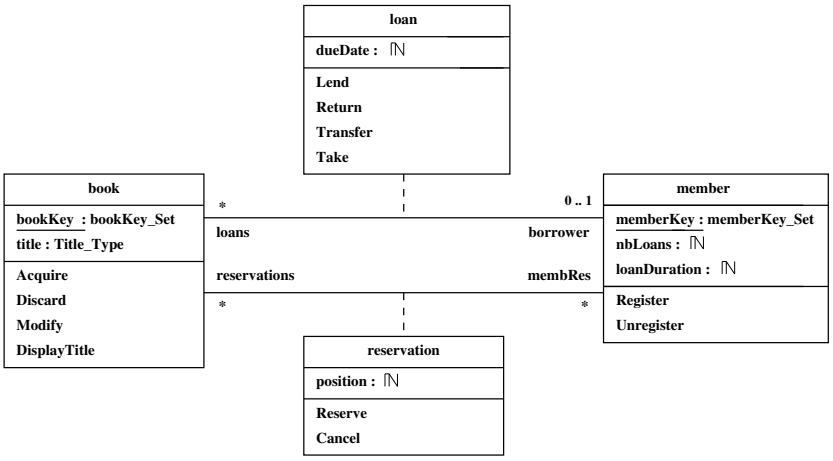

FIG. C.1 – Diagramme EB^3 de la bibliothèque

7. Un membre peut transférer (Transfer) un de ses prêts à un autre membre, si le livre en question n'a pas été réservé.
8. La durée des prêts dépend du type de prêts accordé par la bibliothèque : permanent ou classique. Un prêt de type permanent dure un an, alors qu'un prêt classique dure le temps prévu pour le membre en question (représenté par l'attribut *loanDuration*).

C.2 Spécification avec EB^3

Dans les prochaines sections, nous présentons les quatre composantes suivantes de la spécification EB^3 dans l'exemple :

1. diagramme ER,
2. expression de processus,
3. règles d'entrée-sortie,
4. définitions d'attributs.

La spécification ci-dessous est celle que nous obtenons après application de la méthode EB^4.

C.2.1 Diagramme ER

La figure C.1 représente le diagramme ER de l'exemple. Les signatures des actions sont les suivantes :

$\mathsf{Acquire}(bId : bookKey\_Set, bTitle : Title\_Type^{\perp}) : void$
$\mathsf{Discard}(bId : bookKey\_Set) : void$
$\mathsf{Modify}(bId : bookKey\_Set, newTitle : Title\_Type^{\perp}) : void$
$\mathsf{DisplayTitle}(bId : bookKey\_Set) : (title : Title\_Type^{\perp})$
$\mathsf{Register}(mId : memberKey\_Set, lD : NAT) : void$
$\mathsf{Unregister}(mId : memberKey\_Set) : void$

Lend($bId : bookKey\_Set, mId : memberKey\_Set, type : Loan\_Type$) : $void$
Return($bId : bookKey\_Set$) : $void$
Transfer($bId : bookKey\_Set, mId : memberKey\_Set, type : Loan\_Type$) : $void$
Reserve($bId : bookKey\_Set, mId : memberKey\_Set$) : $void$
Take($bId : bookKey\_Set, mId : memberKey\_Set, type : Loan\_Type$) : $void$
Cancel($bId : bookKey\_Set, mId : memberKey\_Set$) : $void$

Nous rappelons que toute action en EB^3 renvoie implicitement un paramètre de sortie pour indiquer si l'action est valide ("*ok*") ou pas ("*error*").

C.2.2 Expression de processus

Le processus principal du SI s'appelle *main* :

$main =$
($|||bId : bookKey\_Set : book(bId)^*$)
$||$
($|||mId : memberKey\_Set : member(mId)^*$)

Ce processus met en parallèle les différentes entités de type *book* et celles de type *member*. Le processus associé au type d'entité *book* est de la forme suivante :

$book(bId : bookKey\_Set) =$
Acquire($bId, \_$)**.**
(
($\mid mId : memberKey\_Set : loan(mId, bId)$)*
$||$
$loanCycleBook(bId)^*$
$||$
($|||mId : memberKey\_Set : reservation(mId, bId)^*$)
$|||$
Modify($bId, \_$)*
$|||$
DisplayTitle(bId)*
)**.**
Discard(bId)

où *loan* désigne le processus associé à l'association *loan* et *reservation* celui de l'association *reservation*. Le processus *loanCycleBook* permet de contrôler le cycle de vie des prêts pour un livre *bId* donné :

$loanCycleBook(bId : bookKey\_Set) =$
Lend($bId, \_, \_$)**.**
Transfer($bId, \_, \_$)***.**
Return(bId)

Le processus associé au type d'entité *member* est de la même forme que *book* :

$member(mId : memberKey\_Set) =$
Register($mId, \_$)**.**
(
($|||bId : bookKey\_Set : loan(mId, bId)^*$)

$$
\begin{array}{l}
\quad \| \\
\quad\quad (\; |||bId : bookKey\_Set : reservation(mId, bId)^*) \\
\quad). \\
\quad \mathsf{Unregister}(mId)
\end{array}
$$

Le processus de l'association *loan* est de la forme suivante :

$$
\begin{array}{l}
loan(mId : memberKey\_Set, bId : bookKey\_Set) = \\
\quad\quad loanNominal(mId, bId) \\
\quad \| \\
\quad\quad loanController(mId, bId)^*
\end{array}
$$

où les sous-processus *loanNominal* et *loanController* sont définis par :

$$
\begin{array}{l}
loanNominal(mId : memberKey\_Set, bId : bookKey\_Set) = \\
\quad (\mathsf{Lend}(bId, mId, \_) \mid \mathsf{Take}(bId, mId, \_) \mid \mathsf{Transfer}(bId, mId, \_)). \\
\quad (\mathsf{Return}(bId) \mid \lambda)
\end{array}
$$

$$
\begin{array}{l}
loanController(mId : memberKey\_Set, bId : bookKey\_Set) = \\
\quad\quad (nbLoans(trace, mId) < maxNbLoans \wedge \\
\quad\quad\quad membRes(trace, bId) = \varnothing) \\
\quad\quad\quad \Longrightarrow \mathsf{Lend}(bId, mId, \_) \\
\quad | \\
\quad\quad (nbLoans(trace, mId) < maxNbLoans \\
\quad\quad\quad \Longrightarrow \mathsf{Take}(bId, mId, \_) \\
\quad | \\
\quad\quad (nbLoans(trace, mId) < maxNbLoans \wedge \\
\quad\quad\quad mId \neq borrower(trace, bId) \wedge \\
\quad\quad\quad membRes(trace, bId) = \varnothing) \\
\quad\quad\quad \Longrightarrow \mathsf{Transfer}(bId, mId, \_)) \\
\quad\quad)
\end{array}
$$

Le processus de l'association *reservation* est de la forme suivante :

$$
\begin{array}{l}
reservation(mId : memberKey\_Set, bId : bookKey\_Set) = \\
\quad\quad reservationNominal(mId, bId) \\
\quad \| \\
\quad\quad reservationController(mId, bId)^*
\end{array}
$$

où les sous-processus *reservationNominal* et *reservationController* sont définis par :

$$
\begin{array}{l}
reservationNominal(mId : memberKey\_Set, bId : bookKey\_Set) = \\
\quad \mathsf{Reserve}(bId, mId). \\
\quad (\mathsf{Take}(bId, mId, \_) \mid \mathsf{Cancel}(bId, mId))
\end{array}
$$

$$
\begin{array}{l}
reservationController(mId : memberKey\_Set, bId : bookKey\_Set) = \\
\quad\quad mId \neq borrower(trace, bId) \Longrightarrow \mathsf{Reserve}(bId, mId) \\
\quad | \\
\quad\quad position(trace, bId, mId) = 1 \Longrightarrow \mathsf{Take}(bId, mId, \_)
\end{array}
$$

C.2.3 Règles d'entrée-sortie

La règle d'entrée-sortie suivante est définie pour l'action DisplayTitle :

RULE R
INPUT DisplayTitle(bId)
OUTPUT $title(t, bId)$
END ;

Quand l'action DisplayTitle est valide, alors la fonction récursive *title* est appelée pour évaluer l'attribut *title* en fonction de la trace courante du SI. Les autres actions EB^3 n'ont pas de sortie prévue.

C.2.4 Définitions d'attributs

Les définitions d'attributs de l'exemple de la bibliothèque sont regroupées par type d'entité et association.

Type d'entité *book*

$bookKey(s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(bookKey\_Set) \triangleq$
match $last(s)$ **with**
$\bot$: $\varnothing$,
Acquire($bId, \_$) : $bookKey(front(s)) \cup \{bId\}$,
Discard(bId) : $bookKey(front(s)) - \{bId\}$,
$\_$: $bookKey(front(s))$;

$title(s : \mathcal{T}(\mathsf{main}), bId : bookKey\_Set) : Title\_Type \triangleq$
match $last(s)$ **with**
$\bot$: $\bot$,
Acquire($bId, bTitle$) : $bTitle$,
Discard(bId) : $\bot$,
Modify($bId, newTitle$) : $newTitle$,
$\_$: $title(front(s), bId)$;

Type d'entité *member*

$memberKey(s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(memberKey\_Set) \triangleq$
match $last(s)$ **with**
$\bot$: $\varnothing$,
Register($mId, \_$) : $memberKey(front(s)) \cup \{mId\}$,
Unregister(mId) : $memberKey(front(s)) - \{mId\}$,
$\_$: $memberKey(front(s))$;

$nbLoans(s : \mathcal{T}(\mathsf{main}), mId : memberKey\_Set) : \mathbb{N} \triangleq$
match $last(s)$ **with**
$\bot$: $\bot$,
Register($mId, \_$) : 0,
Lend($\_, mId, \_$) : $1 + nbLoans(front(s), mId)$,
Return(bId) : **if** $mId = borrower(front(s), bId)$
then $nbLoans(front(s), mId) - 1$ **end**,

$\mathsf{Transfer}(bId, mId', \_)$: **if** $mId = mId'$
then $nbLoans(front(s), mId) + 1$
else if $mId = borrower(front(s), bId)$
then $nbLoans(front(s), mId) - 1$
end
end,
$\mathsf{Take}(\_, mId, \_)$: $1 + nbLoans(front(s), mId)$,
$\mathsf{Unregister}(mId)$: $\bot$,
$\_$: $nbLoans(front(s), mId)$;

$loanDuration(s : \mathcal{T}(\mathsf{main}), mId : memberKey\_Set) : \mathbb{N} \triangleq$
match $last(s)$ **with**
$\bot$: $\bot$,
$\mathsf{Register}(mId, lD)$: lD,
$\mathsf{Unregister}(mId)$: $\bot$,
$\_$: $loanDuration(front(s), mId)$;

Association *loan*

$loan(s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(bookKey\_Set) \triangleq$
match $last(s)$ **with**
$\bot$: $\varnothing$,
$\mathsf{Lend}(bId, \_, \_)$: $loan(front(s)) \cup \{bId\}$,
$\mathsf{Return}(bId)$: $loan(front(s)) - \{bId\}$,
$\mathsf{Take}(bId, \_, \_)$: $loan(front(s)) \cup \{bId\}$,
$\_$: $loan(front(s))$;

$dueDate(s : \mathcal{T}(\mathsf{main}), bId : bookKey\_Set) : DATE \triangleq$
match $last(s)$ **with**
$\bot$: $\bot$,
$\mathsf{Lend}(bId, \_, Permanent)$: $CurrentDate + 365$,
$\mathsf{Lend}(bId, mId, Classic)$: $CurrentDate$
$+ loanDuration(front(s), mId)$,
$\mathsf{Return}(bId)$: $\bot$,
$\mathsf{Transfer}(bId, mId, type)$: **if** $type = Permanent$
then $CurrentDate + 365$
else $CurrentDate$
$+ loanDuration(front(s), mId)$
end,
$\mathsf{Take}(bId, \_, Permanent)$: $CurrentDate + 365$,
$\mathsf{Take}(bId, mId, Classic)$: $CurrentDate$
$+ loanDuration(front(s), mId)$,
$\_$: $dueDate(front(s), bId)$;

Association *reservation*

$reservation(s : \mathcal{T}(\mathsf{main})) : \mathbb{F}(bookKey\_Set \times memberKey\_Set) \triangleq$
match $last(s)$ **with**
$\bot$: $\varnothing$,
$\mathsf{Reserve}(bId, mId)$: $reservation(front(s)) \cup \{(bId, mId)\}$,

$\mathsf{Cancel}(bId, mId)\ :\ reservation(front(s))-\{(bId, mId)\},$
$\mathsf{Take}(bId, mId, \_)\ :\ reservation(front(s))-\{(bId, mId)\},$
$\_\ :\ reservation(front(s))\,;$

$position(s : \mathcal{T}(\mathsf{main}), bId : bookKey\_Set, mId : memberKey\_Set) : \mathbb{N}\ \triangleq$
match $last(s)$ **with**
$\bot\ :\ \bot,$
$\mathsf{Reserve}(bId, mId)\ :\ card(membRes(bId))+1,$
$\mathsf{Cancel}(bId, mId')\ :$ **if** $mId = mId'$
then $\bot$
else if $mId \in membRes(bId) \wedge$
$position(front(s), bId, mId')$
$< position(front(s), bId, mId)$
then $position(front(s), bId, mId)-1$
end
end,
$\mathsf{Take}(bId, mId', \_)\ :$ **if** $mId = mId'$
then $\bot$
else if $mId \in membRes(bId)$
then $position(front(s), bId, mId)-1$
end
end,
$\_\ :\ position(front(s), bId, mId)\,;$

Le rôle *membRes* est défini par :

$$membRes(bId) = \{mId \mid (bId, mId) \in reservation\}$$

Le rôle *borrower* est défini par :

$borrower(s : \mathcal{T}(\mathsf{main}), bId : bookKey\_Set) : memberKey\_Set\ \triangleq$
match $last(s)$ **with**
$\bot\ :\ \bot,$
$\mathsf{Lend}(bId, mId, \_)\ :\ mId,$
$\mathsf{Return}(bId)\ :\ \bot,$
$\mathsf{Transfer}(bId, mId, \_)\ :\ mId,$
$\mathsf{Take}(bId, mId, \_)\ :\ mId,$
$\_\ :\ borrower(front(s), bId)\,;$

C.3 Traduction en B

La machine B générée en appliquant les algorithmes de traduction décrits dans le chapitre 6 sur l'exemple de la bibliothèque est la suivante :

MACHINE *B_Library*
SEES ... /* machine B qui définit les opérations sur le type DATE */
SETS
memberKey_Set ; *bookKey_Set* ; *Title_Type* ;
Loan_Type = {*Permanent*, *Classic*}
VARIABLES

$memberKey$, $nbLoans$, $loanDuration$, $bookKey$, $title$, $loan$,
$dueDate$, $reservation$, $position$
CONSTANTS
$maxNbLoans$
PROPERTIES
$maxNbLoans \in NAT$
INVARIANT
$memberKey \subseteq memberKey\_Set \ \wedge \ nbLoans \in memberKey \rightarrow NAT \ \wedge$
$loanDuration \in memberKey \rightarrow NAT \ \wedge \ bookKey \subseteq bookKey\_Set \ \wedge$
$title \in bookKey \nrightarrow Title\_Type \ \wedge \ loan \in bookKey \nrightarrow memberKey \ \wedge$
$dueDate \in loan \rightarrow DATE \ \wedge \ reservation \in bookKey \leftrightarrow memberKey \ \wedge$
$position \in reservation \rightarrow NAT$
DEFINITION
$borrower(x) \triangleq loan(x)$;
$membRes(x) \triangleq reservation[\{x\}]$
INITIALISATION
$memberKey, nbLoans, loanDuration, bookKey, title, loan, dueDate,$
$reservation, position := \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing$

OPERATIONS
$res \longleftarrow$ **Acquire**$(bId, bTitle) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ bTitle \in Title\_Type$
THEN
IF $CS1 \wedge CD1$ THEN
$bookKey := bookKey \cup \{bId\} \ ||$
$title := title \lhd\!\!+ \{bId \mapsto bTitle\} \ ||$
$res :=$ "ok"
ELSE
$res :=$ "$error$"
END
END;

$res \longleftarrow$ **Discard**$(bId) \triangleq$
PRE $bId \in bookKey\_Set$
THEN
IF $CS2 \wedge CD2$ THEN
$bookKey := bookKey - \{bId\} \ ||$
$title := \{bId\} \lhd title \ ||$
$res :=$ "ok"
ELSE
$res :=$ "$error$"
END
END;

$res \longleftarrow$ **Modify**$(bId, newTitle) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ newTitle \in Title\_Type$
THEN
IF $CS3 \wedge CD3$ THEN
$title := title \lhd\!\!+ \{bId \mapsto newTitle\} \ ||$

res := "ok"
ELSE
res := "$error$"
END
END ;

$res \longleftarrow$ **Register**$(mId, lD) \triangleq$
PRE $mId \in memberKey\_Set \ \wedge \ lD \in \mathbb{N}$
THEN
IF $CS4 \wedge CD4$ THEN
$memberKey := memberKey \cup \{mId\}$ ||
$nbLoans := nbLoans \leftarrow \{mId \mapsto 0\}$ ||
$loanDuration := loanDuration \leftarrow \{mId \mapsto lD\}$ ||
res := "ok"
ELSE
res := "$error$"
END
END ;

$res \longleftarrow$ **Unregister**$(mId) \triangleq$
PRE $mId \in memberKey\_Set$
THEN
IF $CS5 \wedge CD5$ THEN
$memberKey := memberKey - \{mId\}$ ||
$nbLoans := \{mId\} \lhd nbLoans$ ||
$loanDuration := \{mId\} \lhd loanDuration$ ||
res := "ok"
ELSE
res := "$error$"
END
END ;

$res \longleftarrow$ **Lend**$(bId, mId, type) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ mId \in memberKey\_Set$
$\wedge \ type \in Loan\_Type$
THEN
IF $CS6 \wedge CD6$ THEN
$nbLoans := nbLoans \leftarrow \{mId \mapsto 1{+}nbLoans(mId)\}$ ||
$loan := loan \cup \{(bId, mId)\}$ ||
IF $type = Permanent$
THEN $dueDate := dueDate$
$\leftarrow \{(bId, borrower(bId)) \mapsto CurrentDate{+}365\}$
ELSE $dueDate := dueDate$
$\leftarrow \{(bId, mId) \mapsto CurrentDate{+}loanDuration(mId)\}$
END ||
res := "ok"
ELSE
res := "$error$"

END
END ;

$res \longleftarrow$ **Return**$(bId) \triangleq$
PRE $bId \in bookKey\_Set$
THEN
IF $CS7 \wedge CD7$ THEN
$nbLoans := nbLoans$
$\Leftarrow \{borrower(bId) \mapsto nbLoans(borrower(bId))-1\}$ ||
$loan := loan-\{(bId, borrower(bId))\}$ ||
$dueDate := \{(bId, borrower(bId))\} \lhd dueDate$ ||
$res :=$ "ok"
ELSE
$res :=$ "$error$"
END
END ;

$res \longleftarrow$ **Transfer**$(bId, mId, type) \triangleq$
PRE $bId \in bookKey\_Set \wedge mId \in memberKey\_Set$
$\wedge\ type \in Loan\_Type$
THEN
IF $CS8 \wedge CD8$ THEN
$nbLoans := nbLoans \Leftarrow \{(mId \mapsto nbLoans(mId)+1),$
$(borrower(bId) \mapsto nbLoans(borrower(bId))-1)\}$ ||
$loan := loan \Leftarrow \{(bId, mId)\}$ ||
IF $type = Permanent$
THEN $dueDate := dueDate$
$\Leftarrow \{(bId, borrower(bId)) \mapsto CurrentDate+365\}$
ELSE $dueDate := dueDate$
$\Leftarrow \{(bId, borrower(bId)) \mapsto CurrentDate+loanDuration(mId)\}$
END ||
$res :=$ "ok"
ELSE
$res :=$ "$error$"
END
END ;

$res \longleftarrow$ **Reserve**$(bId, mId) \triangleq$
PRE $bId \in bookKey\_Set \wedge mId \in memberKey\_Set$
THEN
IF $CS9 \wedge CD9$ THEN
$reservation := reservation \cup \{(bId, mId)\}$ ||
$position := position \Leftarrow \{(bId, mId) \mapsto card(membRes(bId))+1\}$ ||
$res :=$ "ok"
ELSE
$res :=$ "$error$"
END
END ;

$res \longleftarrow$ **Take**$(bId, mId, type) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ mId \in memberKey\_Set$
$\wedge \ type \in Loan\_Type$
THEN
IF $CS10 \wedge CD10$ THEN
$nbLoans := nbLoans \lhd\!\!+ \{mId \mapsto nbLoans(mId)+1\} \ ||$
$loan := loan \cup \{(bId, mId)\} \ ||$
IF $type = Permanent$
THEN $dueDate := dueDate$
$\lhd\!\!+ \{(bId, borrower(bId)) \mapsto CurrentDate+365\}$
ELSE $dueDate := dueDate$
$\lhd\!\!+ \{(bId, mId) \mapsto CurrentDate+loanDuration(mId)\}$
END $||$
$reservation := reservation - \{(bId, mId)\} \ ||$
$position := (\{(bId, mId)\} \lhd position) \lhd\!\!+ \{(c, d) \mid \exists\, m \cdot c = (bId, m)$
$\wedge \ m \neq mId \wedge m \in membRes(bId) \wedge d = position(bId, m)-1\} \ ||$
$res :=$ "ok"
ELSE
$res :=$ "$error$"
END
END ;

$res \longleftarrow$ **Cancel**$(bId, mId) \triangleq$
PRE $bId \in bookKey\_Set \ \wedge \ mId \in memberKey\_Set$
THEN
IF $CS11 \wedge CD11$ THEN
$reservation := reservation-\{(bId, mId)\} \ ||$
$position := (\{(bId, mId)\} \lhd position) \lhd\!\!+ \{(c, d) \mid \exists\, m \cdot c = (bId, m)$
$\wedge \ m \neq mId \wedge m \in membRes(bId)$
$\wedge \ position(bId, mId) < position(bId, m)$
$\wedge \ d = position(bId, m)-1\} \ ||$
$res :=$ "ok"
ELSE
$res :=$ "$error$"
END
END
END

C.3.1 Conditions *CS*

Dans cette section, nous décrivons les conditions *CS* obtenues pour chaque opération, en appliquant les règles définies dans les sections 7.2.1 et 7.2.2.1.

- Opération **Acquire** : la condition $CS1$ est $true$.
- Opération **Discard** : la condition $CS2$ est $bId \notin dom(loan)$.
- Opération **Modify** : la condition $CS3$ est $true$.
- Opération **Register** : la condition $CS4$ est $true$.
- Opération **Unregister** : la condition $CS5$ est $mId \notin ran(loan)$.
- Opération **Lend** : la condition $CS6$ est
$nbLoans(mId) < maxNbLoans \wedge membRes(bId) = \varnothing \wedge$

$bId \in bookKey \wedge bId \notin dom(loan) \wedge mId \in memberKey$

- Opération **Return** : la condition $CS7$ est $nbLoans(borrower(bId)) \geq 1$.
- Opération **Transfer** : la condition $CS8$ est
  $nbLoans(mId) < maxNbLoans \wedge$
  $mId \neq borrower(bId) \wedge membRes(bId) = \varnothing \wedge$
  $bId \in bookKey \wedge bId \in dom(loan) \wedge$
  $nbLoans(borrower(bId)) \geq 1 \wedge mId \in memberKey$
- Opération **Reserve** : la condition $CS9$ est
  $mId \neq borrower(bId) \wedge bId \in dom(loan) \wedge$
  $bId \in bookKey \wedge mId \in memberKey$
- Opération **Take** : la condition $CS10$ est
  $position(bId, mId) = 1$
  $bId \in bookKey \wedge bId \notin dom(loan) \wedge mId \in memberKey$
- Opération **Cancel** : la condition $CS11$ est $true$.

C.3.2 Conditions *CD*

Dans cette section, nous décrivons les conditions CD qui sont déduites à partir des patrons de raffinement décrits dans la section 7.3.2 :

- Opération **Acquire** : la condition $CD1$ est $bId \notin bookKey$.
- Opération **Discard** : la condition $CD2$ est $bId \in bookKey$.
- Opération **Modify** : la condition $CD3$ est $bId \in bookKey$.
- Opération **Register** : la condition $CD4$ est $mId \notin memberKey$.
- Opération **Unregister** : la condition $CD5$ est $mId \in memberKey$.
- Opération **Lend** : la condition $CD6$ est
  $bId \in bookKey \wedge mId \in memberKey \wedge$
  $bId \notin dom(loan)$
- Opération **Return** : la condition $CD7$ est
  $bId \in bookKey \wedge bId \in dom(loan)$
- Opération **Transfer** : la condition $CD8$ est
  $bId \in bookKey \wedge mId \in memberKey \wedge$
  $bId \in dom(loan)$
- Opération **Reserve** : la condition $CD9$ est
  $bId \in bookKey \wedge mId \in memberKey \wedge$
  $(bId, mId) \notin reservation$
- Opération **Take** : la condition $CD10$ est
  $bId \in bookKey \wedge mId \in memberKey \wedge$
  $bId \notin dom(loan) \wedge (bId, mId) \in reservation$
- Opération **Cancel** : la condition $CD11$ est
  $bId \in bookKey \wedge mId \in memberKey \wedge$
  $(bId, mId) \in reservation$

C.4 Synthèse de transactions

Dans cette section, nous présentons les résultats obtenus en appliquant les algorithmes présentés dans les chapitres 9 et 10.

C.4.1 Fichiers source

L'outil EB^{3}TG (voir chapitre 11) a besoin de plusieurs fichiers source pour générer les transactions :

- un fichier XML décrivant le diagramme ER,
- un fichier décrivant la signature des actions,
- un fichier avec les définitions d'attributs.

Pour l'exemple de la bibliothèque, le fichier XML est le suivant :

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE entityModel SYSTEM "ER2SQL.DTD">
 <entityModel>
  <property projectName="EB3 Attribute Definition" modelName="Library"
            description="library model" author="Equipe EB3"
            creationDate="8 september 2006" />
  <entity name="book"  type="strong" label="book entity type" >
   <attribute name="bookKey" type="numeric" size="5" scale="2" key="true"/>
   <attribute name="title" type="varchar" size="20"></attribute>
  </entity>
  <entity name="member" type="strong" label="member entity type" >
   <attribute name="memberKey" type="numeric" size="5"  key="true"/>
   <attribute name="nbLoans" type="numeric" size="5" null="false"/>
   <attribute name="loanDuration" type="numeric" size="3" null="false"/>
  </entity>
 <relation name="loan" label="loan association" >
   <attribute name="dueDate" type="date"/>
   <participant refEntity="member" participation="partial" cardinality="1"
    role="borrower"/>
   <participant refEntity="book" participation="partial" cardinality="N"/>
 </relation>
 <relation  name="reservation" label="reservation association" >
   <attribute name="position" type="numeric" size="5"/>
   <participant refEntity="book" cardinality="N"  participation="partial"/>
   <participant refEntity="member" cardinality="N" participation="partial"
                role="membRes"/>
 </relation>
</entityModel>
```

La signature des actions est la suivante :

```
Acquire(bId:int,bTitle:String):void
Discard(bId:int):void
Modify(bId:int,newTitle:String):void
Register(mId:int,lD:int):void
Unregister(mId:int):void
Lend(bId:int,mId:int,typeOfLoan:String):void
Return(bId:int):void
Transfer(bId:int,mId:int,typeOfLoan:String):void
Reserve(bId:int,mId:int):void
Take(bId:int,mId:int,typeOfLoan:String):void
Cancel(bId:int,mId:int):void
```

L'action DisplayTitle n'est pas prise en compte, car elle n'a aucun effet sur les attributs de la bibliothèque. Enfin, les définitions d'attributs sont décrites par le fichier texte suivant :

```
/**
 * Library attribute definition
 */
```

```
/**
 * entity type book
 */
book.bookKey(s)==
  match last(s) with
  NULL-> {},
  Acquire(bId,_) -> book.bookKey(front(s))\/{bId},
  Discard(bId) -> book.bookKey(front(s))-{bId},
  _      -> book.bookKey(front(s));

book.title(s,bId)==
  match last(s) with
  NULL -> NULL,
  Acquire(bId,bTitle)  -> bTitle,
  Discard(bId)   -> NULL,
  Modify(bId,newTitle) -> newTitle,
  _  -> book.Title(front(s),mId);

/**
 * entity type member
 */
member.memberKey(s) ==
  match last(s) with
  NULL -> {},
  Register(mId,_) -> member.memberKey(front(s))\/{mId},
  Unregister(mId) -> member.memberKey(front(s))-{mId},
  _  -> member.memberKey(front(s));

member.nbLoans(s,mId)==
  match last(s) with
  NULL -> NULL,
  Register(mId,_) -> 0,
  Lend(_,mId,_) ->  1 + member.nbLoans(front(s),mId),
  Return(bId) -> if (mId = book.borrower(front(s),bId))
                 then member.nbLoans(front(s),mId) -1 end,
  Transfer(bId,mId2,_) -> if (mId = mId2) then
              member.nbLoans(front(s),mId) + 1
              else if (mId = book.borrower(front(s), bId)) then
                   member.nbLoans(front(s),mId) - 1
                   end
              end,
  Take(_,mId,_) -> 1 + member.nbLoans(front(s),mId),
  Unregister(mId) -> NULL,
  _ -> member.nbLoans(front(s),mId);

member.loanDuration(s,mId)==
  match last(s) with
  NULL -> NULL,
  Register(mId,lD) -> lD,
  Unregister(mId)  -> NULL,
  _  -> member.loanDuration(front(s),mId);

/**
 * Association loan
 */
loan.loan(s)==
  match last(s) with
  NULL -> {},
  Lend(bId,_,_) ->loan.loan(front(s))\/{bId},
  Return(bId) -> loan.loan(front(s))-{bId},
  Take(bId,_,_) -> loan.loan(front(s))\/{bId},
  _  -> loan.loan(front(s));
```

```
loan.dueDate(s,bId)==
  match last(s) with
  NULL -> NULL,
  Lend(bId,mId,typeOfLoan) ->if (typeOfLoan="Permanent") then
                CURRENTDATE +365
                else if (typeOfLoan="Classic") then
                     CURRENTDATE + member.loanDuration(front(s), mId)
                     end
                end,
  Take(bId,mId,typeOfLoan) -> if (typeOfLoan="Permanent") then
                CURRENTDATE +365
                else if (typeOfLoan="Classic") then
                     CURRENTDATE + member.loanDuration(front(s), mId)
                     end
                end,
  Return(bId) -> NULL,
  Transfer(bId,mId,type) -> if(type="Permanent") then
                CURRENTDATE +365
                else if(type="Classic") then
                     CURRENTDATE + member.loanDuration(front(s), mId)
                     end
                end,
  _ ->  loan.dueDate(front(s),bId);


/**
 * Association reservation
 */
reservation.reservation(s)==
  match last(s) with
  NULL -> {},
  Reserve(bId, mId) ->  reservation.reservation(front(s)) \/ {(bId,mId)},
  Cancel(bId, mId) ->  reservation.reservation(front(s)) - {(bId,mId)},
  Take(bId,mId,_) ->  reservation.reservation(front(s))- {(bId,mId)},
  _ ->  reservation.reservation(front(s));

reservation.position(s,bId,mId)==
  match last(s) with
  NULL -> NULL,
  Reserve(bId,mId) -> card(membRes(bId))+1,
  Cancel(bId,mId2) -> if (mId=mId2) then NULL
               else if (mId:membRes(bId) and
                    (reservation.position(front(s),bId,mId2) <
                    reservation.position(front(s),bId,mId))) then
                         reservation.position(front(s),bId,mId)-1
                    end
               end,
  Take(bId, mId2,_) -> if (mId=mId2) then NULL
               else if(mId:membRes(bId)) then
                    reservation.position(front(s),bId,mId)-1
                    end
               end,
  _ -> reservation.position(front(s),bId,mId);

/**
 * roles
 */
book.borrower(s,bId)==
  match last(s) with
  NULL -> NULL,
  Lend(bId, mId,_) ->  mId,
```

```
  Return(bId)  ->  NULL,
  Transfer(bId,mId,_) -> mId,
  Take(bId,mId,_) -> mId,
  _ -> book.borrower(front(s),bId);

membRes(bId) = {mId | (bId, mId) : reservation};
```

C.4.2 Programmes générés

Le script de création de BD généré par l'outil EB^{3}TG est le suivant, lorsque le SGBD choisi est PostgreSQL :

```
/*****************************************/
/* TABLE book */
/*****************************************/
CREATE TABLE book (
bookKey numeric(5,2),
title varchar(20),
CONSTRAINT PKbook PRIMARY KEY(bookKey)
);

/*****************************************/
/* TABLE member */
/*****************************************/
CREATE TABLE member (
memberKey numeric(5),
nbLoans numeric(5)   NOT NULL,
loanDuration numeric(3)   NOT NULL,
CONSTRAINT PKmember PRIMARY KEY(memberKey)
);

/*****************************************/
/* TABLE loan */
/*****************************************/
CREATE TABLE loan (
borrower numeric(5),
bookKey numeric(5,2),
dueDate date,
CONSTRAINT PKloan PRIMARY KEY(bookKey)
);

/*****************************************/
/* TABLE reservation */
/*****************************************/
CREATE TABLE reservation (
bookKey numeric(5,2),
memberKey numeric(5),
position numeric(5),
CONSTRAINT PKreservation PRIMARY KEY(bookKey,memberKey)
);

/*---------------------------------------*/
/*        FOREIGN KEY CONSTRAINTS        */
/*---------------------------------------*/

ALTER TABLE loan ADD CONSTRAINT FKloan_member FOREIGN KEY (borrower)
REFERENCES member (memberKey) INITIALLY DEFERRED;

ALTER TABLE loan ADD CONSTRAINT FKloan_book FOREIGN KEY (bookKey)
REFERENCES book (bookKey) INITIALLY DEFERRED;

ALTER TABLE reservation ADD CONSTRAINT FKreservation_book FOREIGN KEY (bookKey)
```

```
REFERENCES book (bookKey) INITIALLY DEFERRED;

ALTER TABLE reservation ADD CONSTRAINT FKreservation_member
FOREIGN KEY (memberKey) REFERENCES member (memberKey) INITIALLY DEFERRED;
```

Les transactions suivantes sont créées automatiquement par l'outil afin de gérer les connexions au SGBD. Dans cet exemple, le SGBD est PostgreSQL.

```
import java.sql.*;
import java.util.*;

public class Connexion {

private Connection connection;
private List statements;

/**
* Connection to the database in manual commit mode
* @param serverAddress
* @param dbms
* @param dbName
* @param user
* @param pass
* @throws SQLException
*/
public Connexion (String serverAddress, int port, String dbName,String user, String pass)
  throws SQLException{
  try {
    Driver d = (Driver) Class.forName("org.postgresql.Driver").newInstance();
    DriverManager.registerDriver(d);
    connection = DriverManager.getConnection("jdbc:postgresql:" + dbName,user, pass);
    connection.setAutoCommit(false);
    DatabaseMetaData dbmd = connection.getMetaData();
    if (dbmd.supportsTransactionIsolationLevel(Connection.TRANSACTION_SERIALIZABLE)) {
       connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
       }
    statements = new LinkedList();
  } catch (SQLException e) {
      throw e;
  } catch (Exception e) {
      throw new SQLException("Unable to instanciate JDBC");
  }
}

public Statement createStatement() throws SQLException{
  Statement stmt = connection.createStatement();
  statements.add(stmt);
  return stmt;
}

public void commit() throws SQLException{
  connection.commit();
}

public void rollback() {
  try {
    connection.rollback();
  } catch (SQLException s) {
      System.err.println(s.getMessage());
  }
}

public void closeAllStatements() {
  try {
    Iterator it = statements.iterator();
    while (it.hasNext()) {
      ((Statement) it.next()).close();
    }
    statements.clear();
  } catch (SQLException s) {
      System.err.println(s.getMessage());
  }
```

```
}

}
```

Les transactions générées à partir des définitions d'attributs pour l'exemple de la bibliothèque sont les suivantes, lorsque le SGBD choisi est PostgreSQL :

```
import java.sql.*;

public class Systeme {

private static Connexion connection;

/**
* Connection to the database in manual commit mode
* @param serverAddress
* @param dbms
* @param dbName
* @param user
* @param pass
*/
public static void connect(String serverAddress, int port, String dbName,String user,
String pass){
  try {
    connection = new Connexion(serverAddress, port, dbName, user, pass);
  } catch (SQLException e) {
      System.err.println(e.getMessage());
  }
}

public static void Acquire(int bId,String bTitle){
  try {
    int var0 = connection.createStatement().executeUpdate("UPDATE book SET "+
      "title = '"+bTitle+"' "+
      "WHERE bookKey = "+ bId +" ");
    if( var0==0 ){
      connection.createStatement().executeUpdate("INSERT INTO book ( bookKey,title) "+
        " VALUES ( "+ bId +",'"+bTitle+"')");
    }
    connection.commit();
  } catch ( Exception e ) {
      connection.rollback();
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}

public static void Discard(int bId){
  try {
    connection.createStatement().executeUpdate("DELETE FROM book "+
      "WHERE bookKey = "+ bId +" ");
    connection.commit();
  } catch ( Exception e ) {
      connection.rollback();
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}

public static void Modify(int bId,String newTitle){
  try {
    connection.createStatement().executeUpdate("UPDATE book SET "+
      "title = '"+newTitle+"' "+
      "WHERE bookKey = "+ bId +" ");
    connection.commit();
  } catch ( Exception e ) {
      connection.rollback();
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
```

```
}

public static void Register(int mId,int lD){
  try {
    int var0 = connection.createStatement().executeUpdate("UPDATE member SET "+
      "nbLoans = 0, "+
      "loanDuration = "+lD+" "+
      "WHERE memberKey = "+ mId +" ");
    if( var0==0 ){
      connection.createStatement().executeUpdate("INSERT
        INTO member( memberKey,nbLoans,loanDuration) "+
        " VALUES ( "+ mId +",0,"+lD+")");
    }
    connection.commit();
  } catch ( Exception e ) {
      connection.rollback();
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}

public static void Unregister(int mId){
  try {
    connection.createStatement().executeUpdate("DELETE FROM member "+
      "WHERE memberKey = "+ mId +" ");
    connection.commit();
  } catch ( Exception e ) {
      connection.rollback();
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}

public static void Lend(int bId,int mId,String typeOfLoan){
  try {
    ResultSet rset0 = connection.createStatement().executeQuery("SELECT A.nbLoans "+
      "FROM member A "+
      "WHERE A.memberKey = "+ mId+ " ");
    String var0 = ((rset0.next())?rset0.getDouble(1)+"":"null");
    ResultSet rset1 = connection.createStatement().executeQuery("SELECT D.loanDuration "+
      "FROM member D "+
      "WHERE D.memberKey = "+ mId+ " ");
    String var1 = ((rset1.next())?rset1.getInt(1)+"":"null");
    int var2 = connection.createStatement().executeUpdate("UPDATE loan SET "+
      "borrower = "+mId+" "+
      "WHERE bookKey = "+ bId +" ");
    if( var2==0 ){
      connection.createStatement().executeUpdate("INSERT INTO loan ( bookKey,borrower) "+
        " VALUES ( "+ bId +","+mId+")");
    }
    if ( typeOfLoan=="Permanent" ) {
      int var3 = connection.createStatement().executeUpdate("UPDATE loan SET "+
      "dueDate = CURRENT_DATE+365 "+
      "WHERE bookKey = "+ bId +" ");
      if(var3 == 0){
        connection.createStatement().executeUpdate("INSERT INTO loan ( bookKey,dueDate) "+
        " VALUES ( "+ bId +",CURRENT_DATE+365)");
      }
    } else if( typeOfLoan=="Classic" ) {
      int var4 = connection.createStatement().executeUpdate("UPDATE loan SET "+
      "dueDate = CURRENT_DATE+"+var1+" "+
      "WHERE bookKey = "+ bId +" ");
      if(var4 == 0){
        connection.createStatement().executeUpdate("INSERT INTO loan ( bookKey,dueDate) "+
        " VALUES ( "+ bId +",CURRENT_DATE+"+var1+")");
      }
    }
    connection.createStatement().executeUpdate("UPDATE member SET "+
      "nbLoans = 1+"+var0+" "+
      "WHERE memberKey = "+ mId +" ");
    connection.commit();
  } catch ( Exception e ) {
```

```
      connection.rollback();
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}

public static void Return(int bId){
  try {
    ResultSet rset0 = connection.createStatement().executeQuery("SELECT C.borrower,
      A.nbLoans-1 "+
      "FROM loan C,member A "+
      "WHERE C.bookKey = "+ bId+ " "+
      "AND A.memberKey = C.borrower ");
    connection.createStatement().executeUpdate("DELETE FROM loan "+
      "WHERE bookKey = "+ bId +" ");
    while(rset0.next()) {
      connection.createStatement().executeUpdate("UPDATE member SET nbLoans =
        "+rset0.getDouble(2)+ " "+
        "WHERE memberKey = "+ rset0.getDouble(1)+ " ");
    }
    connection.commit();
  } catch ( Exception e ) {
      connection.rollback();
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}

public static void Transfer(int bId,int mId,String typeOfLoan){
  try {
    //create a temporary table eb3Tempmember
    connection.createStatement().executeUpdate("CREATE TABLE eb3Tempmember ( "+
      "memberKey  numeric(5))");
    //Insert parameters in the temporary table eb3Tempmember
    connection.createStatement().executeUpdate("INSERT INTO  eb3Tempmember (memberKey)
      values("+mId+")");
    //end temporary table eb3Tempmember

    ResultSet rset0 = connection.createStatement().executeQuery("SELECT C.memberKey,
      A.nbLoans+1 "+
      "FROM eb3Tempmember C,member A "+
      "WHERE C.memberKey =  "+ mId+ " "+
      "AND A.memberKey = C.memberKey ");
    ResultSet rset1 = connection.createStatement().executeQuery("SELECT G.borrower,
      E.nbLoans-1 "+
      "FROM loan G,member E "+
      "WHERE G.bookKey = "+ bId+ " "+
      "AND G.borrower NOT IN ( "+
      "SELECT C.memberKey "+
      "FROM eb3Tempmember C "+
      "WHERE C.memberKey =  "+ mId+ " ) "+
      "AND E.memberKey = G.borrower ");
    ResultSet rset2 = connection.createStatement().executeQuery("SELECT D.loanDuration "+
      "FROM member D "+
      "WHERE D.memberKey = "+ mId+ " ");
    String var0 = ((rset2.next())?rset2.getInt(1)+"":"null");
    connection.createStatement().executeUpdate("UPDATE loan SET "+
      "borrower = "+mId+" "+
      "WHERE bookKey = "+ bId +" ");
    if ( typeOfLoan=="Permanent" ) {
      connection.createStatement().executeUpdate("UPDATE loan SET "+
      "dueDate = CURRENT_DATE+365 "+
      "WHERE bookKey = "+ bId +" ");
    } else if( typeOfLoan=="Classic" ) {
      connection.createStatement().executeUpdate("UPDATE loan SET "+
      "dueDate = CURRENT_DATE+"+var0+" "+
      "WHERE bookKey = "+ bId +" ");
    }
    while(rset0.next()) {
      connection.createStatement().executeUpdate("UPDATE member SET nbLoans =
        "+rset0.getDouble(2)+ " "+
        "WHERE memberKey = "+ rset0.getDouble(1)+ " ");
```

```
    }
    while(rset1.next()) {
      connection.createStatement().executeUpdate("UPDATE member SET nbLoans =
        "+rset1.getDouble(2)+ " "+
        "WHERE memberKey = "+ rset1.getDouble(1)+ " ");
    }
    connection.createStatement().executeUpdate("DROP TABLE eb3Tempmember");
    connection.commit();
  } catch ( Exception e ) {
    try{
      connection.createStatement().executeUpdate("DROP TABLE eb3Tempmember");
      connection.rollback();
    } catch (SQLException s){
       System.err.println(s.getMessage());
    }
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}

public static void Reserve(int bId,int mId){
  try {
    ResultSet rset0 = connection.createStatement().executeQuery("SELECT
      count(A.bookKey) "+
      "FROM reservation A "+
      "WHERE A.bookKey = "+ bId+ " ");
    String var0 = ((rset0.next())?rset0.getDouble(1)+"":"null");
    int var1 = connection.createStatement().executeUpdate("UPDATE reservation SET "+
      "position = "+var0+"+1 "+
      "WHERE bookKey = "+ bId +" "+
      "AND memberKey = "+ mId +" ");
    if( var1==0 ){
      connection.createStatement().executeUpdate("INSERT INTO reservation
        (bookKey,memberKey,position) "+
        " VALUES ( "+ bId +","+ mId +","+var0+"+1)");
    }
    connection.commit();
  } catch ( Exception e ) {
      connection.rollback();
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}

public static void Take(int bId,int mId,String typeOfLoan){
  try {
    //create a temporary table eb3Tempreservation
    connection.createStatement().executeUpdate("CREATE TABLE eb3Tempreservation ( "+
      "bookKey  numeric(5,2),"+
      "memberKey  numeric(5))");
    //Insert parameters in the temporary table eb3Tempreservation
    connection.createStatement().executeUpdate("INSERT INTO  eb3Tempreservation (memberKey)
      values("+mId+")");
    //end temporary table eb3Tempreservation

    ResultSet rset0 = connection.createStatement().executeQuery("SELECT A.nbLoans "+
      "FROM member A "+
      "WHERE A.memberKey = "+ mId+ " ");
    String var0 = ((rset0.next())?rset0.getDouble(1)+"":"null");
    ResultSet rset1 = connection.createStatement().executeQuery("SELECT D.loanDuration "+
      "FROM member D "+
      "WHERE D.memberKey = "+ mId+ " ");
    String var1 = ((rset1.next())?rset1.getInt(1)+"":"null");
    ResultSet rset2 = connection.createStatement().executeQuery("SELECT "+bId+",
      F.memberKey,D.position-1 "+
      "FROM reservation F,reservation D "+
      "WHERE F.bookKey = "+ bId+ " "+
      "AND ("+bId+",F.memberKey) NOT IN ( "+
      "SELECT "+bId+",B.memberKey "+
      "FROM eb3Tempreservation B "+
      "WHERE B.memberKey =  "+ mId+ " ) "+
      "AND D.bookKey = "+ bId+ " "+
```

```
      "AND D.memberKey = F.memberKey ");
    ResultSet rset3 = connection.createStatement().executeQuery("SELECT "+bId+",
      B.memberKey,NULL "+
      "FROM eb3Tempreservation B "+
      "WHERE B.memberKey =  "+ mId+ " ");
    connection.createStatement().executeUpdate("DELETE FROM reservation "+
      "WHERE bookKey = "+ bId +" "+
      "AND memberKey = "+ mId +" ");
    int var2 = connection.createStatement().executeUpdate("UPDATE loan SET "+
      "borrower = "+mId+" "+
      "WHERE bookKey = "+ bId +" ");
    if( var2==0 ){
      connection.createStatement().executeUpdate("INSERT INTO loan ( bookKey,borrower) "+
        " VALUES ( "+ bId +","+mId+")");
    }
    if ( typeOfLoan=="Permanent" ) {
      int var3 = connection.createStatement().executeUpdate("UPDATE loan SET "+
      "dueDate = CURRENT_DATE+365 "+
      "WHERE bookKey = "+ bId +" ");
      if(var3 == 0){
        connection.createStatement().executeUpdate("INSERT INTO loan ( bookKey,dueDate) "+
        " VALUES ( "+ bId +",CURRENT_DATE+365)");
      }
    } else if( typeOfLoan=="Classic" ) {
      int var4 = connection.createStatement().executeUpdate("UPDATE loan SET "+
      "dueDate = CURRENT_DATE+"+var1+" "+
      "WHERE bookKey = "+ bId +" ");
      if(var4 == 0){
        connection.createStatement().executeUpdate("INSERT INTO loan ( bookKey,dueDate) "+
        " VALUES ( "+ bId +",CURRENT_DATE+"+var1+")");
      }
    }
    connection.createStatement().executeUpdate("UPDATE member SET "+
      "nbLoans = 1+"+var0+" "+
      "WHERE memberKey = "+ mId +" ");
    while(rset2.next()) {
      connection.createStatement().executeUpdate("UPDATE reservation SET position =
        "+rset2.getDouble(3)+ " "+
        "WHERE bookKey = "+ rset2.getDouble(1)+ " "+
        "AND memberKey = "+ rset2.getDouble(2)+ " ");
    }
    while(rset3.next()) {
      connection.createStatement().executeUpdate("UPDATE reservation SET position =
        "+rset3.getDouble(3)+ " "+
        "WHERE bookKey = "+ rset3.getDouble(1)+ " "+
        "AND memberKey = "+ rset3.getDouble(2)+ " ");
    }
    connection.createStatement().executeUpdate("DROP TABLE eb3Tempreservation");
    connection.commit();
  } catch ( Exception e ) {
    try{
      connection.createStatement().executeUpdate("DROP TABLE eb3Tempreservation");
      connection.rollback();
    } catch (SQLException s){
        System.err.println(s.getMessage());
    }
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}

public static void Cancel(int bId,int mId){
  try {
    //create a temporary table eb3Tempreservation
    connection.createStatement().executeUpdate("CREATE TABLE eb3Tempreservation ( "+
      "bookKey  numeric(5,2),"+
      "memberKey  numeric(5))");
    //Insert parameters in the temporary table eb3Tempreservation
    connection.createStatement().executeUpdate("INSERT INTO  eb3Tempreservation (memberKey)
      values("+mId+")");
    //end temporary table eb3Tempreservation

    ResultSet rset0 = connection.createStatement().executeQuery("SELECT "+bId+",
```

```
      G.memberKey,D.position-1 "+
      "FROM reservation G,reservation D "+
      "WHERE  G.memberKey IN (SELECT G.memberKey "+
      "FROM reservation F,reservation G "+
      "WHERE F.bookKey = "+ bId+ " "+
      "AND F.memberKey = "+ mId+ " "+
      "AND G.bookKey = "+ bId+ " "+
      "AND F.position < G.position ) "+
      "AND  G.memberKey IN (SELECT H.memberKey "+
      "FROM reservation H "+
      "WHERE H.bookKey = "+ bId+ " ) "+
      "AND ("+bId+",G.memberKey) NOT IN ( "+
      "SELECT "+bId+",B.memberKey "+
      "FROM eb3Tempreservation B "+
      "WHERE B.memberKey =  "+ mId+ " ) "+
      "AND D.bookKey = "+ bId+ " "+
      "AND D.memberKey = G.memberKey ");
    ResultSet rset1 = connection.createStatement().executeQuery("SELECT "+bId+",
      B.memberKey,NULL "+
      "FROM eb3Tempreservation B "+
      "WHERE B.memberKey =  "+ mId+ " ");
    connection.createStatement().executeUpdate("DELETE FROM reservation "+
      "WHERE bookKey = "+ bId +" "+
      "AND memberKey = "+ mId +" ");
    while(rset0.next()) {
      connection.createStatement().executeUpdate("UPDATE reservation SET position =
        "+rset0.getDouble(3)+ " "+
        "WHERE bookKey = "+ rset0.getDouble(1)+ " "+
        "AND memberKey = "+ rset0.getDouble(2)+ " ");
    }
    while(rset1.next()) {
      connection.createStatement().executeUpdate("UPDATE reservation SET position =
        "+rset1.getDouble(3)+ " "+
        "WHERE bookKey = "+ rset1.getDouble(1)+ " "+
        "AND memberKey = "+ rset1.getDouble(2)+ " ");
    }
    connection.createStatement().executeUpdate("DROP TABLE eb3Tempreservation");
    connection.commit();
  } catch ( Exception e ) {
    try{
      connection.createStatement().executeUpdate("DROP TABLE eb3Tempreservation");
      connection.rollback();
    } catch (SQLException s){
        System.err.println(s.getMessage());
      }
      System.err.println(e.getMessage());
    } finally {
      connection.closeAllStatements();
    }
}

public static void createTables(){
  try {
    connection.createStatement().executeUpdate("CREATE TABLE book ( "+
      "bookKey numeric(5,2), "+
      "title varchar(20), "+
      "CONSTRAINT PKbook PRIMARY KEY(bookKey))");
    connection.createStatement().executeUpdate("CREATE TABLE member ( "+
      "memberKey numeric(5), "+
      "nbLoans numeric(5) NOT NULL, "+
      "loanDuration numeric(3) NOT NULL, "+
      "CONSTRAINT PKmember PRIMARY KEY(memberKey))");
    connection.createStatement().executeUpdate("CREATE TABLE loan ( "+
      "borrower numeric(5), "+
      "bookKey numeric(5,2), "+
      "dueDate date, "+
      "CONSTRAINT PKloan PRIMARY KEY(bookKey))");
    connection.createStatement().executeUpdate("CREATE TABLE reservation ( "+
      "bookKey numeric(5,2), "+
      "memberKey numeric(5), "+
      "position numeric(5), "+
      "CONSTRAINT PKreservation PRIMARY KEY(bookKey,memberKey))");
    connection.createStatement().executeUpdate("ALTER TABLE loan ADD CONSTRAINT
      FKloan_member FOREIGN KEY (borrower) REFERENCES member (memberKey)
```

```
      INITIALLY DEFERRED ");
    connection.createStatement().executeUpdate("ALTER TABLE loan ADD CONSTRAINT
      FKloan_book FOREIGN KEY (bookKey) REFERENCES book (bookKey)
      INITIALLY DEFERRED ");
    connection.createStatement().executeUpdate("ALTER TABLE reservation ADD CONSTRAINT
      FKreservation_book FOREIGN KEY (bookKey) REFERENCES book (bookKey)
      INITIALLY DEFERRED ");
    connection.createStatement().executeUpdate("ALTER TABLE reservation ADD CONSTRAINT
      FKreservation_member FOREIGN KEY (memberKey) REFERENCES member (memberKey)
      INITIALLY DEFERRED ");
  } catch ( Exception e ) {
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}

public static void initTables(){
  try {
    connection.commit();
  } catch ( Exception e ) {
      connection.rollback();
      System.err.println(e.getMessage());
  } finally {
      connection.closeAllStatements();
  }
}
}
```

Annexe D

Liste des publications

Chapitre de livre

- F. Gervais, M. Frappier, R. St-Denis : EB^3. In *Software Specification Methods*, ISTE, ISBN : 1-905209-34-7, chapitre 14, pp. 259-274, 2006.

Conférences internationales avec comité de sélection et actes

- S. Blazy, F. Gervais, R. Laleau : Reuse of specification patterns with the B method. In *ZB2003 : Formal Specification and Development in Z and B, 3rd International Conference of B and Z Users*, Turku, Finlande, 4-6 Juin. Springer-Verlag, LNCS 2651, pp. 40-57, 2003.
- F. Gervais, M. Frappier, R. Laleau : Generating relational database transactions from recursive functions defined on EB^3 traces. In *3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, Coblence, Allemagne, 7-9 Septembre. IEEE Computer Society Press, pp. 117-126, 2005.
- F. Gervais, M. Frappier, R. Laleau : Synthesizing B specifications from EB^3 attribute definitions. In *5th International Conference on Integrated Formal Methods (IFM 2005)*, Eindhoven, Pays-Bas, 29 Novembre - 2 Décembre. Springer-Verlag, LNCS 3771, pp. 207-226, 2005.
- F. Gervais, P. Batanado, M. Frappier, R. Laleau : EB^{3}TG : A tool synthesizing relational database transactions from EB^3 attribute definitions. In *8th International Conference on Enterprise Information Systems (ICEIS 2006)*, Paphos, Chypre, 24-27 Mai. INSTICC Press, Volume Information Systems Analysis and Specification, pp. 44-51, 2006.
- F. Gervais, M. Frappier, R. Laleau : Refinement of EB^3 process patterns into B specifications. In *7th International B Conference (B 2007)*, Besançon, France, 17-19 Janvier. Springer-Verlag, LNCS 4355, pp. 201-215, 2007.

Workshop international avec comité de sélection et actes

- F. Gervais, M. Frappier, R. Laleau : How to synthesize relational database transactions from EB^3 attribute definitions ? In *3rd International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS 2005)*, Miami, USA, 24-25 Mai. INSTICC Press, pp. 83-88, 2005.

Conférences nationales avec comité de sélection et actes

- F. Gervais, P. Batanado, M. Frappier, R. Laleau : Génération automatique de transactions de base de données relationnelle à partir de définitions d'attributs EB^3. In *Atelier Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2006)*, Paris, France, 15-17 Mars. Rapport ENST S 002, pp. 25-39, 2006.
- A. Mammar, F. Gervais, R. Laleau : Systematic identification of preconditions from set-based integrity constraints. In *24ème Congrès INFORSID*, Hammamet, Tunisie, 1-3 Juin. INFORSID, Volume II, pp. 595-610, 2006.
- F. Gervais : EB^4 : Vers une méthode de spécification formelle des systèmes d'information. In *24ème Congrès INFORSID*, Hammamet, Tunisie, 1-3 Juin. INFORSID, Volume II, pp. 561-576, 2006.

Workshops nationaux avec sélection sur la base d'un résumé

- S. Blazy, F. Gervais, R. Laleau : Une démarche outillée pour spécifier formellement des patrons de conception réutilisables. In *Workshop Objets, Composants et Modèles dans l'ingénierie des SI (OCM-SI 2003)*, Nancy, France, 3 Juin. INFORSID, pp. 5-9, 2003.
- P. Batanado, F. Gervais, M. Frappier, R. Laleau : EB^{3}TG : Un outil de génération de transactions de base de données relationnelle pour EB^3. *Session Outils de l'Atelier AFADL 2006*, Paris, France, 15-17 Mars 2006.

Rapports techniques

- F. Gervais : *Réutilisation de composants de spécification en B.* Rapport de stage, DEA Informatique, Université d'Évry, Juillet 2002.
- S. Blazy, F. Gervais, R. Laleau : Un exemple de réutilisation de patterns de spécification avec la méthode B. Rapport technique 395, CEDRIC, Évry, Novembre 2002.
- F. Gervais : *EB^4 : Vers une méthode combinée de spécification formelle des systèmes d'information.* Examen de spécialité, Doctorat Informatique, Université de Sherbrooke (Québec), Juin 2004.

- F. Gervais, M. Frappier, R. Laleau : Synthesizing B substitutions for EB^3 attribute definitions. Rapport technique 683, CEDRIC, Évry, Novembre 2004.
- F. Gervais, M. Frappier, R. Laleau, P. Batanado : EB^3 attribute definitions : Formal language and application. Rapport technique 700, CEDRIC, Évry, Février 2005.
- F. Gervais, M. Frappier, R. Laleau : Vous avez dit raffinement ? Rapport technique 829, CEDRIC, Évry, Mars 2005.
- A. Mammar, F. Gervais, R. Laleau : Generating B preconditions from typical IS invariants. Rapport technique, LASSY, Université du Luxembourg, Luxembourg, Février 2006.
- F. Gervais, M. Frappier, R. Laleau : Defining and proving B preconditions for the patterns of EB^3 process expressions. Rapport technique 996, CEDRIC, Évry, Février 2006.

Présentations

- *Réutilisation de patterns de spécification avec la méthode B.* Séminaire DMI, Université de Sherbrooke, Québec, Canada, 15 mai 2003.
- *Reuse of specification patterns with the B method.* ZB 2003, Turku, Finlande, 4 juin 2003.
- *EB^4 : Vers une méthode combinée de spécification formelle des systèmes d'information.* Soutenance Examen de spécialité, Université de Sherbrooke, Québec, 30 juin 2004.
- *Combinaisons de spécifications formelles orientées états/événements — Application aux SI.* Journée AFADL, ENST, Paris, 3 décembre 2004.
- *EB^4 — Vers une méthode combinée de spécification formelle des systèmes d'information.* Séminaire LACL, Université Paris 12, Créteil, 14 mars 2005.
- *EB^4 : Towards an integrated formal method for specifying information systems.* Session poster, Conférence ZB 2005, Guildford, Royaume-Uni, 13-15 avril 2005.
- *How to synthesize relational database transactions from EB^3 attribute definitions ?* MSVVEIS 2005, Miami, USA, 24 mai 2005.
- *Generating relational database transactions from recursive functions defined on EB^3 traces.* SEFM 2005, Coblence, Allemagne, 7 septembre 2005.
- *Synthesizing B specifications from EB^3 attribute definitions.* IFM 2005, Eindhoven, Pays-Bas, 1 décembre 2005.
- *Génération automatique de transactions de base de données relationnelle à partir de définitions d'attributs EB^3.* AFADL 2006, Paris, France, 15 mars 2006.
- *EB^{3}TG : Un outil de génération de transactions de base de données relationnelle pour EB^3.* Session Outils, Conférence AFADL 2006, Paris, France, 15 mars 2006.
- *EB^{3}TG : A tool synthesizing relational database transactions from EB^3 attribute definitions.* ICEIS 2006, Paphos, Chypre, 25 mai 2006.
- *EB^4 : Vers une méthode de spécification formelle des systèmes d'information.* INFORSID 2006, Hammamet, Tunisie, 2 juin 2006.

Résumé

L'objectif de cette thèse est de profiter des avantages de deux formes de modélisation complémentaires pour représenter de manière formelle les systèmes d'information (SI). Un SI est un système informatisé qui permet de rassembler les informations d'une organisation et qui fournit des opérations pour les manipuler. Les SI considérés sont développés autour de systèmes de gestion de bases de données (SGBD). Notre motivation est d'utiliser des notations et des techniques formelles pour les concevoir, contrairement aux méthodes actuelles qui sont au mieux semi-formelles. D'une part, EB^3 est un langage formel basé sur les traces d'événements qui a été défini pour la spécification des SI. En particulier, EB^3 met en avant le comportement dynamique du système. D'autre part, B est un langage formel basé sur les états qui se prête bien à la spécification des propriétés statiques des SI. Nous avons défini une nouvelle approche, appelée EB^4, qui bénéficie à la fois des avantages d'EB^3 et B. Dans un premier temps, les processus décrits en EB^3 sont utilisés pour représenter et pour valider le comportement du système. Ensuite, la spécification est traduite en B pour spécifier et vérifier les principales propriétés statiques du SI. Enfin, nous avons défini des techniques de synthèse automatique de transactions BD relationnelles à partir du modèle de données d'EB^3 pour compléter le cycle de développement du SI.

Mots-clé Systèmes d'information, Spécification formelle, Combinaison de langages, Vérification, Synthèse de programmes.

Abstract

The objective in this thesis is to benefit from two complementary kinds of modelling in order to formally specify information systems (IS). An IS is a computer system that collects data of an organization and that provides useful operations to manage them. The IS we consider are supported by database management systems (DBMS). Our aim is to use only formal notations and techniques to specify such systems, contrary to current methodologies that are based on semi-formal notations. On one hand, EB^3 is a trace-based formal language specially created for the specification of IS. In particular, EB^3 points out the dynamic behaviour of the system. On the other hand, B is a state-based formal language well adapted for the specification of the IS static properties. We have defined a new approach called EB^4 that integrates both EB^3 and B to specify IS. Process expressions described in EB^3 are first used to represent and validate the behaviour of the system. Then, the specification is translated into B in order to specify and verify the main static properties of the IS. For the implementation, we have defined a set of translation rules to automatically synthesize relational DB transactions from the data model specified in EB^3.

Keywords Information systems, Formal specification, Coupling of languages, Verification, Program synthesis.