N° d'Ordre : 1401

THESE
présentée devant

L'UNIVERSITE DE RENNES I
Institut de Formation Supérieure en Informatique et Communication

pour obtenir
le titre de DOCTEUR DE L'UNIVERSITE DE RENNES I
Mention Informatique

par
Pascal ANDRE

Sujet de la thèse

Méthodes formelles et à objets pour le développement du logiciel : Etudes et propositions.

Soutenue le 7 Juillet 1995 devant la Commission d'Examen

MM.	Daniel	HERMAN	Président
	Jean-Claude	DERNIAME	Rapporteur
	Christine	CHOPPY	Rapporteur
	Paul	LE GUERNIC	Examinateur
	Jean	BEZIVIN	Examinateur
	Jean-Claude	ROYER	Examinateur

N^o d'Ordre : 1401

THESE
présentée devant

L'UNIVERSITE DE RENNES I
Institut de Formation Supérieure en Informatique et Communication

pour obtenir
le titre de DOCTEUR DE L'UNIVERSITE DE RENNES I
Mention Informatique

par
Pascal ANDRE

Sujet de la thèse

Méthodes formelles et à objets pour le développement du logiciel : Etudes et propositions.

Soutenue le 7 Juillet 1995 devant la Commission d'Examen

MM.	Daniel	HERMAN	Président
	Jean-Claude	DERNIAME	Rapporteur
	Christine	CHOPPY	Rapporteur
	Paul	LE GUERNIC	Examinateur
	Jean	BEZIVIN	Examinateur
	Jean-Claude	ROYER	Examinateur

Thèse préparée à l'Institut de Recherche en Informatique de Nantes.

Remerciements

Je remercie Daniel Herman, qui me fait l'honneur de présider ce jury. Ses qualités pédagogiques m'ont marqué durant mes études rennaises et son attention a toujours été réconfortante.

Je remercie Jean-Claude Derniame d'avoir bien voulu accepter la charge de rapporteur et pour sa gentillesse.

Je remercie Christine Choppy, pour ses conseils avisés et pour l'intérêt qu'elle a porté à mon travail en acceptant d'être rapporteur de cette thèse.

Je remercie Paul Le Guernic d'avoir accepté de juger ce travail.

Je remercie Jean Bézivin de m'avoir accueilli dans son équipe ERTO, et sans lequel ce travail n'aurait pu être réalisé.

Je tiens à remercier Jean-Claude Royer de m'avoir fait confiance dès mon mémoire de 3e cycle et d'avoir accepté de participer à ce jury. Nous avons fait une partie du chemin ensemble, il a guidé mes premiers pas dans la recherche. Par son enthousiasme et sa rigueur scientifique, il m'a donné goût à la recherche. Il est bien plus qu'un collègue.

Je ne saurais oublier les membres permanents ou temporaires de l'équipe objet, Nadir, Franck, Régis, Yann, Hervé, Cécile, Marie-Jo, et les autres, pour nos nombreux échanges et leurs encouragements lors des pauses café.

Je remercie tous les collègues d'enseignement de la faculté des Sciences, et les membres de l'IRIN, mon laboratoire d'accueil pour leurs conseils et leur soutien, ainsi que Christine, une figure de l'informatique de la faculté.

Je remercie mes contacts à l'IRISA et à l'IFSIC et en particulier les secrétaires, toujours disponibles ainsi que Jean-Marc et surtout Renée, mon bras à l'Irisa, ainsi que les soeurs de Rennes.

Enfin, j'exprime toute ma reconnaissance à mes parents, pour leur amour sans faille, à Manu, toujours prêt à me changer les idées, à Philippe, Dominique, Julien et Aurélie, et enfin à ma famille et mes amis, toujours prêts à apaiser mes craintes et soutiens inconditionnels.

A Karine

A mes parents

Table des matières

1 Introduction **1**

2 Spécifications formelles et développement à objets **7**
2.1 Introduction au développement du logiciel 7
2.1.1 Méthodes de développement du logiciel 8
2.1.2 Modèles de représentation 8
2.1.3 Processus de développement 9
2.1.4 Spécification 10
2.1.5 Stratégies de développement 12
2.1.6 Qualité 14
2.1.7 Tendances et perspectives du génie logiciel 16
2.2 Introduction aux méthodes formelles 17
2.2.1 Présentation 17
2.2.2 Spécification formelle 17
2.2.3 Utiliser les méthodes formelles 18
2.2.4 Classification des méthodes formelles 20
2.2.5 Spécifications par modèle abstrait 22
2.2.6 Spécifications algébriques 27
2.2.7 Spécifications hybrides 28
2.2.8 Conclusion 29
2.3 Développement à objets 29
2.3.1 Concepts 30
2.3.2 Analyse et conception à objets 34
2.3.3 Utiliser les méthodes à objets 35
2.3.4 Approches remixées 37
2.3.5 Approches innovantes 39
2.3.6 Conclusion 44
2.4 Synthèse 44

3 Spécifications formelles à objets **47**
3.1 Intégration 47
3.1.1 Formalisation dans les méthodes à objets 47
3.1.2 Extensions à objets des méthodes formelles 48
3.1.3 Langage de programmation haut niveau 49
3.2 Panorama 49
3.2.1 ObjectZ 49
3.2.2 OOZE 53
3.2.3 VDM++ 57
3.2.4 COLD 62
3.2.5 OSDL 66
3.2.6 OS/OP 72
3.2.7 Autres langages 75

3.3 Synthèse . . . 77
3.3.1 Langages et modèles . . . 77
3.3.2 Modèle à objets . . . 78
3.3.3 Autres concepts . . . 83
3.3.4 Support formel . . . 84
3.3.5 Méthode de développement . . . 86
3.4 Bilan . . . 87

4 Introduction au modèle TAG/CF 91
4.1 Introduction . . . 91
4.2 Présentation des concepts . . . 93
4.3 Introduction aux Types Abstraits Graphiques . . . 94
4.3.1 Définir le modèle dynamique . . . 94
4.3.2 Construire le modèle fonctionnel . . . 95
4.4 Interface avec ASSPEGIQUE+ . . . 96
4.5 Le modèle des classes formelles . . . 97
4.6 Conception d'un TAG en CF . . . 98
4.7 Prototypage et mise en œuvre dans les classes concrètes . . . 99
4.8 Conclusion . . . 99

5 Types abstraits graphiques 101
5.1 Introduction . . . 101
5.2 Spécification graphique de TAG . . . 101
5.2.1 Représentation graphique . . . 101
5.2.2 Opérations . . . 102
5.2.3 Paramètres et invariant d'une spécification . . . 103
5.2.4 Comportement dynamique . . . 103
5.2.5 Types et relations . . . 104
5.3 Hypothèses . . . 104
5.4 Spécification algébrique de TAG . . . 105
5.4.1 Rappels et définitions . . . 105
5.4.2 Interface . . . 106
5.4.3 Préconditions et prédicats . . . 107
5.4.4 Partitionnement de la spécification . . . 109
5.4.5 Construction d'une axiomatique . . . 112
5.5 Contrôle et validation du TAG . . . 118
5.5.1 Vérifications de la spécification TAG . . . 118
5.5.2 Propriétés de spécification . . . 119
5.5.3 Typage . . . 121
5.5.4 Validation et prototypage . . . 121
5.6 Extensions . . . 122
5.6.1 Structuration des automates . . . 122
5.6.2 Héritage . . . 124
5.6.3 Généricité . . . 128
5.6.4 Concurrence . . . 128
5.7 Conclusion . . . 129

6 Classes formelles 131
6.1 Introduction . . . 131
6.2 Définition et interprétation des classes formelles . . . 132
6.2.1 Principaux concepts . . . 132
6.2.2 Définition formelle des classes . . . 134
6.2.3 Spécification algébrique associée à une classe . . . 136
6.2.4 Structure inductive des instances . . . 137

6.2.5 Propriétés . . . 138
6.2.6 Ecriture des méthodes . . . 139
6.2.7 Classe abstraite, méthode abstraite . . . 140
6.3 Héritage . . . 142
6.3.1 Cœrcition . . . 142
6.3.2 Héritage des méthodes . . . 143
6.3.3 Critère d'héritage . . . 144
6.3.4 Redéfinition des méthodes . . . 144
6.3.5 Conception plate ou ordonnée . . . 145
6.4 Sémantique opérationnelle . . . 147
6.4.1 Déduction équationnelle et réécriture . . . 147
6.4.2 Evaluation symbolique . . . 148
6.5 Typage . . . 149
6.5.1 Un système de type simple . . . 150
6.5.2 Le problème des termes douteux . . . 152
6.5.3 Multi-covariance . . . 153
6.6 Extensions du modèle . . . 155
6.6.1 Multi-générateur et méthodes de classe . . . 155
6.6.2 Déclarations modulaires et méthodes cachées . . . 156
6.6.3 Effets de bord . . . 156
6.6.4 Généricité . . . 156
6.6.5 Multi-sélection . . . 157
6.6.6 Multi-aspect . . . 157
6.6.7 Métaclasses . . . 157
6.7 Conclusion . . . 157

7 Démarche de conception 159
7.1 Introduction . . . 159
7.2 Passage des types abstraits graphiques aux classes . . . 160
7.2.1 Problématique . . . 161
7.2.2 Stratégie de conception . . . 163
7.3 Etape 1 : Définition d'un schéma structuré . . . 164
7.3.1 Réduction de l'automate . . . 165
7.3.2 Calcul des noeuds internes . . . 165
7.3.3 Structuration des classes . . . 168
7.4 Etape 2 : Restructuration . . . 170
7.4.1 Restructuration d'une classe . . . 171
7.4.2 Correction de la représentation . . . 172
7.4.3 Restructuration du schéma d'héritage . . . 174
7.5 Etape 3 : Ecriture des extensions . . . 176
7.5.1 Comparaison et généralisation de méthodes . . . 178
7.6 Validation du raffinement . . . 178
7.6.1 Domaine d'application de la méthode . . . 179
7.6.2 Evaluation du résultat . . . 179
7.7 Récapitulatif . . . 180
7.8 Implantation des classes formelles . . . 181
7.8.1 Optimisation du prototype . . . 181
7.8.2 Traduction directe d'une classe formelle . . . 182
7.8.3 Eiffel . . . 183
7.8.4 Smalltalk . . . 185
7.9 Conclusion . . . 188

8 Conclusion **191**
8.1 Bilan 191
8.2 Perspectives 193

Annexes **197**

A Rappels sur les spécifications algébriques **197**
A.1 Définitions de base 197
A.2 Spécification structurée 201
A.3 Spécification partiellement ordonnée 203
A.4 Héritage et sous-typage 203
A.5 Concurrence 203
A.6 Un exemple de spécification algébrique en ASSPEGIQUE 204

B Un environnement de développement **209**
B.1 Syntaxe des langages 210
B.2 Grammaire des expressions 210
B.3 Présentation rapide des éditeurs 213
B.4 Architecture de l'application 218
B.5 Perspectives 223

C Un exemple plus complexe : l'ascenseur **225**
C.1 Spécification informelle 225
C.2 Spécification TAG 225
C.3 Spécification algébrique 226
C.4 Un exemple de théorème démontré en ASSPEGIQUE 231
C.5 Spécification des classes formelles terminales structurées 232
C.6 Spécification des classes formelles après restructuration 234
C.7 Implantation de la classe `ONFLOORLift` 237
C.7.1 Eiffel 237
C.7.2 Smalltalk 239

D Glossaire **243**

E Notations **249**

Table des figures

1	Compatibilité verticale entre spécifications	12
2	Analyse morphologique selon les niveaux d'abstraction	13
3	Croisement modèle/processus	13
4	Corrélation entre les facteurs et les critères du logiciel	14
5	Corrélation entre les facteurs et les critères du processus	16
6	Relation de causalité dynamique en objet	33
7	Relation entre analyse et conception à objets	34
8	Le modèle à objets OMT de l'application vidéo	38
9	Le modèle dynamique OMT de l'objet `Cassette`	39
10	Le modèle fonctionnel OMT de la fonction `emprunt`	39
11	Les types de classes de OOAD	42
12	Modèle statique OOAD de l'application vidéo	43
13	Modèle dynamique OOAD de l'application vidéo	43
14	Intégration de concepts formels et à objets.	49
15	Spécification OSDL de l'hôpital.	67
16	Spécification OSDL du bloc `curation`	68
17	Spécification OSDL du processus `dispatching`.	69
18	Synthèse : Langages et modèles	78
19	Synthèse : Caractéristiques objet 1/6	78
20	Synthèse : Caractéristiques objet 2/6	79
21	Synthèse : Caractéristiques objet 3/6	80
22	Synthèse : Caractéristiques objet 4/6	81
23	Synthèse : Caractéristiques objet 5/6	82
24	Synthèse : Caractéristiques objet 6/6	83
25	Synthèse : Concepts supplémentaires	83
26	Synthèse : Techniques formelles de la méthode 1/2	84
27	Synthèse : Techniques formelles de la méthode 2/2	86
28	Synthèse : Méthodes de développement	87
29	Concepts et organisation générale	94
30	L'automate du TAG de `Hospital`	95
31	Conception plate et intuitive de la classe formelle `Hospital`	98
32	TAG Compte bancaire.	102
33	Type Pile non bornée.	107
34	Type Switch.	110
35	Dépendances des opérations d'une liste bornée.	112
36	Un exemple de G-dérivation.	113
37	Exemples d'automates refusés	118
38	Généralisation de transition	122
39	Etats frères	123
40	Agrégation d'opérations	123

41 Produit d'automates . . . 123
42 Etat composite . . . 124
43 Transition composite . . . 124
44 Projection/Synchronisation de transitions . . . 129

45 Tableau des méthodes primitives fonctionnelles . . . 135
46 Axiomes des méthodes primitives . . . 136
47 La spécification des booléens par classes formelles. . . . 141
48 Une redéfinition voisine de méthode . . . 144
49 Une redéfinition surchargée de méthode . . . 145
50 La conception ordonnée d'une spécification des entiers naturels. . . . 146
51 La conception plate d'une spécification des entiers naturels. . . . 147
52 Un problème classique de multi-covariance en sélection simple . . . 154

53 Processus de spécification . . . 159
54 La classe formelle intuitive `FlatLift` . . . 162
55 Schéma initial pour l'ascenseur `Lift` . . . 163
56 Stratégie de conception d'un TAG en classes formelles . . . 163
57 Cycle de raffinement . . . 164
58 Schéma initial réduit du type `Lift` . . . 165
59 Comportement dynamique réduit pour l'ascenseur `Lift` . . . 165
60 Algorithme de calcul des profils spécifiques des opérations . . . 167
61 Schéma d'utilisation du type `Lift` . . . 167
62 Schéma de création du type `Lift` . . . 168
63 Projection du comportement dynamique sur la classe MLLift . . . 170
64 Automate interne de la classe MLLift . . . 170
65 Schéma d'utilisation structurée du type `Lift` . . . 174
66 Schéma d'utilisation restructuré du type `Lift` . . . 176
67 Abstraction de méthodes . . . 178

68 Fonction d'interprétation . . . 199

69 Editeur textuel de TAG et gestion de la bibliothèque de TAGs . . . 213
70 Editeur graphique de comportement dynamique . . . 214
71 Editeur graphique de TAG . . . 215
72 Editeur textuel de classes formelles et gestion de la bibliothèque de CFs . . . 216
73 Editeur graphique de classes formelles . . . 217
74 Liens entre les différents composants du système d'édition . . . 218
75 Editeurs de spécifications . . . 219
76 Vues associées aux éditeurs de spécifications . . . 220
77 Compilateurs de spécifications . . . 221

78 Profils du TAG `Lift` . . . 225
79 Comportement dynamique du TAG `Lift` . . . 226

80 La notation du modèle TAG . . . 250
81 Classification des opérations TAG . . . 251
82 Utilisation de la classification . . . 251

Chapitre 1

Introduction

"L'analyse est parfois un moyen de se dégoûter en détail de ce qui était supportable comme ensemble et vivre avec quelqu'un est une manière d'analyse qui obtient les mêmes effets."

Paul Valéry.

Le développement du logiciel est un domaine continuellement en effervescence. Plusieurs raisons expliquent cet état de fait. Les progrès techniques du matériel accroissent la variété et la complexité des applications automatisables. L'informatique est maintenant utilisée dans la plupart des professions. Les ateliers de génie logiciel se sont généralisés. Ils sont un outil indispensable pour appliquer les méthodes de développement et favorisent les développements à grande échelle. De nombreux progrès ont été fait ces dernières années dans le domaine des méthodes d'analyse et de conception. Ces progrès visent à rationaliser de développement, à assurer la qualité des documents produits et du logiciel et à rentabiliser les efforts de développement. Les méthodes traditionnelles de développement, trop cloisonnées dans leur domaine d'application, deviennent insuffisantes pour gérer efficacement ces nouvelles applications. De nouvelles méthodes sont apparues, les méthodes formelles et plus récemment les méthodes à objets.

Les spécifications formelles s'imposent progressivement comme une condition essentielle à un bon développement logiciel. Leurs avantages principaux sont la rigueur dans la spécification du problème, la preuve de propriétés et une conception méthodique, automatisable en partie. Actuellement, l'utilisation des spécifications formelles dans un cadre industriel est marginale. Il semble y avoir plusieurs raisons à cet état de fait. Une première raison est la difficulté d'écriture et d'utilisation des spécifications formelles dans un cadre "réaliste". Une seconde raison est la carence de formation des développeurs du milieu industriel et un manque d'outils adaptés aux spécifications formelles et d'un usage convivial et simple. En résumé, la technique n'est pas encore assez mature et outillée pour être praticable à grande échelle.

Après avoir suscité beaucoup d'engouement en programmation, les objets ont également percé, depuis une décennie, en développement du logiciel. Le modèle à objets permet une meilleure maîtrise du développement du logiciel et plus particulièrement de la phase de conception. Une raison est que les objets constituent un modèle unificateur utile dans toutes les phases du développement et pour des applications très diverses et souvent de plus en plus complexes. La proximité des objets avec la démarche intellectuelle des acteurs du développement facilite les communications entre ceux-ci et donc diminuent les problèmes de transfert d'expertise. Techniquement les objets forcent à penser en terme d'unités autonomes que l'on peut assembler ou décomposer alliant ainsi une démarche qui peut être soit ascendante soit descendante.

Ces objets peuvent être hiérarchisées d'une façon naturelle par héritage. L'héritage favorise la généralité et améliore ainsi l'extensibilité et la réutilisation, deux qualités essentielles pour la rentabilité du logiciel.

Les apports du modèle à objets sont indéniables en ce qui concerne la qualité du logiciel et l'éventail des applications réalisables. Mais les avantages des méthodes à objets ne doivent pas cacher de sérieuses difficultés. En particulier, différents modèles à objets existent et co-existent, se différenciant par leurs concepts, les niveaux d'abstraction ou les plans de description (structurels, comportementaux, fonctionnels). Cette variété induit des erreurs de compréhension et des ambiguïtés. L'uniformisation des différents modèles à objets passe par l'élaboration d'une norme. Il y a un manque de formalisme au niveau des spécifications mais également au niveau du raffinement vers la conception. Le formalisme permet encore une meilleure comparaison de composants, condition indispensable à une bonne réutilisation. Le succès d'une méthode telle qu'OMT semble beaucoup dû à la liberté qui règne dans l'écriture des spécifications. Une multitude de méthodes à objets ont été proposées. Nous avons constaté l'absence de démarche rigoureuse (à quelques rares exceptions près) et de règles méthodologiques précises. A l'heure actuelle, les approches traditionnelles sont plus adaptées à un développement rationnel de l'analyse à la mise en place des systèmes informatisés. De plus elles possèdent des environnements supports complets et bien adaptés. Par ailleurs l'utilisation des objets à tous les niveaux ne doit pas masquer des différences sémantiques réelles. En particulier, cette confusion fait que le résultat de l'analyse est souvent sur-spécifié car pensé en termes trop opérationnels provenant de la pratique des objets en programmation. A l'inverse l'analyse peut être trop superficielle ou sous-spécifiée à cause d'une non maîtrise du formalisme, d'un manque de maturité de la spécification ou d'un manque de compétence de l'analyste.

L'ajout de formalisme aux méthodes d'analyse et conception à objets permet une meilleure structuration et une organisation plus naturelle des spécifications. D'un autre côté, les objets seront plus sûrs et la conception plus rigoureuse. Ce mariage doit se faire impérativement en respectant certaines contraintes industrielles (niveau de compétence, taille de l'application, cadres existant souvent rigides, impératifs économiques ...) qui sont généralement difficiles à tenir tout en respectant la production d'un logiciel de qualité. Des outils, ou mieux un environnement complet de développement, sont indispensables. Il faut préciser, et l'expérience nous le montre, qu'un outillage ou un environnement de développement ne reposant pas sur une méthode formelle est quasi impossible à définir. Il est obtenu, tout au plus, un amalgame de règles plus ou moins laxistes, vagues, voire même contradictoires.

Le prochain pas, décisif dans ce domaine, est donc d'introduire des aspects formels dans le cycle de développement à objets. L'utilisation des spécifications formelles dans un environnement réel de développement se heurte à plusieurs problèmes. Le principal étant la difficulté d'écrire ex nihilo une spécification formelle pour beaucoup d'utilisateur. D'autre part le raffinement d'une spécification en une mise en œuvre efficace et réutilisable n'est en général pas trivial. Par ailleurs, un environnement de spécification est indispensable pour matérialiser l'apport des spécifications formelles. L'utilisateur doit être guidé dans sa démarche, contrôlé dans ses choix, secondé pour les tâches automatisables.

Les travaux présentés dans cette thèse s'inscrivent dans cette problématique. Nous proposons l'introduction de certains concepts, les types abstraits graphiques (TAG en abrégé) et les classes formelles (CF en abrégé), qui sont une réponse partielle à ces problèmes. Nous nous sommes limités aux systèmes séquentiels mais des ouvertures sont possibles pour intégrer ultérieurement la concurrence et les communications asynchrones. Notre approche est une approche multi-langages : langage de spécification, langage de conception abstraite et langage de codage. Une des sources d'incohérences et de difficultés dans le développement du logiciel est qu'il n'y a pas de passerelle facile ni rigoureuse entre les différents plans de description d'un système. Un seul langage couvrant à la fois les différentes phases et les différents plans n'est pas réaliste.

La finesse de description des objets dépend de leur niveau d'abstraction : la réutilisation des objets devient indépendante des environnements de développement. Nous avons privilégiés des langages avec plusieurs facettes (automate/spécification algébrique, TAD/classe) pour limiter le nombre de modèles différents produits à un niveau d'abstraction donné et ainsi renforcer la cohérence globale.

L'introduction d'aspects formels va contraindre les différents acteurs du développement. Ces aspects doivent donc s'intégrer dans le processus de développement classique et être un plus plutôt qu'un bouleversement des habitudes. Ces contraintes doivent être introduites d'une façon naturelle et progressive et avec un support logiciel riche. Des outils d'édition, de preuves, de simplification, d'évaluation, de gestion, et d'autres, doivent être proposés. En bref un bon environnement support est indispensable.

Notre travail se situe à l'intersection de plusieurs axes de recherche : analyse et conception, modèle(s) à objets, méthodes formelles, outils de validation et de preuve de spécifications; outils, démarches et algorithmes de conception formelle à partir de spécifications. Notre but n'est pas de définir une nouvelle méthode d'analyse et de conception mais des concepts et des techniques qui peuvent s'intégrer dans une méthode de développement à objets. Les idées originales de ce travail sont les suivantes :

- séparation nette entre la spécification et la conception de composants logiciels,
- utilisation d'automates pour construire et prouver des spécifications algébriques complexes,
- prise en compte d'aspects dynamiques et fonctionnels dans un formalisme uniforme,
- conception progressive pour et par la réutilisation,
- automatisation du processus de formalisation et de conception,
- ouverture vers d'autres environnements de spécification,
- outils et algorithmes de contrôle des classes avant implantation,
- production de code pour des environnements de programmation différents.

Cette thèse est organisée en trois volets : un état de l'art des méthodes formelles à objets afin de dégager les apports et les manques de ces deux approches, une proposition de méthode de spécification en termes de modèles de description à des niveaux d'abstraction différents et de transition entre ces modèles, et enfin des outils de support de la méthode.

Le chapitre 2 est une synthèse rapide du développement du logiciel sous l'angle des méthodes et les aspects importants que sont les modèles de description, le cycle de vie, la qualité et la validation. Nous examinons ensuite deux courants essentiels : les méthodes formelles et l'approche à objets. Les premières apportent de la rigueur dans l'expression et favorisent l'automatisation du développement et le contrôle des modèles. L'approche à objets fournit une unité cohérente de description qui permet une meilleure structuration des modèles et la capitalisation des résultats produits. Nous passons en revue, pour chaque style, les concepts majeurs, les apports et les limitations.

Comme beaucoup d'auteurs, nous pensons que le mariage entre méthodes formelles et méthodes à objets est une voie prometteuse. Les méthodes formelles ont du mal à s'imposer parce qu'elles ont un caractère rebutant. Toutefois une organisation moins hiérarchique des spécifications les rend plus attrayantes. Les méthodes à objets suscitent beaucoup d'intérêt, mais le manque de formalisme réduit fortement la rigueur du développement. Au cours de ce chapitre 3, nous proposons plusieurs solutions pour intégrer les deux approches, puis nous établissons une synthèse des méthodes formelles à objets actuelles.

Le chapitre 4 est un résumé de notre proposition : trois modèles de description des composants (les objets) et une démarche guidée et outillée pour passer d'un modèle à un autre. La définition des objets doit être la plus abstraite possible, au sens des langages de programmation. Nous pensons en effet que les aspects à spécifier pour un composant logiciel varient en fonction des centres d'intérêt : définir les fonctionnalités et les conditions d'emploi des objets, proposer une architecture fonctionnelle en termes d'objets qui réponde aux fonctionalités attendues, implanter l'architecture dans différents environnements de développement. Le formalisme est indispensable à chaque niveau pour contrôler et certifier les résultats.

Le premier modèle sert à la spécification la plus abstraite des composants logiciels, appelés types abstraits graphiques. Sa sémantique est basés sur les types abstraits de données axiomatiques. Une présentation particulière de ces types permet simultanément la description dynamique et fonctionnelle des objets. Cette particularité fait que le calcul des axiomes est partiellement automatisé tout en respectant les propriétés importantes de cohérence et de complétude suffisante dans les descriptions. Nous donnons les caractéristiques essentielles de ce modèle dans le chapitre 5 et l'ouverture vers d'autres environnements de spécification, notamment les spécifications algébriques de types abstraits de données.

Le chapitre suivant décrit les caractéristiques du modèle intermédiaire basé lui aussi sur une sémantique algébrique. Un certain nombre de contrôles sont mis en oeuvre dans ce chapitre 6, concernant le typage, l'héritage et l'évaluation des spécifications. L'accent est mis sur la rigueur d'un formalisme de description des objets par des classes formelles et la conception à objets d'un logiciel. Un des atouts du modèle est de permettre un passage rapide vers l'implantation dans divers langages de programmation à objets.

Le formalisme des modèles est à la base du contrôle et de la validation des spécifications. D'une part, il oblige le concepteur à décrire clairement et précisément ce qu'il souhaite. Cette rigueur dans l'expression est un premier contrôle. D'autre part il permet d'établir des preuves de propriétés de la spécification. Cependant l'apport du formalisme ne se résume pas au contrôle, il augmente le degré d'automatisation du raffinement des spécification en représentations de plus en plus proche des langages de programmation. Nous nous focalisons sur cet aspect raffinement dans le chapitre 7. Nous détaillons deux processus de raffinement : celui des types abstraits graphiques aux classes formelles et celui des classes formelles aux classes des langages de programmation. Nous montrons l'intérêt d'algorithmes et d'outils pour réaliser ces transitions entre modèles.

La validation de la proposition se fait au travers d'exemples détaillés dans les chapitre dédiés à la proposition et ceux des annexes. La validation se fait aussi au travers d'un prototype, qui est en fait un ensemble d'outils à intégrer dans ASFO (Atelier de Spécification Formelle à Objets), l'atelier de génie logiciel qui supporte les concepts et outils présentés dans cette thèse. Nous décrivons les outils développés dans le chapitre annexe B.

Enfin, la conclusion présente le bilan des travaux menés et un ensemble d'améliorations, d'outils et d'extensions à apporter pour que l'atelier de spécification soit pleinement intégré et utilisable. L'application de la méthode à un exemple industriel significatif est une étape indispensable à sa validation. Les perspectives de nos travaux concernent tant le développement d'outils que les méthodes de gestion de bibliothèques de composants formels ou l'intégration aux méthodes d'analyse et conception existantes.

Etat de l'art

Chapitre 2

Spécifications formelles et développement à objets

"La science ne sert qu'à donner une idée de l'étendue de notre ignorance."
Lammenais.

Ce chapitre présente le contexte de cette thèse. La construction du logiciel nécessite des moyens de réflexion, d'expression et de calcul pour pouvoir aborder des problèmes de plus en plus complexes. En plus des techniques de description de programmes, il faut des techniques et des méthodes de construction toujours plus fiables, plus rapides et plus rentables. C'est l'objectif du développement du logiciel que nous présentons dans la section 2.1. Nous en dégageons les points qui nous semblent essentiels, à savoir les modèles et la démarche, par des critères de comparaison et de qualité. Deux approches du développement s'imposent petit à petit, tant dans la recherche que dans l'industrie : les méthodes formelles et les méthodes à objets. Pour chacune, nous faisons une synthèse dans les sections 2.2 et 2.3 et nous montrons en quoi elles sont une réponse positive à ces besoins.

2.1 Introduction au développement du logiciel

Le développement du logiciel est l'ensemble des moyens mis en œuvre pour produire du logiciel. Le logiciel est un ensemble de programmes qui automatise un système d'information[1]. Le développement du logiciel est une partie de ce qu'on appelle communément la gestion de projet. Un projet est caractérisé par l'expression du besoin, par la définition d'une ou plusieurs solutions techniques, par la gestion des ressources mise en œuvre pour réaliser le projet et par l'achèvement du projet. La gestion de projet au sens large (faisabilité, moyens humains et financiers, planification, impact sur l'organisation automatisée, etc.) sort du cadre de notre exposé. Des théories ont été développées sur ce sujet [BR85].

Un logiciel est une solution au besoin d'informatisation. Ce besoin est exprimé en construisant des modèles ou des **représentation**. La **modélisation** en informatique est le passage du domaine du problème à celui de sa solution informatique [Bar92]. La recherche d'une bonne représentation lors de la résolution de problème est presque toujours une étape indispensable vers la solution [Lau86]. Il faut donc savoir changer de représentation pour mieux modéliser le problème. Le développement est donc une suite de modèles, se rapprochant petit à petit des concepts de la programmation. Consulter [Som92] pour un survol du développement du logiciel.

[1] Pour une synthèse sur les systèmes d'information, consulter [Rol86]

Le développement du logiciel et surtout la maintenance coûtent cher [Boe82]. Les systèmes deviennent rapidement obsolètes vis-à-vis de l'organisation et des techniques nouvelles. Il faut donc les rendre évolutifs. Le développement du logiciel n'est plus une réponse instantanée à un besoin d'automatisation mais un investissement durable. L'objectif du génie logiciel est de mettre en œuvre des moyens pour produire du logiciel de qualité et pour rationaliser et rentabiliser le développement. Ces moyens sont regroupés sous le terme générique de **méthodes de développement du logiciel**.

2.1.1 Méthodes de développement du logiciel

Une méthode est une technique de résolution de problèmes [Lau86]. Le terme de **méthode** recouvre plusieurs notions. C'est à la fois une philosophie dans l'approche des problèmes, une démarche ou un fil conducteur dans la résolution, des outils d'aide et enfin un formalisme ou des normes. Dans la littérature, le terme **méthodologie** est souvent synonyme de méthode.

Dans [Rol86], l'auteur dégage deux grandes classes de méthodes (appliquées à la conception) : les méthodes **cartésiennes** et les méthodes **systémiques**. Les méthodes cartésiennes suivent le principe "diviser pour règner". L'étude d'un phénomène consiste à le diviser en éléments plus simples, étudiés séparément puis à réunir à nouveau ces éléments. La réalisation du processus se fait par une démarche fonctionnelle descendante, inspirée de la programmation structurée. Pour les méthodes systémiques, la démarche n'est pas essentielle, ce qui prime, c'est la compréhension du système d'information en tant que partie de l'organisation. L'approche est conceptuelle et met en valeur différents niveaux d'abstraction par des modèles formels riches sémantiquement. Au début des années 80, l'approche systémique et modulaire s'est imposée face à l'approche cartésienne et structurée pour la réalisation de systèmes complexes caractérisés par une période de développement et une durée de vie allongée.

Dans la suite du document, nous nous focaliserons principalement sur deux aspects, les **modèles de représentation** groupés dans des **spécifications** et les **processus de développement**, pour analyser et comparer les méthodes de développement.

2.1.2 Modèles de représentation

Un **modèle** est une interprétation explicite par son utilisateur de la compréhension d'une situation, ou plus exactement de l'idée qu'il se fait d'une situation [Cal90]. Il peut être exprimé par des mathématiques, des symboles, des mots, mais essentiellement, c'est une description d'entités et de relations entre elles. Un modèle est correct s'il permet de répondre aux questions qu'on se pose. Par abus de langage, dans ce qui suit le terme **modèle** désignera à la fois le résultat de la modélisation, la théorie sous-jacente et la notation utilisée.

Voici quelques modèles utilisés dans le développement du logiciel : la théorie des systèmes (Mélèse, Le Moigne)[LM90], la théorie des processus communicants (CSP [Hoa85], CCS [Mil89]), les systèmes de transitions (Réseaux de Petri, automates) [Arn92], la théorie des ensembles et ses dérivés (Z [Hay92], Warnier), les flots de données (SADT [Ros77]), la programmation (structurée, fonctionnelle, logique, à objets), les bases de données (relationnel, hiérarchique, réseau), le modèle entité/association (Chen) ou Niam, les diagrammes événement/procédures (Corig) ou résultats/données (Minos), la logique [Lau86], etc.

Pour modéliser un système d'information, trois aspects sont distingués : les informations manipulées (appelées aussi les données), quand elles sont manipulées (causalité, contrôle, événements) et comment elles sont manipulées (les opérations ou fonctions) Les méthodes privilégient souvent l'un ou l'autre des aspects.

statique (données)	(opérations)	**dynamique** (traitements)

Entité/ Association	Structures de Données	Langages Formels	Flots de Données	Réseaux de Petri Automates

Trois courants majeurs ont dominé les méthodes d'analyse et de conception :

- l'approche fonctionnelle dans laquelle le système est perçu en termes de fonctions et sous-fonctions munies d'une interface (e.g. JSD),
- l'approche flots de données basée sur la transformation des données (e.g. SA (De-Marco), SADT (Ross), SART (Ward-Mellor/Hatley-Pirbay), ESML (Boeing)),
- l'approche modèle de données pour laquelle l'accent est mis sur la partie statique du système d'information (e.g. Entité-Relation (Chen), NIAM (Verhaijen)).

L'approche à objets, que nous verrons dans la section 2.3, est un mélange des différents courants.

2.1.3 Processus de développement

Tout système a un cycle de vie comprenant principalement sa gestation, sa conception, son exploitation, sa maintenance et sa mort. Le processus de développement d'un système d'information couvre l'intégralité de son cycle de vie. Il comporte les activités suivantes :

1. L'**analyse des besoins** définit les services du système, ses contraintes et ses buts en consultant les utilisateurs du système. Une étude d'opportunité peut être menée pour savoir si le système est réalisable et donner une approximation de la rentabilité de ce système.

2. L'**analyse** est la construction d'un modèle du système à partir de l'analyse des besoins. Ce modèle sert à établir plusieurs scénarii et les comparer dans une étude de faisabilité.

3. La **conception** est une proposition de solution au problème spécifié dans l'analyse. Elle définit la solution retenue par prise en compte des caractéristiques logiques d'usage du futur système d'information et des moyens de réalisation, humains, techniques et organisationnels. Conception système et conception détaillée sont parfois séparées. La première a pour objectif de donner l'architecture globale du systèmes (i.e. les différentes parties) et la seconde décrit chaque partie du système. Cette spécification du logicielle reste indépendante de tout moyen de réalisation.

4. La **réalisation** produit la solution exécutable en termes de programmes. Pour les logiciels complexes, deux phases sont requises : d'une part l'implantation des différentes parties et leur validation par des tests unitaires, et d'autre part l'impantation du système complet par **intégration** des parties et tests systèmes validant la spécification des besoins.

5. L'**installation** du logiciel règle les problèmes de mise en place dans l'organisation.

6. La **maintenance** adapte la solution conceptuelle aux changements organisationnels et aux évolutions technologiques (évolutive). Elle corrige aussi les erreurs accumulées dans les phases précédentes (curative). La maintenance est un processus itératif dont l'activité répétée est un cycle de développement complet.

Des variantes existent, mais elles sont surtout d'ordre terminologique [Cal90, Ner90, Rol86, Som92][2].

[2] Voir aussi la revue Génie Logiciel et Systèmes Experts N° 19 de juin 1990.

Validation

L'étude des facteurs de coûts du logiciel de [Boe82] et [Cal90] montre l'importance de la validation. Plus une erreur est détectée tardivement, plus sa correction est chère. C'est pourquoi l'effort doit être porté sur la spécification plutôt que sur la conception. De plus la validation doit être réalisée par une équipe pluri-disciplinaire, comprenant des informaticiens certes, mais aussi des utilisateurs et des intervenants extérieurs au projet. La confrontation des points de vue devrait aboutir à un compromis raisonnable. De ce fait, la lisibilité des documents est un point clé de la validation. Elle se traduit par un langage unique et sans ambiguïté (langage formel) et des formalismes graphiques.

Le maquettage (ou prototypage) et les tests sont les deux principaux moyens de validation. Le premier permet à petite échelle de se rendre compte approximativement de l'allure générale du résultat et des problèmes pouvant intervenir. Le second permet de vérifier des hypothèses, des objectifs et des contraintes. Il nécessite une formalisation précise des besoins.

Cycles de vie

Plusieurs cycles de vie ont été proposés pour organiser ces tâches. Trois grandes catégories sont retenues :

- Les modèles linéaires (modèle de la “cascade”, modèle en “V”) dans lesquels chaque étape correspond à une des activités ci-dessus. Ce sont les modèles de référence. Les étapes sont franchies dans l'ordre, de l'analyse au codage (approche descendante). En cas d'erreur, un retour sur l'étape précédente est effectué. Dans le modèle en “V”, les diverses spécifications sont nettement séparées par des niveaux d'abstraction (utilisateur/architecture/implantation). La validation se fait a posteriori par des tests unitaires et des tests d'intégration.

- Les modèles contractuels sont formés d'un ensemble de contrats entre client et fournisseurs. Les méthodes formelles sont un bon exemple de tels modèles..

- Les modèles itératifs ou à spirale dans lesquels chaque activité est exécutée en partie. Ils permettent un développement incrémental par prototypage. Ces modèles intègrent des notions de risques à évaluer, de spécifications partielles et de résultats intermédiaires. Ce modèle est évolutif par nature mais rend difficile la planification et il doit éviter les redondances.

- Les modèles mixtes comme le modèle “X” [Hod91] s'inspirent de plusieurs styles. Les modèles linéaires ne dépendent pas de la technologie utilisée, or le processus de développement en dépend par nature. Le modèle “X” prend en compte le modèle à objets. Deux cycles sont en fait décrits, l'un pour une activité de synthèse d'un nouveau système et l'autre, inversée, d'acquisition de sous-systèmes en vue de les réutiliser.

modèles.

2.1.4 Spécification

A chaque étape correspond un document résultat, souvent appelé spécification. Une **spécification** est un ensemble de modèles. La spécification initiale est le cahier des charges et la spécification finale est le logiciel. Les auteurs parlent aussi de spécification des besoins, spécification détaillée, spécification fonctionnelle ou encore spécification formelle. Par convention, le terme **spécification** désignera de façon générique les documents résultant d'une étape dans le processus de développement. Le terme **spécification du logiciel** quand à lui, désignera la description plus ou moins détaillée du comportement attendu du logiciel. Le dossier des spécifications du logiciel est un ensemble de documents produits par le fournisseur de conseils et de services pour répondre aux besoins exprimés par son client dans le cahier des charges ayant donné lieu au contrat. Il s'agit de couvrir TOUS les besoins exprimés et uniquement ceux-là.

Types de spécification

Les spécifications peuvent être classées selon leur forme ou leur degré de formalisme :

- les spécifications informelles, en langue naturelle, rédigées sans contraintes de forme,
- les spécifications standardisées, toujours en langue naturelle mais avec une structure, un format et des règles précises (notations, glossaire, index, historique de modification...),
- les spécifications semi-formelles, qui utilisent un langage de spécification textuel ou graphique, langage doté d'une syntaxe précise et d'une sémantique assez faible permettant certains contrôle et une automatisation de certaines tâches (ex: Merise [TRC83], SADT [Ros77]),
- les spécifications formelles, exprimées dans un langage à syntaxe et sémantique précises, construits sur une base théorique solide et permettant des validations automatisées.

Bien que la dernière forme soit préconisée, ce sont les trois premières qui sont utilisées dans l'industrie[Bra88]. Il est vrai que les notations semi-formelles graphiques sont plus accessibles au spécialiste du domaine de l'application. Mélanger plusieurs formalismes est intéressant s'il y a intégration ou du moins correspondance entre ces formalismes [Dec88].

Relations entre spécifications

Il est intéressant de pouvoir manipuler les spécifications comme des objets mathématiques quelconques, c'est-à-dire de pouvoir les comparer par des relations, les transformer par des fonctions, les structurer par des opérateurs. Nous nous limiterons dans ce paragraphe à la description des relations entre spécifications en deux catégories et non pas à la construction de la spécification.

Relations horizontales La manipulation d'une spécification complexe exige la décomposition en sous-spécifications. Le lien de **structuration** entre ces sous-spécifications peut être affiné en trois relations :

- **complémentaire** : Chaque sous-spécification prend en compte un aspect différent du système à modéliser : c'est la structuration multi-modèle. Une corrélation est donnée, qui assure la complémentarité et la cohérence globale. Par exemple, une spécification classique d'analyse et conception structurée comprend un modèle pour les données, un modèle pour les traitements. Les traitements agissent sur les données.
- **inclusion** : Chaque sous-spécification est une partie de la spécification globale et n'a de sens que vis-à-vis de la spécification globale. C'est une structuration hiérarchique au sens englobant/englobé. La sous-spécification décrit plus en détail un morceau de la spécification dont elle dépend. Le critère de décomposition est souvent simple et monovalué : le temps (séquence), l'espace. Par exemple, l'analyse structurée est une spécification de plus en plus détaillée des fonctions du système.
- **importation** : Chaque sous-spécification décrit une partie relativement homogène et autonome du système. Le modèle de description est le même pour toutes les sous-spécifications (voir spécification modulaire page 14).

Relations verticales La relation verticale exprime le degré de spécialisation entre deux spécifications. La notion majeure ici est l'**abstraction**. Informellement une spécification est moins abstraite qu'une autre si elle contient plus de détails (partitionnement de problèmes, prise en compte de cas d'erreurs, contraintes techniques, sélection d'algorithmes abstraits, représentation de données, implantation d'algorithmes [Abr84, Hay92]). Nous proposons quelques nuances de cette notion très relative en nous inspirant de [Gue94, Wir93, Hay92]. Un objet abstrait peut être vu comme un objet à n paramètres dont p sont fixés et $n-p$ sont libres[Gue94].

- L'**implantation** de l'objet est atteinte lorsqu'il n'y a plus de paramètres libres.
- Le **raffinement** revient à augmenter le nombre de paramètres en remplaçant un paramètre par plusieurs paramètres libres ou fixes. [Hab93] donne la définition suivante du raffinement:
  R est raffiné par S si
  - *lorsque R est applicable alors S l'est aussi (critère d'applicabilité),*
  - *lorsque R est applicable et que S est appliqué, alors le résultat doit être cohérent avec R (critère de correction).*
- La **concrétisation** consiste à fixer certains paramètres de la spécification.

Le raffinement et la concrétisation sont des relations transitives.

Combinaison Les deux axes ci-dessus sont corrélés et permettent la spécification en couches. Par exemple, une spécification est raffinée en ajoutant une relation d'utilisation. Une spécification est concrétisée en changeant une relation d'inclusion ou d'utilisation. L'utilisation conjointe des mécanismes de structuration ci-dessus implique une des règles de compatibilité verticale : ce qui est vrai dans une spécification est vrai dans une sous-spécification verticale et les liens horizontaux ne sont pas modifiés.

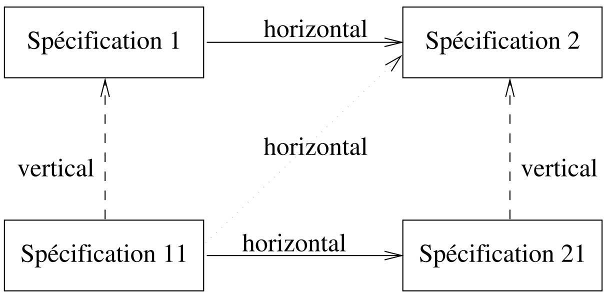


Figure 1 : Compatibilité verticale entre spécifications

Outils de spécification

Des outils logiciels sont indispensables pour écrire, valider, documenter, transformer, stocker et réutiliser des spécifications, interactivement ou automatiquement. Les outils guident et assistent les acteurs du développement dans leur tâche. Ils leur permettent de communiquer en toute sécurité. Le choix de l'outil se fait en fonction de la méthode de spécification utilisée. Mais inversement, plus une méthode sera outillée, plus elle sera utilisable. Un article du monde informatique, du 3 juillet 1989 (pages 22-26), donne une étude comparative des outils de spécification du logiciel, en prenant Idefo/SADT comme référence.

2.1.5 Stratégies de développement

Le processus de développement dépend des modèles choisis, et vice-versa. Nous donnons ici des éléments de comparaison pour une analyse morphologique des méthodes en termes de niveau d'abstraction, de modèles et d'organisation des étapes du processus. des plans de la stratégie.

Niveaux d'abstraction Les spécifications peuvent être classées par niveaux d'abstraction. Habituellement, trois niveaux sont retenus : le niveau conceptuel, qui décrit les fonctionnalités du système, le niveau organisationnel, qui comprend l'architecture logicielle, les interfaces et les contraintes dues à l'organisation (facteurs humains, financiers, temporels), et enfin le niveau opérationnel, qui rassemble toutes les considérations d'implantation. Un niveau d'abstraction peut comprendre plusieurs modèles othogonaux (exemple: plans de description structurelle, dynamique ou fonctionnelle d'un système). Un même modèle peut être décrit à différents niveaux d'abstraction. La traçabilité et le processus "sans couture"[3] sont une application de ce principe dans le modèle à objets. Enfin, d'un niveau à un autre, un modèle peut être **concrétisé** ou **raffiné** , dans un autre modèle (exemple : représentation concrète des données d'un type abstrait de données [Hoa72]) ou **éclaté** en plusieurs modèles (exemple : assemblage des plans de description dans un unique langage de programmation). Inversement, le **multiplexage** groupe plusieurs modèles complémentaires en un seul modèle de niveau inférieur. Ces alternatives sont synthétisées dans la figure 2.

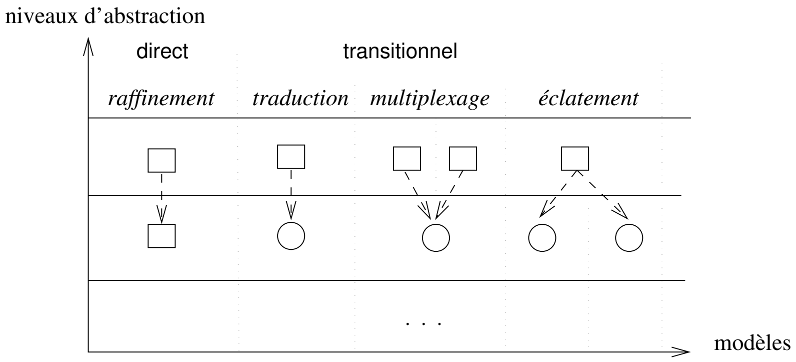


Figure 2 : Analyse morphologique selon les niveaux d'abstraction

Croisement entre modèles et processus Les deux principales familles de cycles sont les processus linéaires et les processus itératifs. Chaque famille comprend les étapes habituelles du développement. Dans le cas linéaire simple, une étape correspond à un niveau d'abstraction, c'est le plus facile à comprendre et à mettre en œuvre. Dans le cycle est itératif et dans le cas linéaire multi-niveau, il y a indépendance entre les modèles et le processus de développement. La méthode doit alors guider précisément le concepteur pour éviter la surspécification.

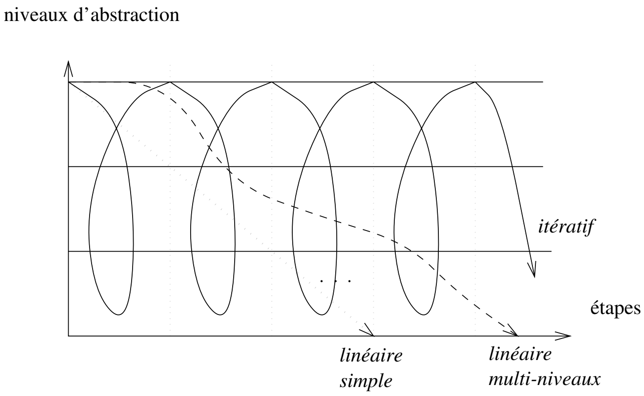


Figure 3 : Croisement modèle/processus

La prise en compte des deux schémas ci-dessus fournit donc les nombreuses stratégies possibles de développement. Une représentation graphique synthétique est donc difficile à exprimer. Les choix doivent être faits selon des critères de qualité.

[3] traduction de *seamless development process*.

2.1.6 Qualité

La comparaison des stratégies et des méthodes passe par la définition de qualités et de critères mesurables influençant ces qualités. Après un bref rappel de la qualité des spécifications et celle du logiciel en particulier, nous étudions la qualité du processus de développement. La qualité de la stratégie se résume principalement à sa complexité et sa facilité de mise en œuvre.

Qualité de la spécification

La qualité d'une spécification est le fait qu'elle exprime uniquement ce qu'il faut et comme il le faut. Une spécification doit être claire, concise, complète et cohérente[4]. Pour être compréhensible et réalisable, elle ne doit pas comporter d'ambiguïtés, de contradictions, d'oublis, de redondances, d'éléments inutiles, sur-spécifiés ou irréalisables. Les méthodes formelles sont un apport fondamental dans l'obtention de telles qualités.

Le logiciel est un cas particulier de spécification, dont les qualités sont maintenant bien connues. Un certain nombre de facteurs influant sur la qualité du logiciel ont été données dans [Mey88]. Des critères internes permettent d'atteindre ces facteurs externes de qualité [Gir91, Mey88]. Ces critères sont perceptibles seulement par les informaticiens. L'influence des critères[5] sur les facteurs de qualités[6] est établie dans [Gir91] comme suit :

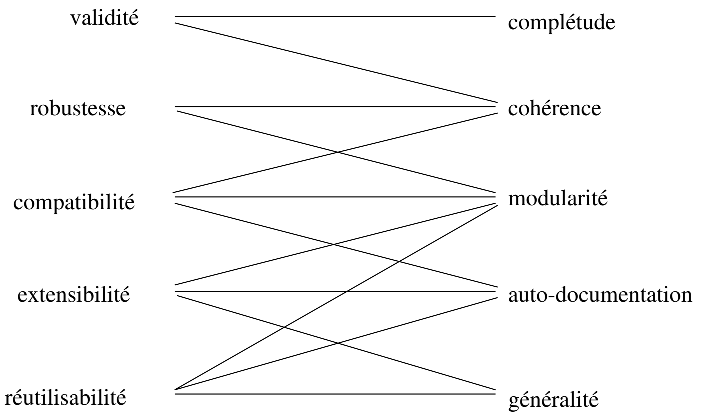


Figure 4 : Corrélation entre les facteurs et les critères du logiciel

Spécification modulaire

Comme le montre la figure 4, la modularité est un critère important. La relation d'importation est à la base des spécifications modulaires. Chaque sous-spécification est appelée module (de spécification) et traite une partie du problème. Le terme **composant** sera aussi utilisé pour décrire un morceau cohérent d'une spécification. La définition d'un module en conception du logiciel est la suivante.

Un ***module*** *est un élément du logiciel, relativement indépendant et contenant à la fois des opérations et des structures de données. Le module satisfait au principe de masquage d'information en proposant une partie interface aux autres modules et en cachant son implantation. Un module peut importer des informations d'autres modules.*

Les qualités d'une bonne conception modulaire sont : un faible couplage entre les modules (i.e. une relative indépendance) et une forte cohésion[7] dans les modules (i.e. les informations

[4] Pour une description détaillée des qualités d'une spécification, consulter [Bra88, Mey85].

[5] Consulter [Som92] pour une description détaillée des critères de cohérence (sept niveaux) et de couplage (page 181). Cinq critères pour évaluer la modularité sont commentés dans [Mey88].

[6] D'autres facteurs sont énoncés dans [Mey88] : l'efficacité, la portabilité, la vérifiabilité, l'intégrité et la facilité d'utilisation.

[7] Dans les approches modulaires, on parle plutôt de cohésion que de cohérence.

détenues dans le module sont logiquement reliées). La cohésion du module peut être assurée par la notion de type. Les types sont utiles pour classer, organiser, abstraire, conceptualiser et transformer des collections de valeurs[Weg90].

*Un **type de données** est caractérisé par un ensemble de valeurs et par un ensemble d'opérations sur ces valeurs.*

La description des types de données peut être affinée. Les types de base sont issus de la représentation interne de la machine (exemples : booléens, entiers, réels). Les types concrets de données sont structurés par des constructeurs de types (union, produit, récursion et parfois fonction). Les opérations sont définies à partir de cette structure. Les types abstraits de données permettent de décrire les données d'un point de vue externe en définissant des ensembles mathématiques, des opérations et une axiomatisation. L'utilisation de types abstraits de données permet d'écrire des algorithmes généraux sans fixer la représentation (concrète) des données. Le type abstrait est ensuite "représenté" par un type concret pour la programmation.

Qualité du processus de développement

Le processus de développement est ce qui permet de faire le lien entre les différents modèles de description. Les principales qualités attendues sont :

- Sûreté : la démarche doit minimiser les retours arrières et permettre des validations périodiques.
- Terminaison : le cycle doit permettre d'obtenir les produits en un temps fini, que ce soit à chaque étape ou globalement.
- Rigueur : l'enchaînement des étapes doit suivre un cheminement logique, correspondant aux habitudes des acteurs du développement.
- Cohérence, complétude : les étapes doivent être cohérentes entre elles (pas de duplication inutile de tâches) et former un tout (pas d'oubli).
- Souplesse : la démarche doit s'adapter en fonction de l'application à développer.
- Accessibilité : c'est la possibilité de comprendre les décisions prises au cours du processus.
- Rentabilité : c'est la capacité à capitaliser l'expérience dans le processus de développement.

Voici quelques critères qui permettent d'atteindre ces qualités :

- Automatisation : c'est le degré de mécanisation du processus,
- Réutilisation : des mécanismes permettent d'intégrer ou modifier des développements antérieurs.
- Facilité d'écriture : simplicité des concepts, notations graphiques, assistance au concepteur.
- Guidage : chaque étape indique clairement les tâches à réaliser.
- Traçabilité : les informations et traitements jugés utiles au départ doit exister sous une forme ou une autre dans les étapes ultérieures.
- Contrôle : un contrôle régulier est indispensable dès les premières étapes.
- Intégration : cohérence entre les modèles de description d'une même étape ou de deux étapes successives.
- Documentation : le raisonnement et les décisions doivent apparaître clairement.
- Ciblage : le domaine d'application de la méthode doit être clairement exprimé pour éviter des blocages dans la modélisation.
- Abstraction : le raisonnement et la preuve doivent progressivement prendre en compte les concepts de programmation.

Il faut noter que ces qualités dépendent non seulement des critères mais aussi des modèles de représentation. Les critères majeurs pour un bon développement sont la facilité d'écriture, le guidage, l'intégration et l'automatisation. Ils permettent d'écrire plus vite et avec moins d'erreurs les différentes spécification du développement.

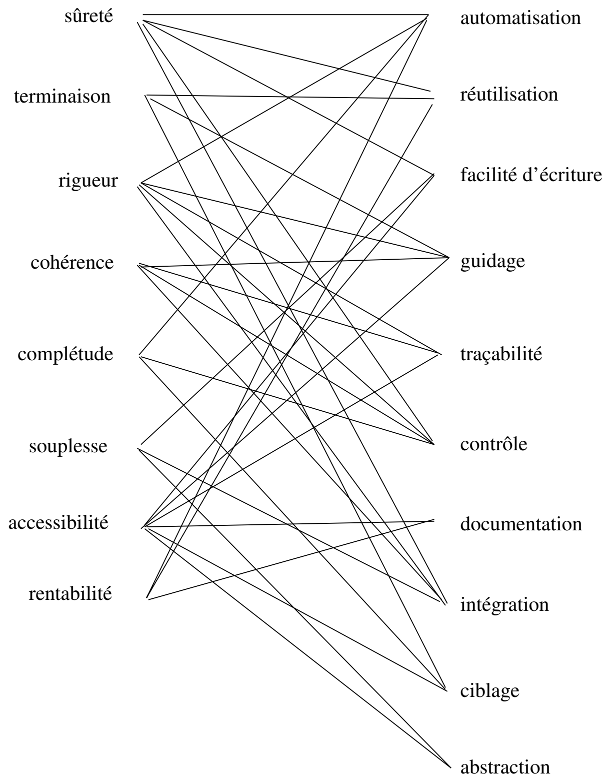


Figure 5 : Corrélation entre les facteurs et les critères du processus

2.1.7 Tendances et perspectives du génie logiciel

Le **génie logiciel** est l'art de construire industriellement du logiciel. Il a pour but de répondre aux différents problèmes posées dans les paragraphes précédents dans trois domaines :

- la *qualité* : il faut contrôler la qualité des composants produits et du résultat perçu par l'utilisateur.
- la *quantité* : les performances sont un critère important pour le génie logiciel. Elles sont évaluées par des mesures de tests et améliorées par des optimisations de code, d'accès et stockage des données ou d'organisation globale.
- la *gestion de projet* : elle comprend la répartition des tâches et responsabilités, la gestion des liaisons entre code, la documentation et spécification, la coordination des développeurs, la gestion de configuration, la gestion des composants logiciels réutilisables, etc.

Pour gérer efficacement le développement des applications complexes, il faut améliorer la qualité des produits (spécifications et logiciel) et celle du processus de développement (voir figure 4 et figure 5). Ces deux points font toujours l'objet de recherches actives. La question majeure est "Quel(s) modèle(s) sont à utiliser pour répondre aux besoins ?". Actuellement, deux techniques émergent : le modèle à objets et les méthodes formelles.

Les méthodes formelles visent à améliorer le coût du développement en imposant un contrôle exigeant dès les premières phases du développement. Les méthodes à objets ont pour objectif d'une part de maîtriser le développement en structurant l'application en composants logiciels relativement indépendants, donc plus facilement appréhendables et d'autre part de capitaliser le développement en généralisant et réutilisant ces composants. Nous allons examiner ces deux approches dans les sections qui suivent.

2.2 Introduction aux méthodes formelles

Cette section effectue un rapide tour d'horizon des méthodes formelles, de leur utilité et des différentes approches actuelles. Trois courants majeurs de la spécification formelle sont ensuite détaillés : les modèles abstraits, les spécifications algébriques et les approches mixtes.

2.2.1 Présentation

Les méthodes formelles sont issues des travaux du Programming Research Group de l'Université d'Oxford. Au début des années 80, ce groupe fut le siège d'un certain nombre de projets préconisant l'utilisation intensive des mathématiques dans la spécification des systèmes informatiques [Hay92], sous l'impulsion notamment de Tony Hoare et Jean-Raymond Abrial. Les méthodes formelles sont associées à trois sortes d'activités : écrire des spécifications formelles, prouver des propriétés et implanter le logiciel.

Parallèlement, d'autres chercheurs ont voulu raisonner plus abstraitement sur les langages de programmation, tant pour la conception d'algorithmes corrects que pour l'amélioration des algorithmes d'implantation de langages. Ils ont donc proposé des modèles formels. Bertrand Meyer synthétise ces idées dans [Mey90a] : "*Pour spécifier, concevoir, implémenter, lire, comprendre, documenter, apprécier, critiquer, tester, qualifier, mettre au point, maintenir, adapter, porter ou améliorer des programmes, il faut maîtriser les notations qui servent à exprimer leur forme finale : les langages de programmation.*". Parmi ces travaux, deux voies principales ont été exploitées : celle de la théorie des types (λ-calcul) [Hue87] et celle des types abstraits (algèbres) [GH78, GTW78].

Nous nous intéressons ici aux modèles formels dans la spécification du logiciel, parfois appelée abusivement étape de spécification formelle. Dans [Abr84], Jean-Raymond Abrial démontre le bien-fondé des mathématiques dans les spécifications. Il décrit la spécification et la construction rigoureuse et systématique d'un algorithme de modification de la mémoire d'un ordinateur. L'auteur procède par étapes, dites de "réalisation successives", de la spécification axiomatique par pré-conditions et post-conditions à l'algorithme du programme. Chaque réalisation est démontrée en utilisant la théorie des ensembles et la logique des prédicats. Dans [Mey85], Bertrand Meyer démontre l'utilité des mathématiques et du formalisme dans la spécification du logiciel. Il souligne notamment les défauts d'une spécification informelle (voir la section 2.1.6) à partir d'un exemple de manipulation de textes, présenté antérieurement comme une bonne spécification en angage naturel. En résumé, la qualité de la spécification des besoins passe par un formalisme rigoureux et abstrait.

2.2.2 Spécification formelle

Une spécification formelle est exprimée dans un langage à syntaxe et sémantique précises, construit sur une base théorique et qui permet des validations automatisées (système formel notamment). Cette base est généralement mathématique. Les réseaux de Petri, les grammaires formelles, les automates à états finis, la logique formelle, l'algèbre, la théorie des graphes, le λ-calcul... sont autant d'exemples de telles techniques [Bra88].

Les relations que nous avions défini pour comparer les spécifications à la page 11 sont utilisées ici comme mécanisme de structuration et de développement. De plus, certaines relations comme le raffinement ou la concrétisation, peuvent être prouvées formellement.

Dans certains domaines d'activité, les spécifications formelles ont acquis un droit de cité incontestable. Les réseaux de Petri sont largement utilisés pour spécifier et valider des protocoles de communication, les logiques temporelles en sont une alternative moins répandue. Les grammaires formelles sont utilisées pour définir et analyser les langages, notamment en théorie des langages de programmation. Les systèmes de transitions permettent la modélisation de mécanismes critiques dont la validation formelle est estimée nécessaire (moniteurs temps réel, postes de commande). La logique formelle[8] et les algèbres permettent de formaliser et de démontrer des algorithmes essentiels. La théorie des ensembles est un support important des bases de données relationnelles et des spécifications abstraites de systèmes.

2.2.3 Utiliser les méthodes formelles

Pendant longtemps, partisans et opposants des méthodes formelles se sont affrontés, les uns présentaient les méthodes formelles comme la solution au problème de la qualité du logiciel et les autres comme un fardeau dans le développement. Hall[Hal90] puis Bowen[BH94] ont depuis réfuté l'argumentation de ces mythes. "*On ne cherche en aucun cas à suggérer que les spécifications formelles sont une panacée contre tous les maux de la programmation*" [Mey90a].

L'étape de spécification du logiciel est universellement reconnue comme étant cruciale dans le processus de développement du logiciel. Des mesures ont prouvé que les erreurs de spécification sont les plus coûteuses à corriger et sont malheureusement fréquentes [Boe82]. La spécification formelle est une bonne réponse à ce problème [Cho87, CG88, JS90].

La spécification formelle met en valeur un certain nombre de propriétés que doit respecter le logiciel. Un plan de test peut donc être établi. Bien qu'une spécification soit plutôt destinée à être lue, il se peut que le langage utilisé soit exécutable. Certains auteurs estiment qu'un langage exécutable diminue la puissance d'expression et l'abstraction[Abr84, ST90, Bid89, Jon93]. Mais cette notion est toute relative, comme le montre [Fuc92]. Quoi qu'il en soit, il semble important de pouvoir extraire un prototype de la spécification pour mesurer l'adéquation au besoin initial mais aussi de guider la conception par la spécification. Ces prototypes peuvent guider à le processus de test et celui de conception en facilitant la comparaison de deux propositions [Cho87].

Enfin, les spécifications formelles peuvent jouer un rôle fondamental dans la réutilisation d'une part en donnant précisément les fonctionalités d'un module réutilisable et d'autre part en facilitant l'intégration d'autres modules lors du raffinement.

Quels sont les intérêts?

Les avantages des méthodes formelles sont dus aux aspects formels, à la précision et à l'abstraction [CLLF93, BH94, Som92].
Aspects formels:

- Des propriétés peuvent être établies par raisonnement formel, ce qui n'est pas le cas des autres formes de spécification. Les intuitions sont démontrables par une argumentation stricte.
- Les conséquences d'une spécification peuvent être mises en évidence. Ainsi, les contradictions sont détectées avant la réalisation.
- Les outils de preuve calculent et vérifient automatiquement et uniquement ce qui a été décrit.

Précision :

- Le domaine du problème est mieux perçu. A force de réfléchir sur le sujet, il est mieux compris, le vocabulaire est mieux défini, les imprécisions sont levées, les contradictions sont mises en valeur.
- Un langage commun lève les ambiguïtés et facilite la communication entre les acteurs. Si le langage est quelque peu hermétique, une conversion de la spécification en langage naturel garantit la qualité de la description.
- La spécification formelle est un avant-projet de la réalisation, qui permet au réalisateur de savoir exactement les modules qu'il doit programmer.

[8] Une description des systèmes formels et de leur interprétation est donnée dans [Lau86, WL88].

Abstraction :

- Seules les caractéristiques essentielles sont retenues, la spécification est plus concise,

- Une description abstraite est plus évolutive et plus perméable aux changements des besoins; elle améliore la maintenabilité du logiciel et facilite l'accès au logiciel de nouveaux acteurs du développement.

- Une distinction plus nette est établie entre étude et réalisation, dont les choix et les décisions sont différents. Ainsi, l'informaticien n'a plus un rôle aussi prépondérant dans les décisions de spécification.

- Les composants logiciels sont plus fiables et plus généraux donc plus réutilisables. Distinguer l'utilisation des composants (modes et contraintes) de leur réalisation est habituel dans d'autres d'autres génies (civil, mécanique, électrique).

Tous ces avantages se traduisent concrètement par des réductions de coûts en aval de la spécification et une rentabilité accrue des investissements en amont de la réalisation. La vérification est un point clé de la certification et de l'assurance qualité, surtout pour les systèmes critiques. Un effet de bord non négligeable est l'amélioration des rapports entre clients et prestataires de service.

Quels sont les problèmes ?

Ils sont avant tout pratiques et de deux ordres : d'une part les chefs de projet et les clients ont du mal à franchir le pas et d'autre part les méthodes formelles sont encore trop éloignées de leur préoccupations (outils de développement, méthode de développement de projets à grande échelle). Expliquons ces raisons en nous inspirant de [Dec88, Mey90a, ST90, Som92] :

- Accessibilité : les praticiens n'ont pas toujours la connaissance mathématique nécessaire. Les notations sont parfois hermétiques ou surchargées (plus de 100 symboles en Z). "*Malgré les progrès significatifs accomplis ces dernières années pour rendre les notations formelles plus compréhensibles et utilisables, la rédaction et l'emploi de spécifications formelles exigent toujours un certain niveau d'aptitude mathématique, ainsi qu'un effort important.*" [Mey90a].

- Rentabilité : l'investissement est lourd en termes de formation, d'écriture et de validation mais rentable à long terme.

- Domaine d'application : actuellement, les spécifications formelles ne sont appliquées qu'à un nombre réduit d'applications : "*Nous avons présenté dans ce texte une certaine manière d'aborder une petite classe de problèmes informatiques,...*" [Abr84]. "*Certains éléments des langages de programmation (le parallélisme, l'arithmétique en virgule flottante, les structures de données complexes) sont encore difficiles à modéliser de manière satisfaisante.*" [Mey90a].

- Diversité des notations : aucun standard ne s'impose (Z, VDM, spécifications algébriques, OSDL,...). Des normes existent mais ne sont pas compatibles. L'interopérabilité des spécifications formelles n'est pas encore prévue.

- Fiabilité relative : la notation formelle n'empêche pas une spécifications de mauvaise qualité. Les preuves sont laborieuses et souvent données de façon anecdotique dans les ouvrages. La spécification formelle aide à cerner les oublis et les contradictions, mais la preuve porte uniquement sur ce qui a été spécifié, pas ce qui était (implicitement) souhaité par le client.

- Faiblesse du processus : la démarche contractuelle respecte les principaux critères de la figure 16. Cependant, la mise en œuvre de la démarche est difficile dans les méthodes actuelles : le processus de formalisation, la preuve et le raffinement sont trop souvent laissés à la charge du concepteur. La théorie du raffinement est simple à comprendre mais son application est laborieuse.

- Faible structuration : la structuration d'applications complexes et leur gestion nécessite des concepts de haut niveau et des outils de manipulation. La dérivation de shéma en Z, le sous-typage ou la relation d'importation dans les spécifications algébriques ne suffisent pas.

- Faible support : les outils sont indispensables à la construction, la consultation, la preuve et la réutilisation des spécifications. Un environnement complet et convivial fait défaut même si des prototypes existent.

Le choix de la notation est primordial dans la compréhension de la spécification. Le formalisme doit être suffisamment abstrait pour conserver les qualités de la spécification mais suffisamment rigoureux pour permettre les preuves.

2.2.4 Classification des méthodes formelles

La spécification formelle de systèmes modulaires comprend une description abstraite des modules (données et traitements) et leur interactions [LB79]. Nous proposons deux types de classement dans ce contexte : un classement selon les modèles utilisés, qui résume les principaux courants de la spécification formelle actuels, et un classement selon les processus de développement. Guelfi propose une dichotomie intéressante entre spécifications orientées propriétés (logique) et spécification orientées modèle (combinaison d'objets fondamentaux) [Gue94], d'autres classification sont proposées dans [CG88, LB79, Wir93].

Classification par modèle

L'approche par **modèle abstrait** (Z [Hay92], VDM [Jon93], CLU [LG90]) définit le module par une structure de données (informatique ou mathématique) et un ensemble d'opérations. Les opérations sont des abstractions procédurales axiomatiques (pré- et post-conditions) ou opérationnelles (algorithmes). La concurrence entre modules est généralement implicite. Les structures de données mathématiques sont souvent abstraites et donc plus simples à spécifier, mais les preuves sont plus difficiles à réaliser et à automatiser [San90].

La **théorie des types** met l'accent sur la définition constructive du type. Les principales approches permettant le polymorphisme sont basées sur des extensions du λ-calcul [DT88, San90, Wir93]. Des résultats importants en sont issus concernant la démonstration automatique et la synthèse de programme.

L'approche **algébrique** [GH78, Mus80, EM85, San90, Bid89, Gau90] est une approche plus déclarative et abstraite que celle de la théorie des types. Le module définit un type de données mais aucune structure particulière n'est exhibée. Les propriétés du système sont décrites par des axiomes et sous certaines conditions, les preuves peuvent être automatisées (par exemple, par les systèmes de réécriture [JL86] ou les langages fonctionnels [San90]). La transition vers la réalisation est parfois automatisable si la logique des axiomes est un sous-ensemble des clauses de Horn.

L'approche **dynamique** modélise un module par un processus. Dans une telle spécification l'accent est mis sur les interactions entre modules plutôt que sur la structure. Ce type de spécification a été étudié dans les algèbres de processus comme CSP [Hoa85] ou CCS [Mil89] ou encore dans les systèmes de transitions (automates [Arn92], les réseaux de Petri) ou enfin dans les logiques temporelles [Arn92].

[AR94b] résume ces approches sur l'exemple classique mais riche des listes génériques. Une dernière approche, dite hybride, regroupe les langages de spécification s'inpirant de plusieurs courants. Elle est discutée dans la section 2.2.7. Enrichissons la classification ci-dessus par un croisement avec les niveaux d'abstraction.

courant / niveau	descriptif	constructif	
		mathématique	algorithmique
modèle abstrait	-	Z, VDM	CLU, ML
théorie des types	DEVA, Typol	Coq	Fun
types abstraits algébriques	Pluss, Obj, Clear, Sacso, Prospectra...	-	Sacso, Prospectra
processus	logiques temporelles LOTOS	CCS, CSP LOTOS	automates réseaux de Petri
hybrides	réseaux algébriques COLD, FP2, SDL, RSL	- RSL	réseaux algébriques SDL, COLD, RSL, FP2

Classification par processus de formalisation

La littérature insiste plus sur les modèles et langages que la démarche de spécification. Nous trouvons dans [FKV94] des éléments de classification des stratégies de développement formel. La taxonomie est dirigée selon deux axes : le processus de formalisation et l'aide à la formalisation.

processus de formalisation	direct		transitionnel			
			séquentiel		parallèle	
support à la formalisation	assisté	non assisté	assisté	non assisté	assisté	non assisté

La formalisation directe est le passage de la description informelle des besoins à une description formelle. Elle est adaptée aux prototypes ou aux projets bien structurés et de petite taille, dans lesquels l'analyste et l'utilisateur final sont proches et se comprennent. Actuellement, les outils d'aide sont syntaxiques et liés à des domaines particuliers.

La formalisation transitionnelle passe par différentes étapes, basées sur des modèles semi-formels puis formels. La formalisation séquentielle est le passage d'un modèle à un autre une fois le premier entièrement décrit. En parallèle, les spécifications formelles et semi-formelles sont produites par raffinements successifs et simultanés. Elle présente l'avantage de mieux faire découvrir et comprendre le problème et sa structuration et s'applique à de gros projets dans lesquels les utilisateurs peu habitués aux notations mathématiques peuvent intervenir. Le passage entre les différents modèles, s'il ne peut être entièrement automatisé doit être bien guidé sous peine d'introduire de nouvelles erreurs. L'aide à la spécification est ici principalement un support de traduction des modèles semi-formels aux modèles formels.

Critique

La classification n'a de sens que si elle permet de choisir un style de spécification en fonction d'un besoin. A notre connaissance, il n'existe pas d'étude comparative des différents courants de spécification formelle. Ceci s'explique par le fait qu'ils ont parfois des concepts orthogonaux (processus et types de données sont a priori disjoints) et des domaines d'application privilégiés : télécommunications et systèmes réactifs pour l'approche dynamique et SDL, sémantique des langages pour la théorie des types, systèmes séquentiels pour les approches algébriques et par modèles abstraits.

L'utilisation d'un courant de spécification dépend aussi fortement de l'intérêt qu'il suscite pour les industriels. En ce sens, l'approche par modèle abstrait, usuelle en programmation, est très attractive pour le concepteur. Pourtant elle pose le problème de la sur-spécification. La théorie des types semble difficilement applicable dans le développement d'applications même si des tentatives ont été faites en ce sens [Cas94b]. L'approche algébrique favorise la conception abstraite des composants logiciels mais pas leur assemblage.

Le mariage de plusieurs courants peut résoudre le problème des concepts orthogonaux si l'intégration ou le lien sont cohérents. En pratique, un courant mixte (voir section 2.2.7) est souvent basé sur un courant dominant: les données sont manipulées par des fonctions et transitent entre processus. Selon [ST90], la conception d'un langage de spécification parfait est illusoire. L'utilisation de différents langages en spécification et/ou en conception pose le problème de la vérification de cohérence.

2.2.5 Spécifications par modèle abstrait

Dans cette approche, un modèle particulier du type à spécifier est construit. Ce modèle possède un état, défini en terme d'autres types de données, et ce récursivement jusqu'à obtenir des combinaisons de types de base. Les constructeurs sont issus des mathématiques (ensembles, produit cartésien, séquences), des langages de programmation ou même des spécifications algébriques (listes, arbres).

Une spécification par modèle abstrait correspond à un système formel dont la syntaxe est donnée par les types et opérations du modèle abstrait et dont la sémantique est donnée par la théorie sous-jacente au modèle abstrait (ex: théorie des ensembles, logique du premier ordre, théorie des types, etc.). Nous allons présenter deux écoles majeures de l'approche par modèle abstrait, l'école mathématique et l'école algorithmique, sur un exemple simple de file d'attente d'hôpital.

Mathématique

Cette école est symbolisée par Z et VDM. La première technique fut VDM (Vienna development method) [Jon86], développée au laboratoire IBM de Vienne à la fin des années 70 et améliorée par la suite [Jon93]. Elle est largement pratiquée [Ari90]. Z est un langage de spécification basé sur la théorie des ensembles et la logique des prédicats [Spi89, WL88]. Il a été développé à partir des travaux de Jean-Raymond Abrial au PRG de l'université d'Oxford.

Les deux techniques reposent sur les même principes (spécification mathématique, preuve de propriétés abstraites, raffinement ou réification). VDM insiste plus sur les fonctions et le λ-calcul tandis que Z favorise le côté axiomatique (pré- et post-conditions) et la théorie des ensembles. VDM a été conçu à l'origine comme une méthode de développement de structures de données concrètes à partir de spécifications abstraites. A tel point que Tony Hoare conseille d'utiliser Z pour la spécification car elle est mieux organisée et VDM pour le développement [Hoa90]. Des éléments de comparaison entre les deux méthodes sont donnés dans [Hab93] et [WL88].

Nous présenterons plutôt Z, car les schémas, notion propre à Z, sont plus lisibles que les spécifications textuelles de VDM. La syntaxe utilisée ici est une version personnalisée du format LaTeXde Mike Spivey [Spi]. Une spécification Z se présente comme une compilation de schémas [Som92], comprenant des déclarations globales (variables, fonctions,...), des schémas d'états et des schémas d'opérations. Un schéma peut être paramétré par un type `T` quelconque. Un schéma d'état regroupe des variables et un prédicat sur ces variables. Un schéma initial est donné pour chaque schéma d'état.

$MAX ::= 20$

$HospitalQueue[T]$

$patients : seq\, T$

$\#patients \leq MAX$

$InitHospitalQueue[T]$

$HospitalQueue[T]$

$patients = \langle\rangle$

La séquence est un type de base de Z défini en extension par $\langle ... \rangle$. L'opération $\frown$ est la concaténation de séquences. Les schémas d'opération définissent des relations entre variables d'état et paramètres. Un schéma de pré-condition avec uniquement des paramètres en entrée (suffixés par un "?") est calculé :

$$
\begin{array}{|l}
\hline
Join[T] \\
\Delta HospitalQueue[T] \\
patient? : T \\
\hline
patient? \notin patients \\
\#patients < MAX \\
patients' = patients \frown \langle patient? \rangle \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
PreJoin[T] \\
HospitalQueue[T] \\
patient? : T \\
\hline
patient? \notin patients \\
\#patients < MAX \\
\hline
\end{array}
$$

La notation $\Delta SCHEMA$ est un raccourcis d'écriture pour définir un nouveau schéma contenant les déclarations du schéma $SCHEMA$ et une duplication de ces déclarations, suffixées par une quote (') exprimant la nouvelle valeur des variables. Les paramètres en sortie sont suffixés par "!". La définition d'une opération complexe, contenant par exemple des erreurs, se fait par fusion de différents schémas de pré-condition complémentaire. Le schéma invariant est noté Ξ.

$$
\begin{array}{|l}
\hline
LeaveN[T] \\
\Delta HospitalQueue[T] \\
patient! : T \\
\hline
patients \neq \langle \rangle \\
patients = \langle patient! \rangle \frown patients' \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
PreLeaveN[T] \\
HospitalQueue[T] \\
\hline
patients \neq \langle \rangle \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
LeaveErr[T] \\
\Xi HospitalQueue[T] \\
\hline
patients = \langle \rangle \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
PreLeaveErr[T] \\
HospitalQueue[T] \\
\hline
patients = \langle \rangle \\
\hline
\end{array}
$$

$$Leave \mathrel{\widehat{=}} LeaveN \ \vee \ LeaveErr$$

Les opérations `Length, IsEmpty, IsFull, IsIn` sont définies similairement par des schémas. Le calcul des schémas [Spi89, Hay92] est une algèbre des schémas avec des opérateurs d'ajout, de fusion, de renommage, etc. Ce mécanisme complexe permet la réutilisation, la preuve et le raffinement de spécifications.

La validation passe par des preuves de la spécification. Par exemple, soit la sortie du n^{ième} patient définie par

$$
\begin{array}{|l}
\hline
LeaveNnary[T] \\
\Delta HospitalQueue[T] \\
patient! : T n? : \mathbb{N} \\
\hline
n? \in 1..\#patients \\
\#patients > 0 \\
(\exists s : seq\,T \bullet \#s = n? - 1 \land patients = s \frown \langle patient! \rangle \frown patients') \\
\hline
\end{array}
$$

Théorème 2.2.1 (équité)
Si un patient entre en n^{ième} position dans la file alors il sera le n^{ième} à en sortir, soit $\vdash \forall Join; LeaveNnary \mid \#patients = n? - 1 \Rightarrow patient? = patient!$

Preuve :

$$
\begin{array}{ll}
\forall Join; LeaveNnary \mid \#patients = n? - 1 & \\
\quad = [patients, patients', patients'' : seq\,T; patient?, patient! : T; n? \mathbb{N} \mid & \\
\quad\quad patient? \notin patients \land \#patients < MAX \land & \\
\quad\quad patients'' = patients \frown \langle patient? \rangle \land & \\
\quad\quad n? \in 1..\#patients'' \land \#patients'' > 0 \land & \\
\quad\quad (\exists s : seq\,T \bullet \#s = n? - 1 & \\
\quad\quad\quad patients'' = s \frown \langle patient! \rangle \frown patients') \land & \\
\quad\quad \#patients = n? - 1] & [\text{développement de `;'}] \\
\quad = [patients, patients' : seq\,T; patient?, patient! : T; n? \mathbb{N} \mid & \\
\quad\quad patient? \notin patients \land \#patients < MAX \land & \\
\quad\quad n? \in 1..\#patients + 1 \land \#patients + 1 > 0 \land & \\
\quad\quad (\exists s : seq\,T \bullet \#s = n? - 1 & \\
\quad\quad\quad patients \frown \langle patient? \rangle = s \frown \langle patient! \rangle \frown patients') \land & \\
\quad\quad \#patients = n? - 1] & [patients'' = patients \frown \langle patient? \rangle] \\
\quad = [patients, patients' : seq\,T; patient?, patient! : T \mid & \\
\quad\quad patient? \notin patients \land \#patients < MAX \land & \\
\quad\quad \#patients + 1 \in 1..\#patients + 1 \land \#patients + 1 > 0 \land & \\
\quad\quad (\exists s : seq\,T \bullet \#s = \#patients & \\
\quad\quad\quad patients \frown \langle patient? \rangle = s \frown \langle patient! \rangle \frown patients')] & [\#patients = n? - 1] \\
\quad = [patients : seq\,T; patient?, patient! : T \mid & \\
\quad\quad patient? \notin patients \land \#patients < MAX \land & \\
\quad\quad (\exists s : seq\,T \bullet patients = s \land patient? = patient!)] & [\text{par } =_{\mathbb{N}} \text{ et } =_{seq}] \\
\quad \Rightarrow patient? = patient! & \\
QED &
\end{array}
$$

La spécification est ensuite raffinée pour introduire des structures de données proches de celles des langages de programmation. Chaque pas de raffinement doit être prouvé. Un assistant de preuve, appelé outil `B`[LLS91], est commercialisé. L'utilisateur doit être familiarisé avec la logique des prédicats. La notation Z nécessite un apprentissage important à cause de la syntaxe complexe. Z souffre du manque de constructions modulaires de plus haut niveau que les schémas pour spécifier des systèmes complexes. Ces remarques sont identiques pour VDM.

Algorithmique

Cette école est symbolisée par le langage CLU [LG90] ou encore par des langages fonctionnels comme ML ou CAML [WL93]. CLU (le nom CLU provient des trois premières lettres de *cluster*) est un langage de programmation supportant l'abstraction. L'abstraction est un mécanisme clé de la méthode de construction de programmes de Guttag et Liskov. Un programme CLU est formé d'un ou de plusieurs modules. Un module est une `procédure`, un `itérateur` ou un `cluster`. A chaque module correspond une abstraction.

Une **abstraction procédurale** est caractérisée par une spécification et par une implantation. Prenons l'exemple d'un hôpital traitant à la fois les humains et les animaux, appelés génériquement `patients`. L'hôpital est supposé défini par un type générique `hospital[patient]`. Soit une constante `max : int`. L'admission d'un patient à l'hôpital serait spécifiée par :

```
leave = proc (h: hospital[patient]) returns (hospital[patient])
     requires the queue is not empty
     modifies % this statement is useless here
     effects  cure the first patient and take it from the queue
```

L'implantation définie à partir de la structure de données du cluster `hospital` serait :

```
leave = proc (h: hospital[patient]) returns (hospital[patient])
     tagcase h
          tag empty: % each case must be mentioned
          tag non_empty(p: pair):
               return (p.last)
          end
     end leave
```

Un **itérateur** est une abstraction d'itération. Il permet d'introduire de nouvelles structures de contrôle munies d'exceptions. Nous nous contentons ici des itérations primitives du langage.

Une **abstraction de données** permet d'introduire de nouveaux types de données, munis de leurs opérations. Elle est spécifiée dans un type de données et implanté par un `cluster`. Un type de données CLU peut être paramétré par un autre type. Le mot-clé `requires` permet d'indiquer les opérations que doit posséder le type paramètre (pour un type de données ou une procédure). Soient les types prédéfinis `int` et `bool`.

```
hospital = data type[patient: type] is create, join, leave, size, is_empty, is_in, equal

Overview
     the hospital represents a queue of patients, waiting for any curation treatment
     the size of the queue is limited to max

Operations
create = proc ( ) returns (hospital[patient])
     effects  provide a new hospital waiting queue

join = proc (h: hospital[patient], p: patient) returns (hospital[patient])
     requires the queue is not full
              the patient is not already in the queue
     effects  provides a new queue with patient p and the patients of h

first = proc (h: hospital[patient]) returns (patient)
     requires the queue is not empty
     effects  provides the first patient to be cured

equal = proc (g,h: hospital[patient]) returns (bool)
```

```
     requires patient belongs an equal operation
          equal: proctype (patient, patient) returns (bool)
                   which is the congruence on patient
     effects  answer true iff g and h have the same elements in the same order
...
end hospital
```

Le `cluster` est une représentation particulière du type de données (mot-clé **rep**) donnée en terme d'union et de produit de type. Le mot-clé **cvt** fait le lien entre la spécification de l'objet et son implantation pour les paramètres et les résultats. Le mot-clé **rep** désigne la représentation du type, ici une union (`oneof`). Les cas d'erreur seront notés par une fonction `failure` bien que CLU possède des primitives pour le traitement exceptionnel.

```
hospital = cluster[patient: type] is create, join, leave, size, is_empty, is_in, equal

     rep  oneof[non_empty: pair, empty: null]
     pair struct[first: patient, last: hospital[patient]]

create = proc ( ) returns (cvt)
     return (rep$make_empty(nil))
     end create

join = proc (h: hospital[patient], p: patient) returns (cvt)
     if size(h) = max
          then failure(''the queue is full'')
          elseif ~is_in(h,p)
              then failure(''the patient is already in the queue'')
              else return (joinLast(h,p))) % fonction intermédiaire
          end
     end join

joinLast = proc (h: hospital[patient], p: patient) returns (cvt)
     tagcase h
          tag empty: return (rep$make_non_empty(pair${first: p, last: h})
          tag non_empty(pa: pair):
              return (rep$make_non_empty(pair${first: pa.first, joinLast(pa.last,p)})
          end
     end joinLast

first = proc (h: hospital[patient]) returns (patient)
     tagcase h
          tag empty:
          tag non_empty(pa: pair):
              return (pa.first)
          end
     end first

equal = proc (g,h: hospital[patient]) returns (bool)
     where patient has equal: proctype (patient, patient) returns (bool)
          if is_empty(g) then return (is_empty(h))
              elseif is_empty(h) then return (false)
                   elseif first(g) = first(h)
                        then return (leave(g) = leave(h))
                        else return (false)
              end  end equal
...
end hospital
```

La clause `where` correspond à la clause `requires` de la spécification. CLU permet la conception détaillée mais abstraite en favorisant la modularité. La spécification des clusters peut être donnée de manière encore plus abstraite via des **spécifications auxiliaires** et correspondant à des spécifications algébriques que nous allons étudier maintenant.

2.2.6 Spécifications algébriques

Une spécification algébrique de type abstrait de donnée, ou **type abstrait algébrique**, est la donnée d'une signature et d'un ensemble d'axiomes. La signature comprend des noms de types (les sortes) et des opérations définies par leur profil. La signature permet de construire toutes les valeurs des types de données par application des opérations. Les axiomes sont des formules logiques.

Une spécification algébrique est un système formel dont le langage est donné par la signature et le système d'inférence est basé sur les axiomes et la déduction équationnelle et dont l'interprétation est donnée en termes d'algèbres [Gal87]. Plus précisément, un type abstrait algébrique est une classe d'algèbres multi-sortes de même signature et de propriétés communes[Gau90]. L'annexe A présente les spécifications algébriques.

Les langages de spécification les plus connus sont CLEAR (Burstall, Goguen), ASL (Wirsing, Sannella), ACT1 (Ehrig, Mahr), LARCH (Guttag, Horning), OBJ2 (Futatsugi, Goguen, Jouannaud, Meseguer), LPG (Bert, Reynaud), PLUSS (Bidoit, Gaudel). AFFIRM (Musser), REVE (Lescanne) sont des exemples d'outils d'analyse et de preuve. Une liste complète des environnements est donnée dans [EC90]. En France, trois environnements opérationnels dominent : ASSPRO [BCC+87], LPG [Ber82] et SACSO développé au CRIN de Nancy [FLSV90].

Une opération a été lancé en 1990 par le GDR de programmation du CNRS, pour fédérer les équipes françaises menant des recherches dans le domaine des spécifications algébriques. Elle a abouti au projet intitulé SALSA [BBC+92], structure d'accueil permettant de communiquer entre les différents environnements cités ci-dessus et d'utiliser les outils spécifiques faisant l'intérêt de certains environnements (interprètes, solveurs logiques, démontrateurs de théorèmes, génération de code, etc.).

ASSPEGIQUE+

ASSPEGIQUE[Cho88] est un environnement intégré de développement de spécifications algébriques. Le langage de spécification algébrique de cet environnement est un sous-ensemble du langage PLUSS [Bid89, BGM87]. PLUSS (a Proposition of a Language Usable for Structured Specifications) est en fait une famille de langages (version standard, version fonctions partielles et logique du premier ordre, traitement d'exception). L'accent est mis sur la lisibilité de la spécification, avec une syntaxe proche du langage naturel pour la description des profils des opérations et des axiomes.

Une spécification PLUSS est découpée en unités élémentaires hiérarchiques qu'on appelle les `modules` de la spécification. Une distinction est faite entre les modules en cours de spécification (`draft`) et les spécifications complètes (`spec`). Les contraintes portant sur les spécifications complètes sont plus fortes (algèbre finiment engendrée) que sur les versions `draft` (tout modèle qui satisfait les axiomes est accepté et il n'y a pas forcément de générateurs). Les spécifications de base `basic spec`, à sémantique initiale, sont les atomes de la spécification.

Un module de spécification est la donnée d'une signature, de pré-conditions et d'axiomes. La signature définit l'ensemble des opérations partielles par leur profil. Les pré-conditions déterminent le domaine de définition. La surcharge et la coercition d'opérations sont autorisées, ainsi que l'union de sortes dans les profils. La syntaxe acceptée est proche du langage naturel, le caractère '_' placé entre les mots désigne les arguments. Dans tous les modules de spécification, les axiomes sont exprimés par des formules de la logique du premier ordre.

La hiérarchisation de la spécification se fait à l'aide de primitives. La primitive *use* enrichit la spécification d'un module par importation d'autres modules avec la contrainte suivante : la réutilisation d'un module de spécification ne doit pas modifier la classe des modèles de ce module. Une spécification qui n'importe pas d'autres spécifications est spécifiée par la construction `basicspec`, sa sémantique est initiale. Notons la possibilité d'enrichir des spécifications incomplètes (*draft*) par une primitive "d'héritage", notée `enrich`, plus souple que `use`.

PLUSS définit des primitives de contrôle de la visibilité des modules (`export, forget`). Le paramétrage de spécification se fait par le constructeur `proc` et l'instanciation par `as`. Une déclaration du module générique `Hospital` serait la suivante.

```
proc: Hospital (Patient);
    sort: Hospital;
    ...
spec: MyHospital as Hospital (Human);
    ...
```

avec un type `Patient` définit par

```
" type patient simple "
spec: Patient;
use: String;
sort: Patient;
generated by:
    toto :        -> Patient;
    tata :        -> Patient;
    tutu :        -> Patient;

end Patient
```

Une spécification ASSPEGIQUE+ de l'hôpital est donnée en annexe A.6. Des exemples concrets de spécification en PLUSS sont donné dans [BGM87, CBB+93, DM91].

2.2.7 Spécifications hybrides

Les approches hybrides cumulent différents styles de la classification de la section 2.2.4. Le panachage peut se faire avec des langages exprimant soit des aspects identiques soit des aspects orthogonaux du système à spécifier.

COLD permet la description de spécifications algébriques et par modèle dans un cadre unique (voir section 3.2.4).

Nombre de langages hybrides modélisent les systèmes par des processus ou des systèmes de transitions. Les données manipulées sont définies par des spécifications algébriques. Les aspects fonctionnels sont soit explicites dans les processus séquentiels soit implicites par transformation des données dans les systèmes de transition. LOTOS [GLO91] est un langage basé sur l'algèbre de processus CCS, enrichie par certains mécanismes de CSP. Les données sont décrites par le langage de spécification algébrique ACT-ONE. LOTOS présente l'intérêt d'être une norme ISO et d'être outillé (simulation, exécution, vérification). SDL définit une structure hiérarchique de circuits de communication dans lesquels circulent des données définies par des spécifications algébriques (voir section 3.2.5). FP2 [Huf89] est un langage applicatif définissant des processus séquentiels qui communiquent par des connecteurs. Les données circulant sur ces connecteurs sont spécifiées algébriquement. Les réseaux algébriques hiérarchiques [Gue94] définissent des réseaux de Petri dont les jetons sont des valeurs de types de données. Les places sont des stocks de valeurs et les transitions des transformations de données. RSL est inspiré de sources diverses telles que VDM pour la notation de base, les spécifications algébriques et les travaux sur ML pour les mécanismes de structuration, les algèbres de processus pour la concurrence CCS et CSP. Une telle diversité élargit le champ d'application de la méthode, appliquée industriellement, mais pose le problème de la cohérence entre les formalismes.

COLD et RSL sont des langages dits à spectre large car ils sont ainsi utilisables à plusieurs stades du processus de développement. Avoir un langage unique facilite le raffinement et l'apprentissage mais aussi la dérive opérationnelle des spécifications. Ce cumul des styles favorise la diversité des applications mais rend difficile la lecture des spécifications.

2.2.8 Conclusion

Les mérites des spécifications formelles sont désormais largement reconnus. Entre autre avantages, les spécifications formelles induisent une compréhension plus poussée du problème à résoudre et une meilleure utilisation de l'abstraction, facteur important de la conception du logiciel. Cependant, les spécifications formelles ne résolvent pas tous les problèmes soulevés par la conception et le développement du logiciel.

En particulier, un problème crucial est l'adéquation au besoin réel du client. La structuration modulaire facilite l'écriture, la compréhension et la modification des spécifications. Le prototypage est une réponse plus pratique à ce problème [CBB+93].

Un autre problème à résoudre est la mise en évidence des propriétés intéressantes de la spécification et leur preuve. La catégorie de théorèmes prouvables (équationnel, inductif, par l'absurde, etc.) et les propriétés globales de la spécification (cohérence, complétude) dépendent à la fois de la puissance de la logique acceptée et des outils présents dans l'environnement. Les alternatives sont : automatiser au maximum la preuve et réduire l'espace des théorèmes et la logique acceptée, laisser la démonstration à la charge du spécifieur mais l'aider dans la démonstration de lemmes.

La conception d'une solution logicielle par raffinement, transformation ou simplement inspiration d'une spécification est un point clé du génie logiciel. La structuration modulaire est encore un élément déterminant ici. Il est plus facile de raffiner des composants bien spécifiés que l'interaction entre ces composants. Si tous les langages proposent des opérateurs de structuration, SACSO dispose d'opérateurs de restructuration [DLS87]. Nous pensons toutefois que le langage de spécification et le langage de conception doivent être distincts pour favoriser l'abstraction. En ce sens, les descriptions algébriques semblent plus adaptées à la spécification que les modèles abstraits.

Enfin une des dernières motivations concerne la structuration d'un ensemble de spécifications dans des bibliothèques. Ce point rejoint le problème du découpage de la spécification, celui de la construction progressive de spécifications à partir de spécifications existantes, celui de l'identification de composants adéquats dans la bibliothèque. La réutilisabilité est un facteur clé de la rentabilité des méthodes formelles.

2.3 Développement à objets

L'approche à objets a des racines profondes et diverses, notamment Simula pour les classes (1960) et la représentation des connaissances en intelligence artificielle pour l'héritage. Elle a été adoptée par bon nombre de disciplines de l'informatique. Si cet aspect facilite l'intégration et la compréhension des différents travaux, il n'en reste pas moins que les définitions varient selon le domaine d'utilisation : un objet est une unité de connaissance, une unité de calcul, un module, une valeur d'un type, une "chose" ayant une existence propre... Un premier travail consiste pour nous à éclaircir le vocabulaire et définir intuitivement les concepts dans la section 2.3.1. Nous nous restreindrons aux modèles à objets et à classes de la classification de [Weg90] : "*Object-Oriented languages support object fonctionality, object management by classes, and class management by inheritance*". Deux autres "styles" à objets existent : les frames et les acteurs. Consulter [MNC+90] pour une description générale des langages à objets.

Le développement à objets marque un changement de culture [Mey89b], il ne s'agit plus de réaliser un projet particulier, mais d'investir dans un domaine en développant des bibliothèques de composants. "*The essence of object-oriented development is the identification and organization of application-domain concepts, rather than their final representation in a programming language, object-oriented or not.*" [RBP+91]. Dans la section 2.3.2 nous positionnerons l'approche à objets vis-à-vis du développement logiciel et une critique sera faite dans la section suivante.

Plus que toute autre classification, celle des méthodes d'analyse et conception est une tâche ardue voire impossible. Des critères de classification ont été donnés dans [DCF92, Gir91, MP92] : phases de développement prises en compte, concepts retenus, formalismes disponibles, notations graphiques, domaine d'application, outils support, sources d'inspiration, type de démarche, etc. Nous nous limiterons à la simple distinction entre les approches issues de méthodes déjà validées et les approches innovantes, que nous présenterons dans les sections 2.3.4 et 2.3.5. Nous illustrons notre discours par un exemple simple de gestion d'un magasin vidéo.

2.3.1 Concepts

Les définitions informelles qui suivent sont inspirées de [Gau90, Gir91, KM90, Mey88, Mey90a, MNC+90, RBP+91, Weg90]. "La conception par objet est une approche modulaire[9] dans laquelle le critère de cohésion s'appuie sur le modèle d'objet ou classe" [Gir91].

Objets et classes

Définir le terme "objet" reste une gageure. Girod en donne une idée à partir des différentes définitions du dictionnaire Larousse : matériel, visible, usage précis ... [Gir91]. Massini et al. le voient comme une entité regroupant des données et des procédures [MNC+90]. Coad en donne la définition suivante [CY91] : "un objet est une abstraction de quelque chose du domaine du problème ou de son implantation, montrant la capacité d'un système à conserver des informations dessus, interagir avec, ou les deux; une encapsulation de valeurs d'attributs et leurs services exclusifs". Nous prendrons la définition suivante inspirée de [Boo92].

Un ***objet*** *est une chose sur laquelle une action est réalisable. Un objet a un état et un comportement.*

L'état est l'ensemble des valeurs que détient l'objet. Le comportement est l'ensemble des opérations (procédures ou fonctions), appelées **méthodes** ou encore services, que l'objet peut réaliser. Un objet **encapsule** des données (l'état) et des traitements sur ces données (le comportement) au sein d'une même entité. En ce sens, un objet est un module. Les objets communiquent par envoi de **message**.

Un message est une requête adressée à un objet demandant l'exécution d'une méthode. Un message comprend un objet destinataire, appelé le ***receveur****, un nom de méthode, appelé sélecteur, des paramètres et parfois une continuation, à laquelle est transmis le résultat.*

Autrement dit, un envoi de message est une invocation d'opération, dans laquelle certains arguments désignent le ou les receveurs. Si le receveur est unique, il s'agit de mono-sélection ou sélection simple. Si plusieurs receveurs sont activables, il s'agit de multi-sélection ou sélection multiple[CL94, ADL91]. L'envoi du message est distingué de la recherche de la méthode. La structure et le comportement des objets similaires sont définis dans leur classe commune.

Une ***classe*** *est la définition d'un module implantant un type abstrait de donnée. Elle est caractérisée par un ensemble d'attributs ou variables d'instance, qui définit la structure des objets de la classe, et un ensemble de méthodes décrivant le comportement des objets.*

Dans le comportement certaines méthodes sont applicables aux objets, ce sont les **méthodes d'instance**, d'autres appartiennent à la classe et sont appelées **méthodes de classe**. Les méthodes de classe servent notamment à créer les objets de la classe. Un objet est instancié par sa classe. Les termes instance et objet sont synonymes. La classe représente à la fois l'ensemble de ses instances et un modèle des instances. La classe est donc une sorte de définition générique

[9] voir page 14.

de module dont le paramètre serait la structure de donnée. Une classe qui n'a pas de méthode d'instanciation est dite **classe abstraite**. De même, une méthode dont la sémantique n'est pas définie est dite **méthode abstraite** ou **virtuelle**.

Le lien entre type et classe n'est pas toujours très clair dans la littérature. Pour certains, ces deux termes sont synonymes [Boo92, MNC+90] pour d'autres il y a séparation [CW85]. Dans la suite de ce document, les deux formulations seront employées. C'est pourquoi, nous exhibons quelques différences, parfois subtiles, entre les deux.

- Une classe définit une structure pour les objets alors qu'il n'y en a pas besoin pour un type. Toutefois cette structure n'est pas a priori visible de l'extérieur et donc d'un point de vue interface, une classe se présente comme un type.

- Il existe des types de données, qui ne sont pas des classes, dans beaucoup de systèmes à objets (les types de base par exemple).

- Le type d'un objet est sa classe mais les valeurs d'une classe ne sont pas forcément ses objets si le modèle distingue les objets par une identité. Dans ce cas, chaque objet est référencé de manière unique dans le système. Les méthodes modifient éventuellement la "valeur" de l'objet mais pas son identité. Ainsi, un objet décrit non pas une valeur mais l'ensemble des valeurs du type associé à sa classe.

- L'envoi de message n'est une simple invocation d'opération car certains arguments, les receveurs, sont différenciés. Définir quel est le receveur d'une méthode quelconque est un problème épineux. Par exemple, `fermer(fenêtre, environnement)` est-elle une méthode de la fenêtre ou une méthode de l'environnement ou des deux? Cette distinction est importante pour l'héritage et le polymorphisme mais aussi pour le stockage et la recherche des méthodes.

- Une classe abstraite n'a pas d'instances mais définit un type muni d'un ensemble potentiellement infini de valeurs.

- Le sous-typage est plus strict que l'héritage. Ce point constitue un débat toujours actif [CW85, CHC90, Bre91, PPP91].

- D'autres mécanismes font que la classe est un instrument plus performant que le type en pratique : comportement dynamique, mécanisme d'instanciation, classe comme objet.

Relations entre objets et classes

Les relations sont celles de la section 2.1.4 auxquelles est ajouté l'**instanciation**, qui est une relation entre l'objet et sa classe.

La relation horizontale exprime ici le fait que pour réaliser ses fonctionalités un objet utilise les services d'autres objets. La relation d'importation[10] est aussi appelée **relation d'utilisation**, **de clientèle** et quelquefois **association**. Nerson distingue cinq types de relation d'utilisation (voir section 2.3.5). La relation d'inclusion est appelée **agrégation** ou relation **tout-partie**. L'objet de l'agrégation n'existe pas en dehors de son objet composite. Ces deux relations sont synthétisées au niveau de la classe.

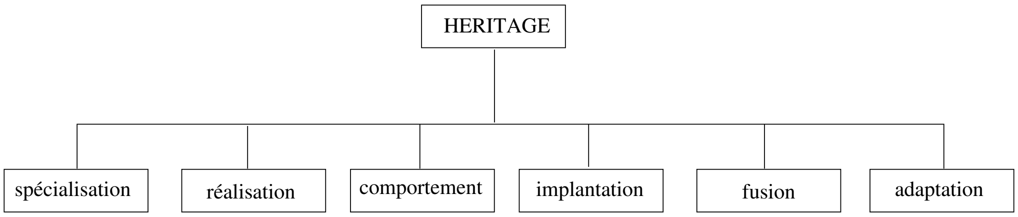
L'**héritage** est l'unique relation verticale, ce qui pose le problème très clair de sa définition formelle. C'est une relation entre classes appelée aussi relation de spécialisation/généralisation. L'héritage est un mécanisme permettant de définir une nouvelle classe (la sous-classe) à partir d'une classe existante (la super-classe) par extension ou restriction [Mey88]. L'extension se

[10] Lorsque cette relation peut être symétrique des problèmes de cohérence se posent pour les modèles sans identité d'objet.

fait en rajoutant des méthodes dans le comportement ou des attributs dans la structure. Par exemple, un employé est une personne qui a un contrat de travail. La restriction consiste à réduire l'espace des valeurs définies par la classe en posant des contraintes (conditions logiques à vérifier) sur ces valeurs. Par exemple, un carré est un rectangle dont les quatre côtés sont égaux. Certains modèles autorisent l'héritage multiple.

L'héritage est à la fois un mécanisme d'inférence permettant de définir des hiérarchies de généralisation/spécialisation, et un mécanisme de construction incrémentale de classes par réutilisation de code. Interprété autrement, une sous-classe représente soit un sous-type soit un raffinement du même type. L'héritage induit le polymorphisme : une instance de la sous-classe est aussi une instance de la superclasse.

Une classification des relations d'héritage est donnée en section 2.3.5. Girod va encore plus loin en donnant une taxonomie de l'héritage [Gir91]. La spécialisation est l'ajout de nouvelles caractéristiques. La réalisation est une implantation particulière. L'implantation désigne une utilisation interne pour implanter le type. L'héritage de comportement est utilisé pour ajouter un comportement donné (par exemple la classe `Model` de Smalltalk indique que ses descendants auront des dépendants). L'héritage combiné mélange plusieurs des styles précédents (héritage multiple uniquement). La fusion désigne plusieurs liens (égaux) de spécialisation indépendants. L'adaptation est la modification de caractéristiques.



Objet dynamique

L'évolution d'un objet au cours du temps constitue son **comportement dynamique**. Nous l'avons vu, les objets communiquent par envoi de messages. Le comportement dynamique d'un objet est donc assujetti aux messages qu'il reçoit. Un objet est dit actif (resp. passif) s'il renferme (resp. ne renferme pas) sa propre tâche de contrôle [Car91]. Il est séquentiel si une seule tâche de contrôle existe.

Avoir des objets dynamiques permet d'enrichir la sémantique de d'envoi de message. Plusieurs protocoles de communication sont possibles du point de vue de l'émetteur et du receveur : l'envoi peut être synchrone (l'émetteur se bloque jusqu'à recevoir le résultat) ou asynchrone (l'émetteur se bloque en attente d'un accusé de réception ou lorsqu'il a besoin du résultat ou pas du tout si le résultat est envoyé à un autre objet); la réception peut interrompre le receveur ou non, des priorités peuvent être accordées; l'envoi peut contenir ou non des informations.

envoi	asynchrone			synchrone
	accusé de réception	utilisation du résultat	continuation	
réception	prioritaire			ordre d'arrivée
	interruptible		non interruptible	
envoi	porteur d'information		non porteur	
	externe	interne	externe	interne

Plus généralement, l'étude des propriétés dynamiques d'un système se fait au travers du concept d'**événement** c'est-à-dire quelque chose qui se passe dans le système à un moment donné. Dans les méthodes temps-réel, telles qu'Electre [RCCE92], la notion d'événement est fondamentale et sa définition est très affinée, en terme de puissance (préemption, interruption), de stockage (fugace, mémorisé une fois, plusieurs fois), simultanéité ou non d'événements (synchrone/asynchrone). Dans les méthodes d'analyse et conception à objets, un envoi de message est considéré comme un événement selon la relation causale définie dans la méthode REMORA

[Rol86] par

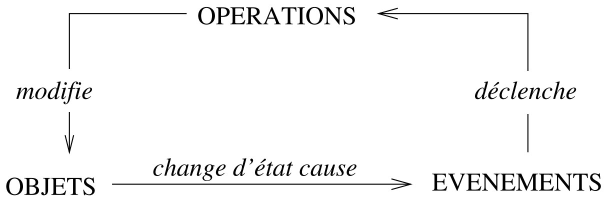


Figure 6 : Relation de causalité dynamique en objet

La méthode de Bari [Bar92], inspirée de REMORA, définit trois types d'événements : interne à un objet, externe (envoi de message) et temporel (définis dans un calendrier). Un prédicat et une priorité y sont éventuellement associés. En OMT [RBP+91] il n'y a pas d'envoi de message mais des émissions asynchrones[11] d'événements porteurs d'information. La réception d'événements déclenche l'exécution de méthodes, appelées actions. Dans la méthode de Shlaer et Mellor [SM92], les événements sont identifiés globalement dans le système et sont aussi porteurs d'informations. De Champeaux propose une notion similaire à celle d'événement : les `trigger` [DCAF92], inspirés des bases de données. Ces `trigger` peuvent être mémorisés ou non, et lever des traitements d'exception. Initialement, il n'étaient pas porteurs d'information. Mais plusieurs formes ont été ajoutées pour mieux modéliser les communications [WBJ90] : déclenchement avec accusé de réception, envoi sans attente (asynchrone), envoi avec accusé de réception, envoi avec attente du retour (synchrone).

Le comportement dynamique d'un objet est parfois explicité par un système de transition ou un terme de la logique temporelle. Le comportement global est obtenu par composition des comportements dynamiques de chacun des objets. Certains modèles comme les flots de données répartissent les fonctions globales du système sur les différents objets.

Classes et sous-système

La classe ne suffit pas en général à structurer un système. En ce sens, la notion de sous-système est utile [Szy92] que ce soit un simple sucre syntaxique ou une entité de première classe avec un protocole de manipulation propre. Les deux axes habituels de la structuration sont une fois de plus appliqués.

Un sous-graphe d'héritage donne les différentes spécifications et implantations d'un type quelconque, appelé `schéma` dans le modèle des classes formelles [ACR94], `domaine` dans Mecano. Elle correspond aussi, en quelques sortes, aux `classes abstraites`.

Un **module** est ensemble composite de classes réalisant un même objectif. Il est appelé `application` dans Mecano [Gir91], `schéma` dans la méthode Class/Relation [Des92], `module` dans OOZE [AG91] ou encore `sujet` dans OOA [CY91], `cluster` dans la notation BON [Ner92], un `sous-système` dans la méthode Class-Responsabilities [WBWW90]. Des modules existent en OMT [RBP+91] mais n'ont pas de représentation graphique particulière. Cet aspect est approfondi à la page 41.

Autres concepts

La puissance d'expression des langages à objets est augmentée en considérant que les méthodes ou les classes sont des objets à part entière (création dynamique, passage en paramètre, affectation, envoi de message). La métaclasse répond à cette demande. Une **métaclasse** est une classe dont les instances sont des classes [Coi87]. Elle permet le partage d'informations entre plusieurs instances d'une classe[12]. Le principal problème est la vérification du code créé dynamiquement (typage, analyse).

[11] Un envoi de message se traduit alors en deux événements distincts : l'émission de la requête et la réception de la réponse.

[12] Ce point est particulièrement intéressant pour la modélisation sans redondance des informations.

2.3.2 Analyse et conception à objets

Après avoir révolutionné la programmation, les objets se généralisent maintenant aux autres couches du développement du logiciel. Ceci s'est fait en deux étapes : d'abord avec dans la conception du logiciel, puis dans l'analyse. En fait, si tout le monde est convaincu de l'apport de l'objet en conception, plusieurs écoles s'affrontent en ce qui concerne l'analyse à objets.

Certains auteurs, comme Rochfeld [Roc91, RB93], considèrent que les méthodes systémiques traditionnelles, comme Merise, suffisent à l'expression du besoin, et que les objets sont utiles pour concevoir des solutions modulaires réutilisables. D'autres proposent des méthodes dédiées, comme Hood [CI88]. D'autres encore définissent uniquement des méthodes de conception [Bar92, Bri93, Des92, Gir91, WBWW90], dans lesquelles les objets et leurs relations sont d'abord identifiés puis implantés. Noter que certaines méthodes de conception sont issues de l'écriture des classes des langages de programmation, ainsi Eiffel est à l'origine de la méthode de Nerson [Mey90b, Ner92] et Smalltalk de la méthode de Wirfs-Brock [WBJ90]. Enfin, dans les méthodes d'analyse et conception à objets, la vue qui s'impose est d'avoir des objets d'analyse (reflétant le domaine d'investigation) et des objets de conception certe issus de l'analyse mais aussi de domaines techniques de la construction du logiciel, comme le montre la figure 7, extraite de [MP92].

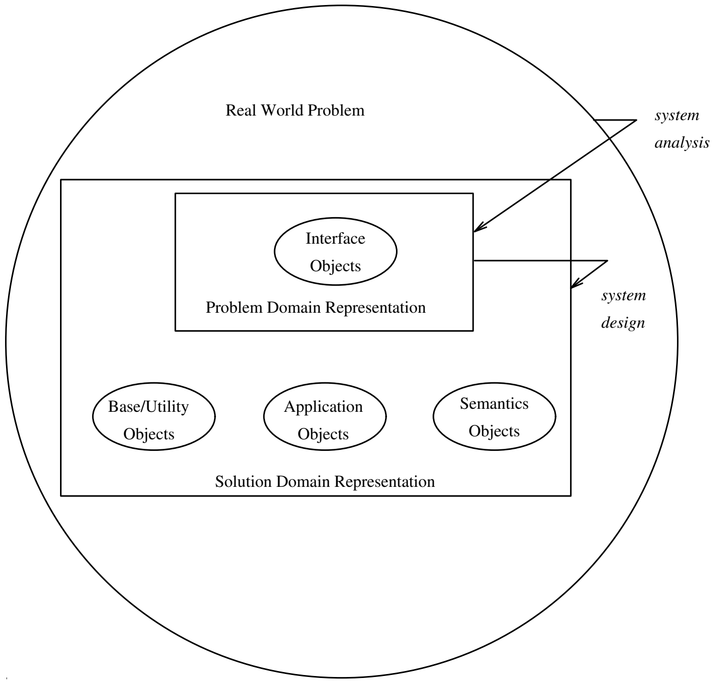


Figure 7 : Relation entre analyse et conception à objets

Les **objets sémantiques** sont des objets décrivant le domaine du problème (client, cassette vidéo, magasin). Les **objets interfaces** sont liés à la communication homme-machine, c'est-à-dire la vue utilisateur sur les objets du domaine (fenêtre, souris, icône). Les **objets d'application** expriment le contrôle général du système tel que le démarrage ou de grosses fonctions (classe root Eiffel, prêt/retour de cassette). Les **objets de base** sont des structures de données indépendantes de l'application et du domaine utilisés par les autres objets (listes, réels, caractères). Shlaer et Mellor [SM93] définissent plusieurs “domaines” relativement indépendants : application, architecture logiciel, matériel, interface, SGBD, dispositifs physiques d'interface homme-machine ou machine-machine. Dans la méthode Objectory [JCJO91], les objets d'analyse (cas d'utilisation, entité, objet interface et service) sont distingués des objets de conception (composant, bloc, objet).

Le modèle à objets est présent à tous les niveaux d'abstraction. L'approche est donc de type raffinement selon la classification des figures 2 et 3. La conception consiste à trouver les objets supplémentaires et à les agencer. En OOD [CY91] la même démarche est utilisée en analyse et en conception mais pour quatre types de composants en conception : les composants du domaine du problème (analyse), les interactions humaines, la gestion des tâches et la gestion des données. La méthode de conception Mosaïc [Bri93] pousse encore plus loin cette démarche multi-domaine en intégrant dans un même modèle les systèmes à objets, les bases de données et les systèmes d'interface.

Modèles

Nous donnons ici quelques exemples de modèles utilisés en analyse et conception à objets [Hod91] pour mettre en valeur le problème de la cohérence d'une spécification multi-modèles (cf section 2.1.4). Ces modèles sont souvent accompagnés d'une notation graphique telle que celles présentées dans [EHS93, HSE90].

- Modèle essentiel : description rigoureuse et non ambiguë de la portée et des effets généraux du système,
- Scenarii d'utilisation : description du flot de contrôle suivi entre les objets pour chaque événement reçu par le système,
- Modèle objet : expression des liens statiques entre les objets.
- Modèle d'interaction entre objets : expression des communications entre les objets. Ces dernières correspondent aux collaborations définies dans les scenarii d'utilisation.
- Modèle des traces d'événements : pour chaque objet, un modèle exprime la réponse aux événements reçus.
- Modèle état-transition : il exprime le comportement dynamique des objets en montrant la coordination entre les événements reçus et éventuellement ceux générés.
- Hiérarchie d'utilisation : vue globale du contrôle entre les objets sans détailler les opérations.
- Graphe des classes : représentation des propriétés communes dans un graphe d'héritage.
- Spécification des classes : définition externe des interfaces des classes,
- Définition des classes : description de l'implantation des classes, leurs caractéristiques privées, les méthodes et les communications internes.

Cycles de vie

En développement objet, il n'y a pas un mais plusieurs cycles de vie possibles, car aucun processus ne domine actuellement [Hod91, HSE90]. Nombre de méthodes utilisent des processus linéaires (OMT [RBP+91], OOAS [SM92], OOA [CY91], classe-relation [Des92]). D'autres méthodes prennent une orientation plus modulaire, symbolisée par le modèle de cluster de Meyer [Mey89b]. Certaines ont même une orientation composant comme dans la méthode OOAD de Nerson [Ner92] : méthodes OORASS [AR92a] ou Class-Responsabilities [WBWW90]. Henderson-Sellers définit un cycle cohérent par une itération sur une analyse ascendante et une composition descendante (modèle de la fontaine) [HSE90].

2.3.3 Utiliser les méthodes à objets

Les méthodes à objets visent à maîtriser le développement en le découpant en parties relativement indépendantes. Il ne s'agit pas simplement d'une application de la technique diviser pour règner car chaque sous-problème est traité comme un problème à part entière et sa solution est généralisé à un ensemble de problèmes similaires. Les méthodes à objets sont donc adaptées aux projets de grande envergure. Le modèle à objets induit donc une nouvelle culture du développement, plus proche des contraintes industrielles. Il s'agit d'investir sur le long terme en définissant des composants généraux réutilisables et adaptables pour rentabiliser les efforts de développement.

Quels sont les intérêts ?

La modularité et sa cohérence intrinsèque du modèle à objets sont le support des avantages suivants :

- concept unificateur : l'objet est un concept simple compris par les différents acteurs du développement. Il améliore la lisibilité et l'uniformité des spécifications ainsi que la traçabilité entre les étapes du développement. Il peut aussi servir à modéliser des applications dans des domaines variés et pour des environnements éclectiques [GM92] (interopérabilité). Enfin, l'objet facilite la distribution du contrôle d'une application sur une architecture parallèle.

- outil de structuration : l'objet est un puissant outil de structuration des spécifications dont il améliore sensiblement la modularité, la cohérence et l'autodocumentation. A la fois l'écriture et la maintenance d'applications sont facilités car le système est évolutif et extensible. La répartition du travail est assouplie car l'écriture des composants et la construction du système entier sont des tâches séparées.

- abstraction et généralisation : l'héritage permet la description d'unités de spécification à des niveaux d'abstraction différents et supporte ainsi le raffinement. Avec un modèle commun de conception, tel que celui des classes formelles (voir chapitre 6), le portage est automatisable en partie.

- réutilisabilité : la relation d'utilisation permet d'incorporer des composants déjà réalisés, donc de limiter les erreurs et d'accélérer le développement. Un composant peut ainsi être remplacé par un autre composant ayant les mêmes fonctionnalités mais plus concret ou plus efficace. La relation d'héritage permet d'adapter des composants à un besoin particulier. L'effort fourni est donc fortement diminué.

Résumons ces qualités : productivité, évolutivité et maintenabilité, applications critiques difficiles à réaliser avec les techniques habituelles, maîtrise de la complexité et prototypage.

Quels sont les problèmes ?

Les méthodes à objets sont un apport indiscutable en termes de modèles, mais pas encore pour les processus. Dans les deux cas, un manque de formalisation et de normalisation engendre une grande confusion et empêche l'obtention des avantages ci-dessus.

- Bien que présenté comme une démarche naturelle de l'esprit humain, la structuration avec des objets n'est pas triviale. Notre compréhension naturelle de beaucoup de systèmes est plutôt fonctionnelle et il est quelquefois difficile de s'adapter à une compréhension orientée objet [Som92]. En ce sens, définir un objet indépendamment du reste est simple mais exprimer un ensemble d'objets coopérants est encore très difficile.

- Pour garantir la réutilisabilité des composants, il faut les exprimer dans un modèle formel. Si la spécification des composants est multi-modèle, il faut prouver sa cohérence [HC91]. Les méthodes actuelles ne définissent pas de modèles formels et les notions de contrat ou de responsabilité sont informelles.

- Il manque une norme. Par exemple, les bases de données objet ont du mal à remplacer le modèle relationnel, bien que pour les transactions complexes l'objet soit plus performant. Actuellement, la réutilisation de composants n'est guère pratiquable qu'à l'intérieur d'un même langage.

- Des critères de bonne analyse ou bonne conception objet font défaut [WBJ90]. L'héritage est souvent utilisé à tort et à travers, il y a parfois même confusion entre héritage et agrégation. En analyse, l'héritage correspond à la spécialisation tandis qu'en conception l'héritage d'implantation est aussi utilisable. Des règles précises de contrôle de l'héritage garantissent un contrôle de type sûr [Car88, Mey88, ACR94].

- La séparation entre analyse et conception s'estompe souvent en pratique, surtout avec un cycle itératif. Des dysfonctionnements existent : la structure des objets apparaît en analyse sous forme d'agrégation mais pas en conception (encapsulation forte). Un objet est-il du domaine du problème ou de la conception? La conception est-elle un raffinement de l'analyse ou une activité indépendante? Doit-on utiliser les mêmes modèles? Ces question montrent l'absence de formalisation du processus de développement, condition essentielle pour son automatisation.

- L'objet est l'unité atomique de structuration, mais l'architecture de systèmes volumineux implique la notion de module. Il est souhaitable de pouvoir intégrer de manière cohérente différentes vues et différents niveaux d'abstraction d'un système [MP92].

- La gestion d'un grand nombre de composants logiciels nécessite des moyens pour cataloguer, comparer, analyser, rechercher des classes ou des groupes de classes dans une bibliothèque. "*La réutilisation massive nécessite une bonne organisation de la communauté des développeurs, prêts à partager des idées, des méthodes, des outils et du code...supporté par des systèmes d'information logiciel qui gérent l'accès aux collections de classes*" [GTE+90].

- D'autres griefs sont faits à la technologie à objets actuelle : maîtrise de l'asynchronisme des envois de messages, gestion de configuration, SGBD industriels supportant un grand nombre d'utilisateurs, catalogues d'objets, règlementation sur la propriété et le droit d'auteur, les plates-formes (absent des gros systèmes), etc.

2.3.4 Approches remixées

Les formalismes utilisés dans l'analyse à objets sont issues de techniques éprouvées par des méthodes plus anciennes. L'"astuce" a consisté à mettre dans un même concept les différents points de vue d'un système, avec quelques enrichissements.

Ainsi le modèle statique est une révision des formalismes de modélisation des données (Entité-Association), incluant l'héritage, la relation d'agrégation, les opérations et parfois les métaclasses. Le modèle dynamique, qui définit le cycle de vie des objets, reprend les modèles des méthodes temps-réel (SA/RT, RdP, Statecharts). Le modèle fonctionnel correspond à l'approche des flots de données des méthodes SADT ou SA/RT.

De même, le processus de conception est assez classique avec des étapes d'analyse, de conception et d'implantation. L'analyse à objets est souvent une modélisation de système via une structure, un comportement dans le temps et des fonctions réalisées. La dynamique du système est en général modélisée par des événements alors que les objets communiquent par envoi de message.

OMT [RBP+91] propose des modèles sur les trois plans mais dont la cohérence est parfois difficile à contrôler et la conception assez fastidieuse. OOA&D de Coad et Yourdon [CY91] enrichit progressivement un modèle des objets par des fonctions et des contraintes dynamiques. OOAS de Shlaer et Mellor met l'accent sur la vue dynamique du système et sur une modélisation classique des informations. OOS [Bai89] de Bailin sépare objets actifs et objets passifs. Des flots de données relient les objets actifs, qui peuvent groupés. La méthode de Bari [Bar92] modélise entièrement le système par des données et des événements. OOM [Roc91, RB93] ajoute des caractéristiques objet à une modélisation des données en Merise.

Un exemple : la méthode Object Modeling Technique

L'exemple traité est celui d'un club vidéo ayant une activité de prêt de cassettes vidéo. Voici une description succincte. Le club dispose de plusieurs boutiques où ses adhérents peuvent emprunter des cassettes, qu'ils rapportent par la suite. Un adhérent peut emprunter un nombre quelconque de cassettes en une ou plusieurs fois mais il ne peut faire plusieurs emprunts le même jour dans la même boutique. Il lui est interdit d'emprunter une cassette s'il a dépassé la date de retour prévue d'une des cassettes qu'il a en emprunt. Lorsqu'il peut emprunter une cassette, l'adhérent donne le nombre de jours pendant lesquels il compte la garder, compte tenu des jours d'ouverture. L'adhérent peut rendre des cassettes (toutes ou en partie) empruntées par lui ou par un autre adhérent et il n'est pas obligé d'effectuer ce retour dans la ou les boutiques où l'emprunt a eu lieu. Le prix de location d'une cassette est proportionnel au nombre de jours d'emprunt. Le club veut pouvoir : gérer son stock de cassettes, les emprunts et les retours des cassettes des adhérents, enregistrer de nouveaux adhérents (la suppression des adhérents n'est pas prévue), effectuer des statistiques.

Que ce soit en analyse ou en conception, trois modèles sont donnés : le modèle des objets (structure statique du système), le modèle dynamique (évolution des objets dans le temps), et le modèle fonctionnel (description des calculs du système). D'autres notations complètent ces modèles.

Modèle des objets Le modèle des objets décrit les objets du système et les relations statiques entre ces objets. Il s'inspire des concepts et notations issues des modèles de données de type Entité/Association et de la programmation à objets. La notion de clé d'entité est parfois redondante avec l'identité implicite des objets.

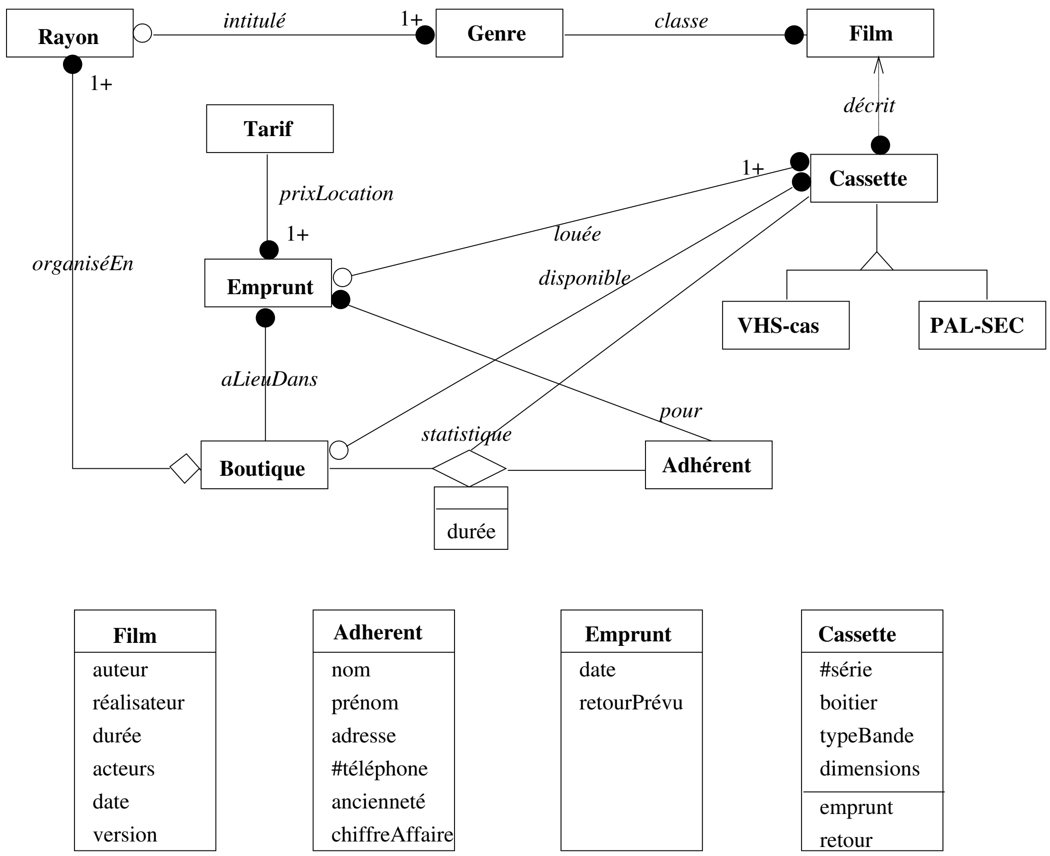


Figure 8 : Le modèle à objets OMT de l'application vidéo

———	association entre deux classes
△	relation de spécialisation/généralisation
⋄	relation d'agrégation (attachée à une classe
◇	relation ternaire (entre trois classes)
•	cardinalité 0-n
∘	cardinalité 0-1
1+	cardinalité 1-n
.... >	relation d'instanciation

Modèle dynamique Le modèle dynamique exprime le contrôle des objets par un diagramme d'état, et parfois des diagrammes supplémentaires (événements, scénarii, etc.). Les événements représentent des stimuli externes tandis que les états représentent des valeurs des objets. Le modèle dynamique est défini par un automate structuré de type StateCharts [HLNP90]. Les états sont des processus interruptibles et les transitions des interruptions (événements) auxquelles sont associés des traitements atomiques et des gardes pour enrichir le formalisme. Un diagramme d'état peut être fini ou infini. Un état initial (•) exprime la création d'un objet et un état final (⊙) sa destruction. Dans un diagramme infini, l'état initial est facultatif et il n'y a pas d'état final.

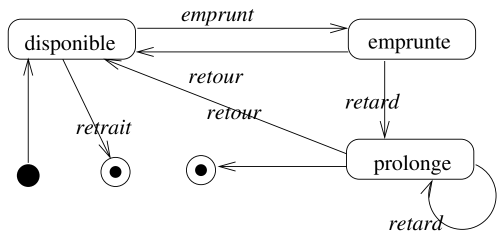


Figure 9 : Le modèle dynamique OMT de l'objet `Cassette`

Le formalisme autorise l'émission d'événements vers un type d'objet, ainsi que la généralisation et l'agrégation d'états. La généralisation est l'expansion d'activités emboitées, elle permet de définir des hiérarchies d'états (inclusion de super-états) et d'événements (hiérarchies d'abstraction). L'agrégation est la concurrence entre états.

Modèle fonctionnel Le modèle fonctionnel exprime les transformations de données dans les traitements par un diagramme de flots de données. Les données et les traitements sont identifiés par les objets qui les détiennent. Ce modèle décrit les fonctions globales du système sans spécifier qui les réalise ni à quel moment elles le sont. Il n'est pas très développé dans la méthode, mais correspond pourtant à un besoin majeur d'abstraction du comportement du système.

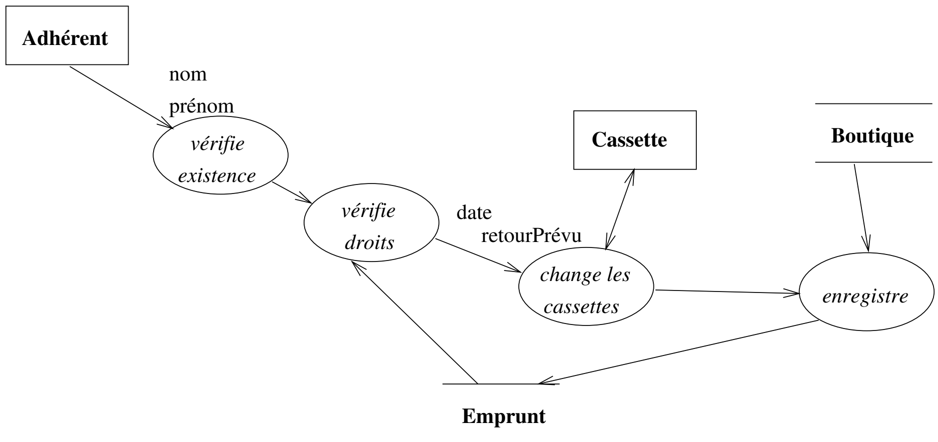


Figure 10 : Le modèle fonctionnel OMT de la fonction `emprunt`

2.3.5 Approches innovantes

Pour les nouvelles approches, ce qui prime c'est la définition individuelle des objets et celle de l'assemblage des objets dans un système. Citons entre autres Objectory [JCJO91], Class-Responsabilities [WBWW90], OORASS [AR92a], HOOD/PNO de Paludetto [Pal91], OOA&D de Nerson [Ner92], OBA de Gibson [DCF92]. Nous distinguerons les points suivants : le composant logiciel, la gestion des composants, la maîtrise d'un ensemble de composants interagissant. Ensuite, nous donnerons un point de vue et un exemple.

Composant logiciel

Un composant (serveur) est défini par les services qu'il offre et les contraintes qu'il implique. Les services sont les opérations du composant. Les contraintes sont de deux ordres : primo, celles qui portent sur le client (type, opérations, etc.) secondo celles qui portent sur les services attendus d'autres composants pour réaliser ses propres services. Les contraintes qui portent sur le client relèvent de la formalisation des interactions.

Les contraintes qui portent sur les autres serveurs concernent a priori l'implantation, c'est-à-dire les composants utilisés pour la réalisation des services. Ce problème est similaire à celui des règles de visibilité des modules. Dans la méthode OMT, les composants utilisés sont précisés dans les événements émis, au risque de sur-spécifier. Les types abstraits en revanche utilisent, pour spécifier les services, des "classes" qui ne sont pas forcément celles utilisées concrètement. La description abstraite est ensuite raffinée par une implantation respectant la description abstraite, avec de nouveaux objets et de nouvelles classes. En ce sens le formalisme de HOOD est agréable, car il met en avant les services (et même le type de communication accepté) tandis que la relation parent-enfant définit les objets de réalisation.

La culture composant implique un effort dans la description et la réalisation des composants [GTE+90]. Pour Meyer une spécification formelle des objets est indispensable "*Object-oriented Design is the construction of software systems as structured collections of abstract data type implementations*" [Mey88].

Gestion des composants

La culture composant implique aussi des moyens d'archivage et d'accès aux composants, désigné sous le concept d'organisation de classe dans [GTE+90]. Les relations entre classes y sont décrites (héritage, instanciation, dépendance). Ces relations servent à "se déplacer" dans le réseau de classes par des fouineurs (browsers Smalltalk ou Eiffel).

La recherche de composants peut aussi se faire par affinités et comparaisons. Il nous faut donc définir une description primitive des classes pour accélérer la comparaison. Cette définition doit être formelle pour éviter des ambiguïtés.

La gestion des composants comprend aussi la maîtrise de leur évolution. L'expérience montre que les classes stables et réutilisables ne sont pas créées à partir de rien, mais découvertes dans un processus itératif de test et d'amélioration [GTE+90, Coa92]. Son champ d'investigation est élargi au fil des développements. La modification des hiérarchies d'héritage ne se fait pas uniquement par extension des classes terminales (i.e. sans descendance) mais par réorganisation de la hiérarchie. Il est intéressant d'introduire des heuristiques de restructuration (métrologie) car la structuration d'un grand nombre de classes, surtout en héritage multiple est loin d'être triviale. La généralisation de comportement (i.e. le regroupement de plusieurs sous-classes dans une super-classe unique) évite l'explosion combinatoire du nombre de classes. Des critères d'appariement entre classes pour automatisent l'opération.

L'évolution peut aussi prendre en compte des environnement de programmation différents. Il est souhaitable de donner des définitions de classes indépendantes de l'environnement, qu'on peut ensuite traduire et personnaliser pour prendre en compte des caractéristiques propres au langage. Cet historique du développement est appelé **versionnement de classe**.

Maîtrise d'un ensemble de composants interagissant

Par essence, un système à objets a un contrôle décentralisé, sans maîtrise de l'ensemble des communications. Il faut donc de la rigueur dans le protocole d'interaction entre les objets. Une fois donnée la description des composants, il faut exprimer "la colle" entre les composants. Cette colle exprime les relations entre des objets de niveau d'abstraction différents. Au niveau atomique, la communication se fait par envoi de message. A des niveaux plus abstraits, des ensembles de communications et des collaborations d'objets synthétisent les communications. Nous examinons quatre voies, non exclusives, pour contrôler l'évolution du système.

Formaliser les interactions Un effort est fait pour préciser les interfaces entre les objets et la spécification des liens. Des contrats établissent clairement les services offerts [Mey92] et les obligations de chacune des parties (le client et le serveur). Ces contrats peuvent se traduire axiomatiquement par des pré-conditions, des post-conditions et des invariants. Dans la méthode OORASS [AR92a], les communications sont modélisées par des rôles. Un modèle de rôle est une unité de conception. Il comprend plusieurs entités, appelées rôles, participant à l'exécution d'un comportement exprimé dans le modèle. Par exemple, l'adhérent A rend une cassette vidéo CV à l'employé du magasin M. Comme pour les vues multiples, les rôles sont ensuite composés jusqu'à obtenir les objets "complets". Les collaborateurs habituels sont ainsi regroupés. Les collaborations de la méthode CRC [WBWW90] représentent des requêtes d'un client vers un serveur. Une collaboration supporte un flot de contrôle et d'information entre deux objets. Un objet collabore avec d'autres objets pour assumer une responsabilité (i.e. connaissance et actions d'un objet) [Gir91].

Etablir des liaisons abstraites Une liaison abstraite est une abstraction des communications entre deux objets. Elle peut être symbolisée soit par des canaux de communication comme en CSP [Hoa85], soit par des méthodes abstraites regroupant un ensemble de communications avec un protocole particulier pour exprimer la sémantique abstraite comme dans les couches ISO des réseaux de données. Les deux types pouvant aussi être rangés dans des classes abstraites. Si la méthode contient une modélisation des données, les associations peuvent parfois être interprétées comme des canaux statiques de communication. Les liaisons abstraites peuvent parfois être décrites via des objets interface, comme dans le paragraphe ci-dessous. Ainsi, le retour de la cassette se fait par une personne représentant le client (souvent le client lui-même) et un employé représentant le magasin.

Définir des objets de contrôle Des grandes fonctions peuvent être affectées à des objets dits d'application dans [MP92] qui prennent en charge le contrôle d'une tâche complexe définie pour un ensemble d'objets et pour laquelle aucun objet n'est "logiquement" responsable. Ainsi définir et imprimer les statistiques d'emprunts du magasin vidéo est une tâche globale, prise en charge par un objet de classe `Statistique` appelé périodiquement. Cette idée se retrouve dans la méthode Objectory [JCJO91]. Dans la méthode MCO [Cas91], des administrateurs d'objets et des serveurs d'objets centralisent le service et les communications des objets. Les notions de classe et d'administrateur sont assez proches.

Regrouper les objets en sous-systèmes Un graphe de collaboration de la méthode CRC [WBJ90, WBWW90] analyse les chemins de communication et identifie les sous-systèmes potentiels. Une notation graphique exprime ces collaborations entre classes et sous-systèmes. De Champeaux définit des `ensembles` comme une sorte d'objet haut niveau avec parallélisme interne [WBJ90]. Un ensemble est un module avec une interface abstraite, éventuellement des attributs et un diagramme de transition d'états. Une différence importante entre objets et ensembles est que les ensembles ont un mécanisme de retardement pour les déclencheurs et les messages entre des entités externes (objet ou ensemble) et des constituants de l'ensemble. Ralph Johnson propose une structuration similaire avec les `framework` [WBJ90]. Un `framework` est est une collection de classes et les interfaces entre elles. La triade MVC `modèle/vue/contrôleur` de Smalltalk en est un exemple [LP90]. Un framework est différent d'un sous-système dans la mesure où il peut être raffiné, comme une classe abstraite. Un dernier terme devient à la mode, ce sont les motifs ou `pattern`. Un motif est une abstraction d'un ensemble de classes, souvent réutilisables dans un développement à objets [Coa92]. Les motifs sont découverts par "essais successifs" et par observation de corrélations entre classes dans divers systèmes existants. L'auteur distingue et motive sept motifs: descriptif, temporel, événementiel, rôle, participation à l'état, participation au comportement, diffusion. La conception par identification/placement/généralisation de motifs devient une démarche de conception et de gestion de classes. Reenskaug définit les objets par un ensemble de rôles, c'est-à-dire d'interactions avec des objets différents [AR92a]. Ces rôles peuvent être abstraits dans des hiérarchies. Dans la méthode HOOD/PNO[Pal91], les objets décrits par des réseaux de Petri sont composés (et donc synchronisé) pour faire des objets de plus haut niveau.

Un exemple : la méthode Object Oriented Analysis and Design

Comme la méthode Mecano de Girod [Gir91], la méthode OOA&D [Ner90, Ner92, Wal93] de Nerson est issue du langage Eiffel. La méthode OOA&D a pour objectif d'apporter une représentation du logiciel plus adaptée à l'analyse et à la conception. L'accent est mis sur l'uniformité du modèle dans les différentes phases du développement. Les concepts sont ceux d'Eiffel : le polymorphisme, la liaison dynamique, l'héritage multiple, l'envoi de message, les structures de classe et les instances d'objets; et ceux de la conception avec la généricité, les assertions et invariants, l'indexation de classe, les routines et classes retardées. Les principes de conception, issus de l'expérience dans le développement des classes du langage, sont les suivants : écrire des classes virtuelles en utilisant des invariants de classe, spécifier les méthodes d'interface avec des pré-conditions et des post-conditions, prévoir les structures candidates au polymorphisme, utiliser la généricité, utiliser les classes de haut niveau pour assurer une meilleure flexibilité.

Modèles Deux modèles représentent les concepts : le modèle statique décrit l'aspect structurel, le modèle dynamique représente les envois de messages et les instances d'objets. Le modèle statique offre deux niveaux de description : la classe et le cluster. La classe décrit les services proposés et les propriétés internes. Une typologie des classes est proposée au travers de la notation graphique ci-dessous.

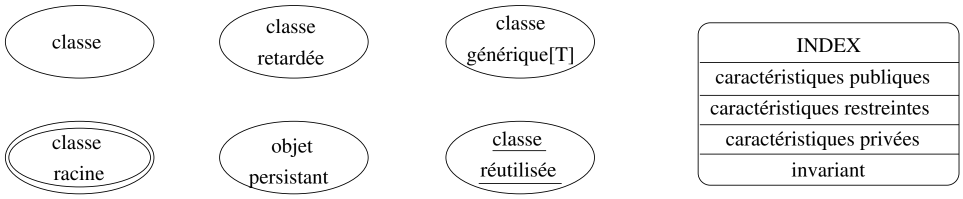


Figure 11 : Les types de classes de OOAD

Les relations entre classes sont de deux types : héritage ou client/serveur. Une cardinalité précise la relation. La relation d'utilisation est classée en cinq sous-relations, issues des cardinalités du modèle des objets et de la dualité association/agrégation : du client vers le serveur (`utilise, a besoin de, a, consiste en`) et du serveur vers le client (`fournit`). Une classification des relations d'héritage distingue la relation du descendant vers le parent (`is-a, behaves-like, implements, combines`) et la généralisation (`defers-to, factors-out`). La notation, appelée BON, permet de plus d'exprimer une relation de partage ou de répartition de plusieurs objets sur des processeurs différents. Les clusters permettent de regrouper des collections de classes répondant à un critère particulier : sous-système de fonctionnalités, classes réutilisées, structures de données partagées. Des règles contraignent les liens classe/cluster (héritage, clientèle, inclusion). Le modèle dynamique exprime les instances des classes et leur couplage. Un graphe de communication établit les connexions.

Démarche La méthode fournit un cadre de conception en huit étapes. Ce processus est un guide et non un cadre rigoureux. Il est dirigé par la simplicité, une transition en douceur vers la programmation, la prise en compte de la réutilisabilité, la facilité d'intégration dans un outil CASE. Les étapes proposées sont les suivantes :

1. trouver les classes du domaine de l'application.
2. définir les clusters et élaborer les contrats par des assertions, assurer les niveaux d'abstraction et isoler les parties critiques,

3. regrouper et supprimer des classes,
4. résoudre les questions liées au modèle à objets (Comment trouver les bonnes abstractions, les structures polymorphes? Comment isoler les informations d'implantation?) et nécessitant de l'expérience,
5. ajouter des classes "de second plan" pour complèter les classifications, découper les classes, assurer les responsabilités, servir les informations partagées,
6. typer les caractéristiques,
7. connecter les classes (pas de laissées-pour-compte),
8. itérer afin d'affiner le passage de l'analyse à la conception.

Nous n'avons volontairement montré qu'une partie de chacun des modèles statiques et dynamiques.

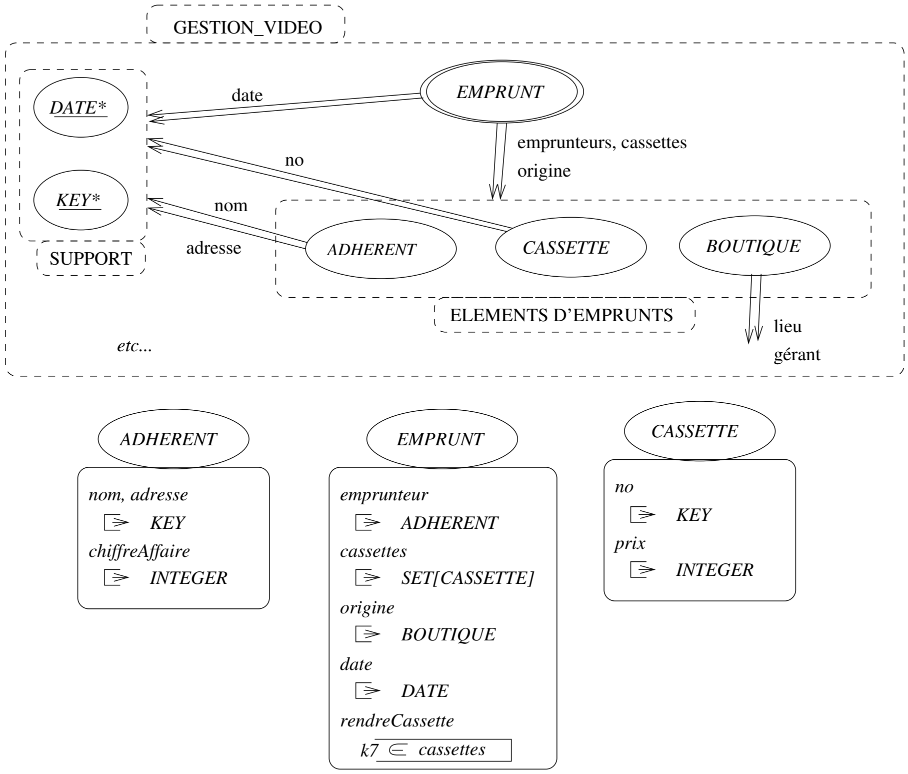


Figure 12 : Modèle statique OOAD de l'application vidéo

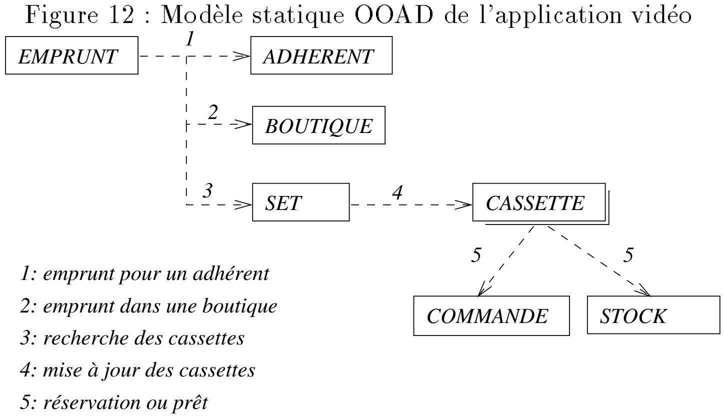


Figure 13 : Modèle dynamique OOAD de l'application vidéo

Dans le modèle statioque, les rectangles en pointillé désignent les clusters. Ces clusters contiennent des classes symbolisées par des ellipses. La description graphique des classes définit le comportement. La flèche sortant d'un carré désigne un argument en sortie. Une description plus précise est donnée par les diagrammes de classes.

Le modèle dynamique montre l'agencement des services des objets pour réaliser les traitements. Les lignes en pointillé désignent des communications entre objets.

[Mey89b] résume la pensée d'OOAD : utiliser les anciennes méthodes pour infiltrer la culture projet[13] et les idées neuves telles que l'abstraction, le contrat, la généralisation, qui mènent à la réutilisabilité.

2.3.6 Conclusion

Il apparaît clairement que les objets offrent une bonne alternative aux méthodes traditionnelles, inadaptées à la complexité des logiciels, que ce soit dans la conception, les langages, les interfaces, les SGBD ou même les systèmes d'exploitation. La dichotomie donnée/programme traditionnelle et la méthode cartésienne disparaissent au profit d'une approche plus systémique correspondant mieux à l'analyse des problèmes concrets. C'est une approche modulaire qui favorise la qualité du développement.

Un avantage prépondérant en développement du logiciel est qu'on "peut remettre en cause les spécifications ou les implantations sans avoir à tout refaire". L'objet améliore ainsi l'évolutivité du système. L'héritage est une caractéristique majeur et innovante du modèle à objets pour la définition incrémentale de composants logiciels. Le développement à objets produit des composants logiciels réutilisables. Plus ces composants logiciels sont généraux, plus ils sont réutilisables. Cependant, lors du développement d'un projet, les contraintes temporelles et budgétaires freinent cette généralisation. Avoir le même paradigme à différents stades du développement favorise l'intégration et la cohérence. Cependant, le niveau d'abstraction doit varier pour éviter les spécifications trop opérationnelles.

L'objet est une "révolution", mais aussi un effet de mode avec ses abus : il n'y a pas un mais plusieurs modèles à objets. La littérature objet foisonne d'ouvrages sur le développement à objets, de l'approche objet pure à l'"orientation" objet. Les premiers impliquent de nouvelles notations et processus tandis que les seconds semblent redéfinir les anciens concepts avec de nouveaux mots [MP92]. Le développement à objets est un domaine en effervescence tant pour l'intérêt scientifique que pour les intérêts économiques en jeu. Les travaux se situent tant sur les représentations que sur les processus, comme le montre [MP92].

Un besoin important de formalisation et de normalisation se fait sentir dans ces deux axes. La description des méthodes référencées, habilitées et reconnues à ce jour par l'OMG[14] a été synthétisée dans [Hut94b] et [Hut94a] où vingt et une méthodes sont décrites sommairement.

2.4 Synthèse

Les apports et les lacunes des méthodes formelles et des méthodes à objets montrent que ces approches sont complémentaires [LH93, CLLF93]. Les méthodes formelles ont besoin des méthodes à objets pour être appliquées à des systèmes complexes. En fournissant des mécanismes de structuration supplémentaires, le modèle à objets

- renforce la modularité des spécifications et permet de travailler séparément sur des parties indépendantes de la spécification; en particulier ces parties sont généralisées ou raffinées,

- permet la séparation des problèmes dans la mesure où la preuve devient modulaire, abstraite et réutilisable.

[13] La dualité donnée/traitement est bien présente dans le modèle.

[14] Object Management Group : organisme de normalisation de la technologie à objets.

- facilite la réutilisation de composants par héritage et par instanciation, notions novatrices en approche modulaire, et ce à tout niveau d'abstraction ,
- fournit un support au raffinement de spécifications; la vérification de l'héritage mène à la preuve du raffinement.
- facilite la maintenance par extensibilité ou substitution de composants, le polymorphisme est alors une forme d'abstraction;
- améliore la lisibilité des spécifications en se rapprochant des concepts du domaine de l'application et non l'application seule,
- permet une description plus cohérente de divers aspects du système, dont la représentation des données, la concurrence, l'expression des responsabilités dans le système,
- unifier les différents styles de spécification formelle modulaires dans un cadre unique (logique, fonctionnel, impératif, algébrique).

Les méthodes à objets ont besoin de formaliser les modèles et le processus de développement. La variété des modèles, leur richesse sémantique et leurs facilités graphiques permettent d'exprimer n'importe quel concept en restant très abstrait. Malheureusement le manque de formalisme est source d'incohérences et va à l'encontre de certains principes de la construction du logiciel (validation dès l'analyse, automatisation de la construction). Par exemple, dans les modèles statiques, l'héritage est difficilement contrôlable, le rôle central des associations dans des modèles statiques pose des problèmes de modélisation dans le modèle à objets en général et implique souvent des décisions prématurées dans une modélisation formelle [LH93]. Les modèles dynamiques posent aussi des problèmes de formalisation et surtout de preuve : les opérations sont souvent décrites en langage naturel, aucune politique précise n'est définie pour traiter les requêtes d'un objet (volatilité des événements). Les problèmes de synchronisation et de concurrence globale sont souvent implicites. Ainsi, les diagrammes fonctionnels de type flots de données déterminent plus un calcul particulier qu'une spécification du comportement du système. Les méthodes formelles sont utilisées pour :

- Définir un cadre sémantique pour le modèle à objets et notamment les relations entre classes, objets et sous-systèmes; et pour l'interface des objets, c'est-à-dire la définition précise des méthodes. Ce cadre est nécessaire à l'étude des propriétés telles que la cohérence l'équivalence de classes et sert de support à la recherche de composant dans une bibliothèque.
- Fournir des bases pour la validation et la vérification des applications à objets, le contrôle de propriétés de la spécification (sûreté du typage, conformité, substituabilité, correction entre une interface et une implantation).
- Induire une approche disciplinée du développement avec preuve mathématique du raffinement de composants.

Pour [Jon91], l'apport croisé le plus important est le développement incrémental. L'objet permet de l'organiser, les méthodes formelles de l'automatiser. Les bénéfices cumulés attendus sont : réutilisation rigoureuse, extensibilité et évolutivité en maintenance, partitionnement des activités, élargissement du champ d'application des techniques formelles, description de systèmes distribués et communicants [LH93].

Chapitre 3

Spécifications formelles à objets

"Les mathématiques ont des inventions, très subtiles et qui peuvent beaucoup servir, tant à contenter les curieux qu'à faciliter tous les arts et diminuer le travail des hommes."
Descartes.

Dans le chapitre précédent, nous avons décrit sommairement le développement du logiciel. Deux types de méthodes ont une influence significative sur la qualité du développement : les méthodes formelles et les méthodes à objets. Ces deux approches sont complémentaires. Dans ce chapitre, nous étudions la combinaison de ces deux techniques. Nous établissons également un état de l'art des méthodes formelles à objets en passant en revue quelques méthodes. Il est important de noter que ces méthodes sont récentes et manquent de validation.

3.1 Intégration

Comment joindre méthodes formelles et méthodes à objets ? Elles ont leur histoire, leurs spécificités et leurs contraintes propres mais aussi certains points communs tel l'influence des travaux sur les types abstraits de données. Plusieurs croisements sont possibles selon le degré de formalisme et la base de départ.

L'objet peut être perçu comme un mécanisme de structuration de la spécification ou comme un mécanisme d'implantation. Dans le premier cas, la difficulté est de concevoir un modèle à objets formel et abstrait, qui unifie plusieurs plans de description. Dans le second cas, la difficulté est de prouver que le modèle de conception vérifie le ou les modèles de spécification.

Pour définir formellement un modèle à objets, il faut choisir les concepts, donner une syntaxe et une sémantique à ces concepts[1]. Le modèle formel doit être suffisamment abstrait pour être utilisable en spécification. Le but est de cacher la complexité en construisant des composants logiciels avec des spécifications externes simples, permettant de gérer les programmes complexes. Deux interprétations abstraites des objets sont couramment données [Dah90]: un objet est une valeur d'un type de donnée, un objet est une trace d'un système de transition (structure modifiable). Les approches impératives posent des problèmes quand à la définition de l'héritage et surtout son contrôle statique.

3.1.1 Formalisation dans les méthodes à objets

Les spécifications formelles par modèle abstrait sont presque prédestinées à ce genre de formalisation. La formalisation peut être conjointe, partielle ou en séquence.

[1] Pour une description précise de la sémantique des langages, consulter [Kah92, Mey90a, Smi93].

Intégration conjointe Donner une sémantique formelle des modèles d'analyse à objets est a priori réalisable dans la mesure où les notations semi-formelles reposent souvent sur un modèle théorique (entité/associations, flots de données, automates). En pratique, ce sont les extensions de ces formalismes bien définis qui posent des problèmes (héritage ou agrégation d'objets statiques ou dynamiques, liens provisoires, communication entre objets, etc.). [HC91] est une tentative de formalisation de OMT, avec la logique des prédicats et des constructeurs de types. Le modèle à objets est raffiné en termes de record, séquence, ensemble, application et d'union avec identité explicite dans un modèle de structure des objets. Les opérations du sytème sont données par pré-post conditions sur la nouvelle structure. Les modèles dynamiques sont aussi exprimés en fonction des structures (gardes) et des pré-post conditions sur les opérations. Des traces sont ainsi calculées. Elles doivent être cohérentes avec les spécifications axiomatiques du modèle fonctionnel. L'approche est intéressante mais ne couvre pas tous les concepts de la méthode OMT (métaclasses, agrégation, émission d'événements, inclusion de comportement dynamique, etc.). De plus elle semble très opérationnelle, car inspirée de VDM.

Intégration partielle La difficulté ici est d'intégrer harmonieusement les parties formelles et informelles. Un niveau d'abstraction suffisant est conservé, tout en exhibant les propriétés importantes du système. Dans [DGI90], HOOD sert à la structuration de la spécification et Z à la spécification formelle de certaines parties, notamment les opérations.

Intégration transitionnelle Cette approche de type processus de formalisation est assez simple à mettre en œuvre et agréable à utiliser. En fait, ce schéma n'est applicable que si les principaux concepts existent dans les deux méthodes. Un bon exemple figure dans [Wil93]. L'auteur propose une méthode de traduction des modèles de la méthode Booch en Object-Z. Le modèle des données est assez facile à traduire en schéma Z, puisque qu'il s'agit dans les deux cas de variations de la théorie des ensemble. Toutefois, l'agrégation et l'héritage sont à contrôler avec précaution. Les opérations deviennent formalisables, ce qui n'était pas le cas avant. Le passage des automates aux historiques (logique temporelle) ne semble pas complet, car les gardes disparaissent. Le reste du modèle est un peu flou. Nous avons aussi utilisé cette approche dans [ABR95]. La difficulté majeure reste la traduction de concepts du modèle informel qui n'existent pas dans le modèle formel (car difficilement exprimables dans la théorie sous-jacente) ou l'inverse. Ils sont soit éliminés soit modélisés artificiellement.

3.1.2 Extensions à objets des méthodes formelles

Intégration conjointe Un langage de spécification formelle à objet est défini par extension d'un langage de spécification. Les preuves sont généralement décrites dans le langage support et parfois dans une sémantique algébrique (OOZE, Z++). L'intérêt évident de ce type de formalisation est la réutilisation des environnement de spécification et de preuve des méthodes formelles, qui sont en général difficiles à écrire. La difficulté majeure est le choix des concepts à objets et leur intégration cohérente dans les modèles théoriques sous-jacents. Beaucoup de méthodes formelles à objets sont de ce type. Object-Z [DD90], VDM++ [D9̈4], OOZE [AG91], Z++ [LH94] et MooZ [MC92] sont des extensions de Z ou VDM.

Intégration partielle Ici, le modèle à objets est la source d'inspiration de nouveux mécanismes de structuration des spécifications, sans définir explicitement de modèle à objets Par exemple, l'héritage est restreint au sous-typage en PLUSS [Bid89], OBJ3 [KKM87].

Intégration transitionnelle Les objets sont utilisés comme mécanisme d'implantation de types abstraits de données. Il y a une séparation nette entre spécification abstraite et conception. Cette approche est souvent choisie dans les méthodes issues des spécifications algébriques telles que OS-OP [Bre91], GSBL [CO88], COLD [FJ92]. Le sous-typage est défini comme mécanisme d'inférence et l'héritage comme un technique de raffinement. La difficulté est de définir une transition sans heurts entre les niveaux d'abstraction. Ce qui se fait par une sémantique algébrique commune.

3.1.3 Langage de programmation haut niveau

Il s'agit de définir un langage à objets haut niveau pour permettre les preuves et le raffinement. Les avantages sont évidemment l'exécutabilité des spécifications et leur sémantique formelle. L'écueil principal à éviter est la sur-spécification. Les langages Eiffel [Mey88], FOOPS [GM87b], POOL [Ame89] et le modèle des classes formelles [ACR94] illustrent cette approche.

Une autre voie est d'ajouter des possibilités de spécification et de vérification dans les langages de programmation à objets. Ceci permet une certification des composants logiciels. Par exemple, Fresco [Wil91] et SmallVDM [LH93], basés sur Smalltalk, ont des caractéristiques inspirées de VDM.

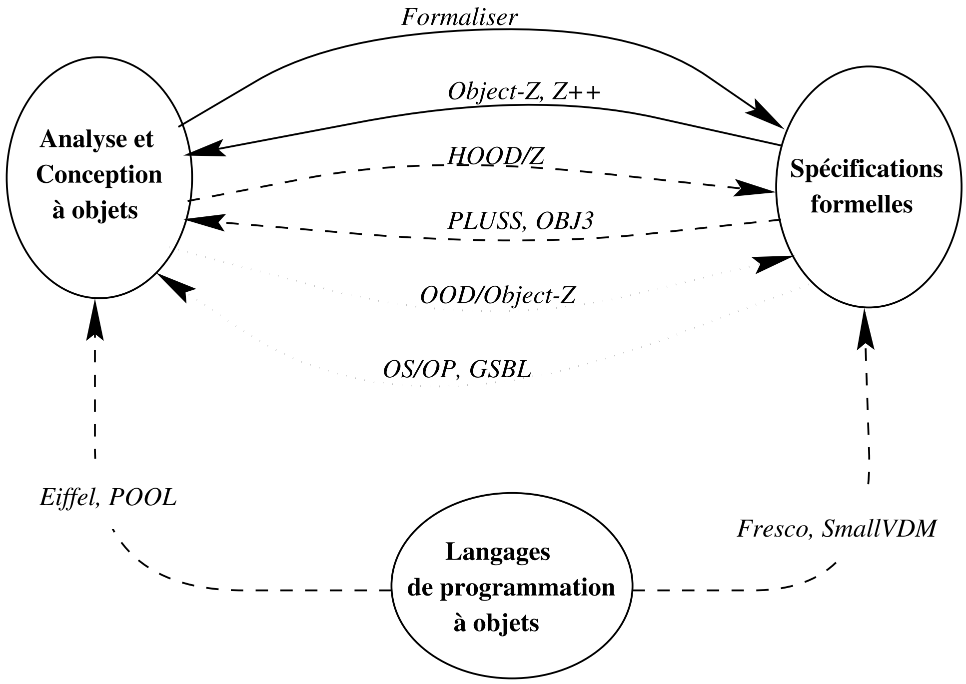


Figure 14 : Intégration de concepts formels et à objets.

3.2 Panorama

Nous passons en revue quelques langages de spécification ou de conception abstraite intégrant des modèles formels à objets.

3.2.1 ObjectZ

Object-Z est une extension du langage de spécification Z (voir section 2.2.5) aux concepts à objets [CLLF93, Duk90, DD90, Smi93, Cus91]. Il a été développé à l'université de Queensland en Australie, sous l'impulsion notamment de Roger Duke et Gordon Rose. C'est à la fois le précurseur dans le domaine et le plus utilisé actuellement.

Modèle à objets

L'objectif initial du langage est de structurer les spécifications Z. Pour cela, un nouveau schéma encapsule différents schémas Z dans une classe. Un schéma de classe permet de regrouper un ensemble cohérent de schémas d'état ou d'opération dans une même structure syntaxique. Les définitions et déclarations locales ont une syntaxe Z.

$$\begin{array}{|l}
\hline
\textit{Nom de Classe [paramètres génériques]} \\
\textit{liste de visibilité} \\
\textit{superclasses} \\
\textit{définitions de types} \\
\textit{définitions de constantes} \\
\textit{schéma d'état} \\
\textit{schéma d'état initial} \\
\textit{schéma d'opérations} \\
\textit{invariant d'historique} \\
\hline
\end{array}$$

Lorsque la liste de visibilité est omise, tous les attributs sont visibles. Dans l'exemple de la file d'attente, la classe `HospitalQueue` est partiellement définie par :

$$\begin{array}{|l}
\hline
HospitalQueue[T] \\
\textit{...déclarations...} \\
\hline
\end{array}$$

avec les déclarations suivantes[2] :

$$| \; MAX ::= 20$$

$$\begin{array}{|l}
\hline
patients : seq\, T \\
\hline
\#patients \leq MAX \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
INIT \\
patients = \langle\rangle \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
Join[T] \\
\Delta(patients) \\
patient? : T \\
\hline
patient? \notin patients \\
\#patients < MAX \\
patients' = item \frown \langle patient? \rangle \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
LeaveN[T] \\
\Delta(patients) \\
patient! : T \\
\hline
patients \neq \langle\rangle \\
patients = \langle patient! \rangle \frown item \\
\hline
\end{array}$$

La constante MAX est propre à la classe. Le schéma d'état comprend une variable appelée attributs et son état initial. Dans les schémas d'opérations, l'inclusion de schémas (Δ) devient

[2] Le format LaTeX-Z utilisé ne permet pas l'inclusion de schémas. Nous les décrirons donc séparément.

implicite. Seuls les attributs modifiés figurent, les autres ne changent pas. Lors de l'instanciation d'une classe générique (appelée application), Object-Z autorise le renommage des attributs ('/') et des paramètres d'opérations. Ainsi, une file d'attente d'un bureau de poste pourrait s'écrire

$$PostOfficeQueue == HospitalQueue[People][patients/suppliers,\ patient?/supplier?, patient!/supplier!]$$

Les déclarations héritées et les nouvelles déclarations sont mixées, y compris l'invariant. Les conflits de noms sont résolus par renommage global pour la classe ou local aux opérations. La surcharge (`redef`) et la suppression (`remove`) sont acceptés.

Une instance de la classe est une affectation de valeurs aux attributs, cohérente avec la définition de la classe. Une déclaration de type `c : C` définit une référence à un objet de classe C. Pour définir deux objets distincts `c,d : C` en Object-Z, il faut ajouter une contrainte $c \neq d$. Les objets peuvent avoir des constituants d'objets. Les références immuables aux objets sont déclarées par des constantes. Par exemple, nous définissons l'hôpital par

Hospital

...déclarations...

$q_{urgency}, q_{normal} : HospitalQueue[People]$

$q_{urgency} \neq q_{normal}$

INIT

$q_{urgency}.INIT \wedge q_{normal}.INIT$

$Admit \mathrel{\widehat=} q_{normal}.Join$

$Urgency \mathrel{\widehat=} q_{urgency}.Join$

$Cure \mathrel{\widehat=} CureUrgency \vee CureNormal \vee CureNobody$

CureUrgency

$\Delta(q_{urgency}, q_{normal})$
$p! : People$

$q_{urgency}.IsEmpty \bullet report! = false$

$q_{urgency}.Leave \bullet patient! = p!$

CureNormal

$\Delta(q_{urgency}, q_{normal})$
$p! : People$

$q_{urgency}.IsEmpty \bullet report! = true$

$q_{normal}.IsEmpty \bullet report! = false$

$q_{normal}.Leave \bullet patient! = p!$

CureNobody

$\Delta(q_{urgency}, q_{normal})$
$p! : People$

$q_{urgency}.IsEmpty \bullet report! = true$

$q_{normal}.IsEmpty \bullet report! = true$

L'instanciation est propagée dans les objets complexes. Par exemple, un hôpital contient un nombre quelconque de files.

$$\mid queues : \mathbb{F}\, HospitalQueue[People]$$

$$\begin{array}{|l} \textit{INIT} \\ \hline \forall q : queues \bullet queue.INIT \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{SelectNonFullQueue} \\ \hline qnf? : HospitalQueue[People] \\ \hline qnf? \in queues \\ qnf?.IsFull \bullet report! = false \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{SelectNonEmptyQueue} \\ \hline qne? : HospitalQueue[People] \\ \hline qne? \in queues \\ qne?.IsEmpty \bullet report! = false \\ \hline \end{array}$$

$$SelectTwoQueues \mathrel{\widehat=} SelectNonEmptyQueue \wedge SelectNonFullQueue \bullet qne? \neq qnf?$$

$$Admit \mathrel{\widehat=} SelectQueue \bullet q?.Join$$

$$Cure \mathrel{\widehat=} SelectQueue \bullet q?.Join$$

$$IsEmpty \mathrel{\widehat=} \forall q : queues \bullet q.IsEmpty \bullet report! = true$$

Caractéristiques propres

Object-Z ajoute des opérateurs de schéma à Z : le choix indéterministe $S[]T$, la conjonction $S \bullet T$ (accessibilité des déclarations) et l'opérateur parallèle $S \mid\mid T$, qui correspond en fait à une sorte de sérialisation car les paramètres de même nom sont unifiés. Le polymorphisme d'un objet peut être explicité par la notation $\downarrow Class$, qui indique que le type de l'objet considéré sera un descendant de $Class$.

$$Transfer \mathrel{\widehat=} SelectTwoQueues \bullet (qne?.Leave \mid\mid qnf?.Join)$$

Une interprétation comportementale est possible au travers de traces, appelées invariant historique. Un historique est une séquence d'événements que subit un objet. Ces événements correspondent à des appels d'opérations. L'invariant d'historique pourrait être donné en termes de prédicats Z, mais la logique temporelle est plus adaptée [DS92, SD92]. Les opérateurs logiques sont : `suivant` ($\bigcirc$) possible au rang suivant, `possible` ($\Diamond$) vrai dans le futur, `toujours` ($\Box$) vrai à tout moment. Par exemple, la classe des hôpitaux dont les files sont équitables est une sous-classe de `HospitalQueue[T]` telle que si la file d'attente de l'hôpital n'est pas vide alors elle se réduira dans le futur :

$$\begin{array}{|l} \textit{FairHospitalQueue}[T] \\ \hline HospitalQueue[T] \\ \hline \Box\,(patients \neq \langle\rangle \Rightarrow \Diamond((\bigcirc \# patients) < \# patients)) \\ \hline \end{array}$$

Aspects formels

La sémantique d'Object-Z est une extension de celle de Z (voir section 2.2.5). En Z, chaque expression est typée et chaque type détermine un ensemble support comme dans les algèbres. Object-Z rajoute le type classe. L'état est une fonction de l'espace des variables vers les ensembles de valeurs. *"This semantics preserves the Z notion of type as set"*[DD90]. L'historique est défini comme une séquence de noms d'opérations et de valeurs d'états, tel que l'ensemble des variables soit le même pour les pré- et post- conditions.

La sémantique d'un schéma est la donnée d'une signature du schéma et d'un ensemble de modèles satisfaisant les contraintes du schéma. Il n'y a pas de surcharge dans les noms de variables des schémas. De même, la sémantique d'une classe est la donnée d'une signature et d'un ensemble de modèles. La signature comprend les constantes et les attributs et méthodes, qui sont définis par des schémas. La logique étendue permet la vérification de l'historique.

Cusack distingue l'héritage incrémental de l'héritage de sous-type [Cus91]. Ce dernier peut être défini par un modèle basé sur la théorie des ensembles tandis que le premier correspond à un raffinement de schéma (ajout de schémas d'opérations, assouplissement de préconditions, renforcement de post-conditions, extension des variables avec renforcement de l'invariant et du schéma d'initialisation). Smith propose une sémantique abstraite en terme d'équivalence observationnelle [Smi93]. Il sépare l'aspect structurel et l'aspect temporel des classes dont les preuves, sont implicitement de nature différente.

Environnement

L'environnement du langage est en cours de développement. Peu d'informations sont disponibles à ce sujet.

Conclusion

Object-Z est une référence dans le monde des spécifications formelles à objets. Le langage possède la plupart des caractéristiques des langages à classes, et inclut en plus des aspects dynamiques. Il ne contient pas de constructions modulaires autres que la classe. Ceci conduit à une structuration avec contrôle centralisé des systèmes spécifiés. Le langage est agréable mais il est à la fois trop opérationnel et trop mathématique dans les preuves.

3.2.2 OOZE

OOZE (Object-Oriented Z Environment) est un langage à spectre large supportant la description des besoins, la spécification, les programmes interprétés et les programmes compilés. Il cumule le style algébrique et celui par modèle abstrait. Il a été développé à l'université d'Oxford par Antonio Alencar et Joseph Goguen [CLLF93, AG91, LH93].

Modèle à objets

Le modèle à objets d'OOZE est complet. Les classes encapsulent des schémas d'état, d'initialisation et d'opérations de Z.

classe < superclasses
constantes
schéma d'état
schéma d'état initial
méthodes

Les caractéristiques des superclasses sont ajoutés et les invariants renforcés. Les constantes sont des valeurs invariantes, communes aux instances de la classe. Les variables d'état sont données par un schéma State. Elles peuvent être privées. Le schéma d'initialisation est paramétrable. Les méthodes sont définies par des schémas Z classiques enrichis.

$$\begin{array}{|l}
\underline{\textit{State}} \\
q_{urgency}, q_{normal} : HospitalQueue[People] \\
\hline
q_{urgency} \neq q_{normal} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\underline{\textit{Init}} \\
q_{urgency} = HospitalQueue[People].Init \\
q_{normal} = HospitalQueue[People].Init \\
\hline
\end{array}$$

$$\begin{array}{|l}
\underline{\textit{Cure}} \\
p! : People \\
\hline
q_{urgency}.Leave \bullet patient! = p! \\
\underline{if} \neg q_{urgency}.IsEmpty \\
\hline
q_{normal}.Leave \bullet patient! = p! \\
\underline{if} q_{urgency}.IsEmpty \\
\quad \neg\ q_{normal}.IsEmpty \\
\hline
\end{array}$$

avec la fonction suivante dans la classe HospitalQueue[People] :

$$\begin{array}{|l}
isEmpty : HospitalQueue[People] \rightarrow Boolean \\
\hline
\forall q : HospitalQueue[People] \bullet isEmpty(q) = (length(q) = 0)
\end{array}$$

Comme dans Object-Z, la notation Δ indique la liste de variables modifiées. Les méthodes privées sont suffixées entre crochets par le mot-clé private. La surcharge et le polymorphisme sont acceptés, le contexte permet de savoir quelle est la méthode à appliquer.

La composition de schémas en Z permet de distinguer les cas normaux des cas d'erreur. En OOZE, les schémas des traitements exceptionnels sont préfixés par le mot-clé Error, les paramètres en entrées apparaissant dans les deux cas.

$$\begin{array}{|l}
\underline{\textit{Error}}\ \textit{Cure} \\
error! : Report \\
\hline
error! = NoPatient \\
\underline{if} q_{urgency}.IsEmpty \\
\quad q_{normal}.IsEmpty \\
\hline
\end{array}$$

L'envoi de message en OOZE est noté par la notation pointée objet.méthode(args). Le receveur est noté self. L'instanciation et la destruction se font par les méthodes de classe Init et Del. Comme en Smalltalk, chaque classe X définit une métaclasse $\overline{X}$ d'unique instance notée $\overline{X}$. Les objets ont un identifiant unique dans le système. Les instances d'une classe sont accessibles par la variable objs, de type séquence.

Caractéristiques propres

Des moyens supplémentaires de structuration sont donnés pour organiser les spécifications. Un **module** est un ensemble de classes fortement couplées. Un module avec une seule classe porte le nom de la classe. Ces modules sont organisés hiérarchiquement selon une relation d'importation. Les modules importés définissent des relations de visibilité entre classes. Les modules génériques sont paramétrés par des théories et instanciés par des vues.

$$\begin{array}{|l}\hline \textit{Hospital} \\ \textit{modules importés} \\ \textit{classes définies} \\ \hline\end{array}$$

Une **théorie** est un cas particulier de module, pour lequel les formules sont exprimées en logique des prédicats. La logique acceptée pour les modules est plus limitée pour assurer l'exécutabilité des spécifications et le prototypage. Les théories servent à la description des interfaces des modules et celle de leurs paramètres, tandis que les modules au sens strict du terme sont plutôt des implantations (assez abstraites). Par exemple, définissons une théorie patient ordonné, qui servira dans un module hôpital avec priorités. La variable `nopa` désigne l'absence de patient et la fonction de comparaison est définie par

$$\begin{array}{|l}
_ < _ : PATIENT \leftrightarrow PATIENT \\ \hline
\forall x, y, x : PATIENT \bullet \\
\quad \neg\,(x < x) \\
\quad (x < y) \wedge (y < z) \Rightarrow (x < z) \\
\quad (x < y) \vee (y < x)
\end{array}$$

$$\begin{array}{|l}\hline
\textit{Theory OrderedPatient} \\
[PRIORITY, PATIENT] \\
nopa : PATIENT \\
PRIORITY \subset \mathbb{R}^{+} \\
priority : PATIENT \rightarrow PRIORITY \\
--\ \textit{fonction de comparaison}\ -- \\
\forall x, y : PATIENT \bullet priority(x) > priority(y) \Rightarrow x < y \\ \hline
\end{array}$$

Une **vue** indique si un module satisfait une théorie. C'est une application des caractéristiques du module source vers celles du module cible. Elles sont utilisées pour instancier des modules génériques. Cette relation sert au raffinement de spécifications. Par exemple, des patients désignés par des réels positifs sont instanciés par :

$$HospitalWaitingRoom[Real\{PATIENT \mapsto \mathbb{R}, nopa \mapsto 0, < \mapsto <\}].$$

L'hôpital est constitué d'une salle d'attente avec un nombre limité de chaises et d'un certain nombre de cabines. Dans la modélisation suivante de la salle d'attente, $\twoheadrightarrow\!\!\!\!\!\!\shortmid\ $ désigne une fonction partielle finie et $\mathbb{F}_1$ désigne un ensemble fini non vide. La fonction `PriorPatient` désigne le prochain patient appelé (i.e. le plus prioritaire).

$$\begin{array}{|l}\hline
HospitalWaitingRoom[P :: OrderedPatient] \\
--\ \textit{déclarations ci} - \textit{dessous}\ -- \\ \hline
\end{array}$$

$$\begin{array}{|l}\hline
\textit{State} \\
seats : \mathbb{N} \twoheadrightarrow\!\!\!\!\!\!\shortmid\ PATIENT[hidden] \\
min : \mathbb{N} \\
max : \mathbb{N} \\ \hline
\end{array}$$

$$\begin{array}{|l}
PriorPatient : \mathbb{F}_1 PATIENT \rightarrow PATIENT \\ \hline
\forall FSP : \mathbb{F}_1 PATIENT; x, y : FSP \bullet \\
\quad PriorPatient(FSP) = x \leftrightarrow y < x
\end{array}$$

$\underline{Init}$

$min?, max? : \mathbb{N}$

$\forall s : min?..max? \bullet seats' s = nopa$

$min' = min?$

$max' = max?$

$\underline{if} min? < max?$

$\underline{Join}$

$\Delta seats$
$patient? : PATIENT$
$seat? : min..max$

$seats' = seats \oplus seat? \mapsto patient?$

$\underline{if} seats(seat?) = nopa$

Les salles d'attente sont maintenant affectées à des cabinets de consultation. Les patients sont dirigés vers la salle la moins occupée, ils choisissent leur place, tandis que pour la sortie, la salle d'attente est connue. Le module précédent est réutilisé par importation.

$\underline{Theory}\ HospitalMultipleWaitingRoom[P :: OrderedPatient]$

$- - déclarations\ ci - dessous - -$

$\underline{Importing}$

$HospitalWaitingRoom[P]$

$\underline{State}$

$cabins : \mathbb{N} \twoheadrightarrow HospitalWaitingRoom[hidden]$
$min : \mathbb{N}$
$max : \mathbb{N}$

$\underline{Init}$

$min?, max? : \mathbb{N}$
$min\_per\_cabin?, max\_per\_cabin? : \mathbb{N}$

$\forall c : min?..max? \bullet cabins'\ c = Init(min\_per\_cabin?, max\_per\_cabin?)$

$min' = min?$

$max' = max?$

$\underline{if} min? < max? \wedge min\_per\_cabin? < max\_per\_cabin?$

$\underline{Join}$

$\Delta cabins$
$patient? : PATIENT$
$seat? : \mathbb{N}$

$cabins' = cabins \oplus cab \mapsto (cabins\ cab).Join(patient?, seat?)$

$\underline{if} \exists\ cab : min..max \bullet$
$\quad (\forall c : min..max \bullet c \neq cab\ \wedge\ (size(cab) < size(c) \vee size(cab) = size(c)))$
$\quad (cabins\ cab).IsFree(seat?) \bullet report! = true$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{Leave}}} \\
\Delta \mathit{cabins} \\
\mathit{cabin}? : \mathit{min}..\mathit{max} \\
\mathit{patient}! : \mathit{PATIENT} \\
\hline
(\mathit{cabins}\ \mathit{cabin}?).\mathit{Leave} \\
\hline
\end{array}
$$

OOZE permet enfin de définir des types de données. Il s'agit de définitions algébriques classiques définies par des modules préfixés par `Data`. Ils peuvent être paramétriques et importer d'autres types de données.

Aspects formels

OOZE ressemble aux spécifications par modèles, mais possède une sémantique algébrique, et plus précisément le langage a actuellement une sémantique équationnelle, celle du langage de spécification algébrique OBJ3 [KKM87].

Les objets et les valeurs de types de données sont distingués, même s'ils partagent certaines caractéristiques de l'héritage (sous-type, sous-classe). Les deux hiérarchies possèdent leur propre ordre partiel. Les types de données ont une sémantique dénotationnelle en terme d'algèbre ordonnée et une sémantique opérationnelle en terme de système de réécriture [JL86]. Ils sont implantables en OBJ3 [KKM87]. Les propriétés et les preuves habituelles des systèmes algébriques sont applicables. La sémantique des classes est aussi donnée par des algèbres, mais seul le comportement observable détermine les équivalences. De telles algèbres sont des machines abstraites dont les états sont les éléments des sortes cachées. Chaque objet est une copie d'une machine avec son état propre, initialisée avec le schéma correspondant à la classe, et dont les méthodes modifient l'état tandis que les attributs observent l'état. Quand toutes les sortes (i.e. nom des types en spécifications algébriques) sont visibles, les concepts de machine abstraite et de type abstrait de données sont identiques.

Une théorie est une algèbre dont les modèles satisfont un ensemble d'axiomes (soit un modèle initial soit une équivalence observationnelle). Les vues sont des morphismes de théories. Les types construits tels que les tableaux, les fonctions, les opérations ou les structures sont utiles pour séparer et classer les entités. Ils facilitent la compréhension des spécifications ou du code en éliminant les expressions inconnues. OOZE a un contrôle de types fort mais flexible.

Environnement

L'environnement est supporté par celui d'OBJ3 [KKM87]. Il comprend la vérification syntaxiques et de type, des bases de données de modules, du prototypage et des facilités de preuve de théorèmes. L'environnement inclut un interpréteur pour exécuter les spécifications. Dans ce cas, les axiomes doivent être conditionnels positifs.

Conclusion

OOZE est un langage large spectre dans le cycle de vie mais aussi dans les concepts utilisés : le langage est très riche. Il présente l'originalité d'avoir des modules et des métaclasses. Cette richesse sémantique influe sur les structures de définition (classe, type de donnée, module, théorie). Par contre, aucun aspect dynamique n'est pris en compte, même si l'appel concurrent des méthodes d'un objet est possible par l'opérateur $||$. Il profite aussi de la richesse de l'environnement algébrique support (théorie, langages, outils). Il semble une bonne alternative industrielle aux standards Z et VDM.

3.2.3 VDM++

VDM++ est une extension du langage VDM aux spécifications modulaires par classes et au parallélisme [D$\ddot{9}$4]. Ce langage a été développé par Eugène Dürr à l'université d'Utrecht et Cap

Gemini dans le cadre d'un projet Afrodite du programme Esprit III. Il est largement inspiré de Smalltalk.

Modèle à objets

Une spécification VDM++ est la données de types, d'un ensemble de classes et d'un espace de travail optionnel. Les types standard VDM sont utilisables, les ensembles (`Set of T` pour `T-Set`), les séquences (`Sequence of T` pour T^* ou T^+), les applications (`Map D to R` pour $D \xrightarrow{m} R$ ou $D \xleftrightarrow{m} R$). Les types correspondant aux classes sont préfixés par '`@`'. La généricité n'est pas autorisée en VDM++, sauf pour des constructeurs classiques de VDM tels que la séquence ou les ensembles.

Bien que dans VDM définisse une notion de module [Jon93], la seule structure possible en VDM++ est la classe. Les modules VDM, par contre, sont utilisés pour décrire la sémantique des classes. Sommairement, la classe définit l'état des objets par un ensemble de variables d'instance, le comportement par un ensemble de méthodes et le comportement dynamique par des traces.

$$
\begin{array}{l}
DescriptionDeClasse ::= CLASS\ class\_id\ is\ subclass\ of\ superclass\_id;\\
\quad [variable\ d'instance]\\
\quad [liste\ des\ méthodes]\\
\quad [héritage\ contrôlé]\\
\quad [comportement\ dynamique]\\
\quad [contrôle\ de\ synchronisation]\\
\quad [définitions\ et\ raisonnement\ auxilliaire]\\
\quad END\ class\_id;
\end{array}
$$

Modélisons une gestion dynamique des salles d'attente (`queues`) et des salles de soin (`rooms`) de l'hôpital. Une salle de soin est libre, occupée ou fermée, elle est définie par un ensemble énuméré. Une file est une séquence de patient, exprimée en VDM par *HospitalQueue* = `seq of` *Patient* ou par une définition similiaire à celle de la section 3.2.1.

$$
\begin{array}{l}
CHAR = \{a, .., Z\}\\
REPORT = \{ok, error\}\\
Service = \{surgery, neurology, cardiology, paediatrics, urgency\}\\
Room = \{free, occupied, closed\}\\
Patient :: name : CHAR^+\\
\quad surname : CHAR^+\\
\quad address : CHAR^+\\
\quad age : \mathbb{N}\\
HospitalQueue = \ldots
\end{array}
$$

La structure des objets est définie par un ensemble de variables d'instances typées (type VDM ou classe), un ensemble de contraintes nommées portant sur les variables d'instance et par un ensemble de clauses d'initialisation (une par variable). Les valeurs initiales sont données lors de l'instanciation par l'unique méthode de classe **new** : `myHospital = Hospital!new`. Les objets sont référencés dans l'environnement par une application *pointeur* $\mapsto$ *objet*. Dans notre exemple, à chaque salle de soin ouverte correspond au moins une salle d'attente. Les autres salles d'attente sont inoccupées. Il existe des salles d'attente et des salles de soin libres.

Class Hospital
  Instance Variables
    queues: `Set of` @HospitalQueue;
    rooms: `Sequence of` Room;
    speciality: `Map` $\mathbb{N}$ `to` Service;

queue-of-room: `Map` @HospitalQueue `to` $I\!N$;
Invar- queues1: `card` queues + `card` `dom` queue-of-room ≥ 1;
Invar- queues2: $\forall$ q $\in$ queues $\bullet$ q!is-empty;
Invar- rooms1: `len` rooms ≥ 1;
Invar- rooms2: $\exists\ r \in rooms \bullet r \neq closed$;
Invar- queues1: `dom` queue-of-room $\cap$ queues = {};
Invar- queue-of-room1: `rng` queue-of-room $\subseteq$ **inds** rooms;
Invar- queue-of-room2: $\forall i \in$ `rng` queue-of-room $\bullet rooms(i) \neq closed$;
Invar- spec1: `dom` speciality = `rng` queue-of-room;
init- queues: {};
init- rooms: [];
init- speciality: {};
init- queue-of-room: {};
End Hospital

Les méthodes sont définies par une pré-condition et un corps. Les fonctions sont suffixées par le mot-clé **value** suivi d'une variable typée pour le résultat. Les clauses **rd** (read-only) et **wr** (writable) marquent les variables modifiées. Le surlignage par le symbole $\leftharpoondown$ signifie qu'il s'agit de l'ancienne valeur de la variable. Ce signe est remplacé ici par $\leftarrow$ pour des commodités LaTeX. Des déclarations de variables privées sont possibles par la clause **internal**. Des traitements d'erreur sont exécutés lorsque la précondition est fausse. Enfin le corps d'une méthode est soit différé par les mots-clé **is not yet specified** ou **is subclass responsability** soit défini par une suite d'instruction ou une post-condition. La forme générale de l'envoi de message est `expression!nom_méthode(param*)`, par exemple : `myHospital!join(cardiology,patient)`.

Class Hospital
Instance Variables
Methodlist
setHospital (nq : $I\!N$, nor : $I\!N$, is: `Sequence of` Service)
/* number of queues, number of rooms, initial service */
ext wr queue-of-room, queues, rooms, speciality;
pre `len` nq $\geq$ nor $\wedge$ nor $\geq$ `len` is
post `len` rooms = nor $\wedge$
($\forall i \in$ `inds` is $\bullet$ rooms(i) = free $\wedge$
speciality = $\overleftarrow{speciality} \cup \{i \mapsto is(i)\}$ $\wedge$
queue-of-room = $\overleftarrow{queue\text{–}of\text{–}room} \cup \{HospitalQueue!new \mapsto i\}$) $\wedge$
`card` queues = nq - `len` is $\wedge$
$\forall q \in$ queues $\bullet$ q = HospitalQueue!new;
exists-free-queue-for-room (r : $I\!N$) **Value** `B`
pre r $\in$ `inds` rooms
post $\exists\ q \in$ @HospitalQueue $\bullet$ queue-of-room(q) = r $\wedge$
$\neg\ q$!is-full;
free-queue-of-room (r : $I\!N$) **Value** queue : @HospitalQueue
pre exists-free-queue-for-room (r)
post $\exists\ q \in$ @HospitalQueue $\bullet$ queue-of-room(q) = r $\wedge$
$\neg\ q$!is-full $\wedge$
queue = q;
current-patients () **Value** sp : **Set of** Patient
/* sp : set of patient */
post $\bigcup_{q \in dom queue-of-room} q$;
join (s : Service, p : Patient) **Value** rep : REPORT
ext wr queue-of-room, queues
rd rooms, speciality;
internal roomNumber : $I\!N$;

pre s $\in$ **rng** speciality $\wedge$
($\exists\, r \in I\!N \bullet$ speciality(r) = s $\wedge$
exists-free-queue-for-room (r) $\wedge$ roomNumber = r $\vee$
queues $\neq \{\}$
post if exists-free-queue-for-room (roomNumber) **then**
queue = free-queue-of-room (roomNumber) $\wedge$
queue!join(p)
else $\exists\, q \in$ queues $\bullet$ queue = q $\wedge$
queues = $\overleftarrow{queues} - \{queue\}$ $\wedge$
queue-of-room = $\overleftarrow{queue{-}of{-}room} \cup \{roomNumber \mapsto queue\}$ $\wedge$
queue!join(p) $\wedge$
rep = ok;
errs rep = error;

etc.
End Hospital

VDM++ distingue l'héritage simple de structure et l'héritage multiple du comportement. La structure d'un objet définit entièrement son type. Une sous-classe est obtenue par extension de type (ajout de variables d'instance), avec compatibilité ascendante (**is subsclass of**). Ce n'est pas un sous-type au sens habituel du terme, mais une sous-classe. L'introduction de deux prédicats **is_base_type_of(T)** (vrai si l'expression évaluée est un objet de classe dérivée de T) et **is_type_of(T)** (vrai si l'expression évaluée est de classe T ou une de ses sous-classes) permettent de vérifier les types dans les pre- et post- conditions. L'héritage des méthodes reste un problème dans un environnement fortement typé, à cause des exceptions notamment. La plupart des langages contrôlent cet héritage en introduisant des clauses de renommage et de redéfinition. En VDM++, l'héritage de comportement est contrôlé en indiquant quelles méthodes sont héritées de quelle classe, par la clause **inherit** (**from** class methods)$^+$ et ses variantes. Il n'y a ainsi pas de conflits.

Caractéristiques propres

VDM++ définit un espace de travail, appelé "*workspace metaphor*", tel que la spécification du système est la description de l'ensemble des classes et celle de l'espace de travail. L'espace de travail correspond aux éléments dynamiques du système, comme dans le langage Smalltalk. Ce concept permet une conception et implantation dans un style interactif et itératif. Pour unifier les concepts du langage, l'espace de travail est une classe.

PID
HID
Class Workspace
Instance Variables
patients: Map PID to @Patient;
clinics: Map HID to @Hospital;
Invar- patients: **card dom** patients ≥ 1
Invar- clinics1: **card dom** clinics ≥ 1
Methodlist
addPatient (p: @Patient) **Value** pid: PID
ext wr patients;
pre pid $\notin$ **dom** patients
post patients = $\overleftarrow{patients} \cup \{pid \mapsto p\}$;
addHospital (h: @Hospital) **Value** hid: HID
ext wr clinics;
pre hid $\notin$ **dom** clinics
post clinics = $\overleftarrow{clinics} \cup \{hid \mapsto h\}$;

admit (p: PID, h: HID, s: Service) **Value** rep: REPORT
/* patient hospital service */
pre patients(pid) $\notin$ clinics(hid)!current-patients
post rep = clinics(hid)!join(p,s);
errs rep = error; ...
End Workspace

La seconde originalité de VDM++ est le traitement de la concurrence et du temps-réel. Ce dernier est traité par des fonctions de manipulation de temps continu et discret (`now, duration`...). Ainsi la durée des méthodes peut être contrainte. Pour la concurrence, les auteurs proposent une panoplie de notations, regroupées dans deux parties : le comportement dynamique pour les objets actifs et le contrôle de la synchronisation entre les méthodes.

Le comportement dynamique est exprimé par des **traits**[3] de contrôle déclaratifs ou procéduraux. Un trait déclaratif est soit un invariant soit une obligation périodique, permettant d'inclure des contraintes temporelles. Un trait procédural est une suite de commandes gardées (routine `Life` en Eiffel// [Car91]). Ces traits de contrôle ne sont pas hérités.

La synchronisation est exprimée ici aussi soit implicitement par des permissions soit explicitement par des traces. Les permissions expriment des conditions sur l'historique des invocations de méthodes, sur l'état de la file d'attente ou celui de l'objet. Les expressions de synchronisation utilisent ainsi des opérateurs de comptage ($\#act$ nombre d'appel de méthode, $\#fin$ nombre de terminaisons de méthodes, $\#req$ nombre de requêtes d'une méthode). Par exemple, soit une permission sur l'opération `Leave` par le mot-clé **per** :

$$\textbf{per}\ \text{Leave} \Rightarrow \#fin(\text{Join}) > \#fin(\text{Leave}).$$

Une trace est une expression régulière constituée de noms d'opérations, d'opérateurs de séquencement et d'expressions de synchronisation. La trace principale est un entrelacement arbitraire (`Weave`) ou séquenciel (`;`) de sous-traces. Une correspondance avec les machines à états finis est calculable.

Enfin, une dernière partie intitulée `raisonnement auxilliaire` sert aux différentes assertions et résultats liés à une classe. Ce sont typiquement des prédicats VDM.

Aspects formels

Le langage est conçu comme un préprocesseur de VDM. Sa sémantique est donc basée sur VDM. La sémantique des constructions parallèles et temps-réel est par contre incomplète. Les objets du système sont une application des identificateurs vers les objets. Les classes sont des modules VDM. Ces modules sont équivalents à ceux de Modula, et définissent des clauses d'importation d'autres modules et d'interface. Le système prend en compte deux sortes de types : les types VDM et les classes. Une indirection est utilisée pour les valeurs d'objets et leur comportement. La séparation héritage de représentation/héritage de comportement facilite l'assimilation des types objets en terme de structures (record).

Un des points essentiels de la méthode VDM est la théorie de la réification[4] et décrite dans [Jon93]. Il s'agit de donner des représentations de plus en plus concrètes des spécifications par reifications des structures de données et décomposition des opérations sur ces structures de données. Il s'agit là, véritablement d'une étape de conception. L'héritage est un bon mécanisme pour implanter le raffinement de composants.

Environnement

L'environnement est en cours de développement. Le système de raisonnement utilise une extension de Mural [JJLM91] l'outil de preuve de VDM. Il repose en partie sur les outils définis dans VDM (éditeurs, proveurs, raffinement). La partie principale est l'écriture d'un outil de traduction vers VDM. D'autres outils de vérification de la concurrence sont en cours d'écriture.

[3] Traduction du mot *thread*.

[4] Ce terme a été préféré au terme raffinement, trop large, pour indiquer la conversion d'une chose abstraite en chose concrète.

Conclusion

VDM++ est issu d'une collaboration dans un projet Esprit. L'objectif est donc, comme en VDM, de fournir un environnement de développement conséquent pour des applications industrielles. VDM donne un support théorique au langage et permet aussi la réutilisation de l'environnement de preuve et de raffinement. Les principales caractéristiques innovantes sont la différence entre héritage de structure et héritage de comportement, la notion d'espace de travail, la concurrence et les traces. L'effort porte sur la modularisation des spécifications et leur raffinement. Toutefois ni la description des riches mécanismes, ni la sémantique ne sont clairement définis, de même que leur intéraction avec les aspects modèle, tels que l'héritage ou l'agrégation.

3.2.4 COLD

COLD (Common Object-Oriented Language for Design) [FJ92] est un langage qui permet d'écrire des spécifications algébriques et des spécifications par modèle abstrait et de les implanter par raffinements successifs. Il a été développé dans les laboratoires de recherche de Philips à Eindhoven dans le cadre d'un projet Meteor du programme Esprit II. Le langage que nous voyons ici est COLD-K, le noyau extensible du langage.

Modèle à objets

Les spécifications peuvent être structurées par des modules (appelés classes). Une classe est un ensemble d'algèbres qui satisfont la spécification d'un module (classe de modèles). Chaque algèbre contient des valeurs : les objets. Le langage possède une bibliothèque de spécifications de base (booléens, entiers, caractères, tuples, ensembles, séquences, map). Il n'y a pas d'envoi de message. Les communications se font par appel d'opérations (prédicats, fonctions, et procédures). Les invariants sont décrits par des axiomes ordinaires ou des prédicats nommés. L'héritage ne fait pas partie des mécanismes de structuration du langage. Les classes sont paramétrables par des types libres (`FREE`), variables (`VAR`), ou dépendants d'autres types (`DEP`). Il n'y a pas de métaclasses, mais les auteurs définissent une classe importable `Inst` symbolisant les fonctions de manipulation d'instances et la procédure de création `create`.

Style algébrique

Le langage est défini sur les algèbres partielles, le caractère "`!`" correspond au prédicat de définition. La classe est définie par les sortes, des fonctions et des prédicats. La classe ci-dessous est paramétrée par le type `Patient`. La sémantique des fonctions et des prédicats est donnée par des axiomes sous forme de clauses de Horn t_1 `AND ... AND` t_m `=>` $r(x_1, ..., x_n)$ ou bien par des définions inductives introduites par le mot-clé `IND`. Le prédicat (privé) `is_gen` indique que tous les termes clos sont finiment engendrés.

```
CLASS
% description de la classe HospitalQueue
     SORT Nat1 % supposé non nul
     SORT Patient FREE
     SORT HospitalQueue
     FUNC open : Nat1 -> HospitalQueue
     FUNC join : HospitalQueue # Patient -> HospitalQueue
     FUNC leave : HospitalQueue -> HospitalQueue
     PRED is_in : HospitalQueue # Patient,
     ...
AXIOM
     FORALL n:Nat1 (
{HQ1}    open(n)!;
{HQ2}    NOT leave(open(n))!;
```

```
PRED is_gen : HospitalQueue
IND  FORALL n:Nat1 (is_gen(open(n) );
     FORALL p:Patient, hq:HospitalQueue
     ( is_gen(hq) AND NOT is_full(hq) => is_gen(join(hq,p)) )
AXIOM
     FORALL n:Nat1, p,p':Patient, hq:HospitalQueue (
{HQ5}    is_gen(hq);
{HQ8}    leave(join(open(n),p)) = open(n);
{HQ9}    leave(join(join(hq,p),p')) = join(leave(join(hq,p)),p');
{HQ13}   NOT is_in(open(n),p);
{HQ14}   is_in(join(hq,p),p') <=> p =p' OR is_in(hq,p') )
END
```

Il est possible de raffiner les spécifications en donnant des versions plus algorithmiques de la spécification en utilisant des constructeurs d'expressions (garde, alternative, composition, application de fonction, sélection) ou d'autres assertions (déclaration de variables locales, affectation, bloc d'instructions). Par exemple,

```
     PRED is_in : HospitalQueue # Patient
     PAR hq : HospitalQueue, p:Patient
     DEF EXISTS hq' : HospitalQueue, p':Patient (
         (hq = join(hq',p') AND ( p = p' OR is_in(hq',p) ) )
% raffiné en
     PRED is_in : HospitalQueue # Patient
     PAR hq : HospitalQueue, p:Patient
     DEF p = first(hq) OR is_in(leave(hq),p) ) )
```

La description de la page 63 est dite plate. Les spécifications plus complexes sont structurées par des clauses d'importation, exportation et renommage. Ces modules sont appelés schémas. Le renommage permet un semblant de sous-typage (même si ce n'est pas présenté comme cela) et de généricité (abstraction par `LAMBDA`, actualisation par `APPLY`). Les spécifications sont nommées par `LET nom : = <spec>`. Modélisons la file des patients par une liste. Soient les modules `NAT1` des entiers positifs et `LIST` des listes génériques bornées. Le terme `open(n):HospitalQueue` indique une coercition du type du terme `open(n)`.

```
LET LISTPATIENT:= RENAME
             SORT Item to Patient,
             SORT BoundedList to ListPatient,
         IN LIST;
LET QUEUE :=
EXPORT
     SORT Nat1,
     SORT Patient,
     SORT HospitalQueue,
     FUNC open : Nat1 -> HospitalQueue,
     FUNC join : HospitalQueue # Patient -> HospitalQueue,
     FUNC leave : HospitalQueue -> HospitalQueue,
     PRED is_in : HospitalQueue # Patient,
FROM
IMPORT NAT1 INTO
IMPORT LISTPATIENT INTO
CLASS
     SORT HospitalQueue

FUNC queue : ListPatient -> HospitalQueue
IND  queue(emptyList(n)) = open(n);
     FORALL p:Patient, l:ListPatient
```

```
     ( queue(cons(p,l)) = join(queue(l),p) )

     FUNC open: Nat1 -> HospitalQueue
     FUNC join: HospitalQueue # Patient -> HospitalQueue
     FUNC leave: HospitalQueue -> HospitalQueue
     PRED is_in: HospitalQueue # Patient

AXIOM
{LHQ1}   FORALL hq:HospitalQueue EXISTS l:ListPatient ( hq = queue(l) );
{LHQ2}   FORALL n:Nat1, p:Patient, k,l:ListPatient
         ( queue(k) = queue(l) <=> equal(k,l)
         ; open(n) = queue(emptyList(n))
         ; join(queue(l),p) = queue(join(l,p))
         ; leave(queue(l)) = queue(leave(l))
         ; is_in(queue(l),p) <=> is_in(l,p)
         ; ... )

% des opérations sont rajoutées aux listes (extensions uniquement)
     FUNC open: Nat1 -> ListPatient
     IND open(n) = emptyList(n)

     FUNC join: ListPatient # Patient -> ListPatient
     IND FORALL p:Patient, l:ListPatient
         ( join(emptyList(n),p) = cons(p,l) )

     FUNC leave: ListPatient -> ListPatient
     IND FORALL n:Nat1, p:Patient, l:ListPatient
         ( leave(emptyList(n)) = emptyList(n)
         leave(cons(l,p)) = leave(l) )
END
```

Style impératif

Le style algébrique permet de définir des types de données à un niveau d'abstraction élevé. Mais il ne convient pas au systèmes complexes, naturellement modélisés en termes d'états. Une autre raison invoquée est l'utilisation toujours actuelle de la programmation impérative. Le langage COLD est donc étendu tout en conservant une sémantique algébrique unique. Certains prédicats et fonctions deviennent variables (`VAR`). Ils varient en fonction des états. Un état correspond à une algèbre d'une classe. Un changement d'état est réalisé par une procédure (`PROC`). Le mot clé `MOD` indique quel prédicat ou fonction variable est perturbée par cette procédure et `USE` donne les droits d'utilisation.

Une classe est maintenant définie par un triplet (`State`, s_0, τ_p), où `State` est une collection d'états munie d'une application des états vers les Σ-algèbres correspondantes, s_0 un élément de `State`, τ_p une collection de transitions, une par procédure p. L'état antérieur d'une opération variable est noté par `PREV`. Une définition axiomatique *pre* $\{proc\}$ *post* est donné. L'axiome `INIT` correspond à l'état initial.

```
LET PATIENT:=
EXPORT
     SORT String,
     SORT Patient,
     FUNC register: String # String -> Patient
     FUNC name: Patient -> String
     FUNC surname: Patient -> String
     FUNC rename: Patient # String # String -> Patient
     PRED equal: Patient # Patient
FROM
     CLASS
```

```
        ...
    END;
LET ROOM:= LAMBDA X : CLASS
                SORT Contents
             END
    OF
    EXPORT
        SORT Room,
        PROC free : Room ->,
        PROC occupied: Room # Contents ->,
        PRED is_free : Room,
        FUNC who : Room -> Contents,
    FROM
    IMPORT X INTO
        CLASS
        PRED is_free : Room VAR
        FUNC who : Room -> Contents VAR

        AXIOM INIT => is_free = TRUE
        AXIOM is_free!

        PROC free : Room ->
        MOD is_free

        PRED pre_free :
        DEF NOT is_free

        PRED post_free :
        DEF is_free

        AXIOM pre_free => [ free ] post_free

        PROC occupied: Room # Contents ->
        MOD is_free,who

        AXIOM {ROOM1} FORALL r:Room,c:Contents
            ( NOT is_free(r) => [ occupied(r,c) ] ( NOT is_free(r) AND who(r) = v )

    END;
LET CURATION_ROOM := APPLY
        RENAME
            SORT Contents TO Patient
        IN ROOM
    TO PATIENT
```

Les descriptions sont ensuite raffinées selon des structures de contrôle plus algorithmiques. Ce n'est pas un modèle à effet de bord : il y a possibilité de définir des variables intermédiaires mais les objets n'ont pas à proprement parler de sructure. Les variables sont vues comme des fonctions sans argument. Un pointeur est un objet et la désignation est une fonction unaire. Un tableau à n dimensions est une fonction n-aire.

Caractéristiques propres

L'interprétation des procédures en morphisme d'algèbres constitue une originialité du modèle. Pour chaque style de spécification, un guide est donné dans la construction, la vérification (validité, correction, cohérence, catégoricité (au moins un modèle vérifie la spécification)) puis l'implantation. Par exemple, la construction des spécifications algébriques se fait en 6 étapes :

1 - donner la signature et l'interprétation associée (au moins une) aux opérations,

2 - choisir un ensemble minimal de fonctions qui génèrent toutes les valeurs de la sorte,
3 - formuler les axiomes spécifiant les domaines de définition et l'égalité sur les sortes,
4 - donner les définitions inductives sur les fonctions génératrices,
5 - donner les axiomes exprimant que la sorte définie est minimale (prédicat `is_gen`),
6 - définir les autres opérations par des inductions et des axiomes.

Aspects formels

Le langage est basé sur la logique mathématique et la programmation. Il est inspiré de plusieurs techniques : les types abstraits de données, les fonctions d'abstraction, les assertions (invariants, pre- et postconditions), la programmation modulaire.

Le langage de spécification algébrique est très souple. L'axiomatique acceptée comprend les axiomes équationnels ou conditionnels, les axiomes du premier ordre avec définitions inductives. Les algèbres support ont des sortes, des prédicats et des fonctions partielles. La sémantique est à classe de modèles sans restrictions implicites sur les algèbres. Ces choix garantissent un pouvoir d'expression élevé et de la flexibilité. Un type est une séquence de sortes. Les opérations partielles évitent le traitement des termes d'erreurs. Par contre la fonction d'interprétation est plus complexe à définir.

Les spécifications par modèle abstrait complètent les spécifications algébriques. Les états sont assimilés à des algèbres, qui deviennent ainsi l'unique cadre sémantique. Les états équivalents sont identifiés par une relation d'équivalence appelée **bisimulation**. Elle permet de regrouper les états ayant même algèbres et même transitions pour donner une forme canonique de la classe.

Environnement

Le langage possède plusieurs versions. Celle présentée ici correspond au noyau et semble lourde à manipuler. Il existe des versions plus conviviales. Des outils sont développés ou à l'étude : éditeurs texte et graphiques (POLAR), générateurs de masques (classes à compléter), vérification de type, sous-ensemble exécutable du langage (PROTOCOLD), etc.

Conclusion

Le langage COLD n'est pas un langage à objets car il n'y a pas d'héritage. Il correspond plutôt au style Ada ou CLU. L'effort porte principalement sur les constructions modulaires. Il n'y a pas de différence entre module et classe. Les définitions algébriques et pseudo-impératives des types de données sont décrites dans un cadre unique. Les dernières sont un peu lourdes à manipuler. Un inconvénient du langage est le besoin d'être systématiquement guidé sous peine de faire de la programmation impérative habituelle. Le langage a été validé par un certain nombre d'applications industrielles.

3.2.5 OSDL

OSDL a été développé dans le projet Mjølner [CLLF93]. Une des parties de ce projet consiste à ajouter une orientation objet au langage SDL. SDL (Specification and Description Language) est une norme définie par le CCITT (Comité Consultatif International Télégraphique et Téléphonique) pour les descriptions et spécifications non ambiguës de comportement de systèmes de télécommunications. Les spécifications sont des représentations abstraites des besoins (boîte noire) tandis que les descriptions caractérisent la structure des systèmes implantés. La norme actuelle date de 1988. Une bonne description du langage est donnée dans [BHS91]. OSDL a été conçu de telle façon que la transition de SDL à OSDL soit facile. Ainsi, la compatibilité ascendante est assurée et les concepts de SDL sont inchangés. Le langage est inspiré de Simula et Beta, mais les termes utilisés sont ceux de SDL (par exemple, les blocs et processus représentent les classes). Nous avons choisi ce langage d'une part parce que c'est une norme (version préliminaire pour OSDL) pour la spécification de systèmes concurrents, et d'autre part parce que la notion de composant enfichable (réutilisable) est mise en évidence, et aussi parce qu'il existe ici des types abstraits et des sortes de classes, mais qui sont disjoints. LOTOS et Estelle font partie de la même famille de langage de spécification formelle de systèmes concurrents [Dia92].

Une spécification définit des types ayant un nombre quelconque d'instances. Un type peut être vu comme une abstraction d'un ensemble d'instances ayant les même propriétés. Contrairement aux autres sections de ce chapitre, nous allons d'abord détailler le vocabulaire propre du langage avant de voir la correspondance en terme de modèle à objets.

Caractéristiques propres

Une spécification OSDL est la donnée d'un système, de données, de blocs et de processus. Une description textuelle précise chacun de ces types. Nous l'omettrons ici. Chacune de ces instances est définie par un type, dont nous allons voir les caractéristiques dans les paragraphes suivants.

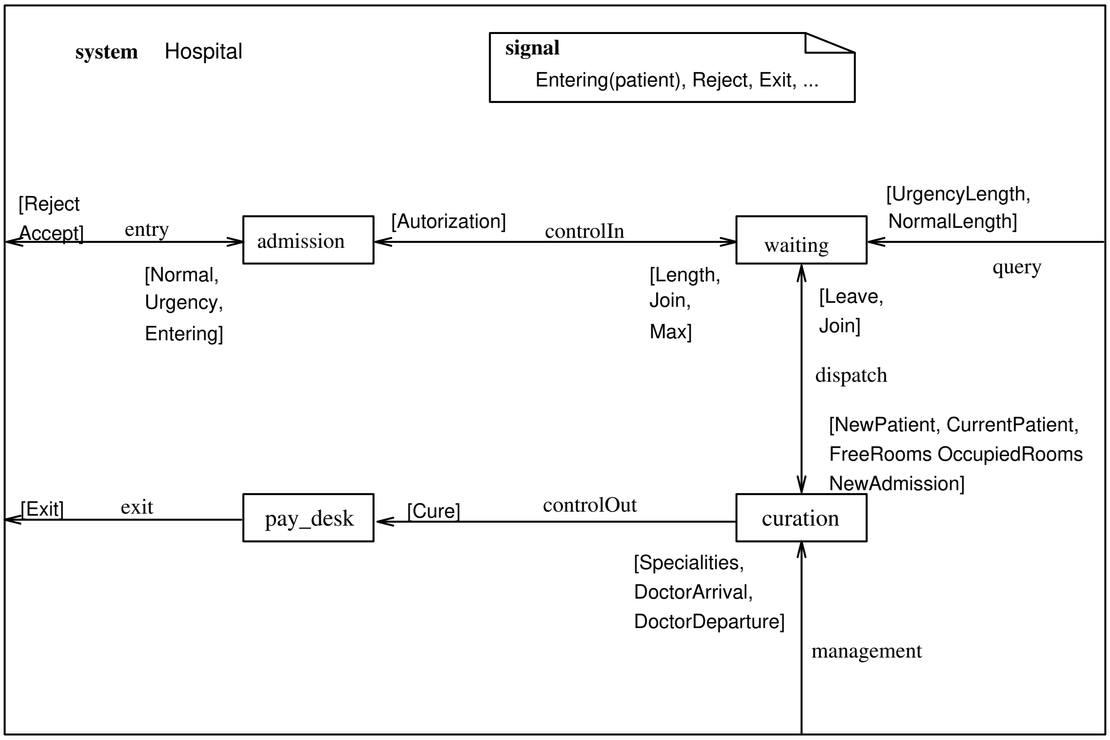


Figure 15 : Spécification OSDL de l'hôpital.

Système La spécification d'un système consiste en un ensemble de `blocs` qui interagissent entre eux et avec l'environnement du système par des `canaux` de communication. Ces interactions sont réalisées via des `signaux`, éventuellement munis d'informations. La spécifications est décomposée en deux parties : une partie déclarative comportant des déclarations de signaux (boîte écornée) et une partie interaction entre blocs définissant l'architecture du système. Une notation graphique agréable synthétise ces spécifications. La figure 15 est un exemple de spécification de l'hôpital.

Les encarts, appelé `text symbol`, contiennent les déclarations de signaux avec leurs paramètres, de types de données (sous forme de type abstrait). Les boîtes représentent des blocs ou des processus (bloc sans sous-bloc), qui décriront plus en détail les signaux, les routages et les actions. Les liens entre les boîtes sont des canaux de communications uni- ou bi- directionnels, pour lesquels les signaux captés sont précisés dans chaque sens.

Nous aurions pu utiliser un type générique patient, introduit par le mot-clé **fpar** (formal parameter). Les spécifications de types locaux se font par `newtype`, qui définit un nouveau type abstrait de données (une seule sorte), ou bien `syntype` qui ajoute un alias avec possibilité de contraindre le domaine de définition. Le mot clé `referenced` signifie que le bloc sera détaillé ultérieurement. Les canaux sont détaillés pour chaque sens, l'interaction avec l'environnement est notée par `env`.

Bloc Une spécification de bloc est soit locale soit différée. Dans les deux cas, l'inclusion des blocs établit une hiérarchie de présentation de la spécification. Une spécification de bloc comprend une partie déclaration et une partie interaction de processus. La partie déclaration est identique à celle du système. La partie interaction de processus décrit les processus en jeu et leurs communications via des routages. Les processus sont représentés par des octogones. La notation est celle des systèmes. Par exemple, détaillons le bloc de soin `curation`. Le lien en pointillé exprime la création (dynamique) de processus.

Les blocs peuvent être imbriqués en sous-structures pour structurer de grosses spécifications et permettre le raffinement des spécifications. Les blocs terminaux contiennent uniquement des processus. Les blocs composites incluent d'autres blocs. Ces deux peuvent être assemblés dans un bloc combiné. Mais dans ce cas, les sous-structures représentent une implantation particulière du bloc (décrit par les processus), il reprend alors les mêmes canaux.

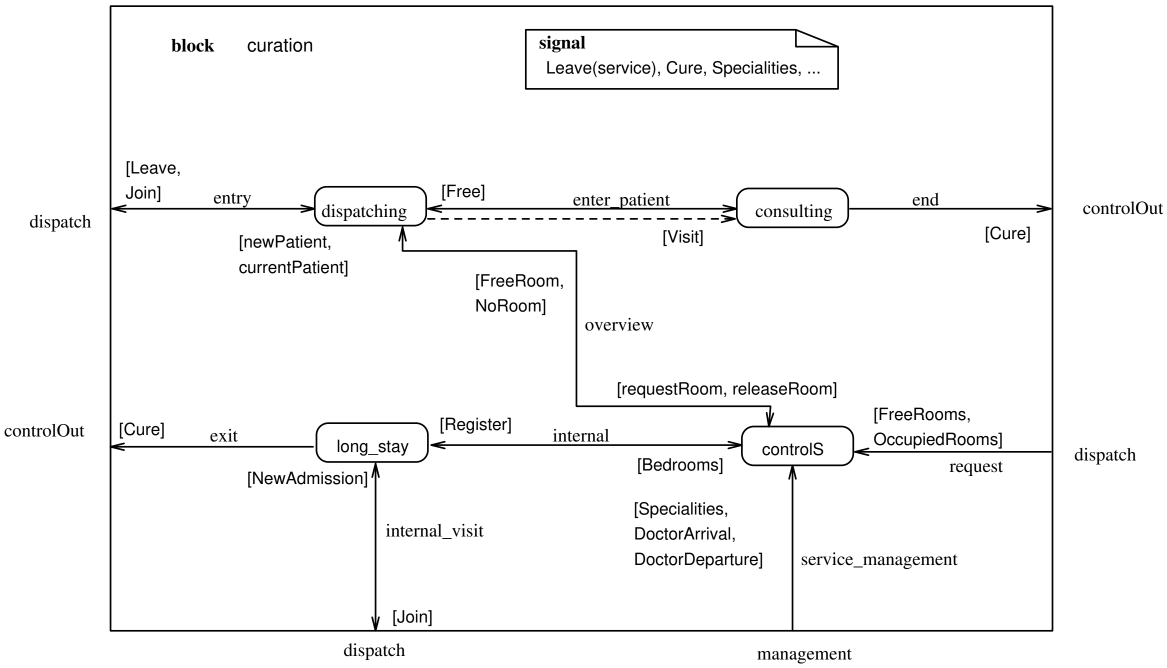


Figure 16 : Spécification OSDL du bloc `curation`

Processus Au niveau élémentaire, les processus décrivent le comportement dynamique du système. Cette partie est la plus connue de SDL. Le comportement des blocs et du système sont dérivés des comportements des différents processus constituants. Il existe une partie déclaration et une partie implantation. Les déclarations, ici aussi, portent sur les types de données locaux, les signaux, et des variables. La description du corps du processus est donnée par un système de transitions fini ayant une syntaxe propre. Contrairement aux autres unités de structuration, la spécification d'un processus peut être décrite par plusieurs diagrammes, les états faisant le lien. La figure 17 illustre ce concept.

Trois concepts sont utilisés : les états, les entrées (ou signaux) et les état suivants. Les transitions indiquent les traitements effectués. Ces traitements sont définis à un niveau opérationnel par des tâches, utilisant des effets de bords sur les variables déclarées. La transition vers un autre état peut être conditionnée par un choix (sorte de case) appelé **décision**. Notez ici une influence Ada. Enfin, remarquons que si les signaux d'entrée déclenchent les transitions, une primitive `output` permet la production d'événements sur un canal précisé (`via`) ou à une adresse donnée (`to`) avec les destinataires spéciaux (`self` pour le processus lui-même, `sender` pour une réponse à l'émetteur du dernier signal, `parent` pour le processus qui a créé le processus courant, `offstring` pour le dernier processus fils). D'autres particularités telles que la création de processus, l'arrêt de processus, la gestion du temps ou la mémorisation des signaux sont décrits dans [BHS91].

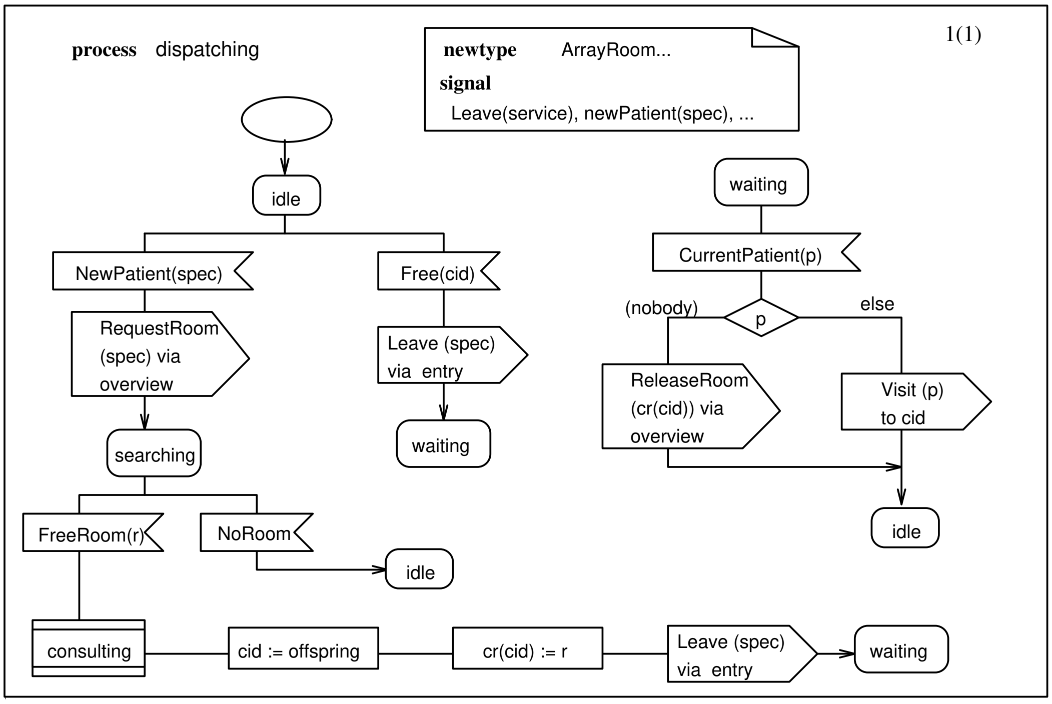


Figure 17 : Spécification OSDL du processus `dispatching`.

Type abstrait de donnée Signalons ici quelques particularités de ces spécifications algébriques. Les axiomes sont des équations conditionnelles quantifiées universellement. Les alternatives sont acceptées (`if-then-else-fi`, `or`). Un terme spécial, noté `error!` et définit pour chaque sorte, désigne les exceptions et provoque une erreur dynamique. Les types génériques sont définis par la clause `generator`. OSDL définit un héritage des spécifications algébriques avec renommage des opérations.

newtype Doctor
    inherits Person **operators all**
    adding operators
        set_speciality: Patient Service -> Patient;
        speciality: Patient -> Service;
endnewtype Doctor;

generator Queue
    (**type** Person, **litterals** empty_queue)
    litterals empty_queue;
    operators
        join: Queue Person -> Queue;
        leave: Queue -> Queue;
        next: Queue -> Person;

```
            length: Queue -> Integer;
        axioms
        for all q in Queue, p in Person
            (leave(empty_queue) == error!;
            length(join(q,p)) = 1 ==> leave(join(q,p)) == q;
            length(join(q,p)) > 1 ==> leave(join(q,p)) == join(leave(q),p);
            next(empty_queue) == error!; /* attention c'est le error de Person */
            length(join(q,p)) = 1 ==> next(join(q,p)) == p;
            length(join(q,p)) > 1 ==> next(join(q,p)) == next(q);
            length(empty_queue) == 0;
            length(join(q,p)) == 1 + length(q););
    endgenerator;

    newtype HospitalQueue Queue(Patient,empty_patient_queue);
    endnewtype;
```

Mécanismes de structuration D'autres mécanismes que ceux vus ci-dessus, permettent de spécifier des systèmes plus complexes : le partitionnement de blocs ou de canaux, le raffinement de signaux, les procédures et les services. D'autres concepts ne seront pas détaillés ici : les macro-definitions, les signaux continues ou conditionnels, le non-déterminisme des transitions, les règles de visibilité.

Si un bloc contient des processus et des sous-structures, alors les sous-structures sont une implantation particulière des processus. Cette perception correspond bien aux couches des réseaux pour lesquels on définit des protocoles (entre entités de même niveau) et des services vers les entités de niveau inférieur ou supérieur. De même, les canaux sont partitionnés en sous-structures en faisant apparaître des blocs supplémentaires, en conservant les signaux d'entrée et de sortie.

L'ajout de détail supplémentaires est appelé raffinement. Un signal est remplacé par un ensemble de signaux de niveau inférieur. Par exemple, à un certain niveau, des fichiers sont communiqués et au niveau inférieur, ce seront des records et des délimiteurs.

Les procédures OSDL sont des procédures classiques définies pour les spécifications de processus ou de services (encapsulation, répétition). La notation graphique associée est celle des processus, elle met en valeur le paramétrage et les résultats. Elles sont interprétées comme des machines à états activées lors des appels de procédure (synchrone).

Les services sont une alternative aux spécifications de processus. Il s'agit en fait conceptuellement de spécifier des processus par d'autres processus, appelés services ayant des routages propres. Le processus est alors décrit par un diagramme proche de celui des blocs terminaux (i.e. ayant des processus) dans lequel le service joue le rôle du processus. La notation graphique associée est l'ellipse. Ils partagent la file de signaux du processus décrit et leur spécification est donnée par un diagramme état/transition similaire à celui des spécifications (terminales) de processus.

Modèle à objets

Le modèle à objets est inspiré de Simula et Beta. Une exécution de programme est vue comme un modèle physique simulant le comportement d'une partie réelle ou imaginaire du monde. Le modèle physique est construit à partir d'objets.

SDL définit des objets passifs (types de données) et des objets actifs (processus, services). Les blocs, sous-structures et le système représentent des objets composites. L'agrégation est explicite et les relations entre objets sont les canaux. Seuls les processus sont instanciables dynamiquement dans SDL, alors que dans OSDL, les blocs le sont aussi par la primitive `create`. La destruction est possible pour les processus par une clause `stop`. La structure des objets comprend les types définis, les variables pour un processus et les composants. Il n'y a pas d'invariant. Le comportement est donné par les signaux acceptés et émis. Il n'y a pas d'exceptions.

Le langage permet une hiérarchisation des composants selon un critère d'inclusion avec pour certains objets une spécification externe et une description interne. C'est le cas des blocs avec sous-structures et processus, ou encore des processus avec spécification en terme d'automates et description en terme de services. Les connexions par canaux ou routages et les déclarations de signaux définissent l'interface des composants.

L'envoi de message se fait soit par signaux soit par appels de procédures pour les émissions internes. Différents modes d'adressages sont possibles, comme nous l'avons vu, et notamment l'appel au receveur lui-même par **`self`**. Des primitives permettent de rendre un accusé de réception suivis d'un signal avec le résultat si nécessaire. Dans OSDL, l'appel de procédure à distance a été rajouté au mécanisme d'envoi de signaux.

La généricité contrainte est possible pour les types de données et les macro. Celle des systèmes l'est via les options déterminant les parties facultatives des spécifications et la possibilité d'introduire des éléments externes par la primitive **`synonym`** ...**`external`**.

Les variables, processus et blocs sont typés. Le polymorphisme est possible pour les blocs ayant des connexions compatibles.

L'apport de OSDL par rapport à SDL est concentré sur l'introduction de l'héritage pour les blocs et processus [CLLF93]. Les processus sont spécialisés par héritage simple : le processus ancêtre est inclu, les états, signaux et transitions sont héritées ou ajoutés. Les procédures et transitions virtuelles sont redéfinies. Il y a trois sortes de transition virtuelle, une entrée virtuelle (i.e. redéfinissable), une sauvegarde virtuelle (redéfinie par une sauvegarde ou une entrée) et un démarrage virtuel redéfini par un processus de démarrage.

Aspects formels

La sémantique du langage SDL est définie dans la recommandation Z.100 du CCITT [BHS91] avec VDM selon [CLLF93]. Une spécification SDL est transformée dans une syntaxe abstraite qui sert aux vérifications statiques. Chaque élément (bloc, processus, procédure, service, signal) est typé. Une description de bloc correspond à une seule instance de bloc[5]. Un outil réalise l'analyse statique.

La sémantique dynamique est donnée en termes de machines abstraites et de processus CSP. Ces machines sont des extensions d'automates à états finis qui communiquent entre eux et avec l'environnement de manière asynchrone par signaux. Ces signaux sont mémorisés dans un tampon pour chaque machine. Les extensions concernent la détention d'informations locales et la gestion du temps. Une description plus approfondie des machines abstraites et processus communicants de SDL est donnée dans [BHS91].

Environnement

La notation graphique normalisée du langage constitue un attrait du langage, dans son exploitation quotidienne. Un environnement, appelé OSDT, enrichit l'environnement SDT du langage SDL disponible depuis 1988. Un éditeur graphique permet la saisie les spécifications et réalise des vérifications syntaxiques. Un flâneur de structure permet de passer d'un niveau à l'autre dans la hiérarchie d'agrégation. Un analyseur teste totalement ou en partie, la correction d'une spécification. Trois passes sont nécessaires : analyse syntaxique, analyse sémantique et analyse dynamique. D'autres outils sont en cours de développement : simulateur, générateur de code C, flâneurs divers, etc.

Conclusion

Même si les termes varient, les concepts d'OSDL sont proches du modèle à objets. Un accent est mis sur la structuration des composants et sur les aspects cycle de vie. Un des intérêt du langage, outre la structuration en termes de couches, proches des définitions de réseaux de communication, est sa normalisation (CCITT) et la notation graphique qui accompagne tous les

[5] Seuls les processus sont instanciés dans SDL.

concepts retenus. Contrairement aux autres langages étudiés, il donne un rôle prépondérant aux processus communicants. L'expression du contrôle, centralisé dans les modélisations Object-Z est complètement décentralisé ici. Le langage met aussi en valeur la difficulté d'introduire de l'héritage de comportement dynamique.

3.2.6 OS/OP

Les langages OS (Object Oriented Algebraic Specification Language) et OP (Object Oriented Kernel Programming Language) ont été développés par Ruth Breu à l'université technique de Munich[Bre91]. Le langage OS est un langage de spécification algébrique basé sur les sortes partiellement ordonnées, permettant la structuration des spécifications en classes. OP est un langage de programmation ayant les mêmes constructions qu'OS, avec une partie structure des objets et définition de méthodes et une partie implantation de méthodes.

Modèle à objets

Les spécifications de classes (`class spec`) sont des spécifications algébriques. Les classes (`class`) sont des représentation particulières de ces spécifications. Les premières sont manipulées dans OS et les secondes dans OP. L'acronyme **TAD** correspondra ici à la fois aux spécifications de classes et aux classes. Une spécification est définie par ses relations, des opérations et des axiomes. Les axiomes sont des équations universellement quantifiées, avec négation et conjonction. Ils peuvent être conditionnés par un prédicat de définition noté `D`. Trois types de relations sont définies entre les TAD : la clientèle (`use`), le sous-typage (`subclass of`) et l'héritage (`inherits`). La spécification peut être générique, le paramètre est alors vu comme un serveur. Les spécifications de base `Int` et `Bool` sont prédéfinies.

```
class spec Patient is
end class spec

class spec AbstQueue[Patient] is
    uses    Int, Bool, Patient
    opns    open: → AbstQueue[Patient],
            join: (AbstQueue[Patient], Patient) → AbstQueue[Patient],
            length: (AbstQueue[Patient]) → Int,
            is_in: (AbstQueue[Patient], Patient) → Bool,
            is_empty: (AbstQueue[Patient], Patient) → Bool,
    axioms  ∀ aq, aq': AbstQueue, p,p': Patient, l: Int.
        is_empty(open) = true,
        is_empty(join(aq, p)) = false,
        length(open) = 0,
        length(join(aq, p)) = succ(aq),
        is_in(open, p) = false,
        equal(p, p') ⇒ is_in(join(aq, p), p') = true,
        ¬ equal(p, p') ⇒ is_in(join(aq, p), p') = true
end class spec
```

Cette spécification est implantée par une classe. Les opérations sont appelées méthodes dans OP. Les axiomes sont regroupés par méthode. Les méthodes peuvent être abstraites. La sémantique des méthodes est donnée dans une clause implantation, sous forme pseudo-algorithmique (séquentialité, alternative, void, envoi de message). Soit `succ(x)` la fonction successeur pour $x \in Int$.

```
class AbstractQueue is
    methods join (p: MyT) is deferred,
            is_empty return Bool is deferred,
            is_in (p: MyT) return Bool is deferred,
            length return Int is deferred,
```

```
            create open is deferred
end class

class ListQueue is
    inherits AbstractQueue
    attributes   head: MyT,
                 tail: ListQueue
    methods join (p: MyT) is concreted,
            is_empty return Bool is concreted,
            is_in (p: MyT) return Bool is concreted,
            length return Int is concreted,
            create open is concreted
implementations
    join(p: MyT) is
        if is_empty then head:= p; tail:= open
                 else tail:= tail.join(p: MyT) fi end,
    is_empty return Bool is tail.is_void end,
    is_in (p: MyT) return Bool is
        if is_empty then false
            else if head.equal(p: MyT) then true
                 else tail.is_in(p: MyT) fi fi end,
    length return Int is
        if is_empty then 0
            else succ(tail.length) fi end,
    create open is standard
end class
```

Les objets OP ont une identité. La référence vide est `void`. L'envoi de message se fait par la notation pointée dans laquelle le receveur noté `Self` peut être omis. Pour manipuler les files de patients, les descriptions ci-dessus sont enrichies. Les patients sont classés par sous-typage. En général, le supertype ne contient pas de générateurs.

```
class spec Queueable is
    uses    Bool, Char
    opns    name: (Queueable) → Char,
            rename: (Queueable, Char, Char) → Queueable,
            surname: (Queueable) → Char,
            equal: (Queueable, Queueable) → Bool
    axioms  ∀ p,p': Queueable, n,s: Char.
        name(rename(p,n)) = n,
        surname(rename(p,s)) = s,
        equal(p,p') =
            equal(name(p),name(p')) ∧ equal(surname(p),surname(p'))
end class spec

class spec Person is
    subclass of Queueable
    opns    register: (Char, Char) → Person
    axioms  ∀ n,s,n',s': Char.
        name(register(n,s)) = n,
        surname(register(n,s)) = s,
        rename(register(n,s),n',s') = register(n',s')
end class spec

class spec Animal is
    uses    Person
    subclass of Queueable
    opns    register: (Char, Char, Person) → Animal,
            owner: (Animal) → Person
```

```
    axioms  ∀ a,a': Animal, n,s,n',s': Char, o: Person.
        name(register(n,s,o)) = n,
        surname(register(n,s,o)) = s,
        owner(register(n,s,o)) = o,
        equal(p,p') =
            equal(name(p),name(p')) ∧ equal(surname(p),surname(p') ∧
                equal(owner(p),owner(p')),
        rename(register(n,s,o),n',s') = register(n',s',o)
end class spec

class spec PriorPerson is
    uses    Int
    subclass of Person
    opns    priority: (PriorPerson) → Int,
            inc_prior: (PriorPerson) → PriorPerson,
            lte: (PriorPerson, PriorPerson) → Bool
    axioms      ∀ p,p': PriorPerson, n,s: Char.
        priority(register(n,s)) = 0,
        priority(inc_prior(p)) = succ(priority(p)),
        lte(p,p') = priority(p) ≤ priority(p'),
        equal(p,p') =
            equal(name(p),name(p')) ∧ equal(surname(p),surname(p')) ∧
                equal_int(priority(p),priority(p'))
end class spec
```

L'actualisation du type générique `AbstQueue` se fait par `AbstQueue[Person:Patient]` ce qui est équivalent à

```
class spec AbstQueue[Person:Patient] is
    uses    Person
    inherits AbstQueue[Patient] refine Patient into Person
end class spec
```

La file d'attente `Queue` est définie par raffinement de la spécification `AbstQueue`, avec une opération privée `single` puis implantée par **FIFOQueue**.

```
class spec Queue[Person] is
    uses    Person
    inherits AbstQueue[Person:Patient] rename by [length → size]
    opns    leave: (Queue[Person]) → Queue[Person],
            first: (Queue[Person]) → Person,
    hidden  single: (Queue[Person]) → Bool,
    axioms  ∀ aq: Queue[Person], p: Person.
        single(p) = equal_int(size(p),succ(o)),
        ¬ D first(open),
        is_empty(aq) ⇒ first(join(aq, p)) = p,
        ¬ is_empty(aq) ⇒ first(join(aq, p)) = first(aq),
        ¬ D leave(open),
        single(aq) ⇒ leave(join(aq, p)) = join(open,p),
        ¬ single(aq) ⇒ leave(join(aq, p)) = join(leave(aq),p)
end class spec

class FIFOQueue is
    inherits ListQueue refine MyT into Person
        rename by [length → size]
    methods leave,
            first return Person,
            single return Bool
implementations
```

```
    single return Bool is equal_int(size,succ(o)) end,
    first return Person is
        if single then head
            else if not empty then tail.first fi fi end,
    leave is
        if single then open
            else if not empty then tail:= tail.leave fi end
end class
```

Aspects formels

Les deux langages sont intégrés dans un même formalisme, les algèbres partiellement ordonnées. Une spécification de classe définit un module par une signature et des axiomes et un type par les ensembles support associés. La conformité des classes vis à vis des spécifications est donnée par des fonctions d'abstractions et des morphismes d'algèbres. La spécification OP est exécutable dans des algèbres à objets. Ces algèbres sont formées d'une algèbre d'environnement (application entre les identités d'objets et leur valeur) et des fonctions de transition locales qui interprètent les changements d'états. Un prototype du système est ainsi obtenu.

Environnement

Un environnement est à l'étude, avec les points suivants : outils de prototypage rapide pour exécuter les spécifications OS, déterminer des stratégies de preuves de correction de spécifications OS, vérification du respect de conformité des classes vis-à-vis de leur spécification algébrique, traduction des spécifications OP en langages à objets.

Conclusion

Un des principaux apports de ce langage est l'étude théorique de la correction entre un type abstrait et une classe qui l'implante avec conservation de la composition horizontale. Il y a unification des deux concepts dans une seule sémantique basée sur les algèbres partiellement ordonnées. Une théorie est développée dans [Bre91] sur l'intégration de l'héritage, le sous-typage et la clientèle. Une séparation entre l'héritage et le sous-typage est donnée ici en terme de modèle et d'ensemble support. Si cette séparation est claire théoriquement, dans la pratique des règles pour la contrôler font défaut et le spécifieur ne sait pas toujours ce qu'il a droit de faire. Les règles pratiques d'héritage et de sous-typage ne sont pas très explicites dans [Bre91]. Ce qui constitue une difficulté dans la construction des spécifications et des classes. Le passage des types aux classes n'est pas trivial. La démonstration de la conformité de représentation doit être guidée. La méthode doit être étendue aux aspects concurrents et doit systématiser le raffinement.

3.2.7 Autres langages

D'autres langages de spécification à objets ont été développés. Nous ne les détaillerons pas, mais indiquerons les proximités avec les langages ci-dessus.

Une description succincte du langage PLUSS a déjà été donnée en section 2.2.6. PLUSS n'est pas un langage objet, mais il propose des constructions inspirées des concepts à objets, telles la clause *enrich* qui exprime une certaine forme d'héritage.

GSBL

GSBL est un langage de spécification algébrique basé sur l'héritage [CO88]. Sa sémantique est à classe de modèle, comme PLUSS, pour des algèbres multi-sortes. Au niveau de la spécification, l'héritage est défini par le sous-typage de spécifications ordonnées : c'est un enrichissement d'un type défini complètement. A l'inverse, l'héritage peut servir au raffinement vertical de spécifications incomplètes. Comme en PLUSS, une distinction est faite entre les spécifications complètes (clause `define`) et les spécifications incomplètes (clause `with`). Une notion de clientèle est donnée par la clause `over`. Il n'y a pas de véritable sous-typage, seul l'héritage est pris en compte. La généricité est une conséquence des deux relations `over` et `subclass`.

Z++

Z++ est langage de spécification et de conception à objets basé sur Z [LH92, LH93, LH94]. Il a été développé à l'université d'Oxford dans le cadre du projet Esprit II REDO. C'est le langage le plus complet de notre panorama. La syntaxe n'utilise pas à priori les schémas Z, mais s'inspire par contre des machines abstraites de B. Le langage comprend des constructions à objets définies dans un style modulaire : importation par **EXTEND**, paramétrage par types ou sous-type, déclarations de types et variables locales, invariant, séparation entre la signature des méthodes (**OPERATIONS**) et leur sémantique axiomatique (**ACTION**) sous forme de code Z. Les préconditions explicites sont suffixées par **&**. Les méthodes suffixées par '*' sont des démons privés et ne sont invocables ni par les clients ni par les descendants.

Il n'y a pas d'identité d'objet, ou de manipulation explicite des instances d'une classe. Une contrainte hiérarchique est imposée entre les classes i.e. pas de circularité et donc de définition récursive. Le polymorphisme est possible par conversion explicite. Une sémantique algébrique du langage est donnée via la théorie des catégories. Elle sert de support au raffinement mais surtout au raisonnement et à la preuve des spécifications. Un paragraphe contrainte permet de définir des équations sur la spécification. Une seconde sémantique, basée sur les machines abstraites de B, est utile pour un autre aspect du raffinement, le prototypage. L'aspect algébrique semble cependant mal intégré à l'aspect modèle. Comme dans Object-Z, une partie historique, définie en logique temporelle, permet de prendre en compte des contraintes temporelles. Une extension aux modules est donnée par le bloc **MODULE**. Les multi-méthodes sont à l'étude.

MooZ

MooZ [MC92, LH93], Modular Object-Oriented Z, est une extension objet de Z qui se différencie des langages Object-Z ou Z++ par le fait que les types Z sont aussi décrits par des classes. Une classe MooZ définit un type, paramétrable par des ensembles supposés existants ou des constantes. Ces ensembles sont soit des paramètres formels soit des déclarations locales. Une définition de classe décrit les liens d'héritage, l'état, l'état initial et les opérations. Toutes ces parties sont données par des schémas Z. Les schémas d'opérations utilisent les notations Δ pour les variables modifiées et Ξ pour les variables lues et la composition d'opérations. Seules les opérations nouvellement définies sont privées.

Les conflits d'héritage multiple sont résolus par renommage, même pour l'héritage répété. L'héritage est indépendant des mécanismes de visibilité. Le schéma d'état est la conjonction des schémas d'état des superclasses. Les exceptions sont redéfinies et la compatibilité ne porte que sur le nom de la méthode.

Une instance est une structure et non un historique comme dans Object-Z. Ces instances communiquent par envoi de message (`obj msg`). Il est possible d'envoyer des messages aux classes pour les opérations qui ne dépendent pas de l'état. Les objets sont polymorphes et l'égalité est possible entre une instance d'une classe et celle d'une sous-classe. Il n'y a pas de modules ni d'expression de la concurrence.

Fresco

Contrairement aux autres langages présentés ici, Fresco [Wil91] n'est pas une extension d'un langage de spécification mais d'un langage de programmation à objets. Il a été développé par Alan Wills à l'université de Manchester. Il est inspiré de Smalltalk et VDM, comme SmallVDM [LH93]. Cette combinaison s'appuie sur le partitionnement du modèle à objets, inclut des définitions génériques, insiste sur la représentation à objets des valeurs plutôt que sur un modèle purement fonctionnel et prend en compte l'aliasing des objets. Fresco est basé sur Mural, l'environnement de spécification VDM écrit en Smalltalk [JJLM91]. Fresco étend Smalltalk par inclusion de **capsules** réutilisables et par l'utilisation d'un langage de spécification pour les vérifier. Les constructions formelles sont dérivés de VDM.

Il y a séparation entre type et classe en Fresco. Les classes implantent des types. Les types décrivent le comportement des objets à réception des messages dans un style VDM incluant la référence à `self`. Les types sont étendus par héritage de conformité. Un type définit un ensemble d'historiques des opérations. Le sous-typage est une inclusion de ces ensembles. L'aboutissement d'une suite de raffinements est une classe Smalltalk. Dans ce type d'héritage, il y a union des préconditions et conjonction des post-conditions.

Une capsule est une collection de spécifications de types, d'implantation de classes avec des preuves formelles. Une capsule peut être créée dans n'importe quel ordre (code $\Leftrightarrow$ spécification). Fresco génère des obligations de preuve quand la cohérence entre spécification et code ne peut être vérifiée automatiquement. Une capsule peut contenir d'autres capsules. Le graphe de dépendance est unidirectionnel et sans circuit.

3.3 Synthèse

Cette section établit des éléments de comparaison et une synthèse comparative de quelques langages de spécification formelle à objets. Dans un premier temps, nous allons examiner les critères d'évaluation puis nous donnerons un tableau récapitulatif pour les langages les plus connus. Nous nous sommes entre autre inspirés de [Bre91, CLLF93, LH93].

De nombreux critères permettent de classer les méthodes. Nous allons classer les critères en cinq classes : langages et modèles, concepts objet, autres concepts de génie logiciel, techniques formelles, méthode et environnement de développement. Cette synthèse n'est pas exhaustive, des caractères propres à certaines méthodes sont omis.
NB: Dans la suite, les critères suffixés par * n'apparaissent pas dans les tableaux.

3.3.1 Langages et modèles

Une première présentation concerne l'origine du langage, ses modèles et son influence dans le développement logiciel :

- Origine* : laboratoire de recherche dont est issu le langage. Souvent, des traditions existent au sein des équipes de recherche, qui expliquent les motivations et les choix faits lors de la conception du langage.
- Inspiration* : théorie ou courant à l'origine du langage. La source d'inspiration conditionne souvent la présentation des concepts objets et surtout les aspects formels. Le langage peut être une extension d'un langage ou une refonte complète du modèle.
- Portée* : situation dans le cycle de développement. Nous n'avons volontairement retenu que des langages permettant au moins la spécification. Nous écartons donc les langages de conception détaillée tels qu'Eiffel.
- Intégration : extension ou nouvelle conception. Plusieurs formalisme et plusieurs modèles sont parfois proposés.
- Notation : style de descriptions et conventions syntaxiques adoptées. La notation influe sur la lisibilité des spécifications.
- Style : le style correspond à une des familles que nous avons discerné dans la section 2.2.4.
- Modèles : ce critère décrit le support théorique du langage.

Langage	intégration	notation	style	modèles
Object-Z	extension	style Z	modèle abstrait axiomatique	Z logique temporelle
OOZE	extension de FOOPS et OBJ	style Z	modèle abstrait algébrique	algèbres ordonnées
VDM++	extension	style VDM	modèle abstrait automates, traces	VDM
OSDL	extension	style SDL Ada	structurelle/ processus	machines abstraites CSP
OS	natif	opérations/ axiomes	algébrique	algèbres partiellement ordonnées
OP	langage OP	impératif	opérationnel	algèbres
PLUSS	natif	opérations/ axiomes	algébrique	algèbres
COLD	natif	modulaire logique	algébrique axiomatique	algèbres logique
GSBL	natif	opérations/ axiomes	algébrique	algèbres
Z++	extension + algèbres	style Z	modèle abstrait axiomatique	Z algèbres
MooZ	extension	style Z	modèle abstrait axiomatique	Z
Fresco	extension	Smalltalk VDM	modèle abstrait axiomatique	VDM typage de classes

Figure 18 : Synthèse : Langages et modèles

3.3.2 Modèle à objets

Le modèle à objets est unificateur, nous l'avons vu. Cependant chaque méthode ne retient qu'une partie des concepts et de plus les définitions peuvent varier en fonction du point de vue choisi.

Langage	entités	abstraction	composition	identité
Object-Z	types Z objets	schémas classes	par variables	oui
OOZE	objets termes	classes, modules théories, tad	par variable importation	implicite self
VDM++	types VDM objets	constructeurs VDM classes	par variable constructeurs VDM	explicite
OSDL	processus blocs termes	type processus type bloc tad	par variables, USE inclusion d'objets et de types	par nommage de blocs et processus
OS	termes	class spec	uses	non
OP	objets	classes	attributs	oui, Self
PLUSS	termes	tad (spec, draft)	use	non
COLD	termes objets	tad, classes	`IMPORT FROM`	nommé
GSBL	termes	sort classes	`over`	non
Z++	objets valeurs Z valeurs	classes types Z tad	par variables `EXTEND` (peu intégré)	explicite (valeurs)
MooZ	objet valeurs Z	classe types Z	par variables	explicite
Fresco	objets	type classe	par variables	implicite &var, self

Figure 19 : Synthèse : Caractéristiques objet 1/6

• Entité : toutes les méthodes proposent des mécanismes de structuration d'entités de base en objets (encapsulation), d'autres types sont parfois acceptés, présence d'abstractions de données dans le langage.

• Abstraction : les objets sont regroupés dans des classes. Les classes servent aussi à la création dynamique des objets. Le terme TAD sous-entend les types abstraits de données sous diverses appellations (types, sortes, spécification).

• Relation : compositions des classes selon des relations de clientèle, d'utilisation ou d'agrégation.

• Identité : l'identification implicite des objets a des conséquences sur le modèle mais aussi la modélisation. L'utilisation des alias facilite la modélisation des systèmes à états mais pas le raisonnement sur les spécifications.
• Généricité : paramétrage des classes par des types ou des valeurs. L'actualisation de ces paramètres crée de nouveaux types. La généricité peut être contrainte.

• Etat : description de la structure des objets en termes de variables. Dans certains langages, il n'y a pas d'état, mais uniquement des accesseurs.

• Invariant : contrainte sur la forme des objets d'une classe. L'invariant a une influence sur l'héritage.

• Interface : séparation entre le service offert et la description interne des objets.

Langage	**généricité**	**état**	**invariant**	**interface**
Object-Z	oui type Z ou classe	variables (accesseurs)	prédicat lié au schéma d'état	liste d'exportation tout par défaut
OOZE	contrainte actualisation par vue	variables (accesseurs)	prédicat lié au schéma d'état	variables et méthodes peuvent être privés
VDM++	non	variables (accesseurs)	prédicat lié aux variables `Invar-`	méthodes publiques variables privées
OSDL	contrainte	variables	non	explicite par signaux et canaux
OS	oui	non	prédicats de définition `D`	opérations privées
OP	non	attributs	non `methods/implementation`	classe OS associée
PLUSS	oui `proc, as`	non	prédicats nommés	`export`
COLD	oui par λ-abstraction	accesseurs	prédicat nommé pour les `PROC`	`EXPORT`
GSBL	simulée/héritage	non	non	non
Z++	contrainte	variables	prédicat `INVARIANT, CONSTRAINTS`	méthodes publiques variables privées
MooZ	non contrainte (given set de Z)	variables	prédicat lié au schéma d'état	méthodes déclarées publiques ou privées
Fresco	non	variables (typées)	prédicat lié à l'état	méthodes pub/priv variables privées

Figure 20 : Synthèse : Caractéristiques objet 2/6

• Méthodes : définition du comportement des objets en termes de prédicats, fonctions, procédures. Dans certaines méthodes, la partie modifiée de l'objet est explicite. Le cas de COLD est à part car la procédure indique les fonctions qu'elle perturbe.

• Définition : définition des méthodes par des assertions ou dans un style algébrique, fonctionnel ou impératif. Les assertions sont données généralement en logique des prédicats. Tandis que les axiomes sont des équations simples ou équations conditionnelles (clauses de Horn), dont la logique est calculable.

• Composition : propriété de composer des méthodes entre elles pour définir de nouvelles méthodes.

• Exceptions : mécanismes permettant de masquer des méthodes du comportement ou traiter les cas d'erreur.

Langage	**comportement**	**méthodes**	**composition**	**exceptions**
Object-Z	procédures (Δ, Ξ-liste) fonctions Z	assertions Z	composition $\vee, \wedge, [], \|\|, \bullet$	technique Z habituelle
OOZE	procédures (classes)(Δ, Ξ) fonctions (TADs)	assertions Z axiomes conditionnels	séquentiel ';' parallèle '——'	méthode `Error`
VDM++	procédures (`wr, rd`) fonctions traces	assertions VDM assertions expressions régulières	non	clauses VDM `errs:`
OSDL	procédures actions	état/transition échanges de signaux pseudo-code	invocation séquentielle de processus	non
OS	fonctions	axiomes prédicats de définition	–	non
OP	procédure / fonction	pseudo-code	séquentiel ';'	non
PLUSS	prédicats fonctions	axiomes préconditions	–	non
COLD	prédicats fonctions procédures (`MOD`)	axiomes, def inductions, assertions	séquentiel ';' alternative '—' garde '?'	
GSBL	fonctions	équations style algorithmique	–	non
Z++	procédures (') fonctions Z contraintes	assertions Z axiomes	non	technique Z par sépa- ration des cas
MooZ	procédures (Δ, Ξ-liste) fonctions Z	assertions Z	composition de schémas Z	non
Fresco	procédures (Δ-liste) fonctions	théorèmes (assertions VDM) code smalltalk	conjonction de spécifica- tions avec &	non : utiliser plusieurs théorèmes

Figure 21 : Synthèse : Caractéristiques objet 3/6

• Héritage : possibilité d'étendre des définitions de classes dans de nouvelles classes. Diverses formes d'héritage coexistent parfois : sous-typage ou raffinement, héritage de structure ou de comportement, sélectif ou global, héritage simple ou multiple. Certaines propriétés comme les invariants et préconditions sont conservées, renforcées ou restreintes.

• Redéfinitions : possibilité de changer la spécification d'une méthode.

• Règles de redéfinition* : selon qu'on différencie la classe et le type, l'héritage et le sous-typage, les règles de redéfinitions sont plus ou moins strictes. Object-Z et MooZ utilisent la conjonction de schémas Z pour combiner les définitions de méthodes : des prédicats sont ajoutés; les paramètres et leurs types sont renommés de façon quelconque par masquage. En OOZE, les invariants doivent être renforcés mais la compatibilité avec les méthodes héritées n'est pas vérifiée. En OSDL, toutes les propriétés (signaux, transitions, états) sont conservées. En VDM++, la conformité de type est basée sur la structure par règle covariante. Des preuves de cohérence sont exigées pour l'héritage de comportement, qui est contrôlé explicitement. En Z++, les invariants sont préservés mais la compatibilité avec les méthodes héritées n'est pas vérifiée. En Fresco, l'héritage de spécialisation est prouvé selon une conformité de type. En OS et GSBL, il y a inclusion de signature pour le sous-typage et morphisme pour l'héritage. Voir aussi *polymorphisme*.

• Conflits : politique suivie pour gérer les conflits de nom dus à l'héritage.

• Polymorphisme : la capacité d'une méthode à être appliqué à des objets de type (classe) différent. Polymorphisme d'inclusion et polymorphisme de paramétrage sont parfois distingués [Bre91]. Le premier correspond au sous-typage et signifie qu'un objet d'un sous-type l'est aussi d'un super-type. Le second caractérise le fait qu'un objet ayant plusieurs types est uniformément représenté pour tous ces types [CW85]. Dans l'approche algébrique, ceci correspond à la paramétrisation des types de données [Bre91]. C'est notamment le cas en OOZE et OS/OP. Le polymorphisme se résout souvent par un contrôle de la conformité du type raffiné (Fresco, Z++).

Langage	**héritage**	**redéfinitions**	**conflits**	**polymorphisme**

Object-Z	Multiple	`redef`
		'/' rename
		`remove`
			renommage	explicite
				(`x:` $\downarrow$)

OOZE	Multiple sous-sortes OBJ	extension d'état, invariant, init surcharge de méthodes	redéfinitions héritage non répété	oui conformité de sous-classe
VDM++	Multiple état/ comportement	héritage explicite extension d'état	non héritage répété possible	
OSDL	Simple	redéfinition de procédures et transitions virtuelles uniquement et	–	oui, substi- tution de blocs de connexions compatibles
OS	Multiple subclass of inherits	`refine` `rename`	renommage	sous-typage
OP	Multiple subclass of inherits	`refine` `rename`	renommage	oui
PLUSS	`enrich` (raffinement)	– `forget`	–	oui draft
COLD	non	`RENAME` (plusieurs opérateurs)	–	–
GSBL	`subclass` (raffinement seul)	–	–	non défini
Z++	Multiple (`EXTENDS`)	surcharge renommage masquage	qualification explicite	oui conformité de sous-classe
MooZ	Multiple	surcharge renommage accès qualifié	renommage	substitution de type
Fresco	Multiple (Fresco) Simple (Smalltalk)	extension compatible de théorèmes surcharge en Smalltalk	qualification explicite	substitution de type

Figure 22 : Synthèse : Caractéristiques objet 4/6

• Instanciation : manière dont les objets sont créés.

• Initialisation : manière dont les objets sont initialisés.

• Destructeurs : manière dont les objets sont détruits.

• Message : notation pour la communication entre objets.

• Sélection de la méthode d'instance* : la sélection est simple pour les langages qui séparent le receveur des autres arguments (Object-Z, OOZE, VDM++, OP, MooZ, Fresco). Elle est multiple pour les langages utilisant l'appel fonctionnel (OS, COLD, GSBL, PLUSS). Z++ supporte les deux types de notation (appel fonctionnel vs notation pointée).

• Méthode comme objet* : une méthode peut être créée, passée en paramètre, recevoir un message. Aucun des langages étudiés ne contient à notre connaissance cette possiblité.

Langage	**instanciation**	**initialisation**	**destructeurs**	**message**
Object-Z	implicite avec `INIT`	arguments `INIT`	non	notation pointée
OOZE	implicite avec <u>`Init`</u>	arguments <u>`Init`</u>	`Del(var)`	notation pointée
VDM++	`new` `start`	clauses `init-`	non	envoi de message `expr!msg(param)`
OSDL	`Create` (processus)	arguments `Create`	`Stop`	
OS	générateurs de la spécification	paramètres du générateur	–	appel fonctionnel
OP	`create` multiple	dans create	non	envoi de message `obj.meth(args)`
PLUSS	générateurs de la spécification	paramètres du générateur	–	appel fonctionnel
COLD	générateurs de la spécification; définition spéciale	paramètres du générateur; create	–	appel fonctionnel
GSBL	générateurs de la spécification	paramètres du générateur	–	appel fonctionnel
Z++	implicite avec `Init`	arguments `Init` ou `INIT`	non	envoi de message `(Op param) obj = (result, newobj)`
MooZ	appel au schéma d'initialisation	clause(s) de `initial state`	non	envoi de message `obj msg (param)`
Fresco	`x := AClass()`	méthode `init`	non	notation pointée

Figure 23 : Synthèse : Caractéristiques objet 5/6

• Différé : utilisation de classes abstraites, classes n'ayant pas d'instances, et possibilité de laisser libre des définitions de méthodes.

• Classe entité : ce critère comprend la possibilié de gérer des variables de classes et les instances de la classe.

• Métaclasses : manipulation dynamique des classes.

Langage	différé	classe entité	métaclasses
Object-Z	classe abstraite (notation $\downarrow$)	non	non
OOZE	non	$\overline{Class}$ contient les instances	non
VDM++	non	non	non
OSDL	procédures et transitions virtuelles	non	non
OS	non	non	non
OP	méthodes différées	non	non
PLUSS	spécifications incomplètes	non	non
COLD	définitions libres `FREE`	non	non
GSBL	spécifications incomplètes méthodes	non	non
Z++	non	non	non
MooZ	non	message de classes pour des constantes	non
Fresco		Attributs de classe Smalltalk seulement	Smalltalk seulement

Figure 24 : Synthèse : Caractéristiques objet 6/6

3.3.3 Autres concepts

Chaque langage fournit des caractéristiques propres issues des modèles sous-jacents ou bien de constructions supplémentaires pour exprimer tel ou tel aspect aspect (concurrence, cycle de vie, communication) ou pour structurer les spécifications (module, domaines).

Langage	module	parallélisme	cycle de vie
Object-Z	non	non compatible avec les historiques	historique logique temporelle
OOZE	modules théories, vues	appel concurrent de méthodes '——'	non
VDM++	non	threads (obligations périodiques, commandes gardées, ...) contraintes temporelles	traces (expressions de chemin) contrôle de synchronisation gardes d'historique sur les les appels de méthodes
OSDL	blocs sous-structures systèmes	processus parallèles signaux asynchrones transitions non déterministes	état/transition
OS/OP	non	non	non
PLUSS	non	non	non
COLD	scheme	non	non
GSBL	non	non	non
Z++	module	composition parallèle de schémas avec partage de variables	historique logique temporelle
MooZ	non	non	non
Fresco	capsules	non	non

Figure 25 : Synthèse : Concepts supplémentaires

- module : possibilité de définir des entités de plus haut niveau que la classe pour grouper un certain nombre d'objets fortement liés dans une partie du système.
- visibilité* : possibilité de définir des classes publiques et des classes privées.
- domaine* : regroupement vertical de classes selon la relation d'héritage permettant de déterminer des ensembles de classes sémantiquement proches, ou une classe avec toutes ses implantations (privé/public, vues). Aucun des langages retenus ne définit une construction particulière, même si certains langages comme OS/OP ou GSBL insistent sur le raffinement et la pluri-implantation d'une spécification.
- parallélisme : constructions disponibles pour définir des objets actifs.
- cycle de vie : expression centralisée du contrôle des objets.

3.3.4 Support formel

Le support formel comprend la sémantique du ou des langages, les outils de preuve, le raffinement et le prototypage.

Langage	**sémantique**	**raisonnement**	**exécutabilité**
Object-Z	basée sur Z une logique temporelle définie en Z est utilisée pour les historiques	déduction logique	via C++
OOZE	opérationnelle en OBJ3 la logique de définition des méthodes est restreinte aux équations conditionnelles pour les modules interprétables les règles sont celles d'OBJ	réécriture induction théorie équationnelle	oui via FOOPS et OBJ
VDM++	opérationnelle en VDM sauf pour les construction concurrentes, incomplètement, définies aucune règle n'est introduite pour les objets ou la concurrence	déduction résolution aide à la preuve	non
OSDL	opérationnelle en termes de machines abstraites et de processus communicants	calculs de processus	oui
OS/OP	algèbres partiellement ordonnées	réécriture induction	oui pour OP
PLUSS	variantes algèbres initiales, partielles, exceptions, erreurs contrainte hiérarchique	réécriture induction équation	oui en partie
COLD	axiomes premier ordre opérateur de point fixe algèbres multi-sortes sémantique “loose”	induction par point fixe équationnel	oui
GSBL	algèbres partiellement ordonnées	réécriture induction	en partie
Z++	Z pour les méthodes et objets algèbres pour les classes et l'héritage	$\mathcal{W}$ déduction	non a-priori via B
MooZ	sémantique Z incomlète	déduction induction	non a-priori via B
Fresco	outils VDM avec conformance de type pour les classes	inférence pre/post et clientèle	partie Smalltalk

Figure 26 : Synthèse : Techniques formelles de la méthode 1/2

• Sémantique : un langage bien défini est basé sur une sémantique formelle. La sémantique des extensions est fonction du langage formel support. Si le langage comporte des aspects orthogonaux une autre sémantique est donnée et la cohérence entre les deux n'est pas triviale. C'est le cas pour VDM défini à partir de VDM, CSP et les machines abstraites. La sémantique d'Object-Z, Z++ et MooZ comporte des lacunes [LH93]. Pour Z++, une théorie du sous-typage et du raffinement a été développée mais n'intègre pas les extensions algébriques et concurrentes. L'ajout de caractéristiques imlique l'ajout de nouvelles règles de contrôle.

• Contrôle* : l'environnement support doit permettre des contrôles syntaxiques mais aussi de cohérence et si possible de complétude. En ce sens les spécifications algébriques ont un avantage certain sur les spécifications par modèle, dont la logique est par nature incomplète.

• Raisonnement : mécanismes de preuve et de validation. La logique est le principal support du raisonnement pour les preuves équationnelles, inductives, par l'absurde, etc. Pour être utilisable, elle doit être supportée par un système de raisonnement. Pour les langages issus de Z, citons l'outil B [Abr84, LLS91] (preuve de théorèmes, inférence, machines abstraites) ou encore le système de raisonnement $\mathcal{W}$ de déduction naturelle [LH93]. Pour VDM, l'environnement mural [JJLM91, Jon93] fournit l'assistance à la preuve de théorèmes, l'aide au raffinement de spécification, etc. D'autres langages utilisent Prolog, les langages fonctionnels, ou la réécriture [JL86]. C'est notamment le cas des outils des spécifications algébriques.

• Exécutabilité : possibilité d'exécuter la spécification. Cette propriété est parfois contestée, comme étant trop proche d'un modèle opérationnel. Le prototypage est la transformation plus ou moins directe en un code donné pour exécuter la spécification.

• Typage : plusieurs paramètres distinguent les politiques de typage

- quels types* : types de base, classes, types complexes,
- politique de typage choisie* : statique/dynamique, fort/faible,
- vérifications de type* : quelles vérifications possibles ? interface, comportement, code.
- base du contrôle de type* : structure, comportement.
- polymorphisme* : implicite, explicite,
- voir aussi le tableau page 81 sur comportement/méthode/polymorphisme.
- règles de sous-typage* (statique)[PS92] :

Quand un type porte sur :	**Le sous-typage devient :**
classe et sous-classe	simple relation d'héritage
compatibilité de nom	plus de méthodes définies
interface	conformité des partamètres des profils
comportement	préconditions et postconditions renforcées ou affaiblies

Les deux derniers points impliquent une notion de variance. La politique adoptée est souvent contravariante : le type des arguments est généralisé dans la sous-classe et la pré-condition est affaiblie; le type du résultat est spécialisé dans la sous classe et la post-condition est renforcée.

• Abstraction : séparation effective entre les niveaux d'abstraction, plusieurs langages, séparation entre spécification et conception détaillée, raisonnement à des niveaux différents.

• Raffinement : moyens offerts pour passer d'un niveau d'abstraction à un autre. Des travaux existent tant au niveau des langages inspirés de Z ou VDM que des fonctions d'abstraction pour les spécifications algébriques.

Langage	**typage**	**abstraction**	**raffinement**
Object-Z	règles de Z	non	modification de la structure / théorie de l'équivalence dynamique
OOZE	sous-typage des méthodes pour les classes inclusion ensembliste pour les types de données	non	relation par prédicat et vues
VDM++	sous-typage basé sur l'héritage de représentation	non	théorie de VDM
OSDL	variables, processus et blocs sont typés	structurel / processus	séparation interface/réalisation
OS/OP	sous-typage : inclusion d'ensembles support héritage : règles et compatibilité de spécification	langage de spécification / langage de programmation	basé sur l'héritage (fonctions de transformation)
PLUSS	types abstraits uniquement	non	par spécification incomplète
COLD	pas de sous-typage	un seul langage pour algèbres et états	fonction d'abstraction
GSBL	pas de sous-typage voir héritage	non	par spécification incomplète
Z++	les classes définissent des types par l'ensemble de leurs intances	non	relation par prédicat / passage vers Uniform (sorte de Pascal), Prolog
MooZ	types sont des ensembles de valeurs pas de sous-typage de comportement	non	Z
Fresco	basé sur les classes	inférence sur les pre-/post-conditions / exécution explicite	raffinement de modèles par théorème et décom-position d'opérations

Figure 27 : Synthèse : Techniques formelles de la méthode 2/2

3.3.5 Méthode de développement

- processus : L'approche est-elle ascendante, descendante ou mixte. Une démarche précise doit guider le spécifieur. Les opérations sont spécifiées par disjonction ou par promotion i.e. délégation à une opération d'un des composants.

- environnement : description de l'environnement supportant le langage .

- outils : outils d'écriture, de contrôle, de preuve, de gestion de classes.

- documentation* : description.

- utilisation* : projets dans lesquels la méthode a été utilisée.

Langage	**processus**	**environ.**	**outils**
Object-Z	approche ascendante	en cours	éditeurs, prouveurs
OOZE	approche mixte	système FOOPS	vérification syntaxique et de types Interpréteur, Base de données Assistance à la preuve
VDM++	approche ascendante	surcouche de Mural	éditeur traducteur en VDM
OSDL	emboitement descendant	SDL Tool	éditeur graphique flâneur de structure analyseur
OS/OP	compatibilité des relations sous-typage/ clientèle/héritage	en cours	éditeurs, preuve de conformité prototypage, stratégies de preuve traducteurs OP - languages objets
PLUSS	mixte	Asspro	éditeurs, aide à la preuve prototypage, génération de code
COLD	ascendante	(indus- triel)	librairie de spécifications éditeurs, bases de connaissance générateur de documents contrôle et preuve, etc.
GSBL	mixte	–	–
Z++	approche plutot descendante une réflexion existe sur le processus	outil B	éditeur interactif pretty printer (C++, Z-style) raffineur
MooZ	approche ascendante	ForMooZ (Smalltalk) en cours	langage d'interrogation de spéci- fications, flâneur graphiques générateur de code Smalltalk générateur de documents LATEX aide à la preuve interactive
Fresco	définition des capsules raffinement de l'implantation ation	–	interpréteur Fresco/Smalltalk flâneur graphique proveur Mural en Smalltalk

Figure 28 : Synthèse : Méthodes de développement

3.4 Bilan

Actuellement les méthodes formelles à objets sont soit des langages de programmation à objets de haut niveau soit des langages de spécification formelle traditionnels étendus aux objets. Ces derniers se classent principalement en deux pôles, les méthodes par modèle abstrait et les méthodes algébriques.

Les méthodes par modèle ont pour inconvénient une surspécification, même si elle est abstraite (en termes mathématiques) et la difficulté d'exhiber certaines propriétés des spécifications que sont la cohérence et la complétude. A priori, l'intérêt de ce style réside dans la facilité à structurer les systèmes à spécifier et aussi dans la possibilité de traduire assez simplement les modèles statiques des méthodes d'analyse et conception. Mais la tendance est de donner une version opérationnelle du système et le raffinement n'est pas trivial pour autant. Notons enfin un avantage indéniable : la plupart de ces langages ont été conçu en collaboration étroite université/industrie dans des projets Esprit (REDO, Afrodite, Procos, etc.).

Les méthodes "plutôt" algébriques correspondent directement à une abstraction des concepts à objets, bien que pour Cook, les types abstraits de données sont des ensembles avec des opérations alors que les objets sont des ensembles d'opérations [BP94]. La séparation entre interface et implantation apparait plus clairement, ce qui facilite la preuve de propriétés abstraites et la validation. Un des problèmes majeurs est l'expression de la communication et de la concurrence dans des systèmes à objets coopérants.

Quelque soit le style choisi, un certain nombre de difficultés persistent :

- la persistance des objets par un identifiant unique reste un choix difficile: facilité de spécification et difficulté de preuve.
- expression du contrôle (de la colle) entre les objets : centralisé ou non, explicite ou non.
- cohésion entre les différents modèles formels exprimant des aspects différents du modèle à objets (structure, comportement, cycle de vie, communication).
- manque de guide et de contrôle dans les méthodes. Le raffinement reste une opération délicate, la différence entre spécification et implantation doit être bien marquée.
- difficulté d'écriture et de preuve. Les formalismes utilisés nécessitent un long temps d'apprentissage, les spécifications sont parfois complexes à aborder. Un formalisme simple et expressif facilite les preuves de la spécification. L'examen ses preuves, lorsqu'elles sont données dans la littérature, montre le besoin d'automatisation mais aussi de documentation de ces preuves. Le degré d'abstraction est aussi un critère important pour qualifier les spécifications, mais il est difficile à mesurer.
- difficultés dans la présentation de systèmes complexes. Il faut des outils de structuration en modules, de gestion des classes, de méthode logicielle (que réutiliser, comment le faire).
- spécification globale de systèmes : l'aspect spécification de composants est de plus en plus maîtrisé, même si certains points sont encore à améliorer. Par contre, l'expression d'un ensemble d'objets communicants reste une gageure dans les modèles actuels. Si l'expression des contraintes temporelles ou de synchronisation et communication (en logique temporelle ou en CSP par exemple) est connue, les liens avec les objets de la spécification ne sont pas bien maîtrisés.
- la sémantique des modèles proposés n'est pas toujours facile à comprendre. Les règles à appliquer pour respecter des propriétés telles que le sous-typage ou l'héritage Le vocabulaire varie d'un auteur à l'autre. Il y a un besoin de normalisation.
- Un dernier besoin se fait sentir, l'interopérabilité entre les langages de spécification.

Quelque soit la méthode de spécification choisie, une méthode d'analyse telle que OMT ou OOAS est nécessaire à l'identification des objets du domaine d'intérêt. Le formalisme de spécification doit permettre de traiter les trois plans (structure/fonction/comportement dynamique) des objets, mais à des niveaux d'abstraction suffisants (pas besoin de voir la structure d'un objet, ou à qui sont envoyés les méthodes). Une approche ascendante-descendante est nécessaire dans l'identification/réalisation des composants: concevoir en réutilisant et concevoir pour réutiliser.

La proposition :
Types Abstraits Graphiques et Classes Formelles

Chapitre 4

Introduction au modèle TAG/CF

"On a pu apprendre à écrire à des chiens et à des singes mais jamais à des oiseaux, et pourtant ce ne sont pas les plumes qui leur manquent."

Henri Monnier.

4.1 Introduction

Nous l'avons vu dans les chapitres précédents, deux courants s'imposent dans le développement de logiciel de qualité : l'approche à objets et les méthodes formelles. Les méthodes à objets sont destinées à maîtriser la complexité des logiciels en le partitionnant en unité de stockage et de traitement de l'information, logiquement indépendantes : les objets. Les méthodes formelles ont pour objectif la validation et l'automatisation du développement.

Ces deux courants sont complémentaires. Le contrôle du logiciel et la répartition du travail sont plus faciles sur des composants de taille raisonnable et (relativement) indépendants. La formalisation des objets permet de mieux les comprendre, d'expliciter leur responsabilité et de connaître leurs conditions d'utilisation. Cette formalisation est aussi nécessaire à la définition même des concepts du modèle à objets et au contrôle du logiciel final (raffinement, tests, prototypage rapide). Enfin, elle sert de référence dans un domaine où des méthodes de toutes sortes prolifèrent.

Nous avons vu que le mariage entre méthodes formelles et méthodes à objets pouvait se faire sur différents plans :

1. intégration de formalisme dans les modèles des méthodes à objets existantes :
   (a) soit directement par formalisation des modèles,
   (b) soit progressivement par changement de formalisme;
2. intégration de concepts objets dans les méthodes formelles existantes,
   (a) soit directement par ajout de concepts à objets dans les modèles formels,
   (b) soit progressivement pour la conception et non la spécification;
3. définition d'un langage de programmation à objets haut niveau pour la conception.

Actuellement, la plupart des méthodes formelles à objets sont du premier type (2.a), ainsi que le montre la synthèse du chapitre précédent. Les spécifications sont plus faciles à organiser, mais la notation formelle reste souvent difficilement accessible. C'est pourquoi, il semble intéressant de faire précéder le processus de spécification formelle d'une analyse à objets habituelle

(1.b). Cette approche semble convenir plutôt aux langages de spécification par modèle abstrait qu'aux langages de spécification algébrique puisqu'une structure des objets est déjà exhibée. L'approche (3) est adaptée aux petits projets et au prototypage rapide. Elle ne convient pas à la spécification abstraite de systèmes complexes. L'alternative (1.a) semble irréaliste à cause du nombre et de la variété des modèles de l'analyse à objets. Enfin, le choix (2.b) n'est avantageux que si le langage de spécification et le langage de conception sont sémantiquement proches.

Pour tirer au mieux profit des deux courants, il semble qu'une dernière alternative soit prometteuse. Le développement est vu comme une suite d'abstractions/constructions de composants et de systèmes. Chaque composant est défini par des notations plus ou moins formelles mais la validation se fait dans un modèle formel à objets. L'abstraction permet de décomposer et spécifier les différents éléments du système à analyser, concevoir ou réaliser. La spécification formelle de ces composants permet de mieux les appréhender pour la construction de nouveaux logiciels (base de la réutilisation). L'expression abstraite du comportement d'un composant (ses réactions aux messages, les résultats produits) est bien distincte de la façon dont il est implanté. Cependant, le lien entre les deux doit être clair et si possible bi-directionnel (abstraction de programmes). Le formalisme doit apparaître progressivement et naturellement pour ne pas rebuter les utilisateurs et il doit servir de support à la fois à la validation et à la construction du logiciel.

Nous nous plaçons dans le cadre des méthodes formelles à objets pour les raisons définies précédemment. Nous nous restreignons aux systèmes concurrents, dans lesquels l'aspect temps réel n'est pas pris en compte. Les phases d'analyse des besoins et de tests de programmes ne sont pas étudiées. Nous soutenons les thèses suivantes :

- *Une méthode de développement doit séparer clairement trois activités : la spécification et la conception et l'implantation.* La spécification est une modélisation du problème, qui se focalise sur les interfaces des objets et leurs conditions d'utilisation. La conception est une architecture de programme en termes d'objets, indépendante des langages de programmation. L'implantation est un programme décrit dans un langage à objets séquentiel. L'approche choisie est donc celle de l'intégration progressive du formalisme avec raffinements successifs (1.b) cumulée avec l'approche (3) pour la définition d'un langage de conception. Le mot-clé ici est l'abstraction. Le modèle à objets est perçu à des niveaux d'abstraction variant selon l'activité considérée. Appréhender le raffinement comme un changement de modèle, plutôt qu'un enrichissement de certaines caractéristiques, limite le risque de sur-spécification. Il ne s'agit pas seulement d'ajouter de nouveaux composants mais d'enrichir les descriptions avec de nouveaux concepts. La clé de l'abstraction est de savoir classer ces concepts. La communication entre les acteurs du développement est améliorée s'ils ne sont pas inondés de détails inutiles. L'abstraction est aussi une des raisons pour lesquelles les spécifications algébriques seront utilisées de préférence aux autres styles de spécification.

- *A un niveau d'abstraction donné, le nombre de modèles produits doit être limité.* Les méthodes d'analyse et de conception multiplient les modèles de représentation (voir section 2.3.2). Ceci est dû à la difficulté de comprendre un système dans son ensemble. Il en résulte des problèmes de vérification de la cohérence entre ces modèles et de raffinement vers les modèles plus concrets. Un modèle unique résoud le problème de la cohérence, mais il est difficilement complet. Au niveau le plus abstrait, nous désirons faire le lien entre le modèle dynamique et le modèle fonctionnel. La structure des objets doit rester informative. Une bonne spécification abstraite d'un composant débute par l'élaboration d'une description du comportement dynamique. Un langage de conception abstraite pour les classes unifie les aspects fonctionnels et structurels indépendamment des langages de programmation.

- *Une méthode formelle doit introduire progressivement le formalisme.* La spécification de départ doit être comprise par tous les acteurs du développement. Une analyse semiformelle dégrossit la compréhension et favorise le découpage modulaire intuitif. L'utilisation de notations graphiques dans l'écriture des spécifications est souhaitable, de même que la traduction informelle des spécifications et de leurs propriétés. Cette description graphique doit cependant être suffisamment détaillée pour servir ensuite à la construction assistée des composants. La formalisation favorise la découverte des erreurs, car la description est de plus en plus précise.

- *Une méthode formelle doit guider le concepteur.* La démarche doit être simple et illustrée, surtout pour un cycle itératif. Ce point requiert de l'expérience et l'application de la méthode à de nombreux cas pratiques. Une assistance dans la construction et le raffinement de spécifications évitera trop de tâtonnements. Des algorithmes de traduction et des conseils dans les choix accélèrent le processus. Une bonne méthode devra proposer un catalogue de preuves et de propriétés à démontrer. Une méthode est indissociable d'un environnement complet. Outre les outils habituels de spécification, l'environnement support devra prendre en compte la gestion et la réutilisation de composants réutilisables.
- *Une méthode s'adapte en souplesse aux problèmes à résoudre et aux participants.* Les modèles proposés peuvent être utilisés indépendamment les uns des autres (écriture, tests et validation, prototypage). Chaque modèle possède un noyau de base qui peut être étendu à des caractéristiques plus complexes.
- *Un environnement de développement réaliste n'est pas autonome.* Il faut fournir des passerelles vers d'autres environnements de développement afin de réutiliser leurs outils pour des parties spécifiques (preuve, test ou prototypage par exemple). Le support de la méthode doit lui-même être modulaire.

Ce chapitre est un rapide tour d'horizon de notre approche (voir aussi [AR94a, AR95, ABR95]). Nous rappelons brièvement les concepts et les objectifs avant de voir les trois niveaux de description sur l'exemple de l'hôpital, les phases de validation et de concrétisation des spécifications.

4.2 Présentation des concepts

La méthode Type Abstrait Graphique/Classe Formelle est le support des idées énoncées ci-dessus. D'autres intérêts en sont attendus : clarification des concepts du modèle objets, proposition de techniques adaptées à d'autres problèmes (assistance à l'écriture de spécifications algébriques, calcul d'une conception ordonnée à partir d'une conception plate) intégration des modèles à des méthodes de développement du commerce.

Nous proposons une approche dans laquelle les composants logiciels sont exprimés à trois niveaux d'abstraction : types abstraits graphiques (TAG en abrégé), classes formelles (CF), langages de programmation. Le TAG est d'abord un modèle dynamique du composant mais également un modèle fonctionnel exprimé par une spécification algébrique de type abstrait de données. Nous voulons nous dégager du niveau trop concret des langages à objets. Pour cela nous avons défini un langage de conception pour les classes, appelé le modèle des classes formelles [ACR94]. Ce langage est d'une implantation aisée dans un langage à objets avec classes. A chaque niveau, la notation est formelle et une validation est possible. Une démarche pour le passage d'un niveau d'expression abstrait à un niveau plus concret est présentée.

Les niveaux d'abstraction n'expriment pas une dichotomie entre analyse, conception et réalisation mais s'interprètent plutôt comme une présentation distincte de raffinement en trois étapes. En effet, les principes présentés ici servent à la fois à la spécification/prototypage des besoins et à la description/implantation du logiciel.

L'organisation générale des concepts est donnée dans la figure 29. Les arcs pleins correspondent aux transformations étudiées dans ce document, les autres sont en cours d'étude. Notez l'ouverture possible vers d'autres environnements et notamment la liaison avec le système ASSPEGIQUE+ tant pour les TAG que les classes formelles. Le TAG est le niveau le plus abstrait, il est le résultat d'une analyse[1] et d'une formalisation préliminaire et incomplète. La classe formelle décrit une conception à objets et abstraite du TAG. La classe concrète décrit une mise en œuvre dans un langage à classes. Nous illustrerons ces concepts par la spécification et la conception d'un système simplifié d'entrées dans un service hospitalier.

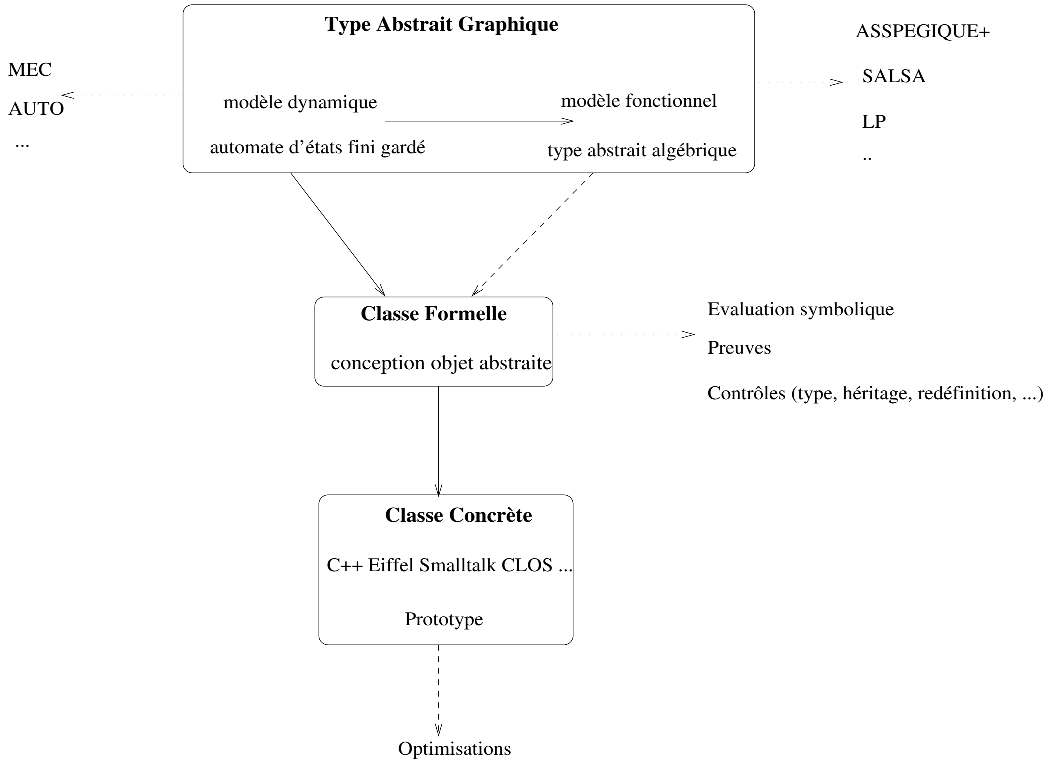


Figure 29 : Concepts et organisation générale

4.3 Introduction aux Types Abstraits Graphiques

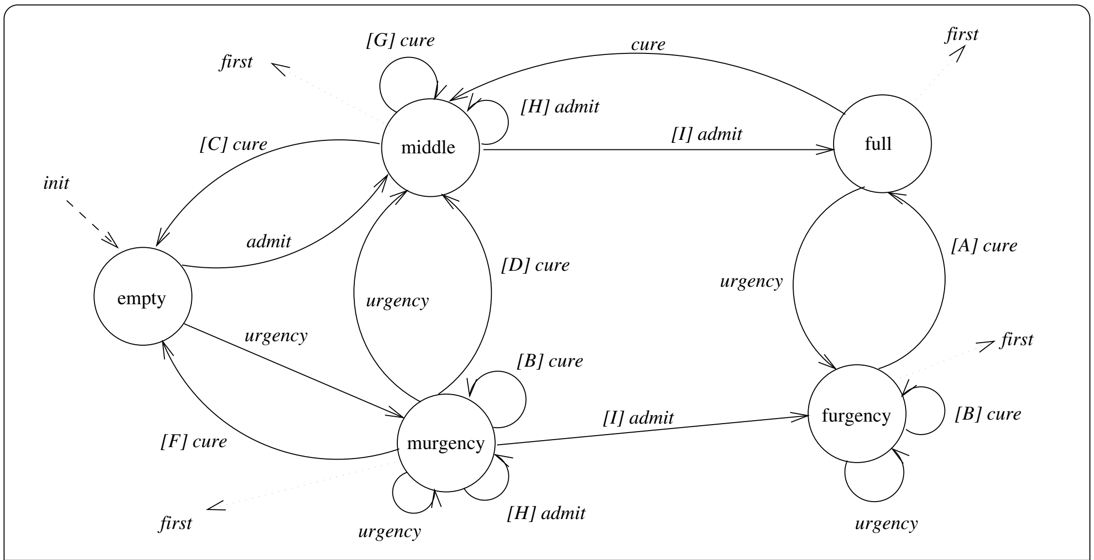
Trois objectifs majeurs ont conduit au modèle TAG : unifier la vue dynamique d'un composant (un automate gardé) et sa vue fonctionnelle (spécification algébrique), introduire pas à pas la formalisation (faciliter l'écriture d'une spécification algébrique) et permettre des preuves locales et globales au système. Ce modèle est présenté en détail dans le chapitre 5.

4.3.1 Définir le modèle dynamique

La spécification débute par une description du modèle dynamique[2] sous forme d'un automate gardé. D'autres formalismes décrivent des systèmes de transition : Réseau de Petri, logiques temporelles, graphes d'événements ou d'état, machines séquentielles... Certains d'entre eux sont plus abstraits que les automates mais nous avons choisi ce formalisme parce qu'il est visuel, modulaire, bien défini (voir [Arn92]), qu'il existe des systèmes de preuve [LMV87, Gri89], et surtout pour sa double interprétation possible. Cette double interprétation est d'une part un système d'états fini et d'autre part un type de données. Un état correspond à une étape dans le cycle de vie et à un sous-type du type de données. La notion d'état est une abstraction de la structure des objets. Une transition correspond à une interruption par événement et à un appel d'opération. Les constructeurs (resp. observateurs) sont des opérations qui changent (resp. observent) l'état.

[1]Nous proposons une transformation des modèles dynamiques d'OMT en TAG dans [ABR95].
[2]En fait, le modèle du comportement dynamique

Le premier travail de spécification est de définir les états du système, les transitions entre ces états, les gardes associées et des informations diverses (section 5.2). Des hypothèses naturelles sur les TAG sont définies dans la section 5.3 : connexité, gardes exclusives, états initiaux, états non vides et finiment engendrés, pas de concurrence intra-objet. Dès ce stade, il est souhaitable de prouver certaines propriétés de l'automate telles que la vivacité, la terminaison ou l'interblocage. Un interfaçage avec des systèmes comme Auto ou Mec est envisagé. Le problème de l'explosion combinatoire du nombre d'états et de la structuration des automates complexes est étudié dans la section 5.6.1.



Operations :	Guards :	Type : Hospital
init (max: Integer) : Hospital	*A: urgencyNumber(Self) = 1*	Description : queues managing
admit (Self: Hospital, p: Patient) : Hospital	*B: urgencyNumber(Self) > 1*	Author : André - Royer
urgency (Self: Hospital, p: Patient) : Hospital	*C: number(Self) = 1*	June 1994
cure (Self: Hospital) : Hospital	*G: number(Self) > 1*	Release : 1
size (Self: Hospital) : Integer	*D: A AND number(Self) > 0*	Properties :
number (Self: Hospital) : Integer	*F: A AND number(Self) = 0*	Constants:
urgencyNumber (Self: Hospital) : Integer	*H: number(Self) > size(Self) - 1*	formal parameters:
first (Self: Hospital) : Patient	*I: number(Self) = size(self) - 1*	Contraint
		0 <= number(Self) <= size(Self)

Figure 30 : L'automate du TAG de `Hospital`

4.3.2 Construire le modèle fonctionnel

La seconde phase d'une spécification TAG va être de produire le modèle fonctionnel sous forme d'un **Type Abstrait Algébrique** (TAA). Le système exploite les informations de l'automate pour aider l'utilisateur dans sa tâche. L'idée est de considérer une transition comme une opération partielle qui transforme une valeur de l'état de départ en une valeur de l'état d'arrivée. Les algorithmes de la section 5.4 permettent d'obtenir automatiquement l'extraction de certaines informations. En particulier, l'extraction de la signature est triviale. La construction de l'axiomatique se fait ensuite en trois étapes.

Premièrement, les opérations sont classées en opérations primitives et opérations secondaires. Cette partition permet, par induction, de réduire le nombre d'axiomes de la spécification. Un parcours d'automate, décrit dans la section 5.4.4, permet de calculer une famille de constructeurs primitifs appelés générateurs. L'extraction des générateurs est fondée sur l'idée que l'on doit pouvoir atteindre chaque état. Nous obtenons ici deux familles: {`init, admit, cure, urgency`} et {`init, admit, urgency`}. Pour la détermination des observateurs primitifs nous ne disposons pas de principe aussi agréable mais de quelques heuristiques. Nous choisissons ici `size, number, urgencyNumber, first`.

Ensuite, les prédicats d'états et les préconditions des opérations sont exprimés en fonction des générateurs. La précondition d'une opération est une union des prédicats d'état et des gardes correspondantes. Les expressions ont été volontairement simplifiées.

```
middle1: middle(init(Xmax)) == false
middle2: middle(admit(Self, Xp)) == empty(Self) OR [middle(Self) AND H(Self)]
middle3: middle(urgency(Self, Xp)) == false.

precondition(admit(Self,Xp)) = empty(Self) OR middle(Self) OR murgency(Self)
```

La dernière étape est l'extraction des axiomes. Nous avons défini dans la section 5.4.5 des algorithmes pour produire un ensemble d'axiomes, en interaction de l'utilisateur. Nous nous sommes basés sur les méthodes usuelles de spécification de types abstraits de données [GH78, Mus80, LG90, Bid82, BB92]. Le principe de base de la génération des axiomes, appelé G-dérivation est défini dans la section 5.4.5. Une G-dérivation d'un terme `Self` dans un état `E` est sa substitution par un générateur arrivant dans cet état. Connaître l'effet d'une opération sur ce terme `Self`, c'est donc analyser toutes les transitions qui mènent à cet état. Prenons l'exemple de l'observateur `first` :

```
first1: empty(Self) == true ==> first(admit(Self, Xp)) == Xp
first2: middle(Self) == true & H(Self) == true ==>
                       first(admit(Self, Xp)) == first(Self)
first3: middle(Self) == true & I(Self) == true ==>
                       first(admit(Self, Xp)) == first(Self)
first4: full(Self) == true ==> first(urgency(Self, Xp)) == Xp
first5: furgency(Self) == true ==> first(urgency(Self, Xp)) == first(Self)
first6: murgency(Self) == true & I(Self) == true ==>
                       first(admit(Self, Xp)) == first(Self)
first7: empty(Self) == true ==> first(urgency(Self, Xp)) == Xp
first8: middle(Self) == true ==> first(urgency(Self, Xp)) == Xp
first9: murgency(Self) == true ==> first(urgency(Self, Xp)) == first(Self)
first10: murgency(Self) == true & H(Self) == true ==>
                       first(admit(Self, Xp)) == first(Self)
```

Notez que les axiomes `first3, first6 et first10` peuvent être groupés. Les propriétés d'une spécification algébrique de TAG sont étudiées dans la section 5.5.2. La démonstration de ces propriétés et la validation de la spécification sont faites via des interfaces avec d'autres environnements de spécification.

4.4 Interface avec ASSPEGIQUE+

Nous n'avons pas la prétention de fabriquer tous les outils nécessaires à un bon environnement de développement. L'utilisation d'un système de manipulation algébrique nous a semblé le plus utile dans un premier temps. Nous avons étudié l'interface avec le système ASSPEGIQUE+ [Cho88, BCC90, Roq92]. Les concepts sont similaires mais la syntaxe varie un peu. L'interface consiste à engendrer un fichier avec quelques modifications syntaxiques et à l'intégrer sous ASSPEGIQUE+. Une spécification ASSPEGIQUE+ de l'hôpital est donnée en annexe A.6. Notons que les hypothèses des TAG sont sur certains aspects plus faibles que la contrainte hiérarchique des TAA. Elles sont plus proche de celles que pose David Musser [Mus80] et sont plus adaptées au contexte objet. Cette contrainte nécessite quelques aménagements dans le graphe d'utilisation. Par exemple, il faut ajouter un type de donnée pour établir le lien entre deux types s'utilisant mutuellement. Nous avons privilégié ASSPEGIQUE+ pour des raisons circonstancielles, tout autre environnement de ce type est a priori possible par l'intermédiaire des FISCs du projet SALSA [BBC+92].

4.5 Le modèle des classes formelles

Le modèle des classes formelles (CF en abrégé) est issu des travaux de Jean-Claude Royer [Roy92, AR92c, Roy93]. C'est une sorte de langage de spécification de classes, qui est intermédiaire entre les TAA et les classes concrètes. Une CF est une spécification particulière de TAA. Le modèle doit permettre une transition plus facile des TAA vers les classes mais également de définir des règles d'écriture et des outils inspirés des spécifications algébriques. Une sémantique algébrique complète est définie pour la notion de classe. Une sémantique opérationnelle abstraite prend en compte l'héritage. L'évaluation symbolique et la preuve de propriétés se font par un mécanisme simple et naturel de réécriture. Ces divers aspects du modèle sont décrits dans le chapitre 6. Une CF est un type et l'héritage implique le sous-typage, un contrôle de type simple mais sûr est décrit dans la section 6.5.

La conception d'une CF est inspirée de la programmation à objets (noyau/extension). Une CF est définie par un aspect et des méthodes secondaires. L'aspect est l'ensemble suffisant des méthodes (primitives) qui caractérisent un objet : création, copie, égalité, accesseurs. Les méthodes dites secondaires sont des extensions fonctionnelles de ce noyau.

```
                                   Hospital
                           inherits from OBJECT
comments: class for hospital
features: init, admit, urgency, cure, number, urgencyNumber, first
param: Patient
from: Integer import: +, 0, 1, =, >, <=
from: Boolean import:
                              aspect : hospital
            field selectors                        constraint
size          : Hospital  —→ Integer
queue         : Hospital  —→ List[Patient]      size(Self) > 1 and
urgencyQueue  : Hospital  —→ List[Patient]   number(Self) <= size(Self)
                            secondary methods
;; isEmpty : the queues are empty
   isEmpty : Hospital  —→  Integer
isEmpty(Self) == empty?(queue(Self)) and empty?(urgencyQueue(Self))
;; number : number of admitted patient
   number : Hospital  —→  Integer
number(Self) == length(queue(Self))
;; isFull : the queue is full
   isFull : Hospital  —→  Integer
isFull(Self) == number(Self) = size(Self)
;; urgencyNumber : number of urgency patient
   urgencyNumber : Hospital  —→  Integer
urgencyNumber(Self) == length(urgencyQueue(Self))
;; isUrgent : the number of urgency patient is not null
   isUrgent : Hospital  —→  Boolean
isUrgent(Self) == urgencyNumber(Self) > 0
;; first : patient to be examined
   first : Hospital  —→  Patient
requires: isEmpty(Self) == false
isUrgent(Self) == true ==> first(Self) == head(urgencyQueue(Self))
isUrgent(Self) == false ==> first(Self) == head(queue(Self))
;; admit : admit a new normal patient
   admit : Hospital Patient  —→  Hospital
var: p:Patient
requires: isFull(Self) == false
admit(Self, p) == copy(Self, queue = cons(queue(Self), p))
;; cure : remove first patient
   cure : Hospital  —→  Hospital
requires: isEmpty(Self) == false
isUrgent(Self) == true ==> cure(Self) == copy(Self, urgencyQueue = tail(urgencyQueue(Self)))
isUrgent(Self) == false ==> cure(Self) == copy(Self, queue = tail(queue(Self)))
```

```
                              Hospital
;; urgency : admit a new urgency patient
   urgency : Hospital Patient  −→  Hospital
var: p:Patient
urgency(Self, p) == copy(Self, urgencyQueue = cons(urgencyQueue(Self), p))
                              class methods
;; init : open the hospital
   init : Integer  −→  Hospital
var: max:Integer
init(Xmax) == new(Hospital, size = max, queue = new(EmptyList),
                  urgencyQueue = new(EmptyList))
```

Figure 31 : Conception plate et intuitive de la classe formelle `Hospital`

D'une manière générale, la conception d'une telle classe se focalise d'abord sur la description des objets (l'aspect) puis sur leur utilisation (les méthodes secondaires). Cette pratique est conforme à l'usage de la programmation à objets. Elle a plusieurs avantages : d'une part, c'est une spécification algébrique particulière qui sous certaines hypothèses est suffisamment complète et non-contradictoire et d'autre part, elle vérifie la propriété naturelle en programmation à objets que l'ajout d'une nouvelle méthode secondaire ne change pas le type des instances. Cette classe a été conçue intuitivement. Nous verrons dans la section suivante une démarche pour construire des classes formelles à partir d'un TAG.

Si globalement une classe formelle est proche d'une classe Eiffel plusieurs originalités existent : la proximité avec les spécifications algébriques et tous les outils ou techniques qui s'en inspirent, les préconditions sur les sélecteurs de champs, qui sans accroître la puissance d'expression du langage permettent plus de souplesse dans la définition des classes.

4.6 Conception d'un TAG en CF

Le problème du raffinement d'un TAG en CF est un problème classique de représentation d'un type de données. Une conception est plate lorsqu'un type de données est défini par une seule spécification. Une conception est ordonnée lorsqu'un type de données est défini par une hiérarchie de spécification basée sur l'héritage et appelée **schéma** dans le modèle des classes formelles. Les avantages d'une conception ordonnée sur une conception plate sont : plus grande finesse du typage, meilleure réutilisabilité, meilleure cohérence des classes, diminution de la complexité de chaque classe.

La conception peut se faire à partir de la partie dynamique ou de la partie fonctionnelle. Il nous semble que la première est plus facile pour un spécialiste de la programmation à objets. Une façon plus rigoureuse d'opérer est de transformer la spécification algébrique en classes formelles. Cette alternative aurait l'avantage de permettre la preuve de la représentation mais une conception ordonnée est difficile. L'approche que nous avons proposée définit des outils communs aux deux démarches et permet d'expérimenter les différents choix et de mesurer leur influence au niveau de la réutilisation et de la structuration.

Le processus de représentation décrit dans le chapitre 7 n'est évidemment pas complètement automatique mais certaines parties le sont. Des choix sont faits par le concepteur au niveau de la sous-hiérarchie d'héritage (le schéma), du placement des opérations, des structures abstraites choisies, etc. Ces différents choix sont influencés par des critères de réutilisabilité, de simplicité, de taille des axiomes, etc. Les principales phases sont :

- Simplification de l'automate par regroupement d'états connexes. Chaque état représente un sous-type.
- Calcul des profils exacts des générateurs en fonction des regroupements d'états (union de types).
- Obtention d'un schéma d'utilisation et d'un schéma de création (regroupement sur le receveur ou sur le résultat).

- Calcul des aspects des classes avec le schéma de création. Un calcul semi-automatique est possible à partir des générateurs mais des simplifications sont nécessaires. Elles sont prouvables en utilisant un schéma inspiré de la représentation des types de données [Hoa72].
- Restructuration de chaque aspect et du schéma d'utilisation.
- Ecriture des axiomes des méthodes secondaires de chaque classe en fonction de l'aspect précédemment défini.

Le résultat obtenu avec notre exemple est une hiérarchie à trois niveaux et cinq classes terminales. La restructuration du graphe est possible en une unique classe `HospitalBis`.

HospitalBis	
inherits from OBJECT	
aspect : hospital	
field selectors	**constraint**
empty? : HospitalBis ⟶ Boolean	
normal? : HospitalBis ⟶ Boolean **requires:** empty?(Self) == false	
entry : HospitalBis ⟶ Patient **requires:** empty?(Self) == false	
oldState : HospitalBis ⟶ Hospital **requires:** empty?(Self) == false	

Cette représentation est équivalente à celle définie de la figure 31 mais moins naturelle et moins réutilisable. Dans une étude de cas plus complexe comme l'exemple de l'ascenseur de l'annexe C, l'intuition seule ne suffirait pas à obtenir un tel résultat. Le processus que nous proposons est méthodique mais il nécessite également de l'expérience dans les choix de conception comme le montrent les exemples de la section 7.4.1.

4.7 Prototypage et mise en œuvre dans les classes concrètes

Les phases précédentes nous fournissent une hiérarchie des classes et des opérations, ainsi qu'une description des classes formelles. Le codage dans les langage à classes est décrit dans la section 7.8. Les langages de programmation ont des caractéristiques propres qui posent quelques problèmes spécifiques mais en général assez simple à résoudre. Pour une traduction automatisée, on distingue deux étapes : la traduction directe et l'optimisation.

Traduction directe La traduction n'est en général pas complètement automatisable. Il est facile de produire la déclaration de la structure de la classe et de l'interface des méthodes. La production automatique du code des méthodes est possible si les axiomes sont orientables en règles de réécriture. Ceci est un schéma simple et évidemment incomplet que l'on doit adapter suivant le langage cible choisi. Le prototype obtenu est fonctionnel et en général assez inefficace.

Optimisations Deux types d'optimisations sont pratiqués jusqu'à obtenir un produit plus utilisable : ceux qui visent à changer la structure des classes ou les liens entre classes et ceux qui dépendent du langage cible. Les premières (introduction d'effets de bord, dérécursivation, économie de structure) seront effectuées avec profit sur la conception abstraite en classes formelles. Les secondes seront réalisées directement sur le prototype.

4.8 Conclusion

Dans ce chapitre, nous avons exploré quelques concepts utiles à une conception rigoureuse en programmation à objets. Une idée majeure est d'utiliser des concepts avec une double interprétation (TAG = automate/TAA et CF = TAA/classe) ce qui permet de faciliter les liens entre les niveaux de description (dynamique/fonctionnel et spécification/conception). Résumons les particularités de notre approche :

- Les composants logiciels sont des objets décrits selon trois niveaux (type abstrait, classe formelle, classe concrète) pour construire progressivement et naturellement les composants logiciels.
- Le modèle TAG associe une approche description dynamique des objets à une approche description fonctionnelle d'un type de donnée.
- Le modèle CF propose une approche algébrique de la programmation à objets pour une spécification et une conception de hauts niveaux.
- Chaque langage est formel et met en œuvre des mécanismes de contrôle et de preuve, indispensable à la qualité des spécifications.
- Il y indépendance complète entre les langages, ce qui permet de les utiliser dans des cadres différents et assouplit l'usage de la méthode. Toutefois, nombre d'outils communs sont développés.
- Le passage d'un niveau d'abstraction à l'autre est guidé par une démarche, partiellement automatisée, qui apporte de la rigueur dans le processus de développement.
- Nous utilisons d'autres environnements de spécification pour supporter les preuves de théorèmes et la validation.

Dans les chapitres suivants, nous détaillons les deux modèles introduits et le processus de conception des types abstraits graphiques en classes formelles. Notre but n'est pas de définir une nouvelle méthode d'analyse et de conception mais des concepts et des techniques qui peuvent s'intégrer dans une méthode de développement à objets.

Chapitre 5

Types abstraits graphiques

"Aimez les choses à double sens, mais assurez-vous qu'elles aient un sens."
Sacha Guitry.

5.1 Introduction

Le modèle **TAG**[1] est un formalisme abstrait de description à objets basé sur les automates et les spécifications algébriques. Le comportement dynamique est la vision la plus abstraite des objets. Cette vision est ensuite affinée par une spécification algébrique. La liaison entre modèle fonctionnel et modèle dynamique est réalisée par une double interprétation du modèle TAG : un TAG décrit le cycle de vie d'un objet et un TAG est un type abstrait de donnée.

Le présent chapitre est organisé comme suit. Dans la section 5.2, nous verrons les caractéristiques principales d'une spécification graphique de TAG. Cette spécification dépend d'un certain nombre d'hypothèses énoncées dans la section 5.3. Ensuite nous verrons comment extraire différents éléments d'une spécification algébrique à partir de cette spécification graphique du TAG dans la section 5.4. Ces deux descriptions forment le noyau du modèle. Dans la section 5.5, nous examinons quelques points sur le contrôle et la validation des spécifications. Un exemple complet est donné en annexe C. Enfin, des extensions du noyau sont proposées en section 5.6, notamment l'héritage.

5.2 Spécification graphique de TAG

Cette section donne la syntaxe de description des types abstraits graphiques. La notation est synthétisée dans la figure 80 de l'annexe E.

5.2.1 Représentation graphique

La représentation graphique est composée de quatre encarts : une entête comprenant notamment le nom du type en cours de définition, appelé **type d'intérêt**, un ensemble de profils d'opérations, des déclarations de paramétrage, un automate du comportement dynamique. Un certain nombre d'informations sont ajoutées à la description (dates, notes) elles servent à commenter la description graphique pour un archivage de la documentation. Chaque propriété est

[1] **T**ype **A**bstrait de Données **G**raphique

nommée, et donnera lieu à la génération d'un ou plusieurs axiomes. Par exemple, modélisons un compte bancaire simple par le TAG suivant :

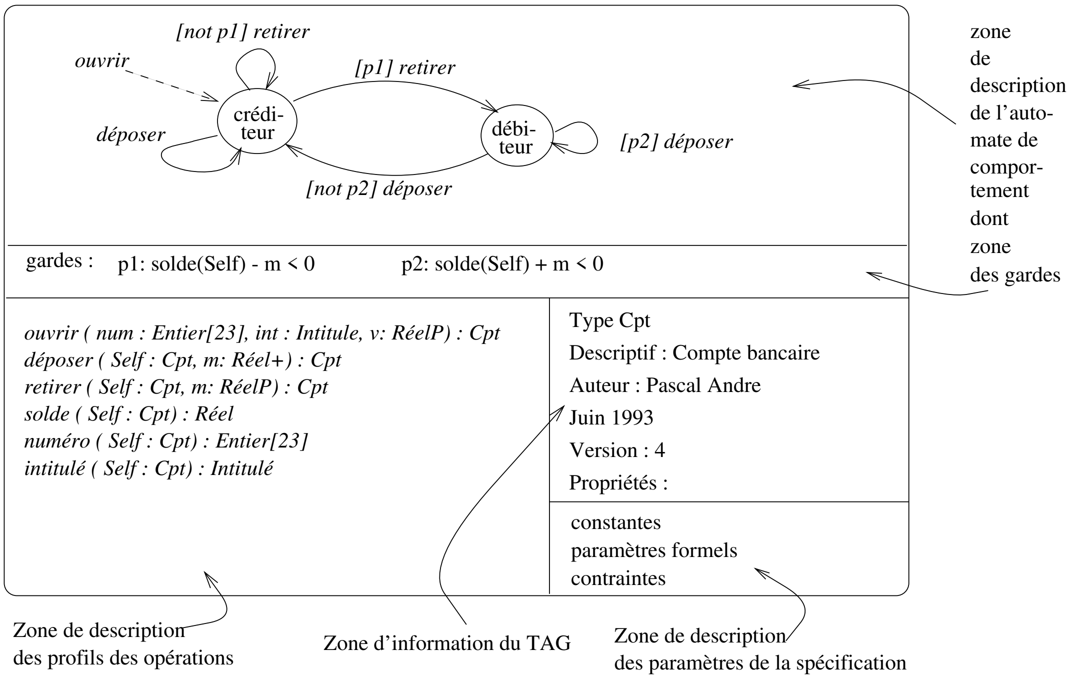


Figure 32 : TAG Compte bancaire.

5.2.2 Opérations

Une **opération** est une abstraction procédurale. Elle est définie d'un point de vue syntaxique par un profil.

Définition 5.2.1 (profil d'opération)
Un ***profil d'opération*** *est un triplet* <*nom, paramètre, résultat*>*, où paramètre est une séquence de couples* <*variable, type*>*. et résultat est un type. Un profil d'opération est noté :*

$$\texttt{oper}\ (p_1 : \mathit{Type}_1,\ \ldots\ p_n : \mathit{Type}_n) : \mathit{Type}_{res} \quad .$$

Le **domaine** (resp. **codomaine**) d'un profil d'opération est le produit cartésien des types du paramètre (resp. le type résultat) de ce profil d'opération. Tout profil d'opération contient au moins une fois le type d'intérêt sinon l'opération associée relèverait d'une autre abstraction de donnée. Nous utilisons la convention syntaxique suivante: si le type d'intérêt appartient au domaine alors il se trouve en première position et le nom de la variable associée est **Self** (connotation d'objet receveur). Comme d'usage, les opérations sont classées en constructeurs et observateurs. Par extension et abus de langage, opération et profil d'opération seront confondus.

Définition 5.2.2 (constructeurs, observateurs)
Soit F un ensemble de profils d'opérations et un nom de type ***TI****. Soient Γ, Γ_b et φ trois sous-ensembles de F définis par :*
$\Gamma = \{po \in F \mid TI = codom(po)\}$ $\Gamma_b = \{po \in \Gamma \mid TI \notin dom(po)\}$ $\varphi = \{po \in F \mid TI \neq codom(po)\}$.
Les opérations respectives de Γ, Γ_b et φ sont appelées respectivement ***constructeur*** *,* ***constructeur de base*** *et* ***observateur*** *de F pour le type* ***TI****.*

Il est évident que $F = \Gamma \cup \varphi$ i.e. Γ et φ forment une partition. Dans l'exemple de la figure 32, les constructeurs sont les opérations nommées `ouvrir, retirer et déposer`. L'opération de nom `ouvrir` est le seul constructeur de base. Les observateurs sont les opérations de nom `solde, numéro et intitulé`. Les transitions associées aux opérations définies sur tous les états et qui ne changent pas d'état ne sont pas indiquées (`solde, numéro, intitulé`).

5.2.3 Paramètres et invariant d'une spécification

Un paramètre d'une spécification graphique et **générique** de TAG est : soit un **paramètre formel de type** (PFT), qui est en fait un nom de type [2], soit une constante , appelée **paramètre formel constant** (PFC). La valeur de ces paramètres est fixée par **actualisation**.

Une **contrainte**, ou **invariant** (INV), est un prédicat basé sur les observateurs, qui est vrai pour toutes les valeurs du type sur lequel il porte.

type Compte Bancaire			
paramètre	paramètre formel	paramètre effectif	variation
type	nom	type complet	sous-type
	Monnaie	Francs	MonnaieParitaire
constante	nom et type	valeur du type	sous-type de la constante
	plafond: $I\!R^+$	5000.0	plafond: 2000.0 .. 10000.0
contrainte	prédicat		renforcee
	solde(Self) <= plafond(Self) (A)		A ET 200 <= solde(Self)

5.2.4 Comportement dynamique

Le **comportement dynamique** définit visuellement l'enchaînement et les conditions d'application des opérations du type de données. Nous utilisons une forme particulière d'automates à états finis et gardés dont les concepts sont proches des types abstraits de données.

Définition 5.2.3 (automate de comportement dynamique)
*L'**automate de comportement dynamique** est un sextuplet $M = (F, K, \Delta, \delta, K_0, K_f)$ où F (l'alphabet d'entrée est l'ensemble des profils d'opérations) et K (l'ensemble des états) sont des ensembles finis non vides, Δ (l'ensemble des gardes) est un ensemble fini, éventuellement vide, $\delta\ : (K \cup \{\varepsilon\}) \times F \times \Delta \rightarrow (K \cup \{\varepsilon\})$ la fonction de transition gardée, $K_0 \subset K$ l'ensemble des états initiaux, $K_f \subset K$ l'ensemble des états finaux.*

L'ensemble des états de la fonction de transition contient ε, l'état indéfini, à cause des constructeurs de base et des observateurs partiels. La notion d'état final n'est pas indispensable. Une transition sera notée : $<k_{origine}, k_{destination}, garde, oper>$ ou encore $\delta(k_{origine}, garde, oper) = k_{destination}$. Un état correspond à un ensemble de valeurs et une transition à une invocation d'opération. A chaque type d'opération correspond un type d'arc : arc entre deux états pour les constructeurs, arc sans état destination pour les observateurs (vecteur en pointillé) et arc sans état origine pour les constructeurs de base (vecteur en ligne discontinue). Par exemple, la transition entre les états `débiteur` et `créditeur`, associée à l'opération `déposer`, signifie *"si l'argent déposé dépasse le découvert, alors le compte 'Self' redevient positif"*. Les états destination de transitions supportant des constructeurs de base sont initiaux e.g. l'état `créditeur`.

Une **garde** est un prédicat nommé associé à une transition. Elle est définie à partir des paramètres de l'opération correspondante et des observateurs du type d'intérêt accessibles dans l'état origine de la transition. Une transition est **passante** si la garde associée est instanciée à vrai. Les gardes sont utilisées pour rendre fini l'automate du comportement dynamique. La partie commune à toutes les gardes d'une opération donnée est factorisée dans un autre prédicat nommé. Supposons par exemple l'existence d'un montant maximal de retrait, comme pour une carte bancaire, alors la condition de retrait du type de la figure 32 devient :

p1(Self, m):	`m <= plafond(Self)`	// le montant est inférieur au plafond autorisé //
p2(Self, m):	`m <= solde(Self)`	// le montant est inférieur au solde //
p3(Self, m):	`p1(Self, m) AND p2(Self, m)`	// le montant est inférieur au plafond et au solde //

[2] Dans la plupart des langages de spécification permettant la généricité contrainte, ce type est défini par une spécification partielle.

5.2.5 Types et relations

Nous pouvons maintenant définir formellement un type abstrait graphique de données.

Définition 5.2.4 (TAG)
Un TAG est un sextuplet $< TI, F, PFT, PFC, INV, M >$ *où* TI *est un nom de type, appelé type d'intérêt,* F *est un ensemble de profils d'opérations,* PFT *est un ensemble de paramètres formels de types,* PFC *est un ensemble de paramètres formels constants,* INV *est un invariant et* $M = (F, K, \Delta, \delta, K_0, K_f)$ *est un automate de comportement dynamique.*

Soit $< TI, F, PFT, PFC, INV, M >$ un TAG. Des relations entre types sont définies à partir des profils d'opérations. Le cas des paramètres formels de type (ensemble $PFT(TI)$) est traité à part dans la section 5.6.3.

Définition 5.2.5 (relation d'importation)
La relation d'importation, appelée aussi relation d'utilisation, signifie que la définition d'un nouveau type de donnée dépend d'autres types. USE est en est la fermeture transitive.

- $use(TI) = \bigcup_{po \in F} \{T \mid T \in dom(po) \vee T = codom(po)\} - (PFT(TI) \cup \{TI\})$
- $USE(TI) = \bigcup_{T \in use(TI)} (\{T\} \cup USE(T))$

Définition 5.2.6 (relation de dépendance forte)
La relation de dépendance est la projection de la relation d'importation sur les constructeurs de base. DPF est en est la fermeture transitive.

- $dpf(TI) = \bigcup_{po \in \Gamma} \{T \mid T \in dom(po) \vee T = codom(po)\} - (PFT(TI) \cup \{TI\})$
- $DPF(TI) = \bigcup_{T \in dpf(TI)} (\{T\} \cup DPF(T))$

5.3 Hypothèses

Nous allons énoncer un certain nombre d'hypothèses , qui seront nécessaires à la démonstration de certaines propriétés de la spécification TAG. Soit $< TI, F, PFT, PFC, INV, M >$ un TAG avec $M = (F, K, \Delta, \delta, K_0, K_f)$. Les hypothèses suivantes sont posées :

- (*HYP1*) une opération est définie dans une seule signature :
$$\forall op \in F,\ \forall M' = (F', K', \Delta', \delta', K_0', K_f') \bullet op \notin F'.$$
Cette hypothèse est facilement généralisée à tous les types définis dans le système, pour prendre en compte les spécifications algébriques de types abstraits de donnée définies autrement que par les TAG (voir annexe A).
- (*HYP2*) pour chaque TAG, il existe au moins un constructeur de base : $\Gamma_b \neq \emptyset$. Une spécification sans constructeur de base est appelée **spécification abstraite** par analogie avec les classes abstraites du modèle objet.
- (*HYP3*) le graphe de la relation de dépendance forte DPF est sans circuit. Les paramètres formels de type ($PFT(TI)$) sont traités de la même façon, une fois ces paramètres instanciés.
- (*HYP4*) les transitions sont déterministes : $\forall <k_1, k_2, g_1, o>, <k_1, k_3, g_2, o> \in \delta \bullet g_1 \wedge g_2 = false$. Cette hypothèse est contraignante mais facilite la démonstration de la convergence du système de réécriture associé au TAG.
- (*HYP5*) les opérations sont distinguée par leur nom et leur domaine : $\forall cb(p_1 : T_1, ...p_n : T_n) : TI \in F_{TI},\ \nexists T \in SORTES \bullet cb(p_1 : T_1, ...p_n : T_n) : T$.

5.4 Spécification algébrique de TAG

La spécification graphique du TAG est insuffisante pour définir complètement le type de donnée. La sémantique est affinée en ajoutant des axiomes pour les opérations. Après quelques rappels et définitions sur les spécifications algébriques, nous verrons comment une spécification algébrique est construite par traduction et enrichissement du TAG.

5.4.1 Rappels et définitions

Nous nous plaçons dans le cadre général des spécifications algébriques structurées dont les définitions et notations usuelles sont rappelées dans l'annexe A. Le terme **type** désignera à la fois la sorte et le type. Les spécifications sont contruites modulairement et incrémentalement. Chaque module définit un unique type de donnée à partir des définitions d'autres types de données. Cette approche est similaire à celle de Guttag[GH78] (**type d'intérêt** vs "types prédéfinis") Bidoit[Bid82] (un type abstrait vs environnement abstrait) Musser[Mus80] (type vs collection de types). Habituellement, l'idée retenue est de supposer que chaque module est bien défini, et que l'assemblage des modules (spécification primitive) et leur enrichissement par une nouvelle présentation ne perturbe pas la définition des types qu'ils spécifient. Pour cela, les auteurs émettent des hypothèses fortes : contrainte hiérarchique (le graphe de la relation d'importation entre modules est un arbre), cohérence et complétude des modules utilisés.

Dans un contexte objet, la contrainte hiérarchique est trop forte. Par exemple, supposons un rectangle construit à partir de deux points. Il est naturel en objet, d'avoir une méthode `extrémité:` de profil `extrémité::` `Point` $\times$ `Point` $\rightarrow$ `Rectangle` dans la spécification du type `Point`, qui à partir de deux points crée un rectangle. Cet observateur sera traduit par un appel à un générateur de base du type `Rectangle`, de profil `new:` `Point` $\times$ `Point` $\rightarrow$ `Rectangle`. C'est donc un observateur secondaire. La relation USE inclut le type `Point` sur `Rectangle` et le type `Rectangle` sur `Point`. Il en est de même pour la référence symétrique entre une fenêtre et l'environnement qui la contient. Il existe d'autres cas typiques où la contrainte hiérarchique est trop forte : ce sont les définitions récursives de types de données, conflits entre un type et un sous-type (exemple : une liste non-vide est définie à partir d'une liste et d'un élément, c'est en même temps une liste), ou les conflits entre un paramètre et un type paramétrique (exemple : une liste de liste).

Plutôt que de définir les propriétés de la spécification par interprétation des opérateurs de structuration dans les algèbres (foncteur d'oubli, réductions, morphismes), nous avons choisi une approche rationnelle basée sur la construction inductive des termes. Ainsi, il n'est pas fait d'hypothèses a priori sur l'organisation des modules de spécification, de même que [Mus80]. La relation d'importation entre types est un graphe orienté (DAG). Cette approche, induite par l'automate de comportement, est donc de type constructive plutot que déductive. Il s'agit de construire une théorie et non d'interpréter en termes de classes de modèles [BB92].

Définition 5.4.1 (Présentation TAG)
*Une **présentation TAG** d'un type TI est un sextuplet $pres(TI) = (TI, S, \Gamma, \varphi, X, E)$ où $\Sigma = (S', F)$ est une signature avec $S' = S \cup \{TI\}$ et $F = \Gamma \cup \varphi$, X un S'-ensemble de variables (i.e. de sorte $s \in S'$) et E un ensemble d'axiomes sur Σ, et tel que*

(i) $TI \notin S$,
(ii) $\forall f \in \Gamma,\ codom(f) = TI$,
(iii) $\forall o \in \varphi,\ \exists s \in S \bullet codom(o) = s \wedge TI \in dom(o)$.
(iv) $\forall\ (cond \Rightarrow t == t') \in E,\ \exists op \in F \bullet t = op(t_1, .., t_n)$.

Soit $pres(TI) = (TI, S, \Gamma, \varphi, X, E)$ une présentation TAG d'un type TI. Elle engendre une spécification algébrique structurée (voir section A.2) appelée **spécification TAG**. La présentation plate correspondante est donnée par :

$$spec(TI) = pres(TI) \cup \Big(\bigcup_{U \in USE(TI)} pres(U) \Big),$$

où $\cup$ représente l'union de deux présentations[3] $((\Sigma, X, E) \cup (\Sigma', X', E') = (\Sigma \cup \Sigma', X \cup X', E \cup E'))$. Le langage $L(TI)$ est l'algèbre des termes $T_{\Sigma[X]}$ engendrée par $spec(TI)$ selon la définition A.1.4.

La relation de congruence "=" définie par les axiomes de E induit une théorie équationnelle dont les classes d'équivalence sont les termes de l'algèbre qui interprète la spécification. La théorie équationnelle d'un type T est l'intersection de la théorie équationnelle engendrée par E avec le langage $L(T)$. Elle est bien-fondée si la signature est raisonnable.

Définition 5.4.2 (complétude suffisante)
Une spécification TAG $spec(TI)$ est ***suffisamment complète*** *si tout terme t clos de $L(TI)$ de type U ($U \in USE(TI)$) peut être prouvé égal à un terme t_U de $L(U)$.*

Définition 5.4.3 (cohérence hiérarchique)
Une spécification TAG $spec(TI)$ est ***cohérente hiérarchiquement*** *si pour tous termes clos t et t' de $L(U)$ ($U \in USE(TI)$) si $t =_{TI} t'$ alors $t =_U t'$*[4].

Une opération partielle est conditionnée par un prédicat de définition notés D. Dans notre cas, le prédicat de définition peut être associé aux états. La relation de congruence "=" est redéfinie en tenant compte de ces prédicats :

$$t = t' \Leftrightarrow \begin{cases} \text{vrai} & \text{si } [\![t]\!]_v^A \text{ et } [\![t']\!]_v^A \text{ sont indéfinis} \\ [\![t]\!]_v^A = [\![t']\!]_v^A & \text{si } [\![t]\!]_v^A \text{ et } [\![t']\!]_v^A \text{ sont définis} \\ \text{indéfini} & \text{sinon} \end{cases}$$

5.4.2 Interface

L'**interface** d'un TAG définit la signature $\Sigma = (S, F)$ du type d'intérêt. Elle est obtenue directement en analysant les profils des opérations du TAG. La différenciation entre constructeurs et observateurs est effectuée en même temps.

```
F := S := Γ_b := Γ := φ := ∅
pour oper(p_1 : Type_1, p_2 : Type_2, ...p_n : Type_n) : Type_res ∈ profil_operations(TI) faire
    F = F ∪ {oper}                                  // ensemble des opérations //
    S = S ∪ {Type_1, Type_2, ...Type_n, Type_res}    // ensemble des sortes //
    si Type_res = TI alors
        si Type_res ≠ Type_1
            alors Γ_b = Γ_b ∪ {oper}                // ensemble des constructeurs de base //
            sinon Γ = Γ ∪ {oper}                    // ensemble des constructeurs //
        fsi
    sinon φ = φ ∪ {oper}                            // ensemble des observateurs //
    fsi
fait
```

Aux observateurs définis par le concepteur, sont ajoutés automatiquement les **observateurs d'état**. Ces observateurs sont des fonctions booléennes, du nom de l'état auquel elles sont associées avec comme paramètre unique $Self : V_{TI}$.

```
pour k_origine ∈ K faire
    profil_operations(TI) = profil_operations(TI) ∪ {nom_k_origine(Self : TI) : Bool}
    F = F ∪ {nom_k_origine}
    φ = φ ∪ {nom_k_origine}
    τ = τ ∪ {nom_k_origine}
fait
```

Les paramètres formels constants sont transformés en observateurs. Un argument est ajouté à chaque constructeur de base, de déclaration le paramètre constant. Ceci est un choix particulier d'implantation.

```
pour (var, type) ∈ PFC faire
```

[3] c'est-à-dire les opérateurs d'enrichissement et d'union habituels pour les spécifications structurées.
[4] La cohérence est encore définie par ($vrai = faux$) n'appartient pas à la théorie équationnelle de E.

$profil\_operations(TI) = profil\_operations(TI) \cup \{var(Self : TI) : type\}$
$F = F \cup \{var\}$
$\varphi = \varphi \cup \{var\}$
$\tau = \tau \cup \{var\}$
pour $oper(p_1 : Type_1, p_2 : Type_2, ...p_n : Type_n) : V_{TI} \in \Gamma_b$ **faire**
$\Gamma_b = \Gamma_b - \{oper(p_1 : Type_1, p_2 : Type_2, ...p_n : Type_n) : V_{TI}\}$
$\Gamma_b = \Gamma_b \cup \{oper(p_1 : Type_1, p_2 : Type_2, ...p_n : Type_n, var : type) : V_{TI}\}$
fait
fait

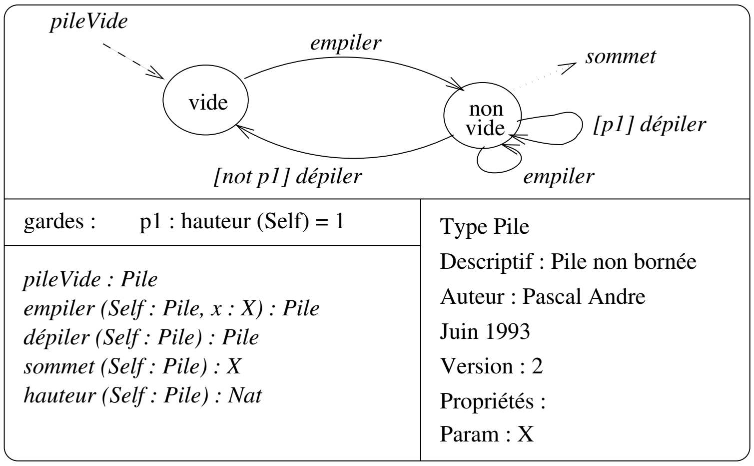


Figure 33 : Type Pile non bornée.

La signature extraite par l'algorithme est :

```
S    = { Pile, Bool, X, Nat }                                  // types (sortes) //
F    = { pileVide, empiler, dépiler, sommet, hauteur }         // opérations //
Γb   = { pileVide }                                            // constructeurs de base //
Γ    = { empiler, dépiler}                                     // constructeurs //
φ    = { sommet, hauteur }                                     // observateurs //
```

Celle du compte bancaire de la figure 32 s'écrit :

```
S    = { Cpt, Entier[23], Intitulé, Réel, RéelP, Bool}
F    = { ouvrir, intitulé, numéro, solde, retirer, déposer, créditeur, débiteur }
Γb   = { ouvrir}
Γ    = { retirer, déposer }
φ    = { intitulé, numéro, solde, créditeur, débiteur }
```

5.4.3 Préconditions et prédicats

Dans la pratique, tous les termes produits par la signature ne sont pas intéressants. Par exemple, `dépiler` un élément de la pile n'est possible que si celle-ci n'est pas vide. Habituellement, deux alternatives, non mutuellement exclusives, sont utilisées : soit les termes inopportuns sont éliminés par des prédicats de définition sur les types, soit ils sont traités à part comme **termes erronés**. Dans le modèle TAG, la notion d'état permet de traiter, de façon élégante et naturelle, le problème des opérations partielles et des traitements d'erreur.

Une partition des opérations selon les domaines de définition est utile pour le développement des axiomes. Une **opération totale** est une opération applicable à un type de donnée quel que soit son état. Une **opération partielle** est une opération dont l'application dépend de l'état courant. Une **opération neutre** est une opération totale qui ne change pas d'état et ne comporte pas de garde. Les opérations neutres ne sont pas indiquées sur l'automate. Dans l'exemple de la figure 32, l'opération `déposer` est totale, `solde` est neutre, et `retirer` est partielle. A cause des gardes, la distinction entre opération totale et opération partielle n'est pas triviale. Une opération est partielle si elle n'est pas définie sur un état, ou si pour un état quelconque, l'union des gardes d'une opération n'est pas égale à vrai. Une classification complète des opérations se trouve en annexe E.

Génération de l'invariant et des prédicats locaux

Un **invariant** de spécification algébrique TAG est un prédicat vrai pour toutes les valeurs du types d'intérêt. Il correspond à un prédicat de définition sur le type d'intérêt. L'invariant est la traduction de la contrainte sous forme d'une conjonction d'équations (i.e. comme les conditions des axiomes). Les prédicats locaux sont des fonctions booléennes nommées, utilisées dans les axiomes et les préconditions.

Génération des préconditions

Une **précondition** d'opération est un prédicat associé à une opération qui doit être valide pour que l'opération puisse être invoquée. La précondition d'une opération (partielle) est l'union des gardes de cette opération, compte-tenu des états origine. Par exemple, pour *dépiler* il ne faut pas que la pile soit vide, la précondition naturelle est *non-vide(self)*. Les préconditions peuvent apparaître dans les axiomes, ou demeurer implicite à l'application des opérations (option de l'outil implantant l'algorithme). C'est ce dernier choix que nous préconisons pour alléger l'écriture.

La génération des préconditions est automatique. Les opérations totales ont pour précondition implicite vrai (elles sont applicables partout). Pour les opérations partielles, il suffit d'examiner toutes les transitions et de construire incrémentalement les préconditions par disjonction. Cependant, le calcul des ensembles τ (opérations totales) et ∂ (complémentaire de τ dans $\Gamma \cup \varphi$) nécessite soit un outil de simplification d'expressions logiques, soit une procédure de décision interactive. La génération des préconditions est donc semi-automatique. Le domaine d'une opération *op* est l'ensemble Dom_{op} des états où elle est définie. La précondition d'une opération *op* est notée $precondition(op)$. La précondition d'une opération *op* pour un état donné e est notée $pre(op,\ e)$. Enfin, la variable du type en paramètre d'une opération est notée V_{TI} et son observateur d'état k est noté $k(V_{TI})$.

```
pour oper ∈ Γ_b faire
precondition(oper):= true
fait
τ := Γ_b                    // totales //
η := ∅                      // neutres //
∂ := ∅                      // partielles //
pour op ∈ (φ ∪ (Γ − Γ_b)) faire
Dom_op := ∅                 // domaine //
precondition(op):= false
    pour k_origine ∈ K faire
        pre(op, k_origine) := false
        pour <k_origine, k_destination, garde, op> ∈ δ faire
            // une seule transition si c'est un observateur //
            pre(op, k_origine) := pre(op, k_origine) ∨ k_origine(V_TI) ∧ garde
            // garde = vrai par défaut //
        fait
        si ¬ pre(op, k_origine) alors
            // pas défini sur cet état //
            ∂ := ∂ ∪ {op}
        sinon
            Dom_op := Dom_op ∪ {k_origine}
            simplifier(pre(op, k_origine));
            si ¬ pre(op, k_origine) alors
                // pas de partition des gardes //
                ∂ := ∂ ∪ {op}
            fsi
            precondition(op):= precondition(op) OR pre(op, k_origine)
        fsi
```

```
        fait
        si ¬ precondition(op) alors
            // l'opération n'est pas sur l'automate: elle est neutre //
            η := η ∪ {op}
            precondition(op) := true
        sinon
            simplifier(precondition(op));
        fsi
    fait
    τ := τ ∪ ((Γ ∪ φ) − ∂)

// Calcul pour l'opération dépiler de la pile //
precondition(dépiler(p)) := false,                    Dom_dpiler := ∅
   1- pre(dépiler(p),non-vide) := false
   2- pre(dépiler(p),non-vide) := false OR (non-vide(p) AND not p1(p))
   3- pre(dépiler(p),non-vide) := false OR (non-vide(p) AND not p1(p)) OR (non-vide(p) AND p1(p))
   4- pre(dépiler(p),non-vide) = non-vide(p) // après simplification // Dom_dpiler := {non-vide}
d'où
   precondition(dépiler(p))      = non-vide(p)
   precondition(pileVide)        = true
   precondition(empiler(p,x))    = true
   precondition(sommet(p))       = non-vide(p)
   precondition(hauteur(p))      = true
   ∂ = {dpiler, sommet}
   τ = {pileVide, empiler, hauteur, vide, non-vide}
   η = {hauteur}
```

L'exemple de l'opération **dépiler** met en évidence la nécessité de simplifier les expressions logiques. L'utilisation de gardes nommées et de connecteurs logiques est une première facilité pour des conditions simples mais elle n'est pas suffisante. Les outils Muffin [JJLM91] ou B-Tool [LLS91] assistent les preuves.

5.4.4 Partitionnement de la spécification

Principe et définitions

Pour réduire le nombre d'axiomes et proposer un schéma de démonstration de propriétés basé sur l'induction structurelle, la présentation est scindée en deux partitions : la **présentation primaire** et la **présentation secondaire**. Les opérations de la partition primaire sont définies à partir des opérations de la partition secondaire. La partition primaire est suffisante pour décrire complètement le type d'intérêt. Cette partition est habituelle dans les spécifications algébriques, et elle est indispensable dans les TAG pour mettre à profit de principe de construction d'axiomes assistée par automate TAG (voir section 5.4.5). Une classification complète des opérations se trouve en annexe E.

Définition 5.4.4 (noyau d'une spécification)
Soit $\Sigma = <S, \Gamma, \varphi>$ *et* $\Sigma_p = <S, \Omega, \varphi_b>$ *deux signatures d'un type* `TI` *telles que* $\Omega \subseteq \Gamma$ *et* $\varphi_b \subseteq \varphi$. Σ' *est un noyau de* Σ *ssi* $\forall t \in \Sigma,\ \exists t' \in \Sigma' \bullet\ t = t'$.

Les éléments de Ω sont appelés générateurs du type `TI` et $\widehat{\Omega_{TI}}$ est l'ensemble des familles génératrices du type `TI`. Cette définition est plus "pratique" que la définition A.1.12 ($\Omega \subseteq C$), car elle s'applique uniquement à la spécification en cours et qu'elle ne tient pas compte de l'interprétation. Les éléments de φ_p sont appelés observateurs primaires du type `TI`.

Détermination des générateurs

Il s'agit de choisir parmi les constructeurs un ensemble de générateurs. L'ensemble des générateurs doit au moins permettre d'atteindre chaque état de l'automate. C'est une condition nécessaire mais non suffisante pour affirmer que chaque terme sera produit par les opérations de cette ensemble, car les gardes ont une sémantique dynamique. Il faut montrer que la famille des constructeurs "*choisis*" est génératrice. Parmi les familles génératrices, celle de plus petite cardinalité sera privilégiée. Dans l'exemple de la figure 30, $\{init, admit, urgency, cure\}$ est une famille, mais $\{init, admit, urgency\}$ est une famille plus petite.

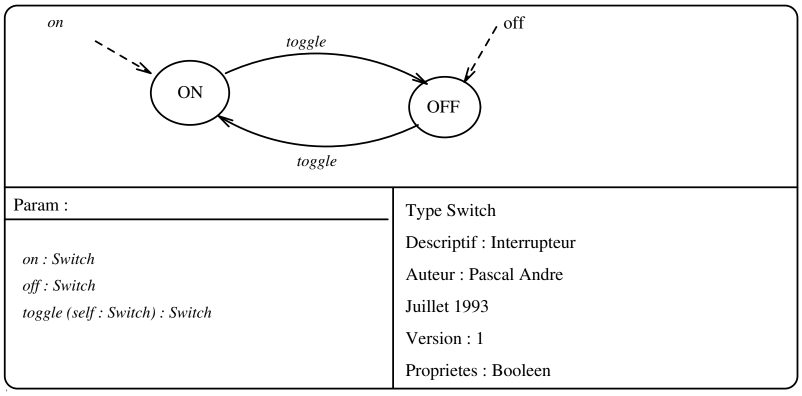


Figure 34 : Type Switch.

L'idée intuitive de l'algorithme de calcul de la famille génératrice est de parcourir le graphe, en partant des états initiaux et de calculer les chemins minimaux pour atteindre chaque état. L'algorithme débute avec un seul état initial ou avec un ensemble d'états initiaux. La première solution revient à privilégier un constructeur de base, c'est une approche intéressante dans une perspective objet, mais elle peut aussi orienter vers une implantation particulière. Nous proposons un algorithme glouton de détermination de cet ensemble, en partant de Γ_b et en utilisant les états potentiellement accessibles, en utilisant si possible les constructeurs déjà dans Ω et permettant à l'utilisateur de choisir s'il en existe plusieurs. Cette simplification marche dans les cas pratiques étudiés. Prendre les gardes en compte revient à associer un ensemble de conditions logiques à chaque chemin, ensemble qu'on ne peut simplifier statiquement, sauf s'il existe des invariants sur l'évolution des états.

```
Ω := Γ_b
etats_atteints := K_0
etats_en_cours := K_0
tant que etats_atteints ≠ K faire
    pour e_o ∈ etats_en_cours faire
        pour op ∈ Ω faire
            pour e_f ∈ δ(e_o, op, g) faire
                si e_f ∉ etats_atteints alors
                    etats_atteints += {e_f}
                fsi
            fait
        fait
        pour op ∈ Γ − Ω faire
            pour e_f ∈ δ(e_o, op, g) faire
                si e_f ∉ etats_atteints alors
                    etats_atteints += {e_f};
                    op_choisie := un_de({op_i : Γ − Ω | δ(e_o, op_i, g) = e_f});
                    Ω += {op_choisie}
                fsi
            fait
        fait
    fait
fait
```

A partir d'un seul état initial, le parcours est simple. Les chemins de longueur 1, 2, etc sont examinés jusqu'à atteindre tous les états, en ne repassant jamais deux fois par le même état. Un meilleur algorithme, applicable dans ce cas est celui qui détermine le plus court chemin d'un sommet (l'état initial) à tous les autres. Lorsqu'on part d'un ensemble d'états initiaux, l'algorithme calcule les plus courts chemins, mais la famille obtenue n'est pas forcément minimale.

Dans l'exemple de la figure 34, l'algorithme par état initial unique donne l'une des deux familles de même longueur, pour $K_0 = \{ON\}$ ou $K_0 = \{OFF\}$.

$1 - etats\_atteints = \{ON\} \;\wedge\; etats\_en\_cours = \{ON\} \;\wedge\; \Omega = \{on\} \Rightarrow \delta(ON) = \{< toggle,\ OFF >\}$
$. \quad etats\_atteints\ = \{ON, OFF\} = K \;\wedge\; etats\_en\_cours\ = \{OFF\} \;\wedge\; (\Omega = \{on, toggle\})$
$2 - etats\_atteints = \{OFF\} \;\wedge\; etats\_en\_cours = \{OFF\} \;\wedge\; \Omega = \{off\} \Rightarrow \delta(OFF) = \{< toggle,\ ON >\}$
$. \quad etats\_atteints\ = \{ON, OFF\} = K \;\wedge\; etats\_en\_cours\ = \{ON\} \;\wedge\; (\Omega = \{off, toggle\})$
Nous en déduisons : $on == toggle(off) \vee off == toggle(on)$.

L'algorithme global donne une famille encore plus simple à calculer, et plus logique intuitivement.

$etats\_atteints = K_0 = \{ON, OFF\} = K \;\wedge\; etats\_en\_cours = K_0 = \{ON, OFF\} \;\wedge\; \Omega = \Gamma_b = \{on, off\}$
Nous en déduisons : $toggle(on) = off \wedge toggle(off) = on$

`Switch` : $\Omega = \{on, toggle\}$
`Pile` : $\Omega = \{pileVide, empiler\}$
`Compte bancaire` : $\Omega = \{ouvrir, retirer\}$

Une autre idée est à creuser, la construction d'un graphe réduit au sens des composantes fortement connexes (cfc) pour déterminer les générateurs entre cfc et à l'intérieur des cfc. Enfin, un dernier algorithme calcule toutes les familles de générateurs, avec éventuellement les gardes associées et le concepteur en choisit une.

Détermination des observateurs primaires

Il n'existe pas d'algorithmes pour calculer φ_p directement car il n'y a pas de corrélation directe entre les observateurs comme le montrent les exemples suivants :

1.	`vide(Self) == hauteur(Self) = 0`	// la pile est vide si la hauteur est nulle //
2.	`last(Self) == car(reverse(Self))`	// le dernier élément de la liste est //
		// le premier élément de la liste inversée //
3.	`OPENED(Self) == true ==>`	// l'origine de la fenêtre est celle//
	`origin(Self) == orig(Self)`	// de la fenêtre ouverte//
	`CLOSED(Self) == true ==>`	// ou celle //
	`origin(Self) == iconOrig(Self)`	// de la fenêtre iconifié //
4.	`x == x(bg) + longueur`	// le coin haut droit d'un rectangle est calculé à //
	`y == y(bg) + largeur`	// partir du coin bas gauche et des dimensions //

Les observateurs peuvent être classés par groupes selon la compatibilité du type résultat ou le domaine de définition ou les deux. Cette classification est une heuristique pour faciliter la comparaison mais elle insuffisante. Deux observateurs de type identique peuvent être liés (e.g. `car` et `last`) ou non (e.g. `nom` et `adresse`). Deux observateurs de type différents peuvent être mis en correspondance (e.g. cas 1). Deux observateurs partiels peuvent avoir le même domaine (e.g. `car` et `last`) ou être complémentaires. Un observateur partiel peut être une extension d'un observateur total et inversement (e.g. cas 3).

Le critère de tri peut être élargi à fermeture transitive `USE`. Cette mise à plat de la hiérarchie n'est pas conseillée. En général, un tel type sera plutôt obtenu en appliquant une séquence ordonnée d'observateurs à une variable du type d'intérêt. Par exemple, supposons une spécification structurée d'une voiture telle que `Voiture` use `Carrosserie` use `Porte` soient les observateurs `porteAvantDroite: Voiture` $\rightarrow$ `Porte`, `carosserie: Voiture` $\rightarrow$ `Carrosserie` et `porteAvantDroite: Carrosserie` $\rightarrow$ `Porte`, l'axiome associé à cet observateur sera
`[...] ==> porteAvantDroite(`V_{TI}`) == porteAvantDroite(carosserie(`V_{TI}`))`.

Représentation des dépendances

Durant la recherche des opérations primaires, il est intéressant de représenter les dépendances entre opérations par un graphe. C'est une autre manière, plus intuitive, de déterminer la partition primaire/secondaire de la présentation. Les dépendances des gardes peuvent être ajoutées. Le graphe suivant met en valeur les dépendances directes et indirectes des opérations d'une liste bornée, définie par un TAG à trois états **`empty`**, **`full`** et l'état intermédiaire **`NENF`**.

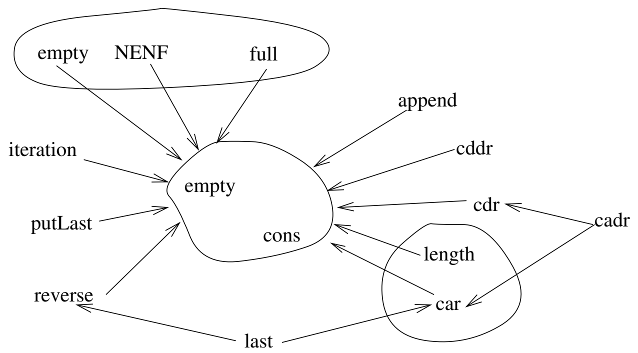


Figure 35 : Dépendances des opérations d'une liste bornée.

Ce graphe constitue un outil d'aide et d'évaluation des choix lors que le nombre d'opérations est important. Il visualise les interdépendances entre opérations, et permet de mieux voir le noyau d'opérations les plus "utilisées". Un travail sur les chemins du graphe peut permettre de détecter le noyau d'opérations primaires.

5.4.5 Construction d'une axiomatique

Soient $< TI, F, PFT, PFC, INV, M >$ un TAG, avec $M = (F, K, \Delta, \delta, K_0, K_f)$ et $pres(TI) = (TI, S, \Gamma, \varphi\ , X, E)$ la présentation TAG correspondante. L'automate va servir à la construction méthodique et systématique des axiomes de E et des variables de X.

Automate et langage

Définissons et calculons l'algèbre des termes de l'automate. Ces termes sont des couples `(p, t)` où `t` un terme avec variable et `p` est un prédicat de définition associé à t.

Définition 5.4.5 (algèbre des termes d'un état)
Etant donnés un automate $M = (F, K, \Delta, \delta, K_0, K_f)$ définissant un type TI, $\Sigma = (S', F)$ la signature correspondante avec $S' = S \cup \{TI\}$ et $F = \Gamma \cup \varphi$, un S'-ensemble de variables X, $L(k)$ (ou $T_{\Sigma[X]_k}$) est défini comme le (plus petit) S-ensemble des Σ-termes avec variables formé à partir des opérations de Σ, par $\forall k \in K, \forall t_k \in L(k), \forall p_i \in L(T_i)$:

(i) $\forall x \in X_s \bullet x \in L(s)$,
(ii) $\forall <\varepsilon, k, g, cb : T_1, ..., T_n \rightarrow TI> \in \delta \bullet (g, cb(p_1, ..p_n)) \in L(k)$,
(iii) $\forall <k', k, g, cons : TI, T_1, ..., T_n \rightarrow TI> \in \delta \bullet (g, op(t_k, p_1, ..p_n)) \in L(k')$,
(iiv) $\forall <k, \varepsilon, g, obs : TI, T_1, ..., T_n \rightarrow T_{n+1}> \in \delta \bullet (g, obs(t_k, p_1, ..p_n)) \in L(k)$.

Le cas (i) correspond aux variables, le cas (ii) correspond aux constructeurs de base, le cas (iii) correspond aux autres constructeurs, le cas (iv) correspond aux observateurs. Le langage associé à l'automate M est défini comme l'union des langages associés aux états de l'automate:

$$L(M) = \bigcup_{k \in K} L(k)$$

Propriété 5.4.6 (terme bien défini)
Soit `(p, t)` un terme de $L(M)$, où `t` est un terme et `p` un prédicat. Si `t` est un terme clos alors p est aussi un terme clos et t est défini si `p` est vrai.

Cette propriété découle de la définition même de la garde.

Génération des termes : le principe de la G-dérivation

Par définition, le langage engendré par l'automate donne tous les termes de $L(TI)$. Un chemin dans cet automate (graphe orienté) représente un terme t de $L(TI)$, de la forme $(cond, op(c_n(c_{n-1}(...(c_1(args_1), ...), args_n n-1), args_n))$, où $c_i \in \Gamma$, $op \in F$ et $cond$ est un prédicat sur les arguments $args_i$ tel que t est défini si $cond(t) = vrai$. En supposant l'existence d'une famille génératrice Ω, les constructeurs c_i sont réduits aux générateurs $g_j \in \Omega$.

La G-dérivation construit toutes les séquences possibles de générateurs par un parcours en profondeur d'abord. Le parcours est stoppé dans trois cas : équation résolue, état déjà visité (K_{vis}), constructeur de base. La construction de l'axiome reprend la séquence de générateurs en tenant compte des gardes.

Les termes obtenus ($T_{\Sigma_{\widehat{\Omega}}(X)}$) sont dits *G-dérivés*. Une G-dérivation d'un terme contenant une variable V_{TI} du type d'intérêt TI et se trouvant dans un état $e(V_{TI})$ est une substitution σ de V_{TI} par un générateur menant à l'état $e(V_{TI})$. Cette définition correspond à celle de Bidoit [Bid82] pour `x = Self`. Les algorithmes de G-dérivation correspondent ainsi à des parcours inverses de l'automate de comportement réduit aux transitions des générateurs.

Prenons un exemple, soit l'automate réduit de la figure 33 et `p` une pile quelconque *non-vide*. Deux G-dérivations sont possibles : `[p = empiler(p', x) AND vide(p1)]` ou `[p = empiler(p'', x) AND non-vide(p'')]`. Si la G-dérivation de `p` est poursuivie, l'unique terme obtenu est `empiler(pileVide, x)`.

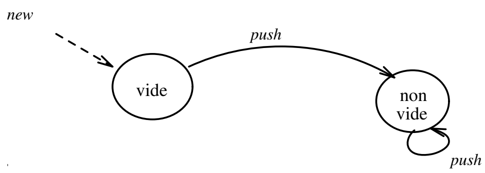


Figure 36 : Un exemple de G-dérivation.

Morphologie des axiomes

Une étude de la forme des axiomes permet de les classer. Cette classification est utile dans le traitement de l'héritage (section 5.6.2). Soit c_1 `& ... &` c_n `==> op(rec,args) == op'(args')` un axiome et `Self` une variable du type d'intérêt.

- `rec = Self` : pas de G-dérivation : l'axiome est dit opérationnel

op $\in \Gamma$	op' $\in \Omega$	opérationnel direct
	op' $\in \Gamma - \Omega$	opérationnel indirect
op $\in \varphi$	op' $\in \varphi$	opérationnel indirect
	op' $\in \Gamma_{type\_resul(op)}$	opérationnel direct
autres		impossible

- `rec = gen(...)` : G-dérivation sur le générateur `gen` : l'axiome est dit G-dérivé

Définition 5.4.7 (extension opérationnelle, extension G-dérivée)
*Une opération est dite **extension opérationnelle** si tous ses axiomes sont opérationnels. Une opération est dite **extension G-dérivée** si tous ses axiomes sont G-dérivés.*

Définition 5.4.8 (présentation G-dérivée)
*Une présentation TAG $pres(TI) = (TI, S, \Gamma, \varphi, X, E)$ est dite **G-dérivée** si les axiomes sont de la forme cond $\Rightarrow op(l) == r$ où l est un terme G-dérivé.*

Génération des axiomes

L'algorithme de G-dérivation est à la base de la construction des axiomes de la spécification TAG (i.e. l'ensemble E). L'idée est de construire automatiquement toutes les parties gauches des axiomes et de compléter ensuite interactivement les parties droites. La construction systématique par l'automate évite les oublis, surtout ceux concernant les conditions d'application. Un axiome est représenté par la structure de donnée suivante :

```
structure axiome c'est
     n : Nom;
     c : ConditionBooléenne;
     d : T_Σ[X];
     g : T_Σ[X];
fin structure
```

L'axiome est noté en extension par `[n, h, d, g]`. Les parties gauches des axiomes et les conditions sont définies par :

- dérivation des constructeurs secondaires à partir des générateurs,
- application des observateurs primaires sur les générateurs,
- des observateurs secondaires à partir des observateurs primaires ou des constructeurs des types utilisés.

L'opération $\sigma : T_{\Sigma,X} \times X \times T_{\Sigma,X} \rightarrow T_{\Sigma,X}$ substitue une variable d'un terme par un autre terme. Par exemple, $\sigma(dépiler(V_{TI}), V_{TI}, empiler(V_{TI}, x)) = dépiler(empiler(V_{TI}, x))$. Les variables sont renommées pour éviter des confusions entre les deux termes paramètre de σ. Une fonction `partie_droite?(a,dom)` demande interactivement à l'analyste, la partie droite d'une équation (soit a.d). Cette fonction effectue des vérifications (opérations utilisées, variables définies...). Ce contrôle est exprimé par le paramètre `dom` exprimant l'ensemble des opérations utilisables dans le terme `a`. L'interface avec l'utilisateur sera améliorée par des questions en langage naturel, la visualisation des chemins dans l'automate, la recherche d'autres chemins pour une boucle, etc.

a) Génération des axiomes des observateurs d'état Il s'agit de déterminer l'état abstrait résultat de l'application d'un constructeur. L'observateur d'état booléen associé à l'état e sera noté $e(V_{TI})$, $axiome_e$ est l'axiome associé.

```
pour e ∈ K faire
     pour op ∈ Γ_b faire
          axiome_e := [e, true, e(op(args)), false];
     fait
     pour op ∈ (Ω − Γ_b) faire
          axiome_e := [e, true, e(op(V_TI, args)), false];
     fait
fait
pour op ∈ Ω faire
     pour e_f ∈ δ(e_d, op, g) faire
               si e_d = ε alors
                    axiome_{e_f}.d := g
               sinon
                    axiome_{e_f}.d := axiome_{e_f}.d ∨ (e_d(V_TI) AND g)
               fsi
     fait
fait
pour e ∈ K faire
     E += axiome_e;
fait
```

```
Pile :
   vide(pileVide) == true
   vide(empiler(p, x)) == false
   non-vide(pileVide) == false
   non-vide(empiler(p, x)) == true
```

```
Compte bancaire :
   débiteur(ouvrir(num,int,v)) == false
   débiteur(retirer(c,m)) == créditeur(c) AND p1(c)
   créditeur(ouvrir(num,int,v)) == true
   créditeur(retirer(c,m)) == créditeur(c) AND NOT p1(c)
```

b) G-dérivation des constructeurs secondaires Avant d'étudier l'action des observateurs sur les générateurs, il faut vérifier que la famille Ω calculée précédemment est génératrice, en calculant les axiomes des constructeurs secondaires. Parmi les constructeurs secondaires, s'il existe des constructeurs de base, il faut les traiter à part car ils doivent forcément être définis par des termes clos (i.e. G-dérivation terminée par un générateur de base).

b1) Cas des constructeurs de base

L'objectif est de trouver un chemin issu de tous les autres états initiaux. La présence des gardes implique de les donner tous. Plusieurs choix d'algorithmes de parcours sont possibles pour trouver un chemin : profondeur, largeur (plus court chemin).

```
pour op ∈ (Γ_b − Ω) faire
    init := {e : K_0 • δ(ε, op, g) = e}; // etat initial du constructeur de base op //
    axiome_op := [op,g,op(args),V_TI];
    K_vis := ∅;
    pour e ∈ init faire
        chemins(axiome_op,e)
    fait
fait

procédure chemins (a : E, e : K) c'est
début
    K_vis += {e}
    pour (e', op) ∈ K × Ω • δ(e', op, g) = e ∧ e' ≠ e faire
        // les boucles ne sont pas prises en compte //
        cas e' dans
            {ε} : // état initial //
                a.d := σ(a.d, V_TI, op(args));
                a.c := g AND σ(a.c, V_TI, op(args));
                E += {a} // ajout d'un axiome //
            K_vis :
                // circuit inutile //
            autre : // état intermédiaire //
                a.d := σ(a.d, V_TI, op(V_TI, args));
                a.c := g AND σ(a.c, V_TI, op(args));
                chemins(a,e')
        fincas
    fait
fin
```

Dans l'exemple de l'interrupteur, si $\Omega = \{on, toggle\}$ est la famille génératrice, alors off est un constucteur de base secondaire. Voici la trace de l'algorithme :

```
   chemins([off, true, off,V_TI],OFF)
K_vis = {OFF}, op = toggle, e' = ON ==> chemins([off, true, off,toggle(V_TI)],ON)
K_vis = {OFF, ON}, op = on, e' = ε ==> E +={[off, true, off,toggle(on)],ON]}
K_vis = {OFF, ON}, op = toggle, e' = OFF ==> "circuit"
```

b2) Cas général

Deux sortes de constructeurs secondaires existent : ceux qui "modifient" des paramètres initiaux (par exemple `déposer de l'argent`) et ceux qui expriment des états réinitialisables (par exemple `dépiler un élément`). L'algorithme général remonte les états jusqu'à trouver le paramètre modifié ou la séquence de réinitialisation.

```
dom := Ω ∪ φ_p
// domaine (opérations) acceptable pour les parties droites //
pour op ∈ (Γ − Ω − Γ_b) faire
    axiome_op := [op,true,op(V_TI,args),V_TI];
    // V_TI en partie droite permet de conserver la structure //
```

```
        dom += {op}
        // les définitions récursives sont autorisées sauf en première position //
        si partie_droite?(axiome_op, dom) alors
            E += {axiome_op}
            // ajout immédiat d'un axiome //
        sinon
            // il faut dériver //
            si op ∈ τ alors // operation totale //
                pour e ∈ K faire
                    K_vis := ∅;
                    // ensemble d'états visités (détecter les circuits) //
                    G-derive-c(axiome_op, e)
                fait
            sinon
                // operation partielle //
                pour {(e, e') ∈ K × K • δ(e, op, g) = e'} faire
                    // les boucles sont prises en compte au départ //
                    K_vis := ∅;
                    a.c := g; // la garde est prise en compte //
                    G-derive-c(axiome_op, e)
                fait
            fsi
        fsi
    fait

    procédure G-derive-c (a : E, e : K) c'est
        // effets de bord sur E et K_vis //
    début
        K_vis += {e}
        si partie_droite?(a,dom) alors
            a.c := a.c AND e(V_TI);
            // e(V_TI) est le prédicat d'état associé Ì'état e //
            E += {a}
            // ajout d'un axiome //
        sinon
            // trouver les générateurs qui arrivent dans e //
            pour (e', op) ∈ K × Ω • δ(e', op, g) = e faire
                cas e' dans
                    {ε} :
                    // état initial //
                        a.g := σ(a.g, V_TI, op(args));
                        a.c := σ(a.c, V_TI, op(args)) AND g;
                        si partie_droite?(a,dom) alors
                            E += {a} // ajout d'un axiome //
                        sinon
                            // erreur //
                        fsi
                    {e} :
                        // boucle //
                        a.g := σ(a.g, V_TI, op(V_TI, args));
                        a.c := σ(a.c, V_TI, op(args)) AND g AND e(V_TI);
                        si partie_droite?(a,dom) alors
                            E += {a}
                            // la boucle a un effet sur le constructeur //
                        sinon
                            // on ne peut rien dire //
                        fsi
                    K_vis :
                        // circuit : simplification du terme //
                        a.g := σ(a.g, V_TI, op(V_TI, args));
                        a.c := σ(a.c, V_TI, op(args)) AND g AND e(V_TI);
                        // la structure est conservée //
                        a.d := σ(a.d, V_TI, a.n(V_TI, args_a.n));
                        E += {a}
                    autre : // état intermédiaire //
                        a.g := σ(a.g, V_TI, op(V_TI, args));
                        // la structure est conservée : système de réécriture //
```

```
                                  // au cas où un état initial sera atteint plus tard //
                                  a.d := σ(a.d, V_TI, op(V_TI, args));
                                  a.c := σ(a.c, V_TI, op(args)) AND g;
                                  si partie_droite?(a,dom) alors
                                       a.c := a.c AND e(V_TI);
                                       E += {a}
                                  sinon
                                       G-derive-c(a,e')
                                  fsi
                       fincas
                  fait
              fsi
         fin
```

Il faut utiliser cet algorithme avec précaution, en effet, il faut toujours vérifier la cohérence des conditions booléennes, comme nous le verrons ci-dessous. Des erreurs peuvent se glisser, dues soit à une redondance entre la notion d'état abstrait et les conditions des gardes ou encore de domaine de définition. Ce ne sont pas des erreurs de modélisation, car lors de l'exécution la précondition étant fausse, l'équation ne sera pas appliquée. Cependant ces axiomes encombrent inutilement l'axiomatique.

Dans l'exemple de la pile, un seul constructeur, `dépiler` est à dériver. Il s'agit d'un cas de séquence réinitialisable. Deux transitions sont à prendre en compte. Elles sont issues d'un même état mais ne sont pas unifiables à cause de la précondition fausse.

```
G-derive-c([dépiler, NOT p1(V_TI), dépiler(V_TI),V_TI],non-vide)
  1.a- K_vis = {non-vide} op = empiler, e' = non-vide ==>
    E += {[dépiler, NOT p1(empiler(V_TI, x)) AND non-vide(V_TI), dépiler(empiler(V_TI, x)),V_TI]}
    // attention précondition fausse : cas rejeté //
  1.b- K_vis = {non-vide} op = empiler, e' = vide ==>
    E += {[dépiler, NOT p1(empiler(V_TI, x)) AND vide(V_TI), dépiler(empiler(V_TI, x)),V_TI]}
G-derive-c([dépiler, p1(V_TI), dépiler(V_TI),V_TI],non-vide)
  1.a- K_vis = {non-vide} op = empiler, e' = non-vide ==>
    E += {[dépiler, p1(empiler(V_TI, x)) AND non-vide(V_TI), dépiler(empiler(V_TI, x)),V_TI]}
  1.b- K_vis = {non-vide} op = empiler, e' = vide ==>
    E += {[dépiler, p1(empiler(V_TI, x)) AND vide(V_TI), dépiler(empiler(V_TI, x)),V_TI]}
    // attention precondition fausse : cas rejeté //
```

Dans l'exemple du compte bancaire, le seul constructeur à dériver est `déposer`. Il s'agit d'un cas de changement de valeur avec possibilité d'erreur sur le domaine de définition. Si déposer de l'argent c'est créer un compte avec comme montant l'ancien solde augmenté de l'apport, alors l'équation s'écrit simplement

`E += {[déposer, true, déposer(`V_{TI}`,m), ouvrir(numéro(`V_{TI}`), intitulé(`V_{TI}`), solde(`V_{TI}`) + m)]}`.

Cette intuition un peu rapide est fausse. En effet, en supposant que le solde fût négatif avec `solde(`V_{TI}`)+m < 0` alors un compte avec un solde initial négatif serait créé. Il faut donc G-dériver et résoudre les même problèmes que pour la pile.

c) G-dérivation des observateurs primitifs sur les générateurs Pour chaque observateur primitif, le résultat de son application aux générateurs est décrit. Pour les observateurs partiels, il faut remonter la séquence d'opérations à partir de l'état où il est défini jusqu'à trouver un générateur qui résolve l'équation. Le résultat pour la pile est :

```
Ω = {pileVide, empiler}, φ_p = {sommet, hauteur}
1. o = hauteur // opération totale // →
  1.a- axiome_hauteur := [hauteur,true,hauteur(pileVide),0]
  1.b- axiome_hauteur := [hauteur,true,hauteur(empiler(V_TI, x)),1+hauteur(V_TI, x)]
2. o = sommet // opération partielle // → G-derive-o([sommet,true,sommet(V_TI),-],non-vide)
    deux transitions : {< vide,non-vide,empiler >, < non-vide,non-vide,empiler >}
  2.a- {(vide, empiler)} → axiome_sommet = [sommet,true,sommet(empiler(V_TI, x)),-]
    ∧ partie_droite = x, E += {[sommet,vide(V_TI),sommet(empiler(V_TI, x)), x]}
  2.b- {(non-vide, empiler)} → axiome_sommet = [sommet,true,sommet(empiler(V_TI, x)),-]
    ∧ partie_droite = x, E += {[sommet,non-vide(V_TI),sommet(empiler(V_TI, x)), x]}
```

d) Ecriture des axiomes des observateurs secondaires L'algorithme de G-dérivation des observateurs principaux est repris avec dom = $\Gamma \cup \varphi_p$. Le principe est le même. Dans la pratique, nous avons remarqué qu'une G-dérivation suffit (équation simple).

e) Détermination des axiomes liés aux propriétés Cette partie est à la charge du concepteur. Il faut cependant vérifier qu'aucune incohérence n'est introduite. Une propriété est souvent exprimable par un observateur secondaire faisant appel à un constructeur de la sorte observée, ce qui ne fait qu'enrichir la spécification.

5.5 Contrôle et validation du TAG

5.5.1 Vérifications de la spécification TAG

Des vérifications sont effectuées sur les profils d'opérations ($\Gamma \cup \varphi$) :

- Chaque opération a un type résultat.
- Le type d'intéret apparaıt dans tous les profils, comme premier paramètre ou comme résultat.
- Si le type d'intéret apparaıt dans les paramètres alors il au moins est en première position.
- Chaque profil d'opération est unique dans le système et défini dans une seule présentation (*HYP1*).
- Les noms des paramètres formels constants n'apparaissent pas dans les profils des opérations.
- Il existe au moins un constructeur de base (*HYP2*).
- La cohérence avec les types importés est vérifiée : pas de dépendance cyclique avec les types des constructeurs de base (*HYP3*).

sur l'automate :

- Il existe au moins un état ($K \neq \emptyset$).
- Il n'y a pas deux transitions du même état au même état avec un même nom d'opération ($\forall k, k' \in K \cup \{\varepsilon\} \bullet \forall op \in F \bullet \forall g, g' \in F \bullet \delta(k, op, g) = k' \wedge \delta(k, op, g') = k' \Rightarrow g = g'$).
- Deux états n'ont pas le même nom.
- Aucune garde n'est égale à faux ($false \notin \Delta$).
- Pour chaque état et chaque couple de transition de même label, les gardes sont disjointes (*HYP4*). Cette propriété est indécidable (évaluateur).
- Chaque état possède au moins une transition entrante non boucle c'est-à-dire syntaxiquement accessible ($\forall e \in K, \exists e' \in K \cup \{\varepsilon\},\ op \in F,\ g \in \Delta \bullet \delta(e', op, g) = e$).
- Tous les états doivent être accessibles à partir d'au moins un état initial.
- Pour tout observateur de type T visible dans un état e et tout constructeur, c menant dans cet état, soit c possède T parmi ses paramètres formels, soit l'observateur est déjà visible dans l'état origine de la transition e (conséquence statique non triviale de la suffisante complétude).
- Les boucles ayant plusieurs paramètres en entrée doivent avoir une incidence observable dans l'état (évaluateur).
- Deux états liés par une transition ne doivent pas être équivalents (i.e. un observateur différent ou un constructeur différent).
- Il existe au moins un état initial et au moins une transition d'origine ε ($K_0 \neq \emptyset$) (*HYP2*).

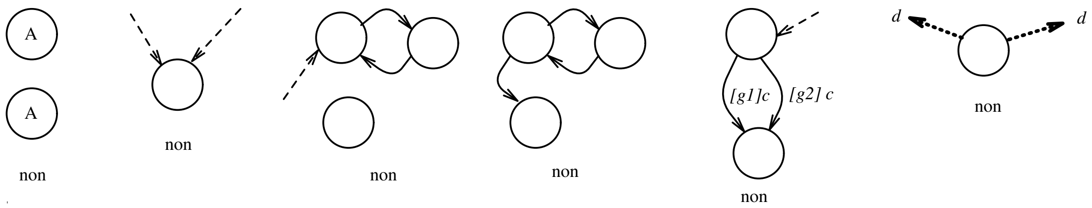


Figure 37 : Exemples d'automates refusés

ou encore sur les liens entre automate et profils :

- Chaque constructeur (resp. constructeur de base, observateur) du profil correspond à une transition entre deux états (resp. à une transition d'origine ε, à une transition d'extrémité ε).
- Chaque transition a pour label le nom d'une des opérations du profil (l'inverse n'est pas forcément vrai).
- Chaque constructeur de base a au plus une transition. Elle a pour état d'origine ε.
- Les observateurs ne provoquent pas de changement d'état.
- Les constructeurs ne sont pas représentés par des vecteurs état destination ε.

5.5.2 Propriétés de spécification

La construction de la spécification et les hypothèses posées induisent certaines propriétés de la spécification algébrique TAG.

Théorème 5.5.1
Le type d'intérêt défini par un TAG est une sorte habitée si et seulement si il existe un générateur de base dont les types des arguments sont des sortes habitées.

Démonstration :
($\Leftarrow$) Par l'hypothèse *HYP2*, la spécification possède au moins un état initial, muni d'un constructeur de base `cb : ` $\mathtt{T}_1$ `...` $\mathtt{T}_n \rightarrow$ `TI`. Par la définition même des constructeurs de base et l'hypothèse *HYP3*, le type d'intérêt n'apparaît pas (transitivement) dans les paramètres. Si les types T_i sont des sortes habitées, chaque algèbre $T_{\Sigma_{T_i}}$ contient au moins un terme clos t_i (on dit que la sorte T_i est habitée), et `cb(`$\mathtt{t}_1$`, ..., `$\mathtt{t}_n$`)` est un terme clos du type d'intérêt.
($\Rightarrow$) Soit `t:TI` un terme clos du type d'intérêt, `t` s'écrit de la forme `t = g(*)`, selon le principe de G-dérivation. Si g n'est pas récursif (i.e. ne contient pas le type d'intérêt en paramètre) alors g est un générateur de base, sinon `t` est de la forme `t = g(g'(*), args`$_g$`)` où `g'(*)` est un terme clos et par récurrence, on montre qu'il existe forcément un générateur de base, puisque `t` est un terme clos.
QED.

Nous en déduisons que les signatures de TAG sont raisonnables, sous les hypothèses *HYP2* et *HYP3*. Une autre propriété est intéressante à démontrer : chaque état décrit une sorte habitée. Nous l'avons montré pour les états initiaux. Pour les autres états on peut faire une induction sur les transitions. Cependant, à cause des gardes, on ne peut prouver que les états sont accessibles comme nous allons le montrer. Supposons que l'état origine de la transition contienne un terme clos, alors par définition, la garde est calculable et si elle est vraie alors il existe un terme clos dans l'état destination, sinon on ne peut savoir. Il faut montrer qu'elle est vraie au moins une fois.

Deux états sont équivalents s'ils acceptent les mêmes séquences d'opérations (bisimulation). La relation d'équivalence $\equiv$ entre états est définie inductivement par :

(a) $\forall k \in K,\ k \not\equiv \varepsilon$,
(b) $\forall k \in K,\ k \equiv k$,
(c) $\forall k, k' \in K,\ k \equiv k' \Leftrightarrow (\forall op \in F,\ \exists k_1, k_2, k_3, k_4 \in K \bullet$
$\delta(k, g, op) = k_1 \Rightarrow \delta(k', g, op) = k_2\ \wedge\ k_1 \equiv k_2)\ \wedge$
$\delta(k_3, g, op) = k \Rightarrow \delta(k_4, g, op) = k'\ \wedge\ k_3 \equiv k_4))$.

Cette relation est possible du fait de l'exclusivité des gardes associées aux transitions (*HYP4*). Les classes d'équivalence sont ainsi calculées pour rendre minimal un automate qui ne l'est pas.

Définition 5.5.2 (automate minimal)
Un automate $M = (F, K, \Delta, \delta, K_0, K_f)$ est minimal si tous ses états sont distincts (i.e. il n'y a pas de classes d'équivalences autres que les états eux-mêmes).

Théorème 5.5.3
L'automate TAG est minimal si et seulement si les observateurs d'état sont exclusifs.

Démonstration :
La démonstration est évidente dans la mesure où la relation d'équivalence et la construction des axiomes des observateurs d'états (qui sont des opérations totales) est la même définition basée inductivement sur la structure des termes.
QED.

Si l'automate est minimal alors l'ensemble des états forme une partition des valeurs du type d'intérêt (au sens du `partitionned by` de Guttag [LG90]). Un axiome est une équivalence conditionnée entre deux chemins de l'automate. Cette propriété découle de la méthode de construction des axiomes.

Intéressons-nous maintenant aux propriétés de complétude et de cohérence. En général, deux alternatives sont utilisées pour montrer ces propriétés : le raisonnement équationnel et ses extensions ou les systèmes de réécriture[JL86]. Nous utiliserons la seconde alternative pour deux raisons : premièrement la forme des axiomes permet assez naturellement de remplacer les axiomes par des règles de réécriture conditionnelle, deuxièmement les systèmes de réécriture possèdent des algorithmes puissants pour corriger un système qui n'a pas les propriétés souhaitées (i.e. Knuth-Bendix). Un autre intérêt des systèmes de réécriture est de pouvoir exécuter les spécifications et ainsi établir des tests et des simulations pour valider la spécification (voir section 5.5.4). L'inconvénient des systèmes de réécriture est une diminution de la puissance d'expression, par rapport au raisonnement équationnel. Notons encore, que dans notre cas, une troisième alternative consiste à traduire dans un langage de spécification algébrique existant, et montrer ces propriétés en utilisant les outils de l'environnement du langage cible. La réécriture conditionnelle permet de prendre en compte les axiomes et les préconditions [JL86, Kap86, KR90]. Les systèmes de réécriture ont été étendus pour prendre en compte la commutativité et les définitions partielles d'opérations.

Supposons les équations orientées de gauche à droite. Chaque axiome $\texttt{t}_1$ `==` $\texttt{t}_1$`' & .. &` $\texttt{t}_n$ `==` $\texttt{t}_n$`' ==> l == r` devient une règle de réécriture, $\texttt{t}_1$ `==` $\texttt{t}_1$`' & .. &` $\texttt{t}_n$ `==` $\texttt{t}_n$`' ==>` $\texttt{l} \rightarrow \texttt{r}$. Une réécriture est un remplacement d'un sous-terme d'un terme par un autre terme selon une règle du système de réécriture. Un terme est irréductible si aucune réécriture ne peut lui être appliquée. Les formes irréductibles (normales) sont calculées à partir des termes par réécriture successive. La réécriture permet de démontrer un théorème équationnel $t = t'$ par un calcul "t et t' ont la même forme normale" (théorème de Church-Rosser) ou encore de prouver des théorèmes inductifs.

Cette affirmation est valide si le système de réécriture est **convergent**. Cette propriété est la conjonction de deux propriétés : la **terminaison** et la **confluence**. La terminaison exprime le fait que les calculs sont finis et le système de réécriture est dit **noethérien**. Cette propriété est indécidable. Elle est démontrée en exhibant un ordre de réduction sur les termes. La confluence exprime le fait que tout terme se réécrit en un seul terme irréductible, quelque soit l'ordre d'application des règles de réécriture. Si le système est noethérien, il suffit alors de montrer la confluence locale en étudiant uniquement les règles de réécriture. Une paire critique est un couple de règles tel que un terme quelconque puisse se réécrire indifféremment par l'une ou l'autre des règles. Le système est localement confluent si les paires critiques sont confluentes.

Théorème 5.5.4 (confluence locale)
Le système de réécriture associé à une spécification algébrique TAG est localement confluent.

Démonstration :
Cette propriété découle de l'algorithme de construction automatique des parties gauches des axiomes. Il y aura une paire critique si pour un état et une opération donnés, deux transitions sont possibles. Cependant à cause de l'hypothèse *HYP4*, si deux transitions sont possibles leurs

gardes sont mutuellement disjointes. Donc un seul des axiomes est applicable.
QED.

Un terme clos vis-à-vis du type d'intérêt est un terme écrit uniquement à partir des générateurs du type d'intérêt. Le système de réécriture associé à un TAG termine si un ordre sur ces générateurs est exhibé. S'il existe un unique état initial, alors un parcours de l'automate en profondeur d'abord donne un ordre sur les générateurs, l'ordre contient un plus petit élément, qui est le constructeur de base associé à l'état initial. Cette intuition reste à démontrer formellement.

5.5.3 Typage

Le typage est un outil essentiel dans la formalisation et le contrôle des spécifications. Le typage est une analyse de cohérence qui vérifie que les opérations qu'on utilise sont déjà définies et que les valeurs qu'on leur applique ont un sens [WL93]. Le modèle TAG est fortement typé. Cela signifie que toutes les expressions ont un type cohérent même si le type lui-même n'est pas connu. Le programme s'exécutera sans erreur de type (i.e. un type employé dans un mauvais contexte). Dans les TAG, le système de type du noyau est simple, mais la prise en compte de l'héritage modifie fortement ces règles (voir section 5.6.2). Le système de type des TAG est proche de celui des classes formelles (voir section 6.5).

5.5.4 Validation et prototypage

“La correction d'une spécification est une notion qui n'est pas complètement formalisable” [Bid82]. Savoir si la spécification correspond réellement aux besoins est une tâche de la validation externe (prototypage par exemple). L'automate est à la fois un outil de preuve dans la spécification algébrique et dans les systèmes de transition. Il joue le rôle de générateur de termes. Par exemple, il existe un chemin dans l'automate qui donne le terme suivant :

```
level(arrived(up(
    arrived(down(
        stop(restart(arrived(up(
            install(X1,X2,X3)
        , X4))))
        , X5))
    , X6)))
```

avec `X1, X2 : Integer`; `X3 : RealGt1`; `X4, X5, X6 : IntegerGt1` et sous réserve du respect de l'invariant et des gardes.

Nous avons vu un certain nombre de contrôles syntaxiques. L'utilisation d'un environnement de spécification algébrique permet d'autres vérifications, notamment celles de complétude et consistance, et la réalisation de preuves. L'objectif est ici de rendre exécutable la spécification. Deux modes sont possibles : définir un évaluateur symbolique ou traduire dans un autre langage.

Evaluateur symbolique L'évaluateur symbolique est définit par un système de réécriture, tel que Reve ou LarchProver. Les preuves sont des théorèmes équationnels ou inductifs. Une version personnalisée, similaire à l'évaluateur des classes formelles dans la section 6.4.2, devra prendre en compte l'héritage (section 5.6.2).

Traduction Nous avons expérimenté des traductions en ASSPEGIQUE [Cho88]. L'intérêt majeur est de pouvoir *réutiliser* un environnement de spécification et de preuve existant. Un exemple de traduction (manuelle) et de preuve est donné en annexe A.6. Le cas des spécifications non hiérachiques doit être traité à part, car le langage PLUSS utilise cette contrainte. La solution retenue est de définir l'interaction entre les deux par une troisième spécification. Une autre, plus générale, est de passer par des identités pour les objets pour éviter les dépendances mutuelles. Ces solutions restent à concevoir. Si le langage permet l'héritage alors la traduction est directe, sinon la spécification est d'abord mise à plat. Une traduction dans le langage prolog est envisagée.

5.6 Extensions

5.6.1 Structuration des automates

Lorsque le nombre d'états est complexe, il est nécessaire de structurer l'automate pour faciliter sa lecture et son écriture. La structuration est basée sur l'agrégation et la généralisation d'états ou de transitions. L'agrégation permet de décrire à un niveau plus fin un sous-ensemble relativement indépendant de l'automate. La généralisation factorise les comportements communs. Les problèmes de structuration d'automate sont proches de ceux du sous-typage des TAG et de leur composition. Contrairement à d'autres formalismes tels que les statecharts [HLNP90], et dans le but d'assister l'utilisateur nous avons choisi des structures calculables en plus de celles qu'il peut proposer.

a) Généralisation d'états

Les transitions similaires sont généralisées dans des super-états.

Définition 5.6.1 (transition similaire)
Deux transitions `t=<op, g, orig, dest>` *et* `t'=<op, g, orig', dest'>` *ayant même opération et même garde sont dites* **similaires** *ssi elles ont même destination* (`dest = dest'`) *ou si ce sont des transitions an boucle* (`orig = dest` $\wedge$ `orig' = dest'`).

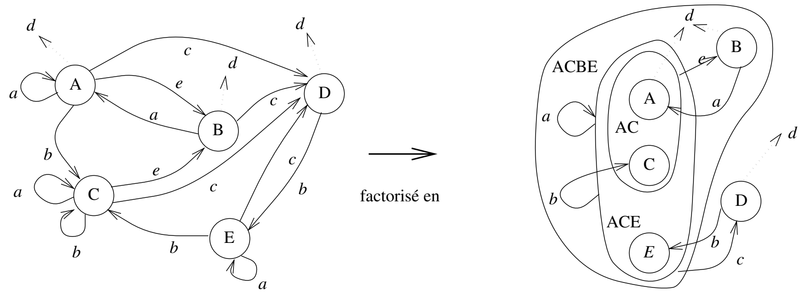


Figure 38 : Généralisation de transition

b) Composition d'états

Un état composite regroupe des états fortement liés. Les transitions associées deviennent des boucles conditionnées par une garde interne. L'ensemble $succ(e)$ des successeurs directs de l'état e est défini par $succ(e) = \{s \in K \mid \exists g \in \Delta,\ \exists op \in F \bullet \delta(e, g, op) = s\}$ (rem. $\varepsilon \notin K$).

Composantes fortement connexes La fermeture réflexive, symétrique et transitive de la relation *succ* définit des classes d'équivalence, appelées composantes fortement connexes. Le graphe réduit forme un automate simplifié. Les états des composantes sont des sous-automates.

Etats frères En pratique, le graphe est souvent fortement connexe (une seule composante) car les automates décrivent des cycles de vie en général réinitialisables. Nous proposons donc une relation plus souple, appelée *is_brother*.

Définition 5.6.2 (états frères)
$\forall k, k' \in K,\ k\ is\_brother\ k' \Leftrightarrow succ(k) = \{k'\} \ \wedge\ k \in succ(k')$.

L'algorithme de restructuration est alors le suivant. L'application de la relation produit des groupes d'états. Ce ne sont pas des classes car ils ont des intersections non-vides. Pour chaque état d'une intersection non vide, il est demandé à l'utilisateur de choisir le groupe d'affectation. Cet algorithme est ré-itéré jusqu'à ce que le graphe soit suffisamment simple.

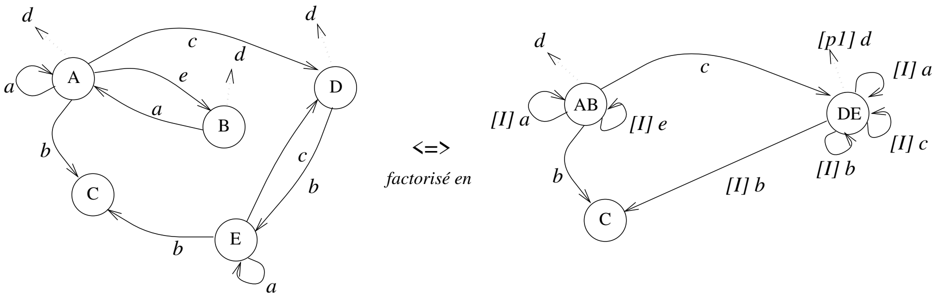


Figure 39 : Etats frères

Composition de transitions Une deuxième manière de factoriser l'automate est de calculer des transitions séquentielles ou parallèles. L'algorithme de composition de transition est la fermeture transitive de l'application successive des deux définitions ci-dessous.

Définition 5.6.3 (transitions séquentielles)
Deux transitions `t=<op, g, orig, dest>` *et* `t'=<op', g', orig', dest'>` *sont dites* ***séquentielles****, noté* $t;t'$*, si elles se recoupent en un unique état* `e` *tel que* `(e = dest = orig')` *et que* $succ^{-1}(e) = \{orig\}$ *et* $succ(e) = \{dest'\}$.

Définition 5.6.4 (transitions parallèles)
Deux transitions `t=<op, g, orig, dest>` *et* `t'=<op', g', orig, dest>` *sont dites* ***parallèles****, noté* t,t'*, si elles ont même origine et même destination.*

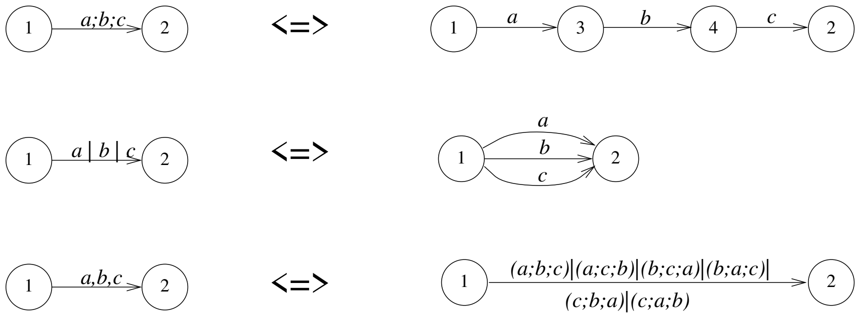


Figure 40 : Agrégation d'opérations

c) Produit synchronisé d'automates

Un produit synchronisé [Arn92] assemble des parties relativement indépendantes. L'état global est le produit de l'ensemble des états de chacun des sous-automates. Les contraintes de synchronisation sont exprimées par un vecteur de synchronisation. Ceci constitue une introduction à la concurrence entre TAG.

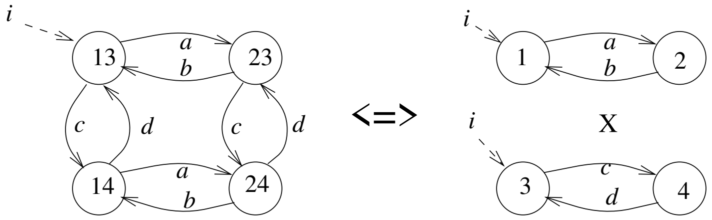
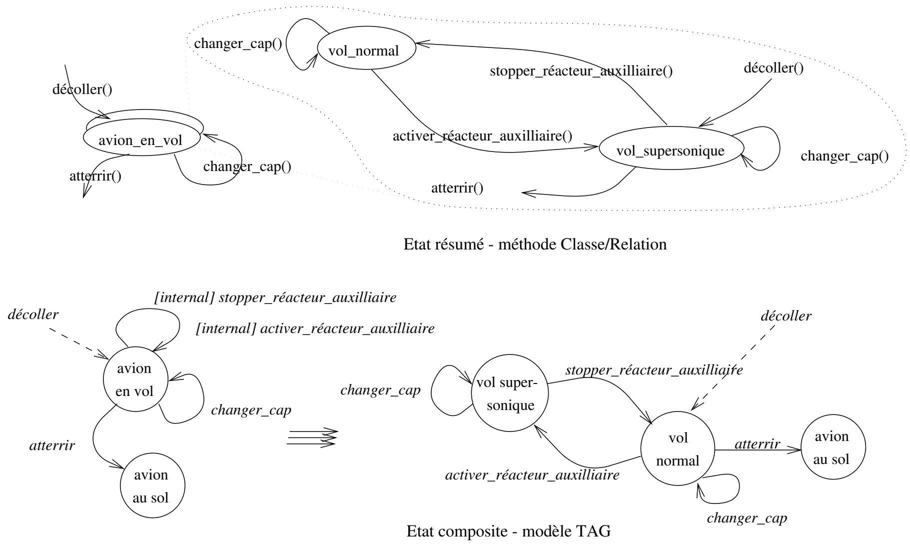


Figure 41 : Produit d'automates

d) Autres notations

Un état ou une transition composite peuvent encapsuler un sous-ensemble relativement complexe de l'automate. Contrairement aux factorisations précédentes, il s'agit d'une facilité uniquement syntaxique car certaines transitions sont cachées. Cette encapsulation forte des états est donnée par la notion d'état résumé dans la méthode classe-relation [Des92] ou les états emboités d'OMT [RBP$^+$91].



Etat résumé - méthode Classe/Relation

Etat composite - modèle TAG

Figure 42 : Etat composite

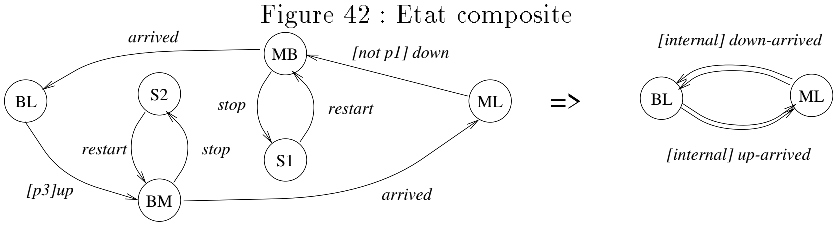


Figure 43 : Transition composite

Effets sur le modèle fonctionnel

Ces ajouts syntaxiques ne sont pas encore utilisés dans la construction des axiomes mais nous avons constaté sur des exemples qu'elles simplifient l'axiomatique en regroupant des paquets d'axiomes et simplifiant les prédicats d'états et les préconditions.

5.6.2 Héritage

L'héritage de TAG est souhaitable pour construire incrémentalement des spécifications, partager les axiomes et classer les spécifications en hiérarchies de spécialisation. Comme nous l'avons décrit dans la section 2.3.1, l'héritage est une notion complexe, qui couvre à la fois le sous-typage, le raffinage, l'extension ou la redéfinition. Dans les approches théoriques [Bre91, CO88, PPP91], l'héritage est défini comme une inclusion de modèles satisfaisant les spécifications, munie d'un morphisme entre les signatures, plus ou moins forte selon qu'on considère des spécifications complètes ou non, le sous-typage ou non. L'héritage peut aussi être vu comme une inclusion de langages engendrés par les automates. Dans un premier temps, nous nous s'attacherons à l'aspect fonctionnel des TAG. Nous souhaitons une relation pratique et vérifiable pour les actions suivantes :

- ajouter une opération supplémentaire,

- renforcer la contrainte,
- renforcer la précondition d'une opération,
- restreindre un type de la relation d'importation *use* à un sous-type,
- restreindre le type du paramètre formel.

Les deux derniers cas sont équivalents si la généricité est simulée par l'héritage. Dans ce cas, les profils des opérations concernées seront substitués.

Nous souhaitons aussi que l'héritage implique le sous-typage en respectant le principe de substitution : "toute opération applicable au super-type est applicable au sous-type"[5]. Le contrôle de type et le polymorphisme ne seront pas étudiés ici. Les solutions adoptées pour les classes formelles peuvent être adaptées aux TAG (voir section 6.5). Quatre cas d'héritage sont dénombrés pour les spécifications algébriques :

1. extension : spécialisation d'une spécification complètement définie (e.g. Pixel inherit Point),

2. concrétisation : spécialisation d'une spécification incomplète (e.g. Natural inherit Comparable),

3. raffinage : changement de représentation (e.g. SetList inherit Set),

4. duplication : définition par renommage des sortes et des opérations (e.g. HospitalQueue inherit FIFOQueue).

Contrairement aux classes, les axiomes sont pas affectés directement aux opérations. Nous utiliserons les termes génériques sous-spécification et super-spécification de la relation d'héritage. Les quatre cas ci-dessus se résument à deux situations : si la super-spécification est finiment engendrée alors c'est une **extension** sinon c'est une **concrétisation**. Les spécifications non finiment engendrées correspondent aux classes abstraites des modèles à objets i.e. sans constructeurs de base. Les solutions ci-dessous sont applicables pour une évaluation par valeur. Les axiomes relatifs à une opération sont mis dans une seule spécification.

PLUSS [BGM87] sépare spécifications complètes et incomplètes mais ne définit pas d'héritage de spécifications complètes. OBJ [FGJM85] définit uniquement le sous-typage strict, par inclusion d'ensembles supports. La notion de sous-classe en GSBL [CO88] correspond à l'héritage bien que les auteurs distinguent définitions complètes et incomplètes de sortes ou même d'opérations. Le sous-typage d'OS/OP [Bre91] est une notion plus forte que la concrétisation des TAG, car elle est définie sur des spécifications de classes complètes, tandis que l'héritage d'OS/OP est équivalent à l'extension. Dans le langage NDL [PPP91], les auteurs séparent sous-typage, héritage de spécialisation et héritage d'implantation. La différence entre les deux premiers porte uniquement sur une partie interface de classe, correspondant à des opérations redéfinissables dans les sous-classes.

Héritage d'extension

Dans OS/OP, l'héritage est un morphisme de signature (renommage) de la spécification ancêtre vers la spécification héritière et une inclusion de modèles. Notre définition est plus contraignante et plus proche de la notion de coercition de type [CW85], qui permet de réutiliser une opération définie dans une super-spécification. Pour définir l'héritage d'extension, nous utilisons une fonction d'abstraction de la sous-spécification vers la super-spécification.

Définition 5.6.5 (héritage d'extension)
Soient $spec_1 = (TI_1, S_1, \Gamma_1, \varphi_1, X_1, E_1)$ *et* $spec_2 = (TI_2, S_2, \Gamma_2, \varphi_2, X_2, E_2)$ *deux spécifications TAG.* $spec_1$ *hérite par extension de* $spec_2$ *si et seulement si il existe une fonction surjective* $f_{abst} : T_{\Sigma_{TI_1}} \to T_{\Sigma_{TI_2}}$ *telle que* $\forall t_1 \in T_{\Sigma_{TI_1}},\ \exists t_2 \in T_{\Sigma_{TI_2}} \bullet f_{abst}(t_1) = t_2$

[5] Si elle n'a pas été redéfinie dans la sous-classe.

La preuve de l'héritage d'extension se fait en exprimant chaque générateur de la sous-spécification comme une extension[6] des générateurs de la super-spécification. Prenons un exemple : un client de la banque est une personne ayant un numéro de client. Une personne est modélisée en TAG par un automate à deux états (A : anonymous, N : named). Un état est ajouté dans l'automate du client, signalant la présence d'un identifiant de client. Les spécifications TAG correspondantes sont les suivantes :

```
name       : Person → String
anonymous  : → Person
rename     : Person String → Person
equal      : Person Person → Boolean
A          : Person → Boolean
N          : Person → Boolean
// Ω = { anonymous, rename } - deux générateurs //
// ∂ = { name } - opération définie sur l'état N //
// S { Person, Boolean, String} //
∀ p : Person; n : String
A1: A(anonymous) == true
A2: A(rename(Self, n)) == false
N1: N(anonymous) == false
N2: N(rename(Self, n)) == true
name1: A(Self) == true ==> name(rename(Self, n)) == n
name2: N(Self) == true ==> name(rename(Self, n)) == n
equal1: equal(anonymous, p) == A(p)
equal2: equal(rename(Self, n), p) == N(p) AND equal(n, name(p))

anonymousC : → Client
rename     : Client String → Client
setId      : Client Nat → Client
id         : Client → Nat
// Ω { anonymousC, rename, setId } : trois générateurs //
// ∂ { id } : opérations définies sur l'état C //
∀ p : Client; n,n's,s' : String; i : Nat
C1: C(anonymousC) == false
C2: C(rename(Self, n)) == C(Self)
C3: C(setId(Self, i)) == true
id1: N(Self) == true ==> id(setId(Self, i)) == i
id2: C(Self) == true ==> id(setId(Self, i)) == i
id3: C(Self) == true ==> id(rename(Self, n)) == id(Self)
```

Montrons maintenant que la spécification des clients hérite par extension de celle des personnes. Soit la fonction d'abstraction suivante :

$\forall$ `Self : Client, n,s : String, i : Nat,`
$\mathtt{abs}_{Client \rightarrow Person}$`(anonymousC) = anonymous,`
$\mathtt{abs}_{Client \rightarrow Person}$`(rename(Self, n)) = rename(`$\mathtt{abs}_{Client \rightarrow Person}$`(Self), n),`
$\mathtt{abs}_{Client \rightarrow Person}$`(setId(Self, i)) = `$\mathtt{abs}_{Client \rightarrow Person}$`(Self).`

La fonction $\mathtt{abs}_{Client \rightarrow Person}$ est surjective, compte tenu de la contrainte et des préconditions, car les générateurs ont les mêmes domaines de définition. L'évaluation du terme clos suivant est alors possible, via la fonction d'abstraction.

`name(setId(rename(anonymousC, 'Ted'))) // name défini dans Person //`
$\Rightarrow$ `name(`$\mathtt{abs}_{Client \rightarrow Person}$`(setId(rename(anonymousC, 'Ted')))) // abstraction //`
$\Rightarrow$ `name(`$\mathtt{abs}_{Client \rightarrow Person}$`(rename(anonymousC, 'Ted'))) // application de abs //`
$\Rightarrow$ `name(rename(`$\mathtt{abs}_{Client \rightarrow Person}$`(anonymousC), 'Ted'))) // idem //`
$\Rightarrow$ `name(rename(anonymous, 'Ted'))) // idem //`
$\Rightarrow$ `'Ted' // name1 //`
`QED.`

[6] cf définition 5.4.7.

Concrétisation

Lorsqu'une spécification concrétise une spécification non finiment engendrée, la fonction d'abstraction est l'identité.

Définition 5.6.6 (héritage de concrétisation)
Soient $spec_1 = (TI_1, S_1, \Gamma_1, \varphi_1, X_1, E_1)$ *et* $spec_2 = (TI_2, S_2, \Gamma_2, \varphi_2, X_2, E_2)$ *deux spécifications TAG,* $spec_1$ *concrétise* $spec_2$ *si et seulement si* $\forall t_1 \in T_{\Sigma_{TI_1}} \bullet t_1 \in T_{\Sigma_{TI_2}}$.

Il s'agit donc de montrer que toutes les opérations de la super-spécification s'appliquent aux termes de la sous-spécification. Premièrement, chaque générateur de la super-spécification est exprimé en fonction des générateurs de la sous-spécification. De tels générateurs correspondent en quelques sorte à des méthodes abtraites dans les modèles à objets. Deuxièmement, toutes les extensions G-dérivées sont complétées avec les nouveaux générateurs. Prenons un exemple simple: "un comparable est un type possédant une relation d'ordre". Dans la spécification suivante des comparables, `<=` et `=` sont les générateurs.

```
eq           : Comparable Comparable → Boolean
leq          : Comparable Comparable → Boolean
gt           : Comparable Comparable → Boolean
∀ Self, a, b : Comparable
gt1: gt(Self, a) == NOT leq(Self, a)
```

Les entiers sont des comparables dont les générateurs sont redéfinis, il n'y a pas d'extension G-dérivée.

```
zero         : → Nat
succ         : Nat → Nat
eq           : Nat Nat → Boolean
leq          : Nat Nat → Boolean ∀ Self, a: Nat
eq1: eq(zero, succ(Self)) == false
eq2: eq(succ(Self), zero) == false
eq3: eq(succ(Self), succ(a)) == eq(Self, a)
leq1: leq(zero, succ(Self)) == true
leq2: leq(succ(Self), zero) == false
leq3: leq(succ(Self), succ(a)) == leq(Self, b)
```

Ainsi l'évaluation suivante est possible, en appliquant les axiomes de la super-spécification :

```
gt(succ(succ(zero)), succ(succ(succ(zero))))
⇒ gt(succ(succ(zero)), succ(succ(succ(zero)))) // par la fonction abstraction identité //
⇒ NOT leq(succ(succ(zero)), succ(succ(succ(zero)))) // gt1 //
⇒ NOT leq(succ(zero), succ(succ(zero))) // leq3 //
⇒ NOT leq(zero, succ(zero)) // leq3 //
⇒ NOT true // leq1 //
⇒ false // not1 //
QED.
```

Une erreur de type serait apparue si le second argument n'était pas de type `Nat`.

Héritage et automate

Il est intéressant de définir la relation d'héritage sur le comportement dynamique . Une règle souvent utilisée est la similarité de trace (équivalence observationnelle) [Smi93]. Cependant, elle est difficile à établir à cause des gardes et des ajouts d'états. Dans [PR94], les états sont définis par des conditions sur les attributs. La spécialisation d'automates est une réduction de l'espace des états, un raffinement de la partition ou une réduction du non-déterminisme.

Nous proposons ici encore une deux relations : le sous-typage et l'héritage. En considérant les états des TAG comme des ensembles de valeurs, le sous-typage est défini comme une suppression d'états ou du moins une restriction des valeurs d'un état i.e. états et transitions en moins, gardes renforcées. Par exemple, une pile non vide est un sous-type de la pile de la figure 33, dans laquelle l'état `vide` n'existe pas. Le constructeur `cons`, renommé en `pileSingleton`, devient un générateur de base. Mais le sous-typage est insuffisant pour les actions de la page 124, une notion d'héritage est nécessaire. L'héritage s'exprime par un éclatement des états et des transitions. C'est ainsi qu'une spécification de l'hôpital de la figure 30 est obtenue à partir d'une liste bornée. L'automate de comportement dynamique de la sous-spécification contient au moins les mêmes états (qui peuvent maintenant être des super-états) que l'automate de comportement dynamique de la super-spécification. Les notions d'extension et de concrétisation se retrouvent encore à ce niveau pour calculer des chemins dans l'automate. Dans les deux cas, l'invariant doit être conservé, de même qu'un sous-typage des paramètres formels et des opérations. Héritage d'automate, structuration d'automates et simulation sont fortement liés.

5.6.3 Généricité

Les spécifications TAG peuvent être paramétrées par des constantes et des types (section 5.2.3). L'actualisation d'un paramètre formel constant est faite lors de la création d'une valeur de ce type. L'actualisation d'un type générique définit un nouveau type. Elle se fait en substituant le paramètre par un autre type. Le nouveau type doit être déjà défini, de telle sorte que les opérations définies pour le paramètre formel existent dans la spécification du paramètre effectif.

La sémantique de la généricité des TAG peut être donnée au travers de l'héritage et de la relation d'importation [Bre91] : le type générique est un client du type paramètre (relation *use*) et il peut être spécialisé par héritage. Actuellement, nous nous contentons de vérifications syntaxiques : les opérations déclarée doivent être implantées par le type effectif. Si le langage algébrique cible permet la généricité, alors la sémantique et les contrôles sont donnés dans ce langage. Sinon, la généricité est purement syntaxique, comme en Ada et le paramètre effectif est donné par une importation de sorte (USE).

5.6.4 Concurrence

Le comportement dynamique d'un système est une relation entre temps et événements. Il peut être affiné par une notion d'événement interne ou observable, par des événements en entrée ou en sortie exprimant les communications entre le système et son environnement ou encore par une notion d'état, qui varie au cours du temps [KS90]. Le temps peut être exprimé concrètement par un temps réel [Kna94] ou abstraitement par un ordre sur les événements. Le contrôle des objets peut être centralisé au niveau des objets ou décentralisé dans les méthodes [Car91].

Nous nous plaçons dans le contexte simple suivant. Un événement est un appel d'opération. Le temps est exprimé par un ordre partiel sur les événements. Il y a non déterminisme entre les événements, les paramètres des opérations mais pas sur les changements d'état pour un événement donné pour l'instant. Le contrôle des objets est explicite et se traduit par un ordre partiel sur les événements internes et les événements externes. La communication se fait par envoi de message synchrone. Aucune contrainte temps réel n'est prise en compte, comme dans VDM++.

Un automate TAG peut être interprété comme un système de transition[Arn92]. Une trace est une suite d'appel d'opérations. Dans les systèmes de transition, un état final correspond à une exécution possible, c'est-à-dire à un ensemble de traces acceptées. La notion d'état final n'a pas de sens dans l'interprétation fonctionnelle, mais elle peut exister dans le modèle objet et correspond aux destructions d'objets.

La difficulté majeure est l'expression de la synchronisation et de la communication par envoi de message. Les envois de message ne nécessitant pas de synchronisation sont réalisés par des appels fonctionnels. La synchronisation est traitée par le produit synchronisé d'automates [Arn92]. Le système est exprimé par un produit d'automates et contraint par un vecteur de synchronisation exprimant le fait que certaines opérations se passent en même temps. La précondition globale est le produit des préconditions de chaque partie. Le passage de paramètres se fait aussi dans le vecteur de synchronisation, soit par variable commune, soit en donnant comme paramètre le résultat d'une autre opération. Dans ce choix de modélisation les synchronisations et communications ne sont pas exprimées dans les objets mais dans le système qui les contient. Cette approche convient bien à l'aspect fonctionnel des TAG. Consulter [Gue94, Huf89, Kna94, Mes90, Vau85] pour d'autres approches algébriques de la concurrence.

Une autre version de la concurrence dans les TAG est possible pour décentraliser le contrôle global. A chaque objet du système est affecté une identité. Les envois de messages se font non plus avec les valeurs fonctionnelles mais des identités d'objets pour les receveurs. Un envoi de message est un événement interne, produit par l'objet lui-même. Ces événements internes sont rajoutés à l'automate et symbolisées dans la figure 44 par des transitions barrées. Un produit synchronisé simple relie les différents automates en reliant les transitions de même nom. Le passage des paramètres se fait par appariement des noms de paramètres. Le comportement observationnel est la restriction de l'ensemble des traces aux événements externes.

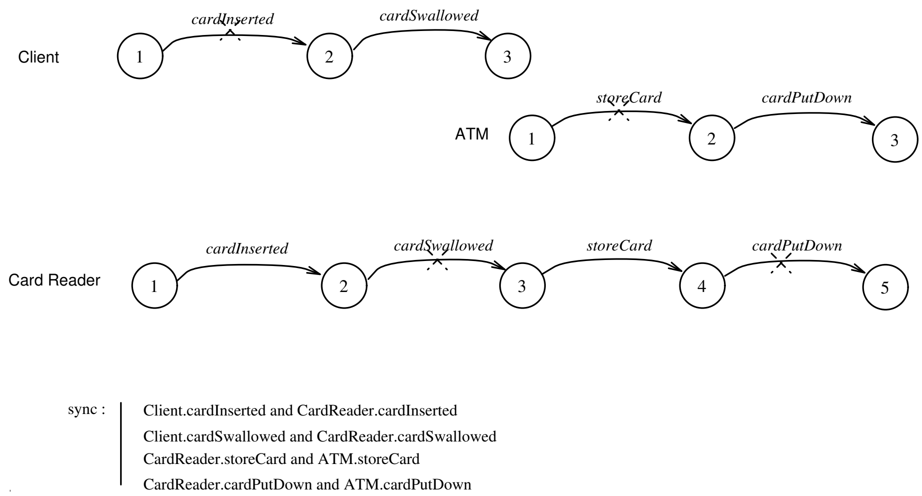


Figure 44 : Projection/Synchronisation de transitions

5.7 Conclusion

Nous avons introduit dans ce chapitre une manière originale de construire des types abstraits de donnée : le modèle TAG. Ce modèle est basé sur la notion d'état abstrait, ce qui donne une représentation à la fois plus opérationnelle que celle des types abstraits algébriques purs mais plus naturelle et plus facile à construire, et à la fois plus abstraite que celle des spécifications par états ou même à objets. Contrairement à d'autres approches utilisant un formalisme basé sur les systèmes de transitions et les types abstraits algébriques (réseaux algébriques hiérarchiques [Gue94], LOTOS [GLO91] ou d'autres [Dia92]), les deux formalismes ne sont pas disjoints. Dans ces approches, les processus et les types de données sont spécifiés distinctement. Les types de données définissent formellement les informations circulant entre processus.

L'enrichissement des spécifications par ajout/extension d'états et de transitions permet une définition incrémentale des composants du système. Ainsi, la modélisation de traitements d'erreurs est possible soit par des assertions soit par des états d'erreur, elle est immédiate et visuelle. Compte tenu des hypothèses, la spécification possède bonnes propriétés, mais un travail non négligeable reste à faire de la part du concepteur. Le noyau présenté ici doit être étendu afin de faciliter la structuration des spécifications et la concurrence.

Les TAG sont particulièrement utiles dans la spécification algébrique de systèmes à états avec plus de trois états. La modélisation des cas d'erreur et des traitements d'exception est naturelle et ne perturbe pas le traitement principal. Pour les structures de données la spécification obtenue est un peu plus riche, mais équivalente.

Ce style de spécification permet de construire des spécifications algébriques plates et des spécifications algébriques ordonnées. En conséquence, c'est une bonne introduction au passage entre spécification algébrique et modèle objet à classes formelles du chapitre 7. La méthode peut sembler un peu lourde sur de petits exemples. Mais l'aspect semi-automatique de la méthode rend la production des axiomes plus facile pour le non spécialiste, et la description est plus naturelle. Toutefois cette lourdeur exprime le besoin d'outils de simplification d'expressions logiques complexes. Par exemple, si nous prouvons que les observateurs d'état sont une extension de certains observateurs totaux, alors d'une part, les axiomes qui leur sont associés sont supprimés car redondants et d'autre part bon nombre d'expressions logiques seront simplifiées dans les axiomes.

Nous avons privilégié la distinction entre modèle générique et langages existants. Des schémas de traduction sont aisés à mettre en place pour décrire la spécification algébrique issue du TAG dans un langage ou un environnement, tels que ASSPRO, LPG, Larch, SACSO... (voir section 2.2.6) en nous inspirant par exemple de l'architecture SALSA [BBC+92].

Chapitre 6

Classes formelles

"N'admettez rien a priori si vous pouvez le vérifier."
R. Kipling, "Souvenirs"..

6.1 Introduction

Dans ce chapitre, nous présentons les bases d'un modèle formel pour les objets, dédié à la conception formelle pour les langages à objets avec classes. Un modèle abstrait et formel permet de poser les bases d'outils de preuves, de contrôles, de génération de tests, etc. L'intérêt de ce modèle réside dans son indépendance vis à vis des langages de programmation avec classes comme Eiffel, C++, Smalltalk ou CLOS. Un but important est de définir un modèle minimal commun à ces langages. Une spécification avec ce modèle va donc permettre de comparer des mises en œuvre différentes dans un même langage ou entre différents langages. Les apports attendus en termes de qualité sont une amélioration de la fiabilité, de la portabilité entre langages, de l'automatisation du codage, de la réutilisation de composants spécifiés formellement. Enfin, il sert de base à l'étude de l'héritage et du typage dans les langages à objets.

Ce modèle est basé sur les types abstraits algébriques pour différentes raisons : proximité des concepts de classe et type abstrait, théorie bien établie maintenant, représentation abstraite des structures de données, et aussi pour garder une certaine cohérence avec les modèle des TAG. Plus précisément une classe s'interprète comme une spécification algébrique d'un type de données constituée d'une spécification projective, d'une éventuelle contrainte et d'extensions successives. Notre démarche se veut avant tout pragmatique.

Dans un premier temps, nous allons définir un modèle simple avec les caractéristiques suivantes : les objets sont des valeurs fonctionnelles, les méthodes ne sont pas des objets, il n'y a pas de méta-classe, le type algébrique associé à la classe est mono-générateur, la sélection des méthodes est simple, la redéfinition des méthodes est possible en suivant une règle de simple covariance i.e. covariante sur le receveur et le résultat uniquement, chaque classe décrit un unique aspect. Ensuite, nous inclurons d'autres concepts.

La première section présente les concepts principaux et définit formellement la notion de classe et quelques propriétés associées. La deuxième section est consacrée à l'héritage des classes formelles. La troisième section s'intéresse à quelques aspects de l'évaluation symbolique et de la déduction. Ces aspects opérationnels sont utiles à l'étude du typage, dont nous présentons une version simple dans la quatrième section. La dernière section propose quelques extensions : multi-aspect, effets de bord, la multi-sélection, métaclasses et généricité. Les notations algébriques sont celles des TAG et données en annexe E.

6.2 Définition et interprétation des classes formelles

Dans cette section nous formalisons la notion de classe formelle en faisant abstraction de l'héritage qui sera introduit dans la section suivante. Nous faisons le lien entre une classe et un type abstrait de données. Ceci conduit à une écriture algébrique des méthodes et à des critères de non-contradiction et de complétude suffisante inspiré des spécifications algébriques. Une classe vue de façon suffisamment abstraite est une spécification algébrique particulière. Une telle classe est un niveau intermédiaire entre TAD et classe concrète. L'intérêt d'un tel niveau est qu'il permet de définir des notions propres aux objets comme l'héritage. Nous commençons par donner informellement les principaux concepts du modèle.

6.2.1 Principaux concepts

Le modèle des classes formelles est un modèle à objets, définissant un langage. Un objet est une valeur à part entière du langage. D'autres valeurs peuvent être considérées, elles sont définies par des types de base. Dans la plupart des langages concrets et en fait dans tous systèmes à classes il est indispensable d'avoir des types de base. Ces types ne sont pas des classes. Il permettent de définir des valeurs nécessaires à la mise en œuvre ou à l'optimisation du langage. Par exemple : CLOS : les *built-in-class*; ST-80 : SmallInteger, ...; C++ : les types usuels de C; Eiffel V 2.3 : Integer, Char, d'une façon générale les *expanded class*.

En programmation à objets, la description des objets du système précède la spécification de ses fonctionnalités (les opérations associées). La phase de description des objets a pour but la caractérisation d'une façon non ambiguë des fragments de connaissance pertinents dans le cadre de l'application. La définition des fonctionnalités dans une classe se fait incrémentalement, c'est un avantage fondamental dans le cadre du développement logiciel.

Objets

Un objet est composé d'une description abstraite et d'un comportement. La description abstraite est un ensemble de **caractéristiques**. Le **comportement** d'un objet décrit l'ensemble des opérations connues de l'objet, appelées **méthodes**. Exemple : un rectangle { `largeur = 10, longueur = 20, périmètre, surface ...` }. Celui-ci peut être implantée par diverses structures concrètes plus ou moins efficace ou minimale. Par exemple :

`largeur`	`10`
`longueur`	`20`

`périmètre`	`60`
`surface`	`200`

`largeur`	`10`
`perimetre`	`60`

`largeur`	`10`
`longueur`	`20`
`périmètre`	`60`

etc.

Chaque objet possède une unique classe qui lui donne son type le plus spécifique. Les objets sont créés dynamiquement à partir de la classe par instanciation. La description abstraite permet de comparer deux objets d'une même classe. Le comportement d'un objet est activable par envoi de message.

Méthodes

Une méthode est définie par un profil et des axiomes algébriques comme en spécification algébrique ou en programmation fonctionnelle. Le nom de la méthode est appelé **sélecteur**. Deux sortes de méthodes sont définies pour un objet : les méthodes primitives et les extensions. Les méthodes **primitives** sont essentielles à la manipulation des objets. Par exemple : création, copie, description, égalité, ... Les **extensions** (ou méthodes secondaires) sont définies par des axiomes à partir des méthodes primitives. L'union des méthodes primitives et des extensions constitue le comportement des instances. Par exemple, la classe `Rectangle` a pour aspect {`largeur : Real, longueur : Real`} et pour comportement : {`périmètre, largeur, longueur, créer, copier, égalité, décrire, ...`}. Dans l'écriture d'une méthode nous prendrons les conventions usuelles en programmation à objets : si la classe courante est le premier type d'argument alors méthode est dite **d'instance** et la variable associée est notée `Self`, sinon la méthode est dite **de classe**.

Classes

La classe d'un objet est appelée une classe formelle . Elle définit deux informations:

- Une description formelle et abstraite des caractéristiques des instances : l'aspect.
- Une description formelle et abstraite de méthodes des instances : les extensions.

L'aspect est un ensemble d'informations nécessaires et suffisantes à la description des objets. Il définit l'ensemble des méthodes primitives. Un aspect est composé d'une structure abstraite et d'une contrainte. La structure abstraite est un ensemble de sélecteurs de champ. Ils sont suffisants pour décrire tous les objets de la classe. Un exemple simple d'aspect est :

Carte	
aspect : carte	
field selectors	**constraint**
idClient : Carte ⟶ Integer	
cumul : Carte ⟶ Integer	
date : Carte ⟶ Date	

Dans certaines situations il est indispensable d'avoir des sélecteurs de champ conditionnels. Ce sont des sélecteurs de champ dont l'existence dépends d'une condition. Par exemple, un client doit avoir une carte de crédit pour tirer de l'argent. Dans d'autres cas une contrainte est indispensable pour décrire précisément les objets. Par exemple, l'identifiant de la carte est celui du client. Notez qu'un aspect doit en pratique posséder des propriétés supplémentaires : minimalité de la description, minimalité de la contrainte, ...

ClientHonnête	
aspect : client honnête	
field selectors	**constraint**
code : ClientHonnête ⟶ Integer	not(a_carte(Self))
idClient : ClientHonnête ⟶ Integer	orelse
a_carte : ClientHonnête ⟶ Boolean	idClient(Self) = idClient(carte(Self))
carte : ClientHonnête ⟶ Carte **requires:** a_carte(Self)	

Une extension est une méthode définie par un profil et des axiomes.

Rectangle
extensions
;; périmètre : calcule le périmètre d'un rectangle périmètre : Rectangle ⟶ Integer périmètre(Self) == mult(add(longueur(Self),largeur(Self)), 2)

Toutes les instances de la classe possèdent les caractéristiques de l'aspect et le comportement définit par la classe (**principe d'instanciation**). Il y a également un **principe d'extension incrémentale** : l'ajout d'une nouvelle extension ne modifie pas l'aspect des instances de la classe. Exemple : l'ajout de la méthode `surface` dans la classe précédente ne remet pas en cause la description des objets. Cette propriété est fondamentale car d'une part elle permet d'étendre incrémentalement le comportement et d'autre part elle permet de définir les notions de type d'une classe et d'héritage.

Envoi de message

L'envoi de message est noté comme une application de fonction, dans lequel `sélecteur` est le nom de la méthode. La sélection est simple et l'objet receveur est identifié par la variable `Self`.

```
<sélecteur>(<receveur> <, argument>*)
```

Héritage

L'héritage est une relation entre classes qui exprime d'abord une similitude des instances. Notre approche se base sur la notion de cœrcition [Roy92]. A partir d'une étude concrète des différents cas d'héritage nous avons synthétisé notre propre notion de cœrcition : la **projection structurelle**.

Les propriétés informelles de l'héritage sont : (i) Toute sous-instance peut être vue ou convertie implicitement en une super-instance. (ii) Une instance peut être substituée par une sous-instance dans un envoi de message (**principe de substitution**). (iii) Toute méthode de la super-classe est applicable à une sous-classe (définition 6.3.7). (iv) Une méthode héritée peut être redéfinie. La propriété la plus importante en pratique est sûrement l'héritage du comportement. Les trois premières propriétés reposent sur la notion de cœrcition La dernière est liée à la notion de surcharge d'opérateur. Une méthode est en quelque sorte générique pour toutes les sous-classes de sa classe de définition. Une telle méthode sera dite **applicable**.

Les classes abstraites sont traitées à part, car elles n'ont pas d'instances. Toutefois la notion de cœrcition reste valable pour donner des critères d'héritage.

6.2.2 Définition formelle des classes

Définition 6.2.1 (classe formelle)
Une classe formelle est un triplet $C = <A,\ E,\ S>$ qui sont respectivement l'aspect, les extensions et les super-classes directes.

La notion de super-classe est relative à l'héritage et sera étudiée dans la section 6.3.

Définition 6.2.2 (aspect)
Un aspect est composé d'une structure abstraite (un ensemble de sélecteurs de champ) et d'une contrainte $A = [fsel, cont]$*

Une description abstraite d'un objet est un n-uplet de valeur : $T_1\ ...\ T_n$ résultant de l'action des sélecteurs de champ sur l'objet : $(fsel_1(o),\ ...,\ fsel_n(o))$.

Définition 6.2.3 (sélecteur de champ)
*Un **sélecteur de champ** est un observateur de la classe de profil : $fsel_i : C \longrightarrow T_i$ muni d'une précondition : $prec_i : T_1\ ...\ T_n \backslash T_i \longrightarrow Boolean$.*

Les T_i sont appelés **types structurants** de la classe. Un sélecteur de champ peut être caractérisé par une **valeur par défaut**. La précondition est définie sur les sélecteurs de champ autres que celui qu'elle contrôle. Toutefois pour simplifier, elle sera notée $prec_i(u)$ où u est une description abstraite d'objet. Les préconditions sont supposées toutes non trivialement fausses. Notez que le sélecteur de champ ressemble plus aux accesseurs définis dans les langages de programmation à objets qu'aux variables d'instance elles-même. En ce sens la précondition est naturelle. Un sélecteur de champ booléen sans précondition est appelé **sélecteur d'état**. Il sert en général de précondition à d'autres sélecteurs de champ.

Définition 6.2.4 (contrainte)
Une contrainte est une condition $cont : T_1\ ...\ T_n \longrightarrow Boolean$

La contrainte est un prédicat qui doit être valide pour tous les objets de la classe. Elle peut être considérée comme une précondition et une postcondition à toute méthode et en particulier à la méthode de création d'instance. Elle joue le rôle d'un invariant de classe comme en Eiffel.

Une méthode est un couple constitué d'un sélecteur et d'un descriptif. Ce descriptif est un profil (type de la méthode) et une liste d'axiomes algébriques. La fonction `sel(m)` donne le sélecteur de la méthode `m`. Par abus de langage, le terme **méthode** `m` désignera souvent une méthode de sélecteur `m`.

La définition de l'aspect entraîne la définition d'un ensemble de méthodes primitives :

le générateur newC
;; newC : crée une instance de la classe et l'initialise newC : T_i* ⟶ C
les sélecteurs de champ fsel$_i$
;; fsel$_i$: récupère l'information correspondante fsel$_i$: C ⟶ T_i
la copie différentielle : copy
;; copy : copie un objet et modifie les valeurs associées aux champ copy : C T_i* ⟶ C
l'égalité sémantique : equal?
;; equal? : teste l'égalité sémantique de deux objets equal? : C OBJECT ⟶ Boolean
la représentation externe : describe
;; describe : représentation externe d'un objet describe : C ⟶ String

Figure 45 : Tableau des méthodes primitives fonctionnelles

Les autres méthodes, définies par l'utilisateur, sont des extensions. Avoir un générateur unique réduit l'éventail des possibilité de définition de méthodes : une méthode n'est définissable que par extension d'autres méthodes ou opérations de types prédéfinis[1]. Une extension est une opération dont toute application est réductible à l'application d'autres opérations.

Définition 6.2.5 (extension)
Une opération e est une extension d'un ensemble F d'opérations ssi $e(*) == t$ *où t est un terme construit sur F.*

Cette notion d'extension est essentiellement la même que les convertible operations chez [GH78], les defined operators de [HH82] ou les opérations secondaires de [LG90]. La définition d'une extension doit être prouvée cohérente. Les techniques de réécriture sont intéressantes dans ce contexte [JL86, Kap86]. Remarquons que les observateurs sont forcément des extensions cohérentes des sélecteurs de champ.

Un dictionnaire de méthodes est un ensemble de méthodes qui possède la propriété : $cardinal(\mathcal{S}el(E)) = cardinal(E)$. L'ensemble des méthodes définies dans une classe C est un dictionnaire noté $Dico(C)$.

[1] Voir aussi la structure des axiomes TAG, section 5.4.5.

Les méthodes comme classées comme suit :

$$\textit{méthodes} \begin{cases} \textit{primitives} \begin{cases} \textit{observateur} \begin{cases} \textit{sélecteursdechamp} \\ \textit{égalité sémantic } (\textit{equal?}) \\ \textit{description externe } (\textit{describe}) \end{cases} \\ \textit{constructeurs} \begin{cases} \textit{générateur } (\textit{new}) \\ \textit{copie d'objets } (\textit{copy}) \end{cases} \end{cases} \\ \textit{extensions} \quad \textit{ensemble ordonné } Ext = (e_i)_{1 \leq i \leq n} \end{cases}$$

La relation constituée des liens entre une classe et ses classes structurantes est appelée **la dépendance structurelle** et notée DPS. La relation d'utilisation USE, plus générale, lie une classe avec toutes les classes qui apparaissent directement ou indirectement dans les profils de ses méthodes.

6.2.3 Spécification algébrique associée à une classe

A une classe formelle nous pouvons associer une présentation algébrique d'un type abstrait de données.

Définition 6.2.6 (spécification algébrique)
La spécification algébrique associée à une classe formelle est composée :

- *des sortes utilisés : ce sont les noms des classes utilisées,*
- *d'une signature : elle est constituée du profil des méthodes du dictionnaire,*
- *d'axiomes : ce sont les axiomes des extensions et ceux des méthodes primitives. Ces derniers sont exprimés par les schémas suivants (avec préconditions explicites) :*

axiomes des sélecteurs de champ: projections
$requires : cont(X_1, ..., X_n) \wedge prec_i(X_1, ..., X_n)$
$fsel_i(newC(X_1, ..., X_n)) == X_i$

axiome de la copie différentielle
requires: $cont(X_1, ..., X_n) \wedge prec_i(X_1, ..., X_n)$
<u>**let**</u> $(\neg c_i \wedge X'_i = fsel_i(Self)) \ \vee \ (c_i \wedge X'_i = X_i)$ <u>**in**</u>
$copy(Self, c_1, X_1, ..., c_n, X_n) == newC(X'_1, ..., X'_n)$

axiomes de l'égalité
$equal?(Self, X) == equalbis?(X, Self)$
<u>**let**</u> $nupletSelf$ = $fsel_1(Self), ..., fsel_n(Self)$
$nupletX \quad fsel_1(X), ..., fsel_n(X)$ <u>**in**</u>
$equalbis?(Self, X) == ((not(prec_1(nupletSelf)) \ \wedge \ not(prec_1(nupletX))) \ \vee$
$(prec_1(nupletSelf) \ \wedge \ prec_1(nupletX) \ \wedge \ equalbis?(fsel_1(Self), fsel_1(X))))$
$\wedge \ ... \ \wedge$
$((not(prec_n(nupletSelf)) \ \wedge \ not(prec_n(nupletX))) \ \vee$
$(prec_n(nupletSelf) \ \wedge \ prec_n(nupletX) \ \wedge \ equalbis?(fsel_n(Self), fsel_n(X))))$
<u>**tel**</u>

axiome de la description
<u>**let**</u> $nupletSelf$ = $fsel_1(Self), ..., fsel_n(Self)$ <u>**in**</u>
requires: $prec_1(nupletSelf) \ \wedge ... \wedge \ prec_n(nupletSelf)$
$describe(Self) == concat("C", describe(fsel_1(Self)), ..., describe(fsel_1(Self)))$
<u>**tel**</u>

Figure 46 : Axiomes des méthodes primitives

La sémantique des méthodes primitives avec mots-clés peut être complètement définie en utilisant autant de déclaration qu'il y a de combinaisons possibles. Pour simplifier nous décrivons un profil maximal uniquement. Pour des questions de typage, ces méthodes notamment `equal?` ne sont pas correctes. En sélection simple, comparer des objets de type compatibles mais différents (instance d'un type et instance d'un sous-type) impose de passer par une fonction intermédiaire `equalbis?`. Pour `equalbis?` nous avons retenu le principe d'une égalité des sélecteurs de champ quand ils sont tous les deux définis (égalité forte). Pour `describe` nous avons uniquement décrit le cas où tous les sélecteurs de champ sont définis.

Soit T_C l'algèbre des termes clos sur la signature de la spécification associée à C. La façon standard de donner une sémantique à une telle spécification est d'en déterminer l'algèbre initiale. L'existence de cette algèbre initiale est assurée par l'utilisation d'axiomes conditionnels positifs. L'algèbre initiale peut se calculer en quotientant l'algèbre des termes clos par la relation d'égalité associée aux axiomes de la spécification. Les instances d'une classe sont sémantiquement les valeurs de l'algèbre initiale de la spécification algébrique associée à la classe. Ainsi un terme clos de T_C, de sorte C, est un représentant particulier d'une instance de C. Une approche de type classe de modèles est également possible [BB92], elle est nécessaire pour donner une sémantique aux classes abstraites. La déduction à partir des axiomes de la classe C est notée $\vdash_C$.

6.2.4 Structure inductive des instances

Nous ne nous intéressons qu'aux classes possédant des instances finiment engendrées et qui sont non vides[2]. La valeur spéciale $\perp$ dénote l'indéfini et ajoutée à tous les types comme en sémantique dénotationelle [R.D91].

Définition 6.2.7 (classe habitée)
Une classe est habitée ssi elle a au moins une instance $o \neq \perp$ et telle que : soit u sa description abstraite
(1) u a une structure conforme à l'aspect de la classe.
(2) u vérifie la contrainte.
(3) si $prec_i(u)$ est vrai alors $fsel_i(o) \neq \perp$.

Une induction peut être facilement construite sur les termes quand il n'y a pas de précondition de champ. Il suffit de considérer le graphe de dépendance structurelle sans circuit. Notre approche étend la notion usuelle de classe et autorise plus de facilité : par exemple une spécification hiérarchique ou plate des listes est possible [AR94b].

Dans les définitions précédentes, la méthode primitive `newC` est considérée comme unique générateur de la classe C mais il n'est pas standard à cause des mots-clés mais également des préconditions et de la contrainte. Suivant les cas, il peut être de base ou multiple (générateur de base et générateur récursif). Sa sémantique exacte peut être donnée par un ensemble de générateurs usuels.

Définition 6.2.8 (vecteur caractéristique)
Le vecteur caractéristique d'une description abstraite $u \in \prod_{1 \leq i \leq n} T_i$ est un n-uplet de booléens tel que $prec_i(u) = v_i$ pour $1 \leq i \leq n$.

Principe 6.2.9 (interprétation de `new`)
L'interprétation algébrique de `new` est un ensemble de générateur g_v. A chaque vecteur caractéristique v correspond un générateur g_v : $profil_v \longrightarrow C$ de précondition $cont \wedge prec_v$ tel que $profil_v$ représente le sous-mot des types structurants qui a un type partout où v est vrai et $pred_v$ est une conjonction des expressions $(v_i \wedge prec_i) \vee (\neg v_i \wedge \neg prec_i)$.

[2] Ces notions sont usuelles en TAA : sorte habitée, signature raisonnable, ...

Théorème 6.2.10
Les propositions suivantes sont équivalentes
(1) la sorte C définie par ces générateurs est habitée
(2) la classe C est habitée
(3) il existe un vecteur caractéristique v tel que $profil_v$ soit composé de classes habitées et non structurellement dépendantes de C et il existe un n-uplet u de $profil_v$ qui satisfait $prec_v$.

Certaines conditions nécessaires pour avoir une classe habitée peuvent être vérifiées facilement. Par exemple, si $n = 0$ alors la classe est un singleton contenant une seule constante, ou encore, si T_i DPS C alors cela implique que $prec_i \not\equiv true$.

Il est plus difficile de définir une forme générale et suffisante pour une classe habitée, en particulier l'usage des préconditions interdit toute chance d'avoir un critère statiquement décidable. La puissance d'expression du schéma suivant n'est pas connu mais en pratique elle semble assez générale.

Principe 6.2.11 (Aspect suffisant)

C	
aspect : aspect suffisant	
field selectors	***constraint***
$fsel_1 \quad : C \longrightarrow S_1$	$cont$
$\ldots \quad : C \longrightarrow \ldots$	
$fsel_k \quad : C \longrightarrow S_k$	
$csel_1 \quad : C \longrightarrow D_1$ **requires:** $prec_1$	
$\ldots \quad : C \longrightarrow \ldots$ **requires:** $\ldots$	
$csel_r \quad : C \longrightarrow D_r$ **requires:** $prec_r$	

- *Soit* $r = 0 \ \wedge \ k = 0$
- *Soit* $k \geq 1$ *et* $r = 0$ *alors* $\neg \ S_i$ *DPS* C *pour* $1 \leq i \leq k$ *et sont habitées.*
- *Soit* $k \geq 1$ *et* $\forall \ 1 \leq j \leq r \ prec_j \not\equiv true \not\equiv false$,
  alors $\neg \ S_i$ *DPS* C *pour* $1 \leq i \leq k$ *et sont habitées et il existe un n-uplet* $u \in \prod_{i=1}^{k} S_i$ *tel que* $prec_j(u)$ $1 \leq j \leq r$ *est faux.*

La structure inductive des termes est définie par les générateurs de l'interprétation précédente. Notons que la définition de ce type de structure nous permet de définir une notion de représentation [Hoa72, Gut80]. Ceci ouvre la voie à des aspects pratiques intéressants : comparaison d'aspects, notion de représentation plus abstraite etc.

6.2.5 Propriétés

Définition 6.2.12 (classe bien définie)
Une classe C est bien définie si elle possède les trois propriétés suivantes :

1. *La classe est habitée.*
2. *La spécification associée est non-contradictoire :*
   $\forall \ t, u \in T_C, \ t, u : P \ (P \in USE(C)), \ \vdash_C t == u \Longrightarrow \ \vdash_P t == u.$
3. *La spécification est pleinement spécifiée :*
   $\forall c \in T_C, \ c : P \ (P \in USE(C)), \ \exists u \in T_P, \ c == u.$

La première condition assure une induction bien fondée sur les instances de la classe. Dans ce cas, la validité de la déduction équationnelle classique est également assurée [GM85, Sun91]. La deuxième condition exprime que des termes prédéfinis qui étaient initialement distincts ne sont pas rendus égaux. La dernière condition dit que l'action de tout observateur est connu sur tous les objets de la classe. Nous nous sommes inspirés de la pleine spécification de [Mus80] plutôt que de la complétude hiérarchique [Gau90] car cette dernière est trop contraignante en objet.

La méthode de conception à objets est incrémentale : le concepteur définit d'abord la caractérisation des objets puis il étend successivement le comportement tout en conservant cette caractérisation.

Propriété 6.2.13 (bonne définition des primitives)
Une classe réduite à son aspect (i.e. sans extension) est bien définie si et seulement si la classe est habitée.

Cette propriété se justifie en considérant d'abord une définition sans contrainte ni condition de sélecteur de champ. Dans ce cas il peut être prouvé que le système d'équations peut s'orienter en règle de réécriture de droite à gauche. Ce système est convergent (donc non-contradictoire) car il est sans superposition et il est facile de trouver un ordre de réduction qui termine. La pleine spécification est facile à vérifier. Dans le cas où il y a des conditions, celles-ci sont définies sur les types structurants donc bien définies par hypothèse. Les préconditions ajoutées aux équations ne changent rien au niveau de la terminaison et n'ajoutent pas de superposition. Elles peuvent avoir une influence sur la pleine spécification, mais nous avons fait l'hypothèse d'un traitement d'erreur complet.

Soit une classe C, $C+m$ désigne l'extension de la classe C par l'ajout de la méthode `m`. Cet ajout peut être une nouvelle méthode ou encore la redéfinition d'une méthode existante.

Définition 6.2.14 (extension bien définie)
Soit C une classe bien définie, `m` *est une extension bien définie pour C si et seulement si $C+m$ est encore une classe bien définie.*

Il existe des critères d'écriture qui assure la bonne définition d'une méthode (cf section 6.2.6). Ils s'inspirent de ceux utilisés en spécification algébriques [Gut80, Bid82].

Propriété 6.2.15 (Extension incrémentale)
L'ajout d'une extension bien définie ne modifie pas l'aspect de la classe.

Cette propriété découle du fait qu'une nouvelle extension ne modifie pas la sémantique des méthodes primitives.

6.2.6 Ecriture des méthodes

Nous commentons dans cette section quelques idées d'écriture des axiomes pour les extensions. Une condition suffisante pour qu'une méthode soit bien définie est qu'elle soit une extension non-contradictoire des méthodes déjà définies dans la classe. Cette situation est simple et générale à utiliser, dans [AR92c] nous avons appelés de telles méthodes **méthodes secondaires**. Les techniques usuelles [GH78, Mus80, Bid82, LG90, BB92] sont une autre sources d'inspiration.

La définition d'une extension peut revêtir plusieurs formes : nous en donnons une ici. Nous en verrons d'autres dans la section 6.6.1.

Définition 6.2.16 (présentation quasi-opérationnelle)
C'est une règle de réécriture de la forme $e(Self, *) \longrightarrow t$ où e est l'extension à définir et t un terme.

extensions

```
;; newtotal : modification du solde de la carte - extension directe
   newtotal : Carte Integer  ⟶  Carte
var: somme:Integer
newtotal(Self, somme)  ⟶ newCarte(idClient = idClient(Self),
                         cumul = cumul(Self) + somme, date = date(Self))
```

```
;; newtotal : modification du solde de la carte - extension indirect
   newtotal : Carte Integer  ⟶  Carte
var: somme:Integer
newtotal(Self, somme)  ⟶ copy(Self, cumul = cumul(Self) + somme)
```

```
;; append : concaténation de deux listes - extension récursive
   append : FullListC ListC  ⟶  FullListC
var: Xl:ListC
append(Self, Xl) == cons(car(Self), append(cdr(Self), Xl))
```

Dans le cas d'une modification de méthode existante ou d'un ajout d'une nouvelle méthode, il faut globalement reconsidérer la propriété de bonne définition de la classe.

6.2.7 Classe abstraite, méthode abstraite

Les propriétés générales communes à un ensemble de classes et les comportements partiels sont définis par des classes et des méthodes abstraites.

Définition 6.2.17 (classe abstraite)
Une classe abstraite est une classe qui ne possède pas de méthode de création d'instance.

Définition 6.2.18 (méthode abstraite)
Une méthode abstraite ou méthode virtuelle est une extension dont les axiomes ne sont pas décrits.

Les termes produits par une extension n'étant pas définis la contradiction et la pleine spécification de la propriété 6.2.12 ne sont pas démontrables. En conséquence, une classe contenant une méthode abstraite doit être déclarée abstraite.

Une méthode (resp. une classe) est déclarée abstraite par le mot-clé `ABSTRACT`. La fonction `abstract(m)` (resp. `abstract(C)`) indique si une méthode `m` (resp. une classe `C`) est abstraite.

Exemple :

```
Boolean
ABSTRACT
```

inherits from `OBJECT`

extensions

```
;; and : conjunction
   and : Boolean Boolean  ⟶  Boolean
ABSTRACT
```

```
;; not : negation
   not : Boolean    ⟶  Boolean
ABSTRACT
```

```
;; or : disjunction
   or : Boolean Boolean  ⟶  Boolean
var: b:Boolean
or(Self, b) == not(and(not(Self),not(b)))
```

```
;; xor : exclusive disjunction
   xor : Boolean Boolean  ⟶  Boolean
var: b:Boolean
xor(Self, b) == and(or(Self,b),not(and(Self,b)))
```

```
;; implies : implication
   implies : Boolean Boolean  ⟶  Boolean
var: b:Boolean
implies(Self, b) == or(not(Self),b)
```

True
inherits from Boolean
extensions
;; and : conjunction   and : True Boolean ⟶ Boolean **var:** b:Boolean and(Self, b) == b
;; not : negation   not : True Boolean ⟶ False not(Self) == newFalse

False
inherits from Boolean
extensions
;; and : conjunction   and : False Boolean ⟶ False **var:** b:Boolean and(Self, b) == Self
;; not : negation   not : False Boolean ⟶ True not(Self) == newTrue

Figure 47 : La spécification des booléens par classes formelles.

Les théorèmes suivants servent à la redéfinition des méthodes `or, xor, implies` dans les sous-classes `True` et `False` :
∀ b:Boolean, t:True, f:False,

or(t, b) == Self or(f, b) == b
xor(t, b) == not(b) xor(f, b) == b
implies(t, b) == b implies(f, b) == not(f) implies(f, b) == newTrue

Les définitions et propriétés précédentes sur les classes et les méthodes doivent être adaptées pour prendre en compte les classes abstraites.

Une classe abstraite n'est pas à proprement parler habitée puisqu'elle n'a pas d'instances. Le problème se pose pour les classes qui dépendent structurellement de cette classe abstraite e.g. la classe `ClientHonnête` dépend structurellement de la classe abstraite `Boolean` (page 133). Plusieurs définitions sont possibles.

Définition 6.2.19 (classe abstraite habitée)
Une classe abstraite est habitée si elle a au moins une sous-classe et que toute ses sous-classes sont habitées.

Cette définition est correcte mais pose le problèmes de la validité dans le temps : la vérification doit être refaite à chaque fois que la sous-hierarchie de la classe abstraite est modifiée. La propriété d'extension incrémentale du système de classe n'est pas conservée. Une définition plus faible serait qu'il existe au moins une sous-classe habitée. Cette contrainte est moins forte mais n'assure pas la bonne définition des classes. Une dernière définition, plus souple, simule une classe abstraite par une classe concrète. La notion de classe habitée devient alors celle de classe (potentiellement) habitable dans les définitions de la section 6.2.5 lorsque les classes sont abstraites.

Définition 6.2.20 (classe abstraite habitable)
Une classe abstraite est habitable si son aspect est suffisant.

Cette définition est possible à cause de la préservation de l'aspect par héritage (voir section suivante) et que toute instance apparaissant dans l'aspect sera une instance d'une sous-classe.

La propriété 6.2.12 n'est pas décidable, mais une bonne définition des extensions est possible en ajoutant les méthodes abstraites dans l'ensemble F des opérations sur lesquelles sont définies les extensions non abstraites, selon la définition 6.2.5.

6.3 Héritage

L'héritage d'aspect et celui des extensions sont distingués, L'héritage d'aspect est une relation proche du sous-typage, qui permet de décider si une classe peut hériter d'une autre. Elle est basée sur la relation de cœrcition appelée projection structurelle [Roy92]. L'héritage de comportement est un mécanisme de réutilisation de code. Cette distinction n'entraîne pas deux hiérarchies, à la différence de VDM++[D9̈4] où l'héritage de structure est simple et l'héritage de comportement est indépendant et multiple.

6.3.1 Cœrcition

Cette définition se base sur une correspondance entre les sélecteurs de champ de S et ceux de C : la contrainte est respectée, les préconditions renforcées, et les types restreints. Cette correspondance se fait par égalité des noms de sélecteurs, pour simplifier nous supposons les sélecteurs ordonnés de telle façon que $\forall i\ 1 \leq i \leq n,\ name(fsel_i^S) = name(fsel_i^C)$, où n représente le nombre de sélecteurs de S.

Définition 6.3.1 (projection structurelle)
Soient S et C deux classes, $coerce_{(C,S)} : T_C \longrightarrow T_S$ est défini par

- $\forall c \in T_C,\ cont^C(fsel_1^C(c),\ ...,\ fsel_m^C(c)) \Longrightarrow cont^S(fsel_1^S(c),\ ...fsel_m^S(c))$ *et*
- $\forall fsel_i^S : S \longrightarrow T_i,\ 1 \leq i \leq n$ *et* $fsel_i^C : C \longrightarrow R_i$
  - *soit* $not(prec_i^C(c))$
  - *soit* $prec_i^C(c) \wedge prec_i^S(c)$ *et*
    - *soit* $T_i = R_i \ \wedge\ fsel_i^C(c) = fsel_i^S(c)$
    - *soit* $\exists coerce_{(R_i,T_i)} \ \wedge\ coerce_{(R_i,T_i)}(fsel_i^C(c)) = fsel_i^S(coerce_{(C,S)}(c))$.
- $\forall c \in T_C,\ abstract(S) \Longrightarrow coerce_{(C,S)}(c) = c$

Propriété 6.3.2 ()
Si cette relation existe elle est unique et elle définit un ordre partiel sur les classes.

Cette définition utilise l'égalité des noms de sélecteurs de champ, en pratique il peut être utile d'envisager la projection structurelle après un renommage adéquat. La définition de la cœrcition se fait en comparant les classes (sans héritage). L'héritage est considéré comme une propriété découlant de cette relation.

Définition 6.3.3 (héritage)
Soient S et C deux classes distinctes alors C peut hériter de S si et seulement si il existe une projection structurelle de T_C vers T_S.

$$C\ ako\ S \Longleftrightarrow \exists coerce_{(C,S)} \ \wedge\ (\neg abstract(S) \Rightarrow \neg abstract(C))$$

L'héritage est donc autorisé si de la super-classe vers la sous-classe :

- L'aspect contient les memes sélecteurs de champ.
- Le codomaine des sélecteurs de champ est plus spécifique,
- Les sélecteurs de champ ont une précondition plus forte.
- La contrainte est plus restrictive.
- La sous-classe ne peut etre abstraite que si la super-classe l'est.

Ces différents cas pouvent se combiner. Le dernier cas correspond plus à une bonne habitude de programmation qu'une condition nécessaire : une classe abstraite représente un comportement partiel, qui est complété dans les sous-classes et il n'est pas logique de définir un comportement partiel à partir d'un comportement total. Noter que cette forme d'héritage entraîne le sous-typage, selon des règles que nous examinerons dans la section 6.5.

6.3.2 Héritage des méthodes

La validité de l'héritage structurel étant établie, étudions ses conséquences au niveau des méthodes. Seules les extensions sont héritées puisque les primitives sont implicitement redéfinies par la classe. Nous nous plaçons dans un contexte restrictif (mais souhaitable en génie logiciel) où il n'y a pas d'exception à l'héritage des méthodes.

Comportement complet

Le comportement complet d'une classe est l'ensemble des méthodes visibles de la classe, i.e. celles définies par la classe et celles héritées. Il existe de nombreuses variantes à l'héritage des méthodes : héritage simple ou multiple, interdictions de collisions ou des conflits, stratégies classiques ou linéaires. Consulter [HD89, DHH$^+$95] pour un panorama plus précis.

Nous donnons ici une définition générale du comportement complet. Si E est un ensemble de méthodes, $\mathcal{S}el(E)$ désigne l'ensemble de ces sélecteurs. La formalisation du comportement complet est la suivante :

Définition 6.3.4 (comportement (complet))
$K(C)$ est un dictionnaire qui vérifie :

- $K(C) \subset Dico(C) \ \# \ H(C)$.
- $\mathcal{S}el(K(C)) \ = \ \mathcal{S}el(Dico(C)) \ \cup \ \mathcal{S}el(H(C))$.

L'opérateur $\#$ est un opérateur de concaténation d'ensembles de méthodes qui réalise le masquage par la première définition trouvée. $H(C)$ désigne l'ensemble des méthodes héritées :

$$H(C) \ = \ \cup_{i=1}^{n} K(S_i) \ \subset \ \cup_{j=1}^{m} Dico(C_j)$$

$(S_i)_{i=1}^{n}$ désigne les super-classes directes et $(C_j)_{j=1}^{m}$ les super-classes de C.

Le calcul de $H(C)$ dépend de la stratégie choisie. Il existe un résultat simple et général concernant l'unicité de la stratégie.

Théorème 6.3.5 (unicité de la stratégie)
La stratégie est unique $\Longleftrightarrow \forall C, \ \forall s \in \mathcal{S}el(K(C)) \wedge card(M_s(C)) > 1 \Longrightarrow \exists (s, \ P_C) \in Dico(C)$

où M_s désigne les méthodes de sélecteurs s en collision pour C. Cette équivalence dit qu'un principe de redéfinition systématique en cas de collision de méthodes évite les ambiguïtés. Les stratégies utilisées en pratique forment une sous-classe des stratégies dites uniformes.

Définition 6.3.6 (parcours de recherche)
Un parcours de recherche est une stratégie uniforme telle que $K(C) \ = \ Dico(C) \ \# \ (\#_{j=1}^{m} Dico(C_j))$.

Applicabilité des méthodes

Une méthode doit être applicable à toutes les instances des sous-classes de sa classe de définition La méthode est applicable si elle est bien définie dans les sous-classes.

Définition 6.3.7 (applicabilité)
Soit m une extension de la classe S.
- Soit S bien définie alors $m \bowtie S \ \Longleftrightarrow \forall C$ *ako* S, *m est bien définie pour C.*
- Soit $abstract(S)$ alors $m \bowtie S \ \Longleftrightarrow \forall C$ *ako* $S, \ m \bowtie C$.

L'idée majeure du contrôle de type est de trouver des critères décidables et nécessaires à l'applicabilité d'une méthode.

6.3.3 Critère d'héritage

Le critère (statique) d'héritage est défini à partir des conditions nécessaires à l'existence de la projection structurelle, étudiées dans [Roy92] et de cohérence pour les classes abstraites.

Définition 6.3.8 (critère d'héritage)

$$
\begin{array}{ll}
\multicolumn{2}{l}{C\ ako\ S \implies} \\
\textit{(i)} & \forall fsel_i : S \longrightarrow T_i\ \exists fsel_i : C \longrightarrow R_i,\ R_i = T_i \lor R_i\ ako\ T_i \\
\textit{(ii)} & \neg abstract(C) \implies \nexists m \in K(C) \bullet abstract(m) \\
\textit{(iii)} & abstract(C) \implies abstract(S)\ \land\ [\forall m \in K(C),\ abstract(m) \implies \\
 & \qquad (\nexists m' \in K(S),\ sel(m) = sel(m') \bullet \neg abstract(m'))].
\end{array}
$$

Une classe concrète n'a pas de méthodes abstraites. Une méthode abstraite ne peut redéfinir qu'une autre méthode abstraite. Il semble difficile de faire mieux dans la mesure où le problème dans le cas général est de comparer des conditions de sélecteurs ou des contraintes, ce qui est indécidable. Ce critère impose une règle de redéfinition a priori simplement covariante.

6.3.4 Redéfinition des méthodes

Ce mécanisme est très utile en programmation à objets. Il est une forme générique de surcharge d'opérateurs. Il a deux principales justifications : premièrement spécialiser une méthode en l'optimisant (conservation de la sémantique) et deuxièmement donner une sémantique voisine à la méthode rédéfinie.

Un usage incontrôlé de la spécialisation de la sémantique peut avoir des effets néfastes sur la qualité logiciel. Il nous semble nécessaire dans notre contexte de limiter les cas de redéfinition (abus de surcharge). Une première difficulté est de définir la notion de **sémantique voisine**. La deuxième est de fournir des critères de vérification décidable. Par exemple, examinons le cas de la méthode `périmètre` d'un rectangle et celle d'un carré :

`Rectangle`	
aspect : `mesuring`	
field selectors	**constraint**
`longueur : Rectangle` $\longrightarrow$ `Integer`	
`largeur : Rectangle` $\longrightarrow$ `Integer`	
extensions	

```
;; périmètre : calcule le périmètre du rectangle
   périmètre : Rectangle    ⟶  Integer
périmètre(Self) == mult(add(longueur(Self),largeur(Self)), 2)
```

`Square`	
inherits from `Rectangle`	
aspect : `mesuring`	
field selectors	**constraint**
`longueur : Square` $\longrightarrow$ `Integer`	`longueur(Self) = largeur(Self)`
`largeur : Square` $\longrightarrow$ `Integer`	
extensions	

```
;; périmètre : calcule le périmètre du carré
   périmètre : Square    ⟶  Integer
périmètre(Self) == mult(longueur(Self),4)
```

Figure 48 : Une redéfinition voisine de méthode

LecteurDeCarte		
aspect : lecteur de carte		
field selectors		**constraint**
stockDeCarte	: LecteurDeCarte ⟶ Set[Carte]	
carteCourante	: LecteurDeCarte ⟶ Carte	
extensions		
;; finDeTransaction : termine la transaction courante en avalant la carte finDeTransaction : LecteurDeCarte ⟶ LecteurDeCarte finDeTransaction(Self) == add(stockDeCarte(Self),carteCourante(Self))		

LecteurDeCarte		
aspect : lecteur de carte		
field selectors		**constraint**
stockDeCarte	: LecteurDeCarte ⟶ Set[Carte]	
carteCourante	: LecteurDeCarte ⟶ Carte	
extensions		
;; finDeTransaction : termine la transaction courante en libérant la carte finDeTransaction : LecteurDeCarte ⟶ LecteurDeCarte finDeTransaction(Self) == release(Self,carteCourante(Self))		

Figure 49 : Une redéfinition surchargée de méthode

Pour la redéfinition des méthodes la position la plus stricte est l'identité sémantique. En pratique plus de liberté est souhaitable et une notion de sémantique voisine est utile. Nous donnons ici une idée d'une telle définition, elle ne semble pas encore recouvrir tous les cas souhaitables.

Définition 6.3.9 (sémantique voisine)
Soient $m_S : S* \longrightarrow T$ *et* $m_C : C* \longrightarrow R$ *et C ako S vérifie le critère d'héritage,*
m_C *a une sémantique voisine de celle de* $m_S \Longleftrightarrow$

$$\forall c \in T_C \ \ cœrce_{R,T}(m_C(c,\ *)) == m_S(cœrce_{C,S}(c),\ *)$$

où m_C *désigne la méthode de sélecteur* **m** *définie dans la classee* **C**.

Cette définition induit une covariance simple (voir la définition 6.5.4). La cœrcition est un morphisme entre classe. La notion de cœrcition est proche de la notion de représentation mais les deux concepts sont disjoints en général. Cela étant la définition précédente rappelle la correspondance des opérations à travers une représentation. Une telle définition est bien sûr indécidable, il reste à trouver des critères syntaxiques, nécessaires et/ou suffisants en pratique.

6.3.5 Conception plate ou ordonnée

Le modèle des classes formelles muni de la relation d'héritage permet deux formes de conception pour les types : la conception plate ou ordonnée. Cette dernière notion correspond aux `subsorts` d'OBJ [FGJM85].

Une conception ***plate*** *est une spécification de type de données par une seule classe formelle. Une conception* ***ordonnée*** *est une spécification de type de données par un sous-graphe d'héritage de racine unique, une classe abstraite de nom le type de données.*

Les avantages d'une conception ordonnée sur une conception plate sont multiples. Ils sont principalement dus à une mise en valeur de classes abstraites :

- séparation claire entre le comportement commun et le comportement spécifique,
- meilleure utilisation du polymorphisme des méthodes,
- meilleure organisation de la hiérarchie d'héritage,

- simplification de l'aspect (moins de préconditions pour les sélecteurs de champ),
- simplification importante dans l'écriture des méthodes,
- est parfois plus adapté au cas mono-générateur (e.g. pas de conception plate mono-générateur pour la spécification des booléens),
- et surtout plus grande précision dans le typage.

L'inconvénient majeur est l'apparition de termes douteux, c'est-à-dire des termes sémantiquement corrects mais mal typés. Une solution est proposée dans la section 6.5.2.

Voici un exemple de spécification plate et ordonnée pour la spécification des entiers naturels :

Natural
ABSTRACT
inherits from OBJECT
extensions

```
;; succ : successor
   succ : Natural    —→  Positive
succ(Self) == newPositive(pred = Self)
```
```
;; plus : addition
   plus : Natural Natural  —→  Natural
ABSTRACT
```
```
;; mult : multiply
   mult : Natural Natural  —→  Natural
ABSTRACT
```
```
;; isEven : answer wether Self is even or not (odd)
   isEven : Natural    —→  Boolean
ABSTRACT
```

Zero
inherits from Natural
extensions

```
;; plus : addition
   plus : Zero Natural  —→  Natural
var: n:Natural
plus(Self, n) == n
```
```
;; mult : multiply
   mult : Zero Natural  —→  Zero
var: n:Natural
mult(Self, n) == Self
```
```
;; isEven : answer wether Self is even or not
   isEven : Zero    —→  True
var: n:Natural
isEven(Self) == newTrue
```

Positive	
inherits from Natural	
aspect : positive natural	
field selectors	**constraint**
pred : Positive —→ Natural	
extensions	

```
;; plus : addition
   plus : Positive Natural  —→  Positive
var: n:Natural
plus(Self, n) == plus(pred(Self),succ(n))
```
```
;; mult : multiply
   mult : Positive Natural  —→  Natural
var: n:Natural
mult(Self, n) == plus(n,mult(pred(Self),n))
```
```
;; isEven : answer wether Self is even or not
   isEven : Positive    —→  Boolean
var: n:Natural
isEven(Self) == not(isEven(pred(Self))
```

Figure 50 : La conception ordonnée d'une spécification des entiers naturels.

Natural	
inherits from OBJECT	
aspect : positive natural	
field selectors	**constraint**
isZero : Natural ⟶ Boolean	
pred : Natural ⟶ Natural requires: isZero(Self) == false	
extensions	
;; succ : successor succ : Natural ⟶ Natural succ(Self) == newNatural(isZero = false, pred = Self)	
;; plus : addition plus : Natural Natural ⟶ Natural var: n:Natural isZero(Self) == true ==> plus(Self, n) == n isZero(Self) == false ==> plus(Self, n) == plus(pred(Self),succ(n))	
;; mult : multiply mult : Natural Natural ⟶ Natural var: n:Natural isZero(Self) == true ==> mult(Self, n) == Self isZero(Self) == false ==> mult(Self, n) == plus(n,mult(pred(Self),n))	
;; isEven : answer wether Self is even or not isEven : Natural ⟶ Boolean var: n:Natural isZero(Self) == true ==> isEven(Self) == newTrue isZero(Self) == false ==> isEven(Self) == not(isEven(pred(Self))	

Figure 51 : La conception plate d'une spécification des entiers naturels.

6.4 Sémantique opérationnelle

La sémantique opérationnelle est donnée par une évaluation par réécriture [JL86, Kap86].

6.4.1 Déduction équationnelle et réécriture

Le principe de base est la déduction équationnelle et repose sur la notion de substitution de termes. Il est mis en œuvre par les techniques de réécriture de termes. Ici nous avons besoin d'une mécanisme de réécriture conditionnelle [Kap86] qui se définit de la façon suivante :

Définition 6.4.1 (réécriture conditionnelle)
Soit $\wedge_{1\leq i\leq n} u_i == v_i ==> l \rightarrow r$ *une règle de réécriture conditionnelle et soit* t *un terme. Une réécriture de* t *est un terme* $t[w \ <- \ \sigma(r)]$ *où :*

- *il existe* $t_w = \sigma(l)$ *et*
- $\forall i \ \sigma(u_i) == \sigma(v_i)$.
- $Var(l) \supseteq Var(r) \cup (\bigcup_{i=1}^{n} Var(u_i) \cup Var(v_i))$.

t_w désigne le sous-terme de t occurrant en w, $t[w \ <- \ b]$ le terme t dans lequel l'occurrence en w est remplacée par b et σ une substitution.

Ce type de déduction est insuffisant en pratique, il nous faut un principe d'induction. La théorie inductive [GG88] se définit relativement à des générateurs[3].

Ce schéma de base permet de faire des preuves dans une classe mais si il y a plusieurs classes avec héritage il n'est pas suffisant. L'héritage apparait alors comme une stratégie particulière de réécriture.

[3] uniquement newC dans la version de base du modèle.

6.4.2 Evaluation symbolique

Pour intégrer ces mécanismes de façon cohérente, nous nous basons sur une vision opérationnelle de la programmation à objets et nous l'adaptons à notre contexte. Nous proposons donc un mécanisme d'évaluation symbolique, permettant de faire des preuves et basé sur une stratégie d'évaluation par valeur avec sélection relativement au type dunamique d'un terme.

Soit une opération primitive `rewrite(<un terme>, <des règles>)`. Le calcul sur les types de base (`BasicId`) est réalisé par la primitive `computation`. La valeur d'un terme est sa forme normale.

Définition 6.4.2 (forme normale)
Un terme est en forme normale si il est construit uniquement sur des générateurs :
NFTerm ::= Constant | newClassId(NFTerm)*

Définition 6.4.3 (type dynamique)
La fonction `dynamic` *récupère le type dynamique d'une expression en forme normale. Elle est donnée par :*

- *dynamic(aConstant) == BasicId,*
- *dynamic(newClassId(nft$_1$, ..., nft$_n$)) == ClassId.*

Un terme est évalué dans un contexte définit par des classes, ce contexte est implicite ici. Soient deux fonctions `profile` et `get-rules` qui récupèrent respectivement la déclaration et les règles associées à une méthode. Nous rappellons ici l'hypothèse de sélection simple : la recherche de la méthode se fait à partir de la classe du receveur.

Algorithm 6.4.4 (Evaluateur par valeur)

```
eval : Term Class*  ⟶ NFTerm + {type-error, dynamic-error, loop}
eval(e, C1 ... Cn) ==
   IF e = c Constant
       THEN e
       ELSE LET e = m(t1, ..., tn)
           fi = eval(ti)
           CASE m : Operation: S1 ... Sn  ⟶ S
                THEN IF ∀i dynamic(fi) = Si
                    THEN computation(m, f1, ..., fn)
                    ELSE type-error
                    ENDIF
                m : newClassId: S1 ... Sn  ⟶ ClassId
                THEN IF ∀i dynamic(fi) ≤ Si
                    THEN newClassId(f1, ..., fn)
                    ELSE type-error
                    ENDIF
                OTHERWISE LET R = dynamic(f1)
                    IF Rm = get-rules(m, R) : S1 ... Sn  ⟶ S
                         THEN IF ∀2 ≤ i ≤ n dynamic(fi) ≤ Si
                             THEN eval(rewrite(m(f1, ..., fn), Rm), C1 ... Cn)
                             ELSE type-error
                             ENDIF
                         ELSE type-error
                         ENDIF
                TEL
           ENDCASE
           TEL
       ENDIF
```

Le “résultat” de `eval` est : soit une forme normale, un calcul infini (loop), une erreur de type. Notez que seule l'erreur `type-error` apparait explicitement comme une erreur de type lors de l'évaluation. La différence entre `type-error` et `dynamic-error` est que la première peut être détectée statiquement par un contrôleur de type. La dernière erreur de type exprime le fait qu'il n'y a pas de règles applicables et englobe plusieurs erreurs différentes : `m` est une méthode abstraite, `m` a un profil incompatible avec types des paramètres dynamiques, `m` est une méthode inconnue pour cette classe. Le traitement des méthodes primitives est traitable à part pour éviter d'alourdir la syntaxe des classes.

En général nous n'avons aucune certitude sur l'obtention d'une forme normale et donc de la terminaison de l'évaluation. Des questions théoriques apparaissent ici : critères de terminaisons, stratégies plus efficaces, extension à un mode par nom, confluence, etc.

6.5 Typage

Le typage des langages à objets est connu pour être difficile. [Car84, CHC90, CGL92, Cas94a, PS92, PT93]. Plusieurs raisons expliquent cette difficulté : confusions et controverses sur la terminologie (type/classe, sous-typage/héritage, covariance/contravariance, statique/dynamique, ...) [PS92], variété des approches pour la sémantique des langages [DT88], puissance mais aussi diversité des concepts tels que le polymorphisme [Car84], etc. Les modèles de la programmation à objets varient en fonction de l'importance donnée aux caractéristiques. Selon [KR91] deux modèles principaux existent : le modèle de la fermeture ou du point fixe et le modèle des structures de données ou auto-application. Le premier est adapté aux démonstrations théoriques mais il n'est pas naturel. Le second est plus opérationnel, il sépare les objets de leur opérations. Contrairement à [Ame89, CHC90] nous nous plaçons dans le cadre où une classe est un type et l'héritage entraîne le sous-typage, comme en Eiffel, où le sous-typage est défini par une relation de conformité sur l'héritage[Mey88].

Dans la pratique, la plupart des langages à objets typés statiquement ont un système de type pauvre ou mal adapté. Eiffel a un système de type riche mais non sûr [Coo89].

D'un point de vue théorique, beaucoup de travaux sont basés sur les types *record* de Cardelli [Car84, DT88]. Cette approche, basée sur le λ-calcul typé étendu à la quantification existentielle et universelle, est très puissante. Elle permet un système bien typé comprenant les fonctions d'ordre supérieur. Les types sont considérés comme des ensembles de valeurs et le sous-typage comme une inclusion ensembliste contrainte (les idéaux). Un des problèmes de cette approche, est qu'elle implique une règle de contravariance pour la redéfinition des méthodes qui n'est pas naturelle en programmation à objets. D'autres travaux plus récentes proposent des alternatives [AK86, HJW$^+$92, Cas94a]. Notons toutefois que Haskell[HJW$^+$92] a une approche différente des classes, et pas de liaison dynamique.

Notre approche, similaire à celle d'Eiffel sans les types associatifs (**`like`**), se fonde d'abord sur un point de vue pratique basé sur les types abstraits de données et un certain polymorphisme des valeurs, qui ne sont pas des fonctions. Les résultats sont proches des travaux de Castagna [Cas94a] qui utilise le λ-calcul typé. Une syntaxe abstraite est donnée dans [Roy94]. Elle permet d'unifier aspect et extensions dans le comportement. Dans cette section nous énonçons les principes d'un type simple mais sûr pour les classes formelles : déclaration explicites des variables et opérateurs, simple covariance dans la redéfinition des méthodes. Puis nous examinons quelques cas particuliers, notamment le problème des termes douteux et celui de la covariance multiple.

6.5.1 Un système de type simple

L'objectif du système de type est de montrer que toutes les expressions d'un programme ont un sens. Les expressions statiquement typables sont : les variables, les messages et les méthodes. Les expressions qui ont un type dynamique (les valeurs) sont les mêmes. La forme générale des expressions de types statiques est :

Définition 6.5.1 (expressions de type)
`TypeId ::= BasicId | ClassId | TypeId* ⟶ TypeId`

La difficulté du typage des langages à objets vis-à-vis des langages conventionnels comme C ou Pascal est que les objets et méthodes peuvent être polymorphes i.e. avoir plusieurs types. [CW85] est une bonne synthèse de ce problème, en particulier pour la clarification de la notion de polymorphisme. Nous considérons une unique forme pour l'instant : le sous-typage, noté $\leq$. La définition du sous-typage varie selon les auteurs. Pour beaucoup, il s'agit d'une inclusion ensembliste. Une définition plus générale est le respect de deux règles : la substitution et la cohérence. Le principe de substitution exprime le fait qu'une valeur du sous-type peut être substituée à une valeur du super-type partout où une valeur du super-type est attendue. Le code de la méthode exécutée peut varier selon le type dynamique des paramètres, au moins le receveur. La cohérence correspond au fait que quelque soit le code exécuté, le résultat sera correct. La cohérence implique entre autre une contrainte de variance sur le profil des méthodes redéfinies. Nous en donnons les différentes formes générales ici.

Définition 6.5.2
Soit $\leq$ une relation de sous-typage. Soient $op_S \;\hat{=}\; \mathtt{op(s : S,}\ a_1 : T_1\mathtt{,\ ...,}\ a_n : T_n\mathtt{)} \longrightarrow \mathtt{T)}$ et $op_C \;\hat{=}\; \mathtt{op(c : C,}\ b_1 : U_1\mathtt{,\ ...,}\ b_n : U_n\mathtt{)} \longrightarrow \mathtt{U)}$ deux méthodes avec $C \leq S$, alors :

- *règle de novariance :* $op_C \leq_{nv} op_S \Rightarrow \forall 1 \leq i \leq n,\ U_i = T_i \ \wedge\ U = T$
- *règle de covariance :* $op_C \leq_{cv} op_S \Rightarrow \forall 1 \leq i \leq n,\ U_i \leq T_i \ \wedge\ U \leq T$
- *règle de contravariance :* $op_C \leq_{ctv} op_S \Rightarrow \forall 1 \leq i \leq n,\ T_i \leq U_i \ \wedge\ U \leq T$
- *règle de simple covariance :* $op_C \leq_{scv} op_S \Rightarrow \forall 1 \leq i \leq n,\ T_i = U_i \ \wedge\ U \leq T$
- *règle de multi-covariance :* $op_C \leq_{mcv} op_S \Leftrightarrow op_C \leq_{cv} op_S$

Les définitions sont restreintes au cas où les deux méthodes ont le même nombre d'arguments. Une définition plus générale est donnée dans [ABDS95], en séparant les arguments receveurs et les non-receveurs. La définition de la multi-covariance serait alors différente de celle de la covariance et s'appliquerait uniquement sur les arguments non cibles.

Sous-typage

Nous pouvons maintenant définir la relation de sous-typage sur les types de la définition 6.5.1. La relation de sous-typage porte uniquement sur les classes.

Définition 6.5.3 (sous-typage)
$TypeA \leq TypeB \iff$

- *either* $TypeA = TypeB$
- *or* $TypeA \in ClassId \ \wedge\ TypeB \in ClassId \ \wedge\ \forall m_B \in K(TypeB),\ \exists m_A \in K(TypeA) \bullet m_A \leq_{scv} m_B$

où la relation $\leq_{scv}$ est celle de la définition 6.5.2. La règle de sous-typage des classes est liée à l'héritage, puisqu'on tient compte du comportement complet. Mais contrairement à *ako*, qui est un ordre partiel strict, $\leq$ est un ordre partiel.

La règle de redéfinition en simple covariance est cohérente avec les règles d'héritage.

Définition 6.5.4 (règle de redéfinition des méthodes)
$m_S \in K(S) \Longrightarrow \forall C \leq S,\ \exists m_C \in K(C) \bullet m_C \neq m_S \ \wedge\ m_C \leq_{scv} m_S.$

La fonction `profile` trouve la plus petite (relative à $\leq$) définition au-dessus de C; elle est supposée monotone.

Définition 6.5.5 (recherche monotone)
$\forall C \leq S,\ profile(m, C) \leq_{scv} profile(m, S)$

Règles de typage

Les règles de typage sont assez classiques, le type statique d'une expression est celui donné à sa définition. Un premier travail de semi-inférence pour les méthodes primitives est réalisé lors de la traduction en syntaxe abstraite.

Un contexte de typage Γ est constitué d'identificateurs de variable et de méthodes associées à des types. Si `Self` est présent son type est celui de la classes courante `CFC`. De plus le contexte contient les relations de sous-type issues des relations d'héritage. A cause du polymorphisme, une expression a plusieurs types, `e : S` signifie que le terme `e` a au moins `S` comme type (statique) principal. Les règles d'inférence de type sont les suivantes :

Définition 6.5.6 (type statique)
Soient C, C_i, S, S_i des types,

1. *Typage des constantes*

$$\frac{}{\Gamma \vdash c\ :\ BasicId}$$

2. *Typage explicite des méthodes*

$$\frac{}{\Gamma \cup \{m : S_1\ S_2\ ...\ S_n \longrightarrow S\} \vdash m : S_1\ S_2\ ...S_n \longrightarrow S}$$

3. *Typage explicite des variables*

$$\frac{}{\Gamma \cup \{v : C\} \vdash v : C}$$

4. *typage d'un appel d'opération (de base)*

$$\frac{\forall i\ \Gamma \vdash e_i : S_i \qquad \Gamma \vdash op : S_1\ S_2\ ...\ S_n \longrightarrow S}{\Gamma \vdash op(e_1,\ ...,\ e_n) : S}$$

5. *typage d'un envoi de message*

$$\frac{\forall i\ \Gamma \vdash e_i : C_i \leq S_i \qquad \Gamma \vdash profile(m,\ C_1) = S_1\ S_2\ ...\ S_n \longrightarrow S}{\Gamma \vdash m(e_1,\ ...,\ e_n) : S}$$

Le principe de substitution n'est valable que pour les instances des classes. Il n'y a pas substitution sur les méthodes, celles-ci ne sont pas des valeurs à part entières.

Théorème 6.5.7 (principe de substitution)
Soient `t : R` et `e : S` deux expressions telles que `e` soit un sous-terme de `t`. Pour tout `e' : C` $\leq$ `S`, `t[e'/e] : T` $\leq$ `R`.

où `t[e'/e]` est le remplacement des occurrences de `e` de `t` par `e'`. Ce théorème signifie que la substitution est monotone vis-à-vis du typage. C'est une propriété[4] forte, qui n'est pas vraie en général. Elle est démontrée dans [Roy94].

[4] Elle est appelée Type Update Lemma dans [CGL92].

Contrôle de type

Le contrôle de type est réalisé par la fonction `check : Program Context` $\longrightarrow$ `Boolean` $\times$ `Context`. Le résultat est soit un échec soit un succès et un environnement. Le principe de l'algorithme de vérification est simplement de parcourir les définitions des classes et des méthodes et de vérifier que toutes les expressions ont bien un type statique.

Avec les règles précédentes, le contrôle de type est sûr : il n'y a pas d'erreurs de type à l'exécution. La démonstration de la sûreté du contrôle de type est donnée dans [Roy94]. Cependant les règles sont trop strictes, notamment pour la redéfinition multi-covariante des méthodes que nous examinons dans la section 6.5.3.

6.5.2 Le problème des termes douteux

Le problème et sa solution sont décrites dans [GM87a, JKKM89] dans le cas des algèbres à sortes ordonnées, et dans [AR94b] pour les classes formelles.

Prenons l'exemple de la liste, modélisée par les classes formelles de [AR94b]. Le profil exact de la fonction `cdr` est `FullListC` $\longrightarrow$ `ListC`. Le terme `cdr(Xl)`, suivant que `Xl` désigne dynamiquement une liste de longueur 1 ou plus, a un pour type dynamique `FullListC` ou `ListC`. Mais le type statique est toujours `ListC` donc le terme `car(cdr(Xl))` est toujours incorrectement typé. Pourtant ceci est parfois une opération tout à fait légitime. Par exemple, `car(cdr(newFullListC(car= 1, cdr= newFullListC(car= 2, cdr= newEmptyListC()))))` est sémantiquement correct. Le problème est que ces termes, qui ont un sens, sont éliminés par le contrôle de type. Ces termes sont dits **termes douteux**.

Définition 6.5.8 (terme douteux)
Un terme douteux est un message `m(r, *)` *avec* `r : S`, `m` $\notin$ `K(S)` *tel qu'il existe un sous-type* `C` *de* `S` *tel que* `m` $\in$ `K(C)`.

La solution utilisée n'est pas d'assouplir le contrôle de type mais de faire en sorte qu'il considère ces termes comme typables. Avant l'évaluation, une **rétraction** est insérée dans le terme. Une rétraction est une opération (partielle) inverse du polymorphisme des instances i.e. de la coercition implicite de l'héritage. Elle permet de convertir (si possible) une instance de la super-classe en l'instance correspondante de la sous-classe.

Définition 6.5.9 (rétraction)
$\mathtt{retract}_{S \rightarrow C}$ `: S` $\longrightarrow$ `C`
`Self : C ==>` $\mathtt{retract}_{S \rightarrow C}$`(Self) == Self`

Si la rétraction est évaluée avec un mauvais argument, elle produit une erreur dynamique appelée `retract-error`. Dans notre exemple, la rétraction est la fonction `ListC` $\longrightarrow$ `FullListC` et le terme précédent est écrit un un terme bien typé
`car(`$\mathtt{retract}_{ListC \rightarrow FullListC}$`(cdr(newFullListC(car= 1, cdr= newFullListC(car= 2, cdr= newEmptyList())))))`. La rétraction est une opération partielle car une liste vide considérée comme une liste n'a pas de rétraction en une liste pleine.

La généralisation d'une telle solution suppose l'unicité de la rétraction à insérer. Cette condition (CUR : Condition d'Unicité de la Rétraction) est : pour toute classe `S` qui ne voit pas la méthode `m`, si `m` est visible dans le sous-graphe d'héritage de `S` alors il existe une unique plus grande super-classe du sous-graphe où `m` est visible. Cette condition est semblable à la régularité des signatures d'OBJ mais adaptée à notre contexte.

Définition 6.5.10 (CUR)
$\forall\ S,\ m\ \notin\ K(S),\ \exists C < S,\ m\ \in\ K(C)\ \wedge\ (\exists!\ T \leq S \bullet \forall\ R \leq S,\ m\ \in K(R) \Rightarrow R \leq T).$

La mise en place de cette solution utilise des algorithmes et des concepts propres à l'optimisation de la recherche des méthodes [AR92b]. Une table indique pour chaque classe et chaque sélecteur si le sélecteur est `visible` ou `invisible` ou la classe cible de la rétraction. Le calcul de cette table se fait par un parcours en largeur à partir de la frontière du graphe d'héritage. L'algorithme de calcul permet de corriger le graphe (ajout de méthodes et de classes abstraites) si la CUR n'est pas vérifiée. A partir de cette table, les rétractions sont calculées. Soient `S` une classe et `m` un sélecteur et `C = table(S, m)`, si `C` est différent de `visible` ou `invisible` alors l'axiome de la rétraction est :

```
;; retract_{S,m} : fonction inverse du polymorphisme d'héritage
   retract_{S,m} : S  ⟶  C
Self : C ==> retract_{S,m}(Self) == Self
```

La définition des méthodes de rétraction est facile à faire dans un LOC, notons qu'Eiffel dispose d'un opérateur explicite pour cela (`?=`).

Cette table est ensuite exploitée par l'analyseur syntaxique des termes. Un terme est douteux s'il contient un envoi, à un objet de type `S` d'un message `m` et si `table(S, m)` $\neq$ `visible + invisible`. La rétraction définie pour `S` et `m` est alors insérée. Finalement, après correction des doutes, les termes sont correctement typés, donc évaluables, ou incorrectement typés et donc rejetés. La complexité d'un tel mécanisme n'est pas un problème puisque le calcul des rétractions doit se faire statiquement.

6.5.3 Multi-covariance

Dans certaines circonstances, il peut être utile d'autoriser une redéfinition d'une méthode d'instance sur d'autres arguments que le receveur ou le résultat. C'est notamment le cas pour `equal?` et `copy`, pour lesquels les paramètres sont spécialisés dans les sous-classes. Pour cela, la règle 6.5.3 de sous-typage est étendue à la multi-covariance $\leq_{mcv}$ de la définition 6.5.2. Une telle méthode est dite multi-covariante et notée MCV.

En sélection simple , cela pose un certain nombre de problèmes. En sélection multiple, il n'y a pas de problème, mais les règles peuvent être affinées.

Sélection simple

Le problème décrit ci-dessous est très général : c'est le premier problème d'Eiffel cité dans [Coo89], il est discuté dans [Cas95] pour la sélection multiple.

Base
...

Extra
inherits from Base
...

Super	
aspect :	
field selectors	**constraint**
a : Super ⟶ Base	
extensions	

```
;; setup : a problematic method
   setup : Super  ⟶  Super
setup(Self) == copy(Self, a= newBase())
```

Sous	
inherits from Super	
aspect :	
field selectors	**constraint**
a : Sous $\longrightarrow$ Extra	

main: `setup(newSous(a= newExtra()))`

Figure 52 : Un problème classique de multi-covariance en sélection simple

La méthode `copy` est MVC et redéfinie dans la sous-classe. Le terme principal est correctement typé mais son évaluation produit une erreur. Ce genre de problème est inverse de celui des termes douteux. Dans ce cas il est clair que les règles précédentes n'assurent plus un contrôle sûr.

Ces problèmes sont résolus en utilisant des définitions contravariantes, comme le montre [Coo89, CL94]. D'autres travaux [Joh86, OPS92] existent, mais utilisent une extension des méthodes héritées dans les sous-classes. Comme pour Eiffel, nous avons choisi d'utiliser uniquement la covariance. Meyer [Mey89a] présent une solution générale et complexe.

Une solution classique est de transformer la multi-sélection en simple par ajoût de méthodes auxiliaires. C'est ainsi que nous avons écrit les axiomes de la méthode d'égalité de la figure 46. Cette fonction auxilliaire est privée et rédéfinie dans toutes les sous-classes. La définition est un peu lourde mais si elle est effectuée par le système (primitive) ce n'est pas gênant.

Une solution générale mais restrictive a été proposée dans [Roy94]. Il y a un problème si lors de l'évaluation une méthode MVC, la méthode dynamique est différente de la méthode statique. L'idée est d'autoriser les MCV mais elles ne doivent par être héritées. Par exemple, la définition suivante non-équivalent de `setup` est correcte : `setup(Self) == copy(Self, a= a(Self))`. La vérification est faite en calculant les domaines statiques et dynamiques des méthodes MCV pour chaque classe de définition et en montrant que la couverture dynamique est toujours incluse dans la couverture statique. Le domaine statique pour une méthode MCV et une classe qui la définit est la classe et ses sous-classes qui ne redéfinissent pas la méthode. Le domaine dynamique pour cette classe et une méthode qui a un objet de cette classe en receveur est l'ensemble des classes possible dynamiquement. Le système de type est très contraignant car on ne peut utiliser dynamiquement une méthode MCV surchargée.

Sélection multiple

En sélection multiple, l'évaluateur est modifié pour prendre tous les arguments en compte et non pas seulement le receveur. A cause de la monotonie (définition 6.5.5) la bonne méthode est trouvée par l'évaluateur et le problème précédent disparaît.

Cette partie est proche de [CGL92] et [ADL91]. Les hypothèses et preuves sont similaires. La covariance entre les méthodes statiques et dynamiques est assurée par la monotonie de la recherche de la méthode. Une autre façon de faire est de trouver un ordre total sur les méthodes [ADL91]. Cette pratique est contraignante car elle oblige la relation sur des classes non liées par typage. En supposant que cet ordre est une extension de la relation de sous-typage, ce qui est naturel et fréquemment réalisé, alors il y a multi-covariance et recherche monotone dans [ADL91].

Certains systèmes préconisent la multi-sélection sur une partie seulement des arguments. Pour avoir un typage sûr, les résultats précédents sont adaptés. Les règles informelles sont : un argument cible peut être covariant, la recherche est monotone sur les arguments cible, les règles de vérification MVC est appliquée aux arguments non cibles covariants. Une telle approche est présentée dans [CL94] : les arguments cibles sont covariants et les non-cibles sont contravariants. Elle nécessite la cohérence et la complétude des méthodes, ce qui semble contraignant.

6.6 Extensions du modèle

L'intégration de nouveaux concepts enrichit le modèle de base et facilite son utilisation. Nous développons ici quelques extensions à apporter au modèle des classes formelles.

6.6.1 Multi-générateur et méthodes de classe

Jusqu'à maintenant, nous avons considéré l'unique générateur `newC`. Avoir d'autres méthodes de classes facilite les descriptions de comportement généraux. A ce niveau, les méta-classes ne sont pas encore considérées et les méthodes de classe sont uniques dans le système. Elles ont un profil unique, ne contenant pas la classe dans leur domaine et n'ont pas la variable `Self`. La définition d'un nouveau générateur peut ou non être une extension des générateurs existant. L'ensemble des générateurs est noté `newC, copy,` $\texttt{gbuser}_m$, $\texttt{gruser}_n$.

Les règles de typage sont étendues pour prendre en compte les appels de générateurs. De nouveaux contrôles doivent être mis en place pour vérifier que les générateurs ne perturbent pas l'aspect de la classe.

Ecriture des méthodes

Avoir plusieurs générateurs offre de nouvelles possibilités de définition des extensions. Pour les constructeurs il existe une forme simple qui consiste à les définir relativement aux sélecteurs de champ par des axiomes sûres [Gut80]. Ce type de définition ajoute un nouveau générateur dans la classe. D'autres types de définitions sont possibles notamment en utilisant la G-dérivation définie dans la section 5.4.5.

Définition 6.6.1 (présentation sûre)
C'est un ensemble de règles de réécriture du type $< fsel_i > (g(*), *) \longrightarrow t_p$ *où* $fsel_i$ *décrit les sélecteurs de champ et* g *est le générateur à définir.*

Par exemple `newtotal` pour une carte bancaire :

```
idClient(newtotal(Self, Xsomme))  ⟶ idClient(Self)
cumul(newtotal(Self, Xsomme))  ⟶ cumul(Self) + Xsomme
date(newtotal(Self, Xsomme))  ⟶ date(Self)
```

Une deuxième forme plus sophistiquée peut parfois être utile.

Définition 6.6.2 (présentation G-développée)
C'est un ensemble de règles de réécriture de la forme $e(g(y_1, ..., y_n), x_1, ..., x_m) \longrightarrow t_c$ *où* e *est l'extension à définir,* g *décrit l'ensemble des générateurs et les* y_i, x_j *sont des variables distinctes.*

Dans le cas des générateurs ayant une syntaxe simplifié comme `new` et `copy`, seul le profil le plus général avec tous les arguments est pris en compte. Dans l'exemple de `newtotal` cela donne :

```
newtotal(newCarte(Xin, Xid, Xdate), Xsomme))
  ⟶ newCarte(idClient = Xid, cumul = Xin + Xsomme, date = Xdate)
newtotal(copy(Self, Xin, Xid, Xdate), Xsomme))
  ⟶ copy(Self, idClient = Xid, cumul = Xin + Xsomme, date = Xdate)
```

L'éventail des présentations quasi-opérationnelles est étendu.

Définition 6.6.3 (présentation quasi-opérationnelle)
C'est une règle de réécriture de la forme $e(Self, *) \longrightarrow t_c$ *où* e *est l'extension à définir.*

```
newtotal(Self, Xsomme)  ⟶ copy(Self, cumul = cumul(Self) + Xsomme)
-------------------------------------------------------------------
newtotal(Self, Xsomme)  ⟶ newCarte(idClient = idClient(Self),
cumul = cumul(Self) + Xsomme, date = date(Self))
```

6.6.2 Déclarations modulaires et méthodes cachées

Jusqu'à maintenant, les méthodes étaient publiques et donc exportées tandis que les déclarations d'importation sont implicitement liées aux paramètres des méthodes. Dans beaucoup de langages, la visibilités des méthodes peu être affinée en étant déclarée privées ou protégées. Une **méthode privée** est visible uniquement dans la classe de définition. Une **méthode protégée** est visible dans la classe de définition et ses sous-classes. La règle suivante est retenue pour l'héritage de telles méthodes : une méthode publique reste publique; une méthode protégée est redéfinie protégée ou publique.

Une erreur de type supplémentaire est ajoutée dans l'évaluateur si la méthode est appelée en dehors de son contexte. Le mode protégé ne pose pas de problèmes pour le typage, qui vérifie simplement le contexte syntaxique. Par contre, l'héritage des méthodes privées, nécessaire à un héritage sans exceptions, en pose car la portée ne peut être vérifiée statiquement. Une solution consiste à redéfinir ces méthodes privées dans les sous-classes mais le résultat est plus proche des méthodes protégées.

6.6.3 Effets de bord

Beaucoup de programmeur "à objets" ne conçoivent pas la programmation à objets sans effet de bord. Il y a deux façons d'introduire des effets de bord. D'un point de vue typage, les deux voies sont équivalentes et ne rajoutent pas de problèmes supplémentaires à ceux de la section précédente. La première est d'ajouter un opérateur d'affectation et de modifier l'évaluateur. Mais ceci alourdit le modèle et le rend moins naturel et surtout les preuves deviennent beaucoup plus difficiles. La seconde est de considérer une identité d'objet facilement intégrable dans le modèle fonctionnel. Cette notion stipule les principes suivants :

- Un objet à une identité propre qui lui est assigné au moment de sa création.
- Cette identité est immuable par modification.
- Tout générateur utilise une nouvelle identité à chaque appel.
- L'égalité d'implantation permet de comparer les identités de deux objets.

La technique est relativement simple : définir un type d'identificateur avec les propriétés nécessaires, ajouter dans chaque classe un champ `identity: Ident` et définir les axiomes primitifs correspondants. La méthode primitive `modify!` modifie la valeur de l'objet receveur.

6.6.4 Généricité

Nous utilisons le fait que l'héritage permet de simuler la généricité. Pour cela, nous généralisons les expérimentations faites avec les listes dans [AR94b], où nous avions introduit un nouveau concept : le *schéma*.

Définition 6.6.4 (schéma)
Le schéma `S` est une conception ordonnée d'un type de données telle qu'il existe une conception plate équivalente.

Un schéma est donc un ensemble de classes, appelées CTS (Classe Terminale du Schéma), liées par la relation d'héritage à une racine unique, représentant le type de données. Il peut être actualisé par un type paramètre ou une contrainte. Par exemple, la conception ordonnée des listes constitue un schéma paramétré par le type `T`. Une première approche est de donner un statut uniquement syntaxique aux schémas. Il semble plus intéressant de donner un statut d'objet aux schémas. Un héritage des schéma a été défini. Dans cet ordre d'idée, une bonne solution passe par l'utilisation des métaclasses (voir [AR94b]). Mais une autre alternative est celle choisie par Bertand Meyer [Mey88].

6.6.5 Multi-sélection

Il est possible d'introduire la multi-sélection [ADL91, CL94]. Notons qu'il est possible de transformer la multi-sélection en sélection simple de façon plus ou moins élégante [AR94b]. Les avantages de la multi-sélection sont principalement une amélioration des problèmes de typage que nous avons rencontré (redéfinitions inutiles de méthodes, etc.), un encadrement du polymorphisme des méthodes, un rapprochement évident avec les types abstraits algébriques facilitant leur conception immédiate. Les inconvénients majeurs concernent l'algorithme de recherche des méthodes et la stratégie d'héritage (règles de priorité, détermination des conflits, ordre sur les classes et les méthodes), ainsi que l'organisation modulaire des méthodes vis-à-vis des classes. Nous n'allons pas discuter du problème général des multi-méthodes (méthodes pour lesquelles la sélection est multiple i.e. se fait sur plusieurs arguments) car il est très complexe. Consulter pour cela [DHH+95, ABDS95, ADL91, CL94]. Nous avons fait des travaux sur ces problèmes de recherche de méthodes en sélection simple [AR92b], ils peuvent être adaptés à la sélection multiple.

6.6.6 Multi-aspect

Notre modèle permet de donner plusieurs caractérisations équivalentes d'un même objet. Les différents aspects d'un objet s'utilisent d'une manière uniforme, on dit que la classe est multi-aspect [AR92c]. Par exemple, un point peut être polaire ou cartésien. L'équivalence des aspects est assurée par les fonctions de conversion qui définissent des bijections entre les objets. Ces fonctions permettent de transformer une description associée à un aspect en une autre description (du même objet) associée à un autre aspect. Pour lever les ambiguïtés lors de la création d'une instance plusieurs solutions sont possibles. Nous avons adopté une solution simple : les noms des champ des différents aspects sont tous distincts et préfixent les arguments correspondant à la création de l'objet.

La principale différence par rapport au cas d'une classe mono-aspect est l'existence des fonctions de conversion. Les multi-aspects permettent une meilleure convivialité du modèle. Par exemple la méthode `rotate` est plus facile à définir sur l'aspect polaire que sur l'aspect cartésien, et inversement pour la méthode `move`. Ils permettent aussi de donner une sémantique de l'héritage rigoureuse et plus conforme à l'intuition du concepteur. La contrainte et les fonctions de conversions ne sont pas visibles directement par l'utilisateur. Leur utilisation s'effectue au travers des méthodes primitives. D'autres informations sur ces possibilités se trouvent dans [BW85].

6.6.7 Métaclasses

Les classes du modèle précédent ne sont pas a priori des objets. Un système de métaclasse est envisageable. L'intérêt est d'étendre le modèle, ou de personnaliser les mécanismes de base comme l'héritage ou l'instanciation. Nous avons réalisé des expérimentations qui montre que l'intégration est réalisable d'un point de vue opérationnel [AR94b]. Cependant l'introduction dans une théorie du premier ordre est difficile. Nous adoptons comme base le modèle ObjVLisp de P. Cointe [Coi87] pour ses qualités d'uniformité, de simplicité et de minimalité.

6.7 Conclusion

Dans ce chapitre, nous avons posé les bases d'un modèle de programmation à objets avec classes et sans méta-objets. Nous avons donné une trame pour l'interprétation algébrique de ce modèle en essayant d'être assez général au niveau des définitions de structures comme des définitions de méthodes. Un certains nombre de propriétés et de définitions sont exposés, notamment pour le contrôle de l'héritage et du typage. Un certain nombre de points, sur le typage notamment restent à approfondir, de même que l'intégration de nouveaux concepts tels que ceux proposés dans la section 6.6.

Le modèle est déjà opérationnel et nous l'avons expérimenté sur un certain nombre de cas concrets. Ce modèle a déjà servi à spécifier et concevoir des exemples simples mais non triviaux comme la commande **makeindex** de LaTeX [Roy93], un système d'ascenceur de l'annexe C, l'unification de termes, ... Par ailleurs des spécifications de composants ont été étudiées comme les listes [AR94b]. L'utilisation de ce modèle pour la conception à objets fait l'objet de travaux.

La conception des classes formelles retient les aspects suivants : description minimale et formelle facilitant la réutilisabilité et la correction. La description des objets débute par la description des aspects (sélecteurs de champ et contrainte). Elle est suffisante pour étudier la mise en place dans le graphe d'héritage avec respect des contraintes. Ce point comprend la réutilisation de composants existants par restructuration de l'aspect et les relations d'héritage définies sur des classes existantes. Les extensions sont alors définies et permettent le contrôle de type et des preuves de propriétés sur les classes. Enfin, un schéma de traduction permet d'implanter, au moins partiellement, les classes formelles dans divers langages de programmation à objets. Ce qui constitue un avantage du modèle en termes de réutilisation inter-langages. Nous verrons cet aspect processus de développement dans le chapitre suivant.

Chapitre 7

Démarche de conception

"C'est une leçon que vous devriez observer : Essayez, essayez, essayez encore. Si, tout d'abord, vous ne réussissez pas, Essayez, essayez, essayez encore. "
W.-E. HICKSON, "Essayer et essayez encore"..

7.1 Introduction

Un objectif majeur des spécifications formelles est la dérivation assistée des spécifications en programmes tant pour le prototypage que la conception finalisée des composants logiciels. Ainsi que nous l'avons introduit dans le chapitre 4, nous avons privilégié une approche multi-langage, un langage de spécification de haut niveau, un langage de conception abstrait et les langages de programmation à objets. Une démarche de conception est indispensable pour guider le concepteur dans l'élaboration d'un modèle concret à partir d'un modèle plus abstrait ou tout au moins pour obtenir plus facilement un prototype correct. La démarche doit être souple et s'adapter aux besoins du concepteur. Voici une vue générale de la démarche de conception associée aux modèles TAG/CF.

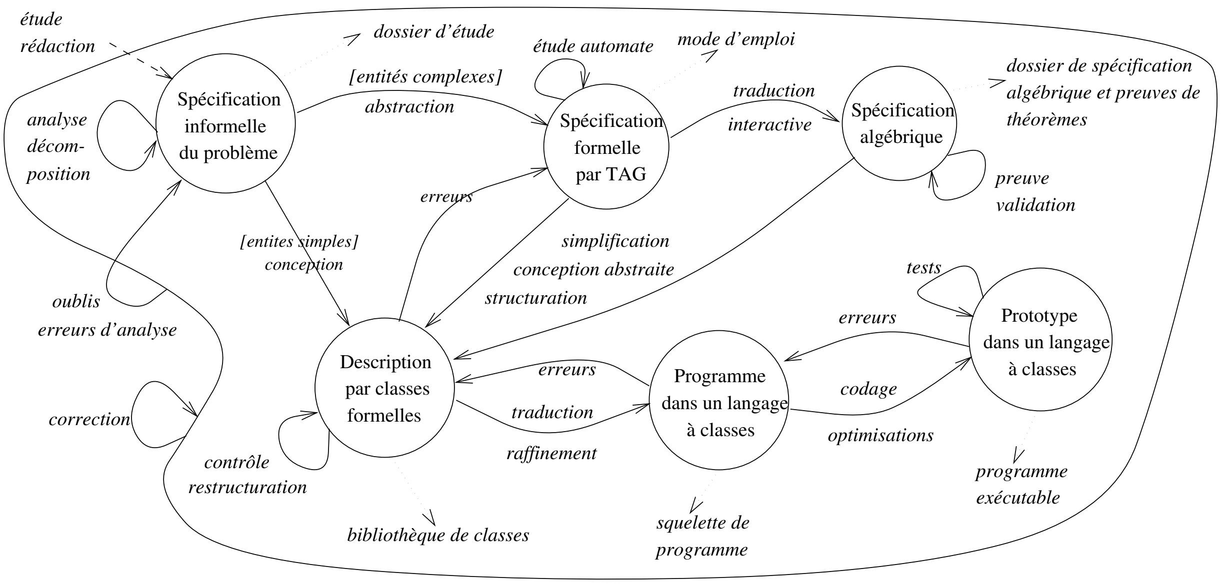


Figure 53 : Processus de spécification

Le modèle TAG peut être utilisé pour décrire sommairement les objets de l'analyse ou bien en complément d'autres formalismes que nous avons abordé dans la section 2.3. Il permet une description abstraite des objets sans se soucier de leur structure (agrégation, attributs) mais en tenant compte des contraintes d'enchaînement des opérations. Cette description peut ensuite être affinée dans une spécification algébrique pour mettre en valeur les propriétés fonctionnelles ou simulée par exécution concurrente d'automates communicants pour dégager les propriétés dynamiques.

Le modèle des classes formelles est utilisé pour la conception abstraite des classes. La conception par classes formelles comprend plusieurs tâches : extraire des classes à partir des TAG (structure et comportement), restructurer les classes obtenues pour les intégrer aux classes existantes (raffinement pour réutiliser des classes existantes, généraliser pour prendre en compte la hiérarchie d'héritage, optimisations pour rendre plus efficace). Il sert enfin de support pour la traduction dans différents environnements de programmation.

Une spécification abstraite décrit un ensemble de représentations concrètes. Dans les spécifications formelles, le raffinement consiste à trouver une représentation de plus en plus proche des concepts des langages de programmation. C'est un problème crucial pour les méthodes formelles [Abr84, Gog90, Jon93, San90, Wir93]. Le problème du changement de représentation a été initialement présenté dans [Hoa72]. Il s'agit de montrer que la spécification raffinée, i.e. plus proche des concepts de la programmation, est correcte vis-à-vis de la spécification initiale. Dans les spécifications algébriques, la notion de morphisme permet d'établir la relation de raffinement, appelée aussi simulation dans [BG94b]. Dans les spécifications orientées modèles, des théories ont été définies, supportées par des environnements (outil B[LLS91] pour Z, Mural[Jon91, JJLM91] pour VDM). Le raffinement est vu alors comme un raffinement des données puis des opérations (promotion en Z). L'héritage permet aussi de donner une sémantique formelle au raffinement [Bre91, BG94b, CO88, LH92, LH93, PPP91, BGM87] et la plupart des méthodes formelles à objet incluent cette notion [Bre91, AG91, CO88, CSM94, DGP94, LH94].

Dans le chapitre 5, nous avons détaillé un algorithme de spécification algébrique à partir des TAG. Dans cette section, nous nous intéresserons aux deux autres étapes majeures de transition entre modèles, symbolisées par un trait plein dans la figure 29 :

- le raffinement des TAG aux classes formelles,
- la traduction des classes formelles en classes des langages de programmation.

La démarche se doit d'automatiser autant que se peut ces deux transitions. Cette section discute de manière générale du passage des TAD aux classes et présente l'approche proposée dans le cadre TAG/CF. A notre avis, une séparation nette est indispensable entre conception des classes et implantation des classes est nécessaire, d'où l'intérêt des classes formelles. Nous allons illustrer la démarche sur un exemple : l'ascenseur `Lift`. La spécification du TAG `Lift` est donnée en annexe C.

7.2 Passage des types abstraits graphiques aux classes

L'objectif de cette étape est de donner une représentation des TAG en terme de classes. Les types abstraits sont structurés en classes et les opérations sont raffinées en méthodes. L'originalité de la méthode réside dans le fait qu'un type abstrait est raffiné non par une classe mais par un ensemble de classes. Ce qui favorise l'extension du système et sa vérification.

Dans ce qui suit, nous considérons que la spécification algébrique du TAG est facultative. Le raffinement est une suite d'étapes utilisant des techniques relativement indépendantes les unes des autres. Ces techniques s'appliquent à des problèmes plus généraux, que nous indiquerons en introduisant chaque étape. Le processus nécessite l'intervention du concepteur pour fixer certains choix, notamment de représentation. L'objectif est de fournir un certain nombre d'outils et d'algorithmes pour supporter la démarche et assister le concepteur.

7.2.1 Problématique

Les classes formelles raffinent les TAG en donnant une représentation particulière mais abstraite. Les opérations deviennent des méthodes, et les axiomes, s'ils existent sont groupés dans les méthodes. Deux approches extrêmes sont possibles pour passer d'un type (abstrait) de données à une classe (formelle) : la conception plate et la conception ordonnée.

Conception plate

En conception **plate**, une seule classe décrit le TAG. Si l'automate est représenté par des prédicats d'état ou une fonction de transition alors les préconditions des méthodes sont calculables directement par l'algorithme de la section 5.4.3. Dans le cas contraire, le concepteur les exprime en fonction d'une structure abstraite qu'il aura dégagé. Supposons maintenant une structure abstraite quelconque. Puisque les observateurs peuvent être partiels, certains sélecteurs de champ peuvent ne pas avoir de valeur à un instant donné. Nous utiliserons de préférence un champ conditionnel à la valeur indéfinie (`nil` ou $\perp$) usuelle en programmation. Examinons une conception plate et intuitive du TAG `Lift` par la classe formelle `FlatLift`.

FlatLift
inherits from `OBJECT`
comments: `classe formelle de l'ascenseur Lift`
param: `Weightable`

features: `limits, bottomLevel, topLevel, level, capacity, weight,`
`up, down, arrived, chgCapacity, getIn, getOut, getOOO, repaired, restart, stop,`
`new, copy, isEqual, describe, install`

aspect : `lift`

field selectors	constraint
`bottomLevel : FlatLift ⟶ Integer`	
`topLevel : FlatLift ⟶ Integer`	`bottomLevel(Self) < topLevel(Self)`
`capacity : FlatLift ⟶ RealGt1`	`and`
`contents : FlatLift ⟶ Set[Weightable]`	`0 <= weight(Self)`
`isOnFloor : FlatLift ⟶ Boolean`	`and`
`level : FlatLift ⟶ Integer` **requires:** `isOnFloor(Self) == true`	`bottomLevel(Self) <= level(Self) <=`
`isOverloaded : FlatLift ⟶ Boolean` **requires:** `isOnFloor(Self) == true`	`topLevel(Self)`
`isOnTop : FlatLift ⟶ Boolean` **requires:** `isOnFloor(Self) == true`	`and`
`isOnBottom : FlatLift ⟶ Boolean` **requires:** `isOnFloor(Self) == true`	`not(isOnTop(Self) and isOnBottom(Self))`
`isGoingUp : FlatLift ⟶ Boolean` **requires:** `isOnFloor(Self) == false`	`and`
`distance : FlatLift ⟶ IntegerGt1` **requires:** `isOnFloor(Self) == false`	`bottomLevel(Self) <= distance(Self) <=`
`isStopped : FlatLift ⟶ Boolean` **requires:** `isOnFloor(Self) == false`	`topLevel(Self)`
`isOutOfOrder : FlatLift ⟶ Boolean`	

extensions

```
;; limits : provides the lift limits (top and bottom levels)
   limits : FlatLift    ⟶  Tuple(Integer Integer)
limits(Self) == newTuple(bottomLevel(Self), topLevel(Self))
```

```
;; chgCapacity : set a new capacity for the lift
   chgCapacity : FlatLift RealGt1  ⟶  FlatLift
var: Xcap : RealGt1
chgCapacity(Self, Xcap) == copy(Self, capacity = Xcap)
```

```
;; up : gets up Xd levels
   up : FlatLift IntegerGt1  ⟶  FlatLift
var: Xd : IntegerGt1
isOnTop(Self) == false ==> up(Self, Xd) ==
         copy(Self, isOnFloor = false, distance = Xd, isGoingUp = true)
```

```
                              FlatLift
                             extensions
;; down : gets down Xd levels
   down : FlatLift IntegerGt1  −→  FlatLift
var: Xd : IntegerGt1
isOnBottom(Self) == false ==> down(Self, Xd) ==
         copy(Self, isOnFloor = false, distance = Xd, isGoingUp = false)
;; arrived : the lift reaches the required level
   arrived : FlatLift    −→  FlatLift
isGoingUp(Self) == true ∧ (level(Self) + distance (Self) = topLevel(Self)) == true ==>
         arrived(Self) == copy(Self, isOnFloor = true, isOnBottom = false,
                          isOnTop = true, level = topLevel(Self))
isGoingUp(Self) == true ∧ (level(Self) + distance (Self) < topLevel(Self)) == true ==>
         arrived(Self) == copy(Self, isOnFloor = true, isOnBottom = false,
                          isOnTop = false, level = level(Self) + distance (Self))
isGoingUp(Self) == false ∧ (level(Self) - distance (Self) = bottomLevel(Self)) == true ==>
         arrived(Self) == copy(Self, isOnFloor = true, isOnBottom = true,
                          isOnTop = false, level = bottomLevel(Self))
isGoingUp(Self) == false ∧ (level(Self) - distance (Self) > bottomLevel(Self)) == true ==>
         arrived(Self) == copy(Self, isOnFloor = true, isOnBottom = false,
                          isOnTop = false, level = level(Self) - distance (Self))
;; weight : provides the total weight the lift
   weight : FlatLift    −→  Real
weight(Self) == sum(contents(Self),weight)
;; getIn : a weightable gets inside the lift
   getIn : FlatLift Weightable  −→  FlatLift
var: Xw : Weightable
isOverloaded(Self) == false ∧ (weight(Self) + weight(Xw) <= capacity(Self)) == true ==>
         getIn(Self, Xw) == copy(Self, contents = contents + {weight(Xw)})
isOverloaded(Self) == false ∧ (weight(Self) + weight(Xw) > capacity(Self)) == true ==>
         getIn(Self, Xw) == copy(Self, contents = contents + {weight(Xw)}, isOverloaded = true)
;; isIn : indicates if the lift contains a weightable
   isIn : FlatLift Weightable  −→  Boolean
var: Xw : Weightable
isIn(Self,Xw) == belongsTo(contents(Self),Xw)
;; getOut : a weightable gets outside the cage
   getOut : FlatLift Weightable  −→  FlatLift
var: Xw : Weightable
isOverloaded(Self) == false ∧ isIn(Self,Xw) == true ==>
            getOut(Self, Xw) == copy(Self, contents = contents - {Xw})
isOverloaded(Self) == true ∧ isIn(Self,Xw) == true ∧
         (weight(Self) - weight(Xw) > capacity(Self)) == true ==>
            getOut(Self, Xw) == copy(Self, contents = contents - {Xw})
isOverloaded(Self) == true ∧ isIn(Self,Xw) == true ∧
         (weight(Self) - weight(Xw) <= capacity(Self)) == true ==>
            getOut(Self, Xw) == copy(Self, contents = contents - {Xw}, isOverloaded = false)
;; stop : stops the lift
   stop : FlatLift    −→  FlatLift
isStopped(Self) == false ==> stop(Self) == copy(Self, isStopped = true)
;; restart : restarts the lift
   restart : FlatLift    −→  FlatLift
isStopped(Self) == true ==> restart(Self) == copy(Self, isStopped = false)
;; getOOO : the lift becomes out of order
   getOOO : FlatLift    −→  FlatLift
isOutOfOrder(Self) == false ==> getOOO(Self) == copy(Self, isOutOfOrder = true)
;; repaired : the lift is being repaired
   repaired : FlatLift    −→  FlatLift
isOutOfOrder(Self) == true ==> repaired(Self) ==
         install(FlatLift, bottomLevel(Self), topLevel(Self), capacity(Self))
                            class methods
;; install : build an initial lift
   install : Integer Integer RealGt1  −→  FlatLift
var: Xbot : Integer,Xtop : Integer,Xcap : RealGt1
install(Xbot, Xtop, Xcap) == newFlatLift(bottomLevel = Xbot, topLevel = Xtop,
                   capacity = Xcap)
```

Figure 54 : La classe formelle intuitive **FlatLift**

La conception d'une telle classe n'est pas triviale à partir du TAG. Les inconvénients de cette approche sont : descriptions souvent opérationnelles, préconditions complexes, classes volumineuses pas facile à lire, à comprendre et à spécialiser.

Conception ordonnée

En conception **ordonnée**, un TAG est défini par un schéma. Rappelons qu'un schéma est un sous-ensemble du graphe d'héritage de racine unique une classe abstraite représentant le type de données. Par défaut, le schéma est initial , c'est-à-dire qu'il comporte deux niveaux : une racine et des feuilles appelées classes terminales. La racine est une classe abstraite correspondant au TAG. Les classes terminales représentent les états (le nom de la classe est la concaténation du nom du type et du nom de l'état). Voici le schéma initial du TAG `Lift`.

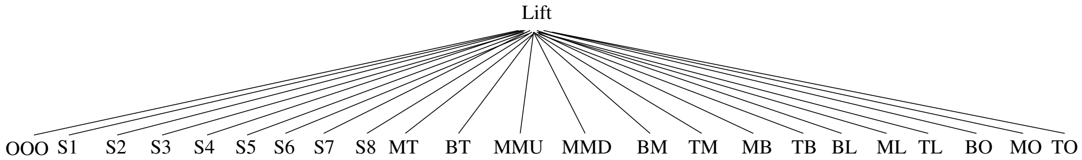


Figure 55 : Schéma initial pour l'ascenseur `Lift`

Cette représentation est induite par la sémantique algébrique des états : un état est un sous-type (une algèbre) du type d'intérêt. Les sélecteurs de champ sont des fonctions totales et les préconditions sont plus simples. Le partitionnement facilite la compréhension et la réutilisation mais risque d'entraîner une explosion combinatoire du nombre de classes avec duplication de méthodes identiques (celles des opérations applicables à plusieurs états) : pas de partage de code tant qu'on n'introduit pas des classes abstraites intermédiaires. Un autre inconvénient important est la médiocre représentation en termes d'objet qui conduit à de l'héritage dynamique, c'est-à-dire qu'un objet est instance de différentes classes au cours de son existence.

7.2.2 Stratégie de conception

Nous souhaitons une approche plus fine que la conception plate et plus efficace que la conception ordonnée ci-dessus. Cette approche intermédiaire est résumée dans la figure suivante.

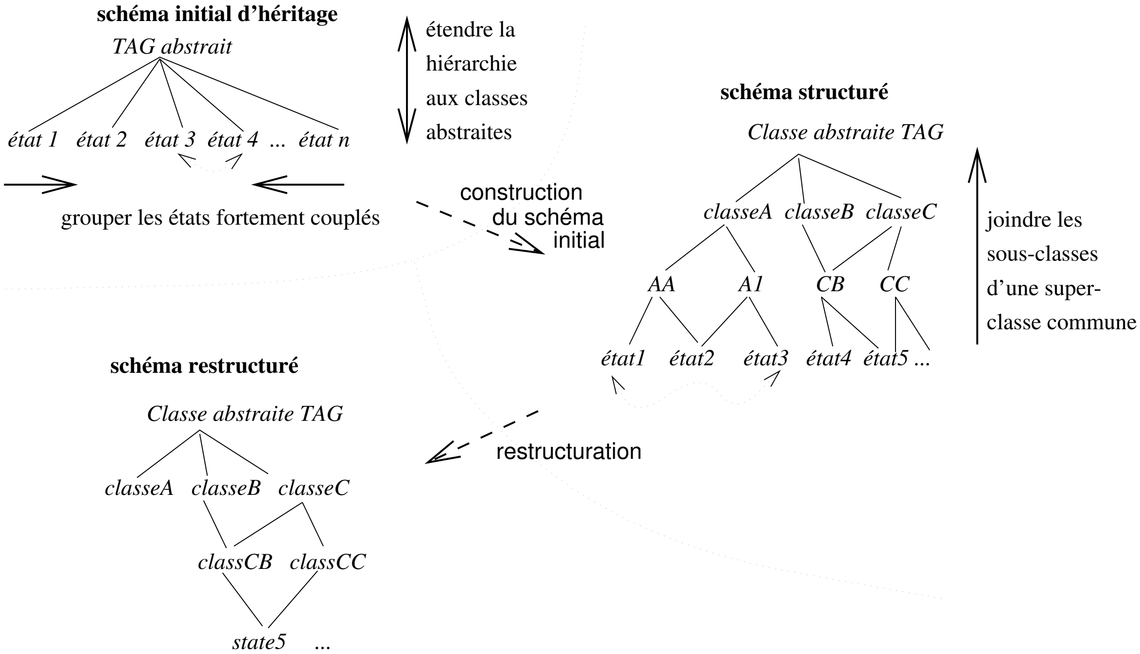


Figure 56 : Stratégie de conception d'un TAG en classes formelles

L'idée est de partir du schéma initial, de le factoriser puis de le structurer afin de mettre en évidence les comportements communs dans des classes abstraites intermédiaires et enfin de factoriser les sous-classes d'une super-classe pour optimiser la représentation. Les classes terminales sont regroupées soit parce qu'elles ont un comportement très proche (réduction verticale) soit parce que les états dont elles sont issues sont fortement liés (réduction horizontale). Nous suivons en cela les principes habituels de la conception modulaire : réduction du couplage modulaire, renforcement de la cohérence modulaire.

Cette approche est adaptée aux TAG complexes tels que l'ascenseur de la figure 79. Par restructuration successive, le schéma est parfois réduit à une seule classe. Ce qui constitue une démarche rigoureuse pour atteindre la conception plate.

Nous proposons donc une méthode de raffinement des TAG en classes formelles en trois étapes principales : (1) extraire un schéma d'héritage à partir du graphe de comportement dynamique, (2) restructurer et regrouper les classes formelles, (3) écrire les extensions.

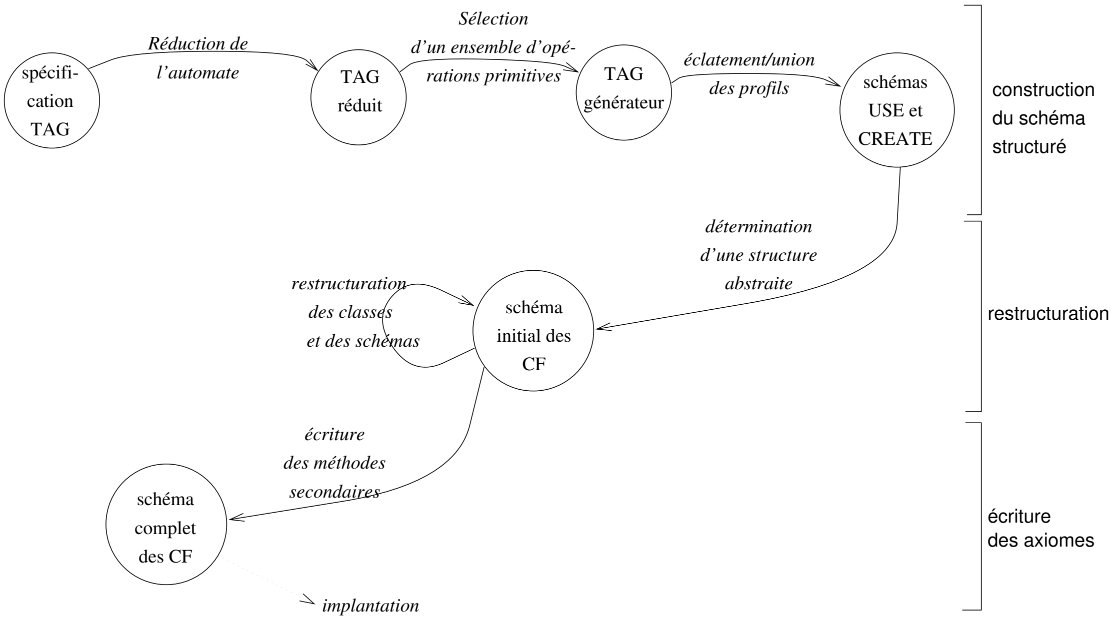


Figure 57 : Cycle de raffinement

Les sections suivantes décrivent les trois étapes de la démarche.

7.3 Etape 1 : Définition d'un schéma structuré

Le but de cette étape est de calculer un premier schéma structuré à partir du schéma initial en groupant les classes terminales représentant des états fortement liés, en calculant des niveaux intermédiaires dans le schéma initial et en donnant une structure abstraite aux classes terminales du schéma. Les niveaux intermédiaires sont des classes abstraites qui serviront à factoriser des comportements communs à plusieurs classes terminales. Afin de minimiser le travail, nous nous focaliserons uniquement sur le comportement primitif, i.e. les opérations primitives du TAG. La définition d'un schéma structuré se fait selon un algorithme en trois phases :

– (i) réduction de l'automate et définition d'un noyau d'opérations primitives;

– (ii) calcul des noeuds internes du schéma d'héritage avec leurs opérations et enfin

– (iii) calcul d'une structure abstraite pour chaque classe terminale.

7.3.1 Réduction de l'automate

Cette phase préliminaire sert à supprimer des transitions et à regrouper des états pour simplifier l'automate et les calculs qui en découle pour les étapes suivantes. L'heuristique choisie est la suivante :

– regrouper les états par l'algorithme de calcul des états frères de la section 5.6.1.

– supprimer des transitions des opérations secondaires par l'algorithme de la section 5.4.5.

Dans notre exemple, les regroupements d'états obtenus sont les suivants : `MB-S1, BM-S2, TM-S5, MT-S6, TL-T0, TB-S7, BT-S8, BL-B0 ML-M0, MMD-S3, MMU-S4`. Pour alléger l'écriture, seul le nom du premier état de l'ensemble sera noté dans la suite. La partition opération primitive/opération secondaire ne tient pas compte des observateurs d'état.

générateurs	$\{install, up, down, stop, arrived, getIn, getOOO\}$
constructeurs (secondaires)	$\{restart, chgCapacity, repaired, getOut\}$
observateurs primitifs	$\{level, capacity, weight, bottomLevel, topLevel, isIn\}$
observateurs (secondaires)	$\{limits\}$

Le schéma initial réduit est :

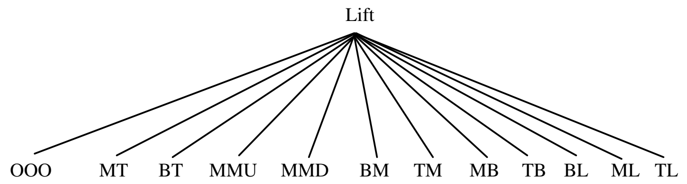


Figure 58 : Schéma initial réduit du type `Lift`

Il correspond à l'automate suivant :

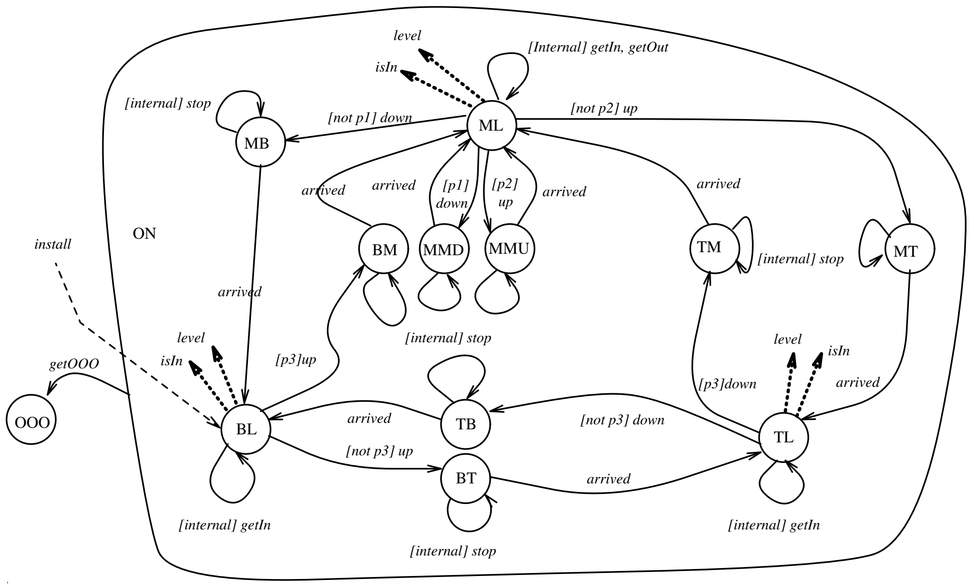


Figure 59 : Comportement dynamique réduit pour l'ascenseur `Lift`

7.3.2 Calcul des noeuds internes

Une analyse plus fine des opérations en fonction des états où elles sont définies permet de calculer des profils plus spécifiques. Un regroupement adéquat de ces profils fait apparaître de nouveaux types, noeuds intermédiaires de la hiérarchie. Illustrons ceci par l'étude des opérations `arrived` et `up`. Les profils les plus spécifiques sont obtenus à partir des transitions elle-mêmes.

```
up        :  MLLift IntegerGt1  ⟶ MTLift
          :  MLLift IntegerGt1  ⟶ MMULift
          :  BLLift IntegerGt1  ⟶ BTLift
          :  BLLift IntegerGt1  ⟶ BMLift

arrived   :  BTLift  ⟶ TLLift
          :  MTLift  ⟶ TLLift
          :  MBLift  ⟶ BLLift
          :  TBLift  ⟶ BLLift
          :  BMLift  ⟶ MLLift
          :  TMLift  ⟶ MLLift
          :  MMDLift  ⟶ MLLift
          :  MMULift  ⟶ MLLift
```

Les profils synthétisés sont de trois sortes :

1. *profil de méthode* : c'est le profil associé à une opération pour un état (une classe) donné(e). Il s'agit de l'union des transitions sortantes d'un état pour une opération donnée. Le type argument est inchangé, le type résultat est l'union des types associés aux états d'arrivée. Pour `arrived`, aucun regroupement n'est possible, pour `up` les profils sont :

```
up        :  MLLift IntegerGt1  ⟶ MUPLift
          :  BLLift IntegerGt1  ⟶ BUPLift
```

   où `MUPLift = {MTLift, MMULift}` et `BUPLift = {BMLift, BTLift}`.

2. *profil d'aspect*: c'est le profil associé à une méthode créant des instances d'une classe donnée. Il s'agit de l'union des transitions entrantes d'un état pour une opération donnée. Le type du premier argument est l'union des types associés aux états de départ, le type résultat est inchangé. Pour `up`, aucun regroupement n'est possible, pour `arrived` les profils sont :

```
arrived   :  TUPLift  ⟶ TLLift
          :  BDOLift  ⟶ BLLift
          :  MLift  ⟶ MLLift
```

   où `TUP = {BT, MT}`, `BDO = {MB, TB}` et `M = {MMU, MMD, BM, TM}` (voir figure 63).

3. *profil général*: c'est le plus grand profil spécifique associé à une opération (i.e. tel que l'opération soit totale pour le receveur). Le type du premier argument est l'union des types associés aux états de départ des transitions associées à cette opération, le type résultat est l'union des types associés aux états d'arrivée des transitions associées à cette opération. Les profils généraux de `up` et `arrived` sont :

```
up        :  BMLLift IntegerGt1  ⟶ BMUPLift

arrived   :  INTERMEDIATELift  ⟶ ONFLOORLift
```

   où `INTERMEDIATE = {MT, BT, MMU, MMD, BM, TM, MB, TB}`, `ONFLOOR = {BL, TL, ML}`, `BML = {ML, BL}`, `BMUP = {MT, MMU, BT, BM}` .

Un algorithme, appelé *split/join*, permet de calculer les profils ci-dessus.

```
for each partial and primitive operation op do
    in(op) = ∅                              // domaine //
    out(op) = ∅                             // co-domaine //
end
```

```
for each state s_i do
     for each transition δ(s_i, op, guard) = s_{i+1} do
          in(op) = in(op) ∪ {s_{i+1}}
          // transition sortant de l'état s_i //
          if s_{i+1} = ε
               then                          // observateur primitif //
               else out(op) = out(op) ∪ {s_{i+1}}   // générateur //
          endif
     end
end
// nommage des ensembles //
domains := emptyDictionary
targets := emptyDictionary
for each operation op do
     exists := find(domains, in(op))
     if not exists then add(domains, name, in(op)) endif
          exists := find(targets, out(op))
     if not exists then add(targets, name, out(op)) endif
end
// calcul des sous-ensembles //
// les domaines sont comparés et une relation d'inclusion est exhibée //
// cette relation est notée en pointillé dans les figures 61 et 62 //
```

Figure 60 : Algorithme de calcul des profils spécifiques des opérations

Notez qu'il s'agit là d'une méthode possible pour calculer les transitions généralisées de la figure 38 ou encore pour dégager une spécification algébrique ordonnée à partir d'une spécification algébrique plate munie d'états.

Les nouveaux types apparaissant dans ces profils sont des classes abstraites, unions de classes terminales. Ils sont classés par inclusion ensembliste en deux schémas :

- Le schéma d'utilisation comprend les profils d'aspect et les profils généraux. Il décrit pour chaque type, l'ensemble des opérations invocables. C'est le graphe d'héritage habituel. Noter que ce schéma respecte la condition d'unicité de la rétraction définie dans la section 6.5.10.
- Le schéma de création comprend les profils de méthode et les profils généraux. Il décrit pour chaque type les opérations qui créent une valeur de ce type (i.e. une instance de la classe correspondante). Il sert à déterminer l'aspect de chaque classe formelle associée à ce type (phase (iii) en section 7.3.3).

Dans la pratique, le nombre de nouveaux types peut être important (13 ici) et un certain nombre d'entre eux n'existent qu'à cause d'une seule opération. Ce qui ne facilite pas la restructuration de la hiérarchie. Dans ce cas, nous utiliserons une nouvelle heuristique : seuls les profils généraux sont conservés pour chaque opération.

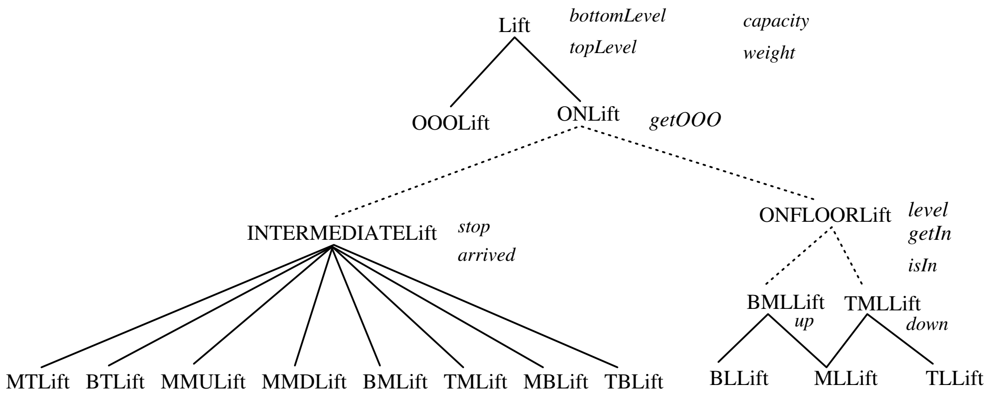


Figure 61 : Schéma d'utilisation du type `Lift`

Cette heuristique et celle de l'étape (i) précédente portant sur le comportement primitif posent le problème suivant : certains types, apparaissant normalement dans les profils des méthodes des classes terminales n'existent plus. Il ne s'agit que des types désignant le receveur ou le résultat d'un constructeur. Par exemple, le profil de méthode `up : MLLift IntegerGt1` $\longrightarrow$ `MUPLift` contient le type inexistant `MUPLift`. La solution est de **relaxer** le type, c'est-à-dire remplacer dans le profil, le type supprimé par le plus petit type commun. Ainsi, le profil de méthode `up` devient `up : MLLift IntegerGt1` $\longrightarrow$ `BMUPLift`. Il n'y a pas de problème de termes douteux, car la condition d'unicité de la rétraction est respectée.

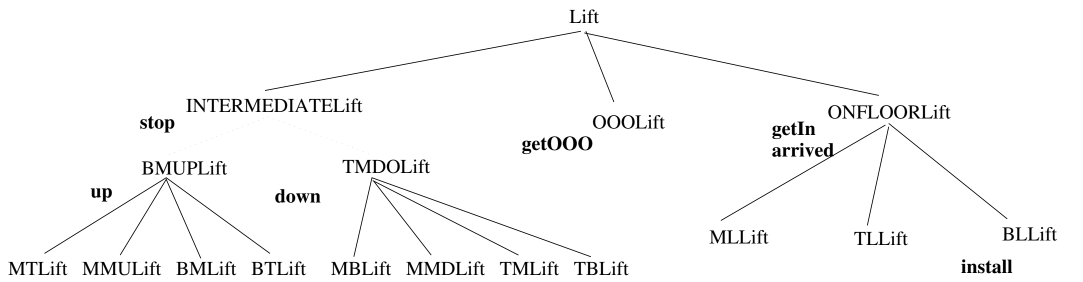


Figure 62 : Schéma de création du type `Lift`

L'intérêt du schéma de création est de calculer des aspect partiels. Mais la fusion des deux schémas crée de nouveaux héritages multiples et introduit des conflits d'héritage. Pour éviter ce problème nous conservons uniquement le schéma d'utilisation. Cette heuristique est acceptable pour les raisons suivantes : premièrement l'héritage de spécialisation se fait non pas sur la structure mais le comportement, deuxièment les types créés n'ont pas de comportement propre. Cette simplification entraîne aussi une relaxation du type des opérations (type résultat uniquement).

7.3.3 Structuration des classes

Dans cette phase, l'aspect de chaque classe terminale est calculé, i.e. une contrainte et une structure abstraite. La contrainte du TAG, est commune à toutes les classes de la hiérarchie; elle sera notée `INVT` pour simplifier. Cette contrainte pourra être affinée par le concepteur dans chaque classe formelle. Toutes les valeurs du sous-type associé à l'état sont obtenues à partir des générateurs qui arrivent dans cet état (schéma de création). Le générateur de la classe est donc une combinaison de ces géérateurs. Ce problème s'inscrit dans le problème général de la détermination d'une classe mono-générateur à partir d'une classe multi-générateur[1]. Il faut définir une application des paramètres des générateurs dans les sélecteurs de champ pour affirmer que les deux modélisations représentent bien les mêmes données. L'objectif de la structuration est de dégager une structure abstraite minimale qui permet d'exprimer chaque opération invocable dans l'état correspondant de l'automate.

La stratégie choisie est de calculer un ensemble de structures abstraites, chacune correspondant à un générateur du type de la classe. Cette structure abstraite minimale sera raffinée successivement jusqu'à obtenir une unique structure abstraite dans l'étape de restructuration de la section 7.4.1. Commençons par définir une notion de sous-ensemble de l'aspect lié à un générateur particulier.

Définition 7.3.1 (sous-aspect)
Un sous-aspect est un sous-ensemble d'un aspect conditionné par un sélecteur d'état.

[1]Attention : une classe multi-générateur n'est pas une classe multi-aspect, car il n'y a pas équivalence entre les générateurs.

Les différentes alternatives de la phase de structuration sont :

1. description des opérations génératrices :

   (a) un sous-aspect par transition : c'est le profil le plus spécifique de chaque générateur de la classe.

   (b) un sous-aspect par opération : c'est le profil d'aspect calculé dans la phase précédente pour chaque générateur de la classe (e.g. `arrived : MLift` $\longrightarrow$ `MLLift`).

2. valuation des gardes :

   (a) non : la représentation des gardes n'est pas indispensable, mais elle enrichit la représentation.

   (b) oui : si l'alternative 1.(a) est choisie alors la contrainte du sous-aspect est la garde de la transition, sinon il faut trouver un prédicat vrai pour toutes les transitions du générateur considéré.

3. prise en compte de l'automate interne

   (a) une sous-classe par état : la structure de chaque sous-classe est calculée indépendamment; elles sont ensuite regroupés dans une classe unique par restructuration.

   (b) un sous-aspect par état : cette solution pose le problème du conflit de représentation des sous-aspects des générateurs et de ceux des sous-états.

   (c) une représentation interne de l'automate : chaque générateur interne est considéré comme une opération récursive et donc une extension. La structure prend en compte une représentation concrète de l'automate. Plusieurs représentations sont possibles :

      i. matrice de transition définie par un état courant prenant sa valeur dans un intervalle énuméré (ici {`ML`,`MO`}) et modifié par `new` ou `copy`.

      ii. fonction de transitions `transit` définie sur les générateurs de la classe et invoquée à chaque appel de constructeur. Le protocole d'envoi de message doit prendre en compte cette fonction de transition.

      iii. observateurs booléens définis sur les générateurs (comme pour la description algébrique du TAG),

      iv. sélecteurs d'état (proche du sous-aspect).

   En pratique, la dernière représentation sera utilisée parce qu'elle est simple, facile à mettre en oeuvre et qu'elle est compatible avec la notion de sous-aspect.

La composition des alternatives 1.(a), 2.(b) et 3.(b) est une solution générale et suffisante, mais elle implique par contre plus de travail de restructuration. La technique de structuration est décrite comme suit. Chaque transition entrante, supportée par un générateur, constitue un sous-aspect de la classe. La structure initiale est constituée de l'union des paramètres des générateurs, munie d'un renommage adéquat. Les sélecteurs de champ de cet aspect sont les paramètres du générateur et la contrainte la garde associée à la transition. Une contrainte (globale et implicite) est "un et un seul sous-aspect est vrai à un instant donné".

Application de la structuration à l'exemple de l'ascenseur

Nous illustrerons cette phase par l'étude de la classe terminale `MLLift` et des profils calculés dans la section précédente. La figure 63 présente les générateurs associés à cette classe. La structure doit prendre en compte l'automate interne, s'il existe.

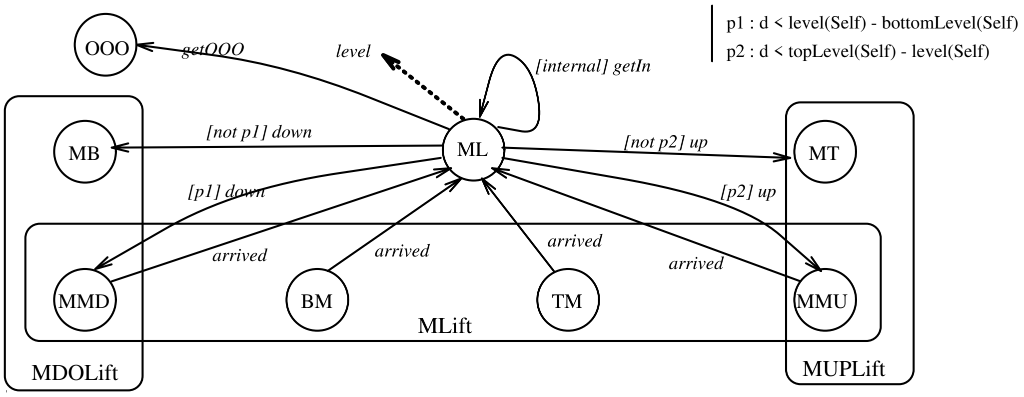


Figure 63 : Projection du comportement dynamique sur la classe MLLift

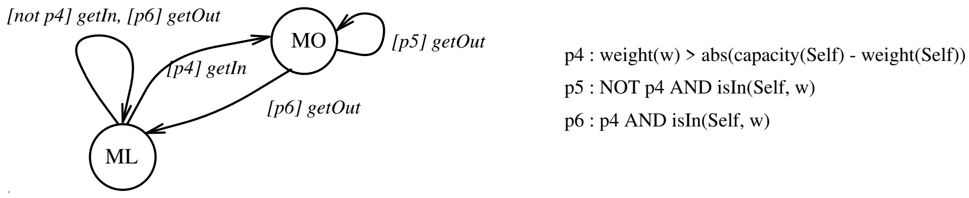


Figure 64 : Automate interne de la classe MLLift

Un sous-aspect est donné par transition. La contrainte associée est la garde. L'automate interne, projeté sur le générateur unique `getIn`, est représenté par les sous-aspect `MLgetIn` et `MO-MLgetIn`.

MLLift_i	
aspect : ML lift - initial structure	
field selectors	**constraint**
subaspect : MLgetIn	
aSelf1 : MLLift_i ⟶ MLLift_i	NOT p4(aSelf1(Self), aXw(Self))
aXw : MLLift_i ⟶ Weightable	
subaspect : MMDarrived	
aSelf2 : MLLift_i ⟶ MMDLift	
subaspect : MMUarrived	
aSelf3 : MLLift_i ⟶ MMULift	
subaspect : BMarrived	
aSelf4 : MLLift_i ⟶ BMLift	
subaspect : TMarrived	
aSelf5 : MLLift_i ⟶ TMLift	
subaspect : MO-MLgetInt	
aSelf6 : MLLift_i ⟶ MLLift_i	p4(aSelf6(Self), aXw1(Self))
aXw1 : MLLift_i ⟶ Weightable	

avec `p4(Self, Xw) = (weight(Xw) > capacity(Self) - weight(Self))`

En supposant une implantation des sous-aspects telle que le nom du sous-aspect représente un sélecteur d'état, conditionnant les sélecteurs de champ du sous-aspect, la contrainte d'unicité du sous-aspect s'exprimerait par
`isMLgetIn(Self)` $\oplus$ `isMMDarrived(Self)` $\oplus$ `isMMUarrived(Self)` $\oplus$ `isBMarrived(Self)` $\oplus$ `isTMarrived(Self)` $\oplus$ `isMO-MLgetInt(Self)`, où $\oplus$ désigne la disjonction exclusive.

7.4 Etape 2 : Restructuration

La restructuration d'une hiérarchie de classes déborde du présent cadre de travail. Voir une introduction dans [DDHL94]. Nous donnons ici quelques éléments de réflexion visant à compacter le sous-graphe d'héritage du type d'intérêt. Ces éléments de réflexion sont applicables à d'autres cas de restructuration. Nous distinguons la restructuration d'une classe qui est un changement de représentation et la restructuration d'un ensemble de classes qui est une recomposition du graphe d'héritage.

La restructuration a une influence sur l'écriture des méthodes et vice-versa. Si les extensions sont écrites avant la restructuration alors une comparaison en termes de sémantique voisine est envisageable pour regrouper les méthodes communes dans la super-classe. Dans ce cas, les sous-classes sans extensions sont supprimées après un certain nombre de contrôles (relaxation de type, cohérence de définitions chez les clients de la classe supprimée). Au contraire, si l'écriture a lieu après restructuration alors d'une part il y aura moins de méthodes à écrire, d'autre part fusionner est plus difficile qu'écrire directement et surtout il est plus difficile de démontrer le raffinement sur l'aspect seul que sur l'aspect et les les extensions. Notons cependant que certaines comparaisons sont difficiles (par exemple celles où un sélecteur de champ d'une classe est une extension dans une autre classe).

La section suivante concerne la restructuration individuelle d'une classe et la suivante celle d'un ensemble de classes.

7.4.1 Restructuration d'une classe

Dans cette section, nous étudions la transformation d'une structure abstraite en une structure abstraite plus adaptée (plus simple, plus réutilisable, plus réutilisatrice, plus concrète). L'idée est de donner une nouvelle structure à la classe puis de montrer la correspondance entre les deux structures. Notre notion de représentation est similaire à la réification en VDM [Jon93].

Définition 7.4.1 (représentation)
Une classe R est une représentation d'une classe A si et seulement si les conditions suivantes sont respectées.

1. Il existe une fonction d'abstraction des instances de R dans les instances de A
   $\exists abs_{R\rightarrow A} : R \rightarrow A \bullet \forall r \in R, \exists a \in A \bullet abs_{R\rightarrow A}(r) = a.$
2. Cette fonction d'abstraction est surjective
   $\forall a \in A, \exists r \in R \bullet abs_{R\rightarrow A}(r) = a.$
3. L'invariant est respecté par la fonction d'abstraction
   $\forall r \in R \bullet inv^A(abs_{R\rightarrow A}(r)) \Rightarrow inv^R(r)$
4. Les méthode de A sont applicables aux instances de R et produisent le même résultat.
   $\forall m^A \in K(A),\ \exists m^R \in K(R) \ \bullet\ \forall r \in R,\ prec^A_m(abs_{R\rightarrow A}(r)) \Rightarrow prec^R_m(r) \wedge$
   - $codom(m^R) = R \Rightarrow abs_{R\rightarrow A}(m^R(r)) == m^A(abs_{R\rightarrow A}(r))$ $(constructeurs)$
   - $codom(m^R) \neq R \Rightarrow m^R(r) == m^A(abs_{R\rightarrow A}(r))$ $(observateurs)$

$$\begin{tikzcd} a \arrow[r, "m^A"] & v_i^A \\ r \arrow[u, "abs_{R\rightarrow A}"] \arrow[r, "m^R"] & v_i^R \arrow[u, "abs_{R\rightarrow A}"'] \end{tikzcd} \qquad \begin{tikzcd} a \arrow[r, "m^A"] & v \\ r \arrow[u, "abs_{R\rightarrow A}"] \arrow[ur, "m^R"'] & \end{tikzcd}$$

Comme les classes sont mono-génératrices, les instances de A (resp. de B) sont de la forme `newA(...)` sous la contrainte `cont_A` (resp. `newB(...)` sous la contrainte `cont_B`). La démonstration se fait en quatre temps : donner une fonction d'abstraction, montrer qu'elle est surjective, vérifier que les sélecteurs de champ de l'ancien aspect sont des extensions du nouvel aspect, vérifier la contrainte.

Application à la classe `MLLift`

Nous proposons une restructuration en trois raffinements. Le premier raffinement groupe les différents sous-aspects issus d'une même opération en relaxant le type correspondant. Par exemple, les sous-aspects `MMDarrived`, `MMUarrived`, `BMarrived`, `TMarrived` sont tels qu'il existe une super-classe `INTERMEDIATELift` commune aux classes `BMLift`, `TMLift`, `MMULift`, `MMDLift`. Comme la super-classe possède d'autres sous-classes, il faut contraindre le type du sélecteur de champ `oldState` : $\texttt{cont}_{oldState}$ `= classOf(oldState)` $\in$ {`BMLift, TMLift, MMULift, MMDLift`} Une optimisation consiste à supprimer la contrainte $\texttt{cont}_{oldState}$ par relaxation de type. Cette relaxation est possible car elle ne change pas le type dynamique. En ce qui concerne les sous-aspects de `MLgetIn`, les types des sélecteurs de champ sont identiques et la contrainte est complémentaire.

MLLift_a	
aspect : ML lift - release a	
field selectors	**constraint**
subaspect : getIn	
s : MLLift_a ⟶ MLLift_a	
w : MLLift_a ⟶ Weightable	
subaspect : arrived	
oldState : MLLift_a ⟶ INTERMEDIATELift	

Le second raffinement consiste à joindre les sous-aspects des opérations différentes en les distinguant par des sélecteurs d'état.

MLLift_b	
aspect : ML lift - release b	
field selectors	**constraint**
isGetIn : MLLift_b ⟶ Boolean	
aSelf1 : MLLift_b ⟶ MLLift_b **requires**: isGetIn(Self) == true	
aXw : MLLift_b ⟶ Weightable **requires**: isGetIn(Self) == true	
oldState : MLLift_b ⟶ INTERMEDIATELift **requires**: isGetIn(Self) == false	

Enfin le troisième raffinement est un changement de représentation, qui supprime le sélecteur de champ récursif `aSelf1` en réutilisant le type de données `List[Weightable]`. L'invariant `INVT` du TAG est ajouté et pourra être simplifié plus tard.

MLLift_o	
aspect : ML lift - optimized abstract structure	
field selectors	**constraint**
oldState : MLLift_o ⟶ INTERMEDIATELift	
contents : MLLift_o ⟶ List[Weightable]	INVT

7.4.2 Correction de la représentation

Le changement de structure est possible s'il existe une fonction d'abstraction surjective de l'aspect `MLLift_o` dans l'aspect `MLLift_i`. Nous allons donner les fonctions d'abstraction successives puis nous détaillerons la démonstration pour le passage de `MLLift_b` à `MLLift_o` avec l'introduction du type `List`.

Fonctions d'abstraction

Soient les variables suivantes `Xw : Weightable; Xs : MLLift; Xold = INTERMEDIATELift; Y1, Y2 : Boolean`. Les sous-aspects sont représentés par des sélecteurs d'état de nom le nom du sous-aspect.

$\texttt{abs}_{a\rightarrow i}$: `MLLift_a` ⟶ `MLLift_i`

```
abs_{a→i}(newMLLift_a(s = Xs, w = Xw, oldState = Xold, isgetIn = Y1, isArrived = Y2)) ==
    newMLLift_i( aSelf1 = Xs, aXw = Xw, aSelf2 = Xold, aSelf3 = Xold, aSelf4 = Xold,
                 aSelf5 = Xold, aSelf6 = Xs, aXw = Xw,
                 isMLgetIn = Y1 AND NOT p4(Xs, Xw),
                 isMMDarrived = Y2 AND classOf(Xold) = MMDLift
                 isMMUarrived = Y2 AND classOf(Xold) = MMULift,
                 isBMarrived = Y2 AND classOf(Xold) = BMLift,
                 isTMarrived = Y2 AND classOf(Xold) = TMLift,
                 isMO-MLgetIn = Y1 AND p4(Xs, Xw) )

abs_{b→a} : MLLift_b  ⟶ MLLift_a
abs_{b→a}(newMLLift_b(isgetIn = Y1, aSelf1 = Xs, aXw = Xw, oldState = Xold)) ==
    newMLLift_a( s = Xs, w = Xw, oldState = Xold, isgetIn = Y1, isArrived = NOT Y1 )

abs_{o→b} : MLLift_o  ⟶ MLLift_b
abs_{o→b}(newMLLift_o(oldState = Xold, contents = Xcontents)) ==
    newMLLift_b( isGetIn = NOT isEmpty(Xcontents),
                 aSelf1 = abs_{o→b}(newMLLift_o(oldState = Xold, contents = cdr(Xcontents))),
                 aXw = car(Xcontents), oldState = Xold)
```

Preuve du passage de `MLLift_b` à `MLLift_o`

Soient les variables supplémentaires suivantes `Yold : INTERMEDIATELift; Xcontents, Xl : List[Weightable]`. Montrons la surjectivité de la fonction d'abstraction par induction, ou plus formellement que $\forall$ `Xmlb` $\in$ `MLLift_bInstance`, $\exists$ `Xmlo` $\in$ `MLLift_oInstance` • $\text{abs}_{o\to b}$`(Xmlo) = Xmlb` $\wedge$ $\text{abs}_{o\to b}$`(cont(Xmlo))` $\Rightarrow$ `cont(Xmlb)`.
Soit `newMLLift_b(isGetIn = Y1, aSelf1 = Xs, aXw = Xw, oldState = Xold)`.

- Si `Y1 == false` alors `newMLLift_o(oldState = Xold, contents = newEmptyList())` est un $Xmlo$ correct.
- Si `Y1 == true` alors par hypothèse d'induction $\exists$ `newMLLift_o(oldState = Yold, contents = Xl) == Xs` tel que `newMLLift_o(oldState = Yold, contents = cons(Xl, Xw))` est un $Xmlo$ correct.

La contrainte est vérifiée implicitement par la fonction d'abstraction.

La dernière chose à prouver est que les sélecteurs de champ de `MLLift_b` sont des opérations sur `MLLift_o`, c'est-à-dire des extensions sur les sélecteurs de champ de `MLLift_o` modulo la fonction d'abstraction. Soient les extensions suivantes :

```
(e1) isGetIn(Self) == NOT isEmpty(contents(Self))
(e2) aSelf1(Self) == newMLLift_o(oldState = oldState(Self), contents = cdr(contents(Self)))
     requires: isEmpty(contents(Self)) == false
(e3) aXw(Self) == car(contents(Self))
     requires: isEmpty(contents(Self)) == false
(e4) oldState(Self) == oldState(Self)
     requires: isEmpty(contents(Self)) == true.
```

Prenons deux exemples : `isGetIn` et `aSelf1`. Dans le premier cas, il faut montrer que
`isGetIn(newMLLift_o(oldState = Xold, contents = Xl))`
`== isGetIn(abs`$_{o\to b}$`(newMLLift_o(oldState = Xold, contents = Xl)))`
($\Rightarrow$) `isGetIn(newMLLift_o(oldState = Xold, contents = Xl))`
`== NOT isEmpty(contents(newMLLift_o(oldState = Xold, contents = Xl))` par (e1)
`== NOT isEmpty(Xl)` par définition de contents.
($\Leftarrow$) `isGetIn(abs`$_{o\to b}$`(newMLLift_o(oldState = Xold, contents = Xl)))`
`== isGetIn(newMLLift_b(isGetIn = NOT isEmpty(Xl), ...))` par abstraction,
`== NOT isEmpty(Xl) par définition de isGetIn`

`QED.`

Dans le deuxième cas, il faut montrer que
`abs`$_{o\rightarrow b}$`(aSelf1(newMLLift_o(oldState = Yold, contents = Xl)))`
`== aSelf1(abs`$_{o\rightarrow b}$`(newMLLift_o( oldState = Yold, contents = Xl)))`.
Par définition de `aSelf1` sur `MLLift_o`,
`abs`$_{o\rightarrow b}$`(aSelf1(newMLLift_o(oldState = Yold, contents = Xl)))`
`== abs`$_{o\rightarrow b}$`(newMLLift_o(oldState = Yold, contents = cdr(Xl)))`
par la fonction d'abstraction,
`aSelf1(abs`$_{o\rightarrow b}$`(newMLLift_o(oldState = Yold, contents = Xl)))`
`== aSelf1(newMLLift_b(isGetIn = NOT isEmpty(Xl), oldState = Yold,`
`aSelf1 = abs`$_{o\rightarrow b}$`(newMLLift_o( oldState = Xold,`
`contents = cdr(Xl))), aXw = car(Xl)))`
par la définition de `aSelf1` sur `MLLift`, l'égalité est vérifiée.
Examinons la précondition sur `aSelf1`: sur `MLLift_o`, `isEmpty(contents(Self)) == false` et sur `MLLift_b`, `isGetIn(Self) == true`.
Ces préconditions sont égales par le résultat précédent sur `isGetIn`.
`QED`.

La méthode est rigoureuse mais un peu lourde à manipuler. Plusieurs alternatives sont possibles dans chaque phase. Il faut envisager des techniques plus fines pour obtenir des aspects plus simples et plus réutilisables, comme l'introduction de nouvelles classes serveur (exemple : classe `List`). Une études des outils utiles à la manipulation de structures est envisagée et constitue un travail à part entière.

7.4.3 Restructuration du schéma d'héritage

Dans la section précédente, nous avons vu comment restructurer une classe (terminale) en changeant sa représentation. Dans cette section, nous voulons réduire le graphe d'héritage en regroupant des classes proches sémantiquement.

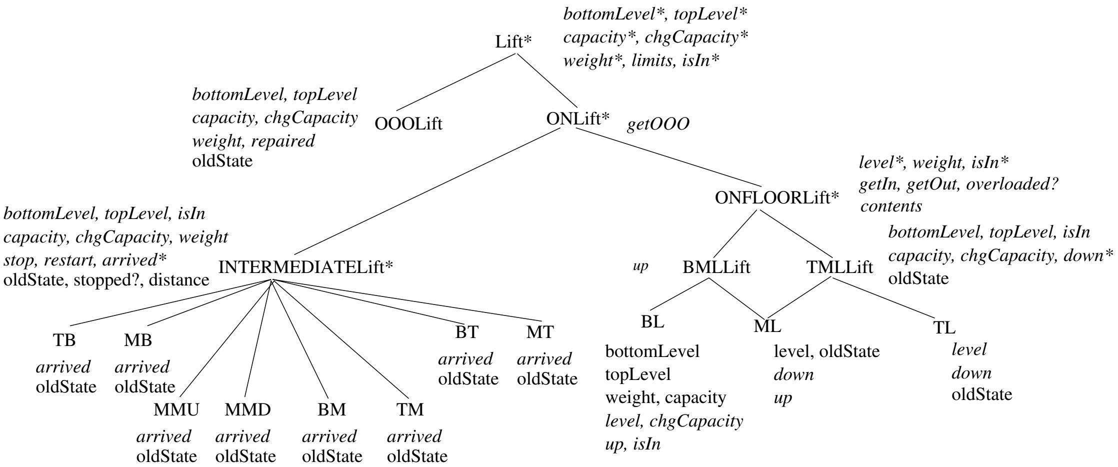


Figure 65 : Schéma d'utilisation structurée du type `Lift`

Proximité entre classes

La proximité sémantique des classes est difficile à mesurer. Nous définissons un critère statique simple pour mesurer la proximité deux classes C_i et C_j :

$$(eq7.4.3) \qquad 1 - \frac{\mid K(C_i) \cap K(C_j) \mid}{\mid K(C_i) \cup K(C_j) \mid}$$

où $K(C)$ représente le comportement complet, en termes d'opérations du TAG de la classe C (i.e. sans les méthodes primitives autres que les sélecteurs de champ).

Le comportement complet se calcule comme la construction des profils par l'algorithme de la figure 60 à ceci près qu'on ne rajoute pas de nouveaux noeuds dans la hiérarchie. Se pose alors le problème du non respect de la condition d'unicité de la rétraction pour les opérations secondaires. Par exemple, comparons les classes `MMDLift`, `MBLift`, `BLLift`, `TLLift` et `OOOLift` de la figure 65.

	MMDLift	MBLift	BLLift	TLLift	OOOLift
MMDLift	-	0	13/21	13/21	7/14
MBLift	0	-	13/21	13/21	7/14
BLLift	13/21	13/21	-	2/16	10/17
TLLift	13/21	13/21	2/16	-	10/17
OOOLift	7/14	7/14	10/17	10/17	-

Plus la valeur est proche de zéro, plus le regroupement est intéressant (e.g. `MMDLift-MBLift (0)` et `BLLift-TLLift (0.125)`).

Le critère est basé sur les noms des méthodes. Ce qui est significatif de notre point de vue, car les méthodes des différentes classes formelles sont issues des opérations d'un unique TAG. Nous avons envisagé une comparaison sur les profils des méthodes pour prendre en compte un éventuel renommage, mais le résultat ne s'est pas avéré payant (bon nombre de méthodes différentes ont le même profil, deux méthodes représentatant la même opération ont en général un profil (spécifique) différent., etc.).

Restructuration des classes proches

Si les sous-classes d'une classe sont proches alors elles sont regroupées puis restructurées. Lorsqu'il s'agit d'héritage simple, grouper les classes terminales dans leur super-classe commune est suffisant. En héritage multiple, il faut chercher la plus petite super-classe commune à toutes les classes terminales concernées.

La technique de restructuration est similaire à celle de la structuration des classes formelles. Une structure triviale est exhibée, contenant un sous-aspect par classe jointe. Puis les sous-aspects équivalents sont représentés exclusivement par des sous-aspects disjoints. Enfin, le concepteur réalise plusieurs changements de représentation jusqu'à aboutir à la structure la plus simple possible, avec preuve des représentations. La détermination d'une structure commune aux sous-aspects comprend des équivalences ou inclusions de structures, des homonymies, des valeurs par défaut (0, nil)), une inclusion de contrainte. Le bon sens du concepteur guide bien évidemment la restructuration.

Par exemple, examinons les sous-classes de `INTERMEDIATELift` et de `ONFLOORLift`. Les sélecteurs de champ des sous-classes de `INTERMEDIATELift` sont équivalents mais la contrainte varie. Les cas sont différenciés par le sélecteur de champ `isUp`. Un premier regroupement des sous-aspects {`BMLift, BTLift`}, {`MBLift, MMDLift`}, {`MTLift, MMULift`}, {`TMLift, TBLift`} est possible car les structures sont équivalentes et les contraintes complémentaires. Ensuite, nous changeons la structure par relâchement du type de `oldState`. La séparation des cas se fait en utilisant le sélecteur d'état `isUp`. La contrainte est réduite car la méthode `level` n'est pas définie ici.

INTERMEDIATELift	
inherits from ONLift	
aspect : lift	
field selectors	**constraint**
oldState : INTERMEDIATELift ⟶ ONLiftFLOOR	bottomLevel(Self) < topLevel(Self) AND 0 <= weight(Self)
offset : INTERMEDIATELift ⟶ IntegerGt1	
isUp : INTERMEDIATELift ⟶ Boolean	
isStopped : INTERMEDIATELift ⟶ Boolean	

Le cas des sous-classes de `ONFLOORLift` est un peu plus complexe.

`ONFLOORLift`	
inherits from `ONLift`	
aspect : `lift`	
field selectors	**constraint**

`isInstalled : ONFLOORLift ⟶ Boolean`	`bottomLevel(Self) < topLevel(Self)`
	`AND 0 <= weight(Self)`
	`AND bottomLevel(Self) <= level(Self)`
	`AND level(Self) <= topLevel(Self)`

`bottomLevel : ONFLOORLift ⟶ Integer` **requires:** `isInstalled(Self) == true`	
`topLevel : ONFLOORLift ⟶ Integer` **requires:** `isInstalled(Self) == true`	
`capacity : ONFLOORLift ⟶ RealGt1` **requires:** `isInstalled(Self) == true`	
`contents : ONFLOORLift ⟶ List[Weightable]`	
`oldState : ONFLOORLift ⟶ INTERMEDIATELift` **requires:** `isInstalled(Self) == false`	

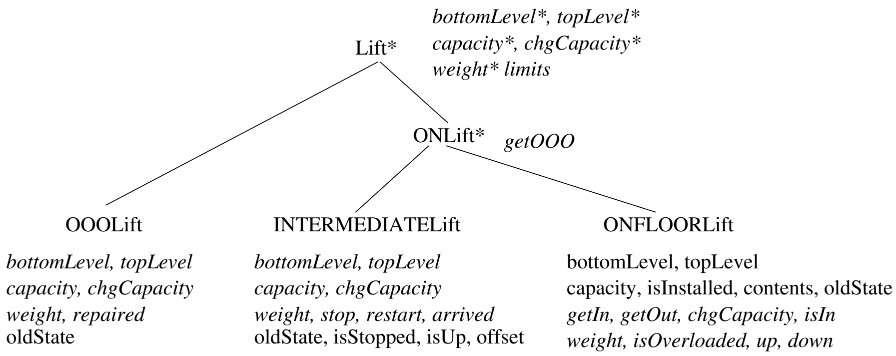


Figure 66 : Schéma d'utilisation restructuré du type `Lift`

7.5 Etape 3 : Ecriture des extensions

Une fois la hiérarchie achevée, le concepteur écrit les extensions selon la méthode de la section 6.2.6. Il peut s'inspirer de la spécification algébrique, si elle a été construite.

A partir de chaque opération du TAG, le concepteur écrit les axiomes dans les classes où une extensions correspondant à cette opération est définie. Deux alternatives existent : soit le concepteur écrit d'abord les extensions des classes terminales puis par comparaison de méthodes il généralise les comportement communs, soit il commence par les extensions des classes abstraites et affine si nécessaire les descriptions dans les sous-classes. La comparaison de méthodes est étudiée dans la section 7.5.1.

Nous préférons la seconde option car l'écriture des axiomes d'une méthode (sélecteur de champ ou extension) correspondant à un constructeur du TAG est souvent remise en cause par la restructuration des classes. De plus la comparaison et l'union de méthodes n'étant pas triviales, il vaut mieux les éviter. Par contre la conformité des classes avec le TAG est plus difficile à montrer lorsque sa spécification algébrique existe.

générateurs de base

Si le modèle des classes formelles contient des métaclasses, les générateurs de base deviennent des méthodes de classe des classes correspondant aux états initiaux, sinon ce sont des méthodes générales, uniques dans le système.

```
ONFLOORLift
class methods
;; install : build an initial lift
   install : Integer Integer RealGt1  ⟶  ONFLOORLift
var: Xbot : Integer,Xtop : Integer,Xcap : RealGt1
install(Xbot, Xtop, Xcap) == newONFLOORLift(bottomLevel = Xbot, topLevel = Xtop,
         capacity = Xcap, contents = newEmptyList(), isInstalled = false)
```

générateurs

Les générateurs non récursifs définis dans une classe deviennent des extensions. Tandis que les générateurs récursifs sont exprimés à partir de la méthode `copy`.

```
ONFLOORLift
extensions
;; down : gets down Xd levels
   down : ONFLOORLift IntegerGt1  ⟶  INTERMEDIATELift
var: Xd : IntegerGt1
requires: Xd <= level(Self) - bottomLevel(Self) == true & isOverloaded(Self) == false
down(Self, Xd) == newINTERMEDIATELift, oldState = Self, offset = Xd,
                          isStopped == false, isUp == false)
```

constructeurs secondaires

Les constructeurs secondaires sont des extensions exprimées si possible en fonction des extensions issues des générateurs plutôt que des méthodes primitives.

```
ONFLOORLift
extensions
;; getOut : a weightable gets outside the cage
   getOut : ONFLOORLift Weightable  ⟶  ONFLOORLift
var: Xw : Weightable
requires: isIn(Self, Xw) == true
getOut(Self, Xw) == copy(Self, contents = remove(contents(Self),Xw))
```

Remarque : Le lecteur attentif aura remarqué que les axiomes de la méthode `getOut` sont différents de ceux associés à l'opération `getOut` de l'axiomatique algébrique. La question à se poser est "peut-on spécifier de façon plus abstraite ?". Oui si `contents` est défini en fonction de `oldState` et `getIn`. Mais, puisque nous avons choisi une option par mono-générateur, toutes les opérations définies pour la classe sont secondaires, elles se décrivent à partir de la structure.

observateurs

Les observateurs sont définis à partir de la structure abstraite. Par exemple,

```
ONFLOORLift
extensions
;; bottomLevel : bottom level of the lift
   bottomLevel : ONFLOORLift    ⟶  Integer
isInstalled(Self) == true ==> bottomLevel(Self) == bottom(Self)
isInstalled(Self) == false ==> bottomLevel(Self) == bottomLevel(oldState(Self))
```

```
INTERMEDIATELift
extensions
;; bottomLevel : bottom level of the lift (bottomLevel3:, bottomLevel5:)
   bottomLevel : INTERMEDIATELift    ⟶  Integer
bottomLevel(Self) == bottomLevel(oldState(Self))
```

Les méthodes privées comme `isIn` ne se trouvent pas dans la partie interface (`features`) de la classe racine, ni des sous-classes. La description complète des classes se trouvent en annexe C.6.

7.5.1 Comparaison et généralisation de méthodes

Si la comparaison de méthodes est indécidable, une définition axiomatique est plus facilement comparable. Six critères permettent de comparer les méthodes : nom, nombre de paramètres, noms et types des paramètres type résultat, catégorie (observateur, constructeur, primitif, secondaire), axiomes. Le niveau de compatibilité est fonction des variations de profils et d'axiomes :

- La compatibilité est sémantique lorsque seul le receveur varie. Si les axiomes peuvent être unifiés, la méthode est mise dans la super-classe, (exemple : $\mathtt{topLevel}_{STOPPEDLift}$, $\mathtt{limits}_{Lift}$). Dans le cas contraire, la méthode reste abstraite mais une optimisation sera de dégager une sous-partie commune. Exemple : $level_{ONFLOORLift}$ est une extension dans `TLLift` et un sélecteur de champ dans `MLLift`.

- La compatibilité est de résultat si le receveur et le résultat varient. La méthode est définie avec le plus petit profil commun. Exemple : $restart_{STOPPEDLift}$, $arrived_{INTERMEDIATELift}$). Il y a union des axiomes.

- La compatibilité est de paramètre lorsque le receveur et soit le type soit le nombre de paramètres varie. La méthode commune doit suivre la règle de sous-typage. La méthode est définie avec le plus petit profil commun. Il y a union des axiomes.

- Il y a surcharge lorsque seuls les noms de méthode sont identiques. Le plus grand profil est exhibé pour la méthode principale à partir duquel les autres méthodes sont exprimées. C'est par exemple le cas pour les sélecteurs de champ conditionnels qu'on ne veut pas mettre dans une méthode de création.

- La compatibilité de renommage est explicite. Il semble laborieux de comparer toutes les méthodes, même si le profil est identique.

Dans notre cas, la comparaison porte uniquement sur les méthodes de même nom, ce qui simplifie la généralisation du comportement. De plus le cas de surcharge n'est pas possible puisqu'il s'agit des méthodes d'un même sous-type. Détaillons les cas où $profil(op_{subclass}) < profil(op_{superclass})$. Comme pour la restructuration des classes, une définition par défaut est donnée par union disjointe des axiomes et le concepteur doit exhiber une représentation plus adéquate. Les conditions communes aux axiomes deviennent des préconditions de la méthode. Les préconditions communes à toutes les méthodes sont ajoutées à la contrainte.

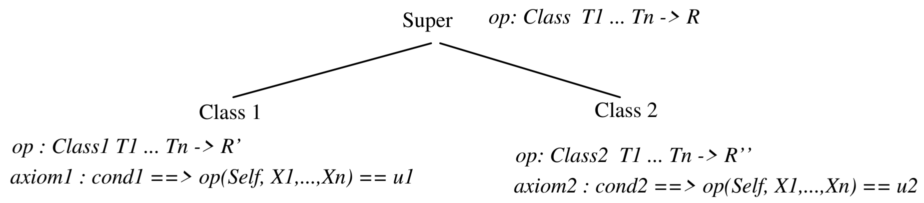


Figure 67 : Abstraction de méthodes

7.6 Validation du raffinement

Le fait de pouvoir définir les méthodes secondaires est en soi une première étape de validation. La validation du raffinement regroupe un ensemble de mesure qui permettent d'évaluer l'intérêt de cette méthode de raffinement selon la structure des types de données décrits, de mesurer la qualité des hiérarchies de classe obtenues et les optimisations envisageables. Plus qu'une métrication rigoureuse, nous donnons quelques indices et critères qui nous semblent important. Enfin la validation concerne la conformité entre le TAG et les classes formelles qui en découle.

7.6.1 Domaine d'application de la méthode

Selon la forme du TAG, la méthode rigoureuse ou l'approche intuitive sera plus adaptée. Nous donnons ici quatre mesures statiques pour évaluer si l'approche constructive est la plus adaptée. $\mid s \mid$ désigne la cardinalité de l'ensemble s.

1. Le nombre d'états est important.
2. Il existe un taux minimal R_∂ d'opérations partielles.
$$\frac{\mid \partial \mid}{\mid F \mid} > R_\partial$$
dans la pratique, $R_\partial = 0.4$ semble suffisant.
3. Il existe un taux minimal de redéfinition des méthodes sur plusieurs états R_{K*}, $(dom(op) = \{k \in K, \exists k' \in K, g \in \Delta \bullet \delta(k, op, g) = k'\}$ c'est-à-dire que le schéma d'utilisation comporte plusieurs niveaux intermédiaires .
$$\frac{\sum_{op \in \partial} \mid dom(op) \mid}{\mid \partial \mid} > R_{K^*}$$
dans la pratique, $R_{K^*} = 1.5$ semble suffisant.
4. Le schéma d'utilisation est équilibré (nombre de niveaux intermédiaires vis-à-vis du nombre de classes terminales $R_{bal} = 1/3$).
$$\frac{\overline{C}}{K} > R_{bal}$$
où $\overline{C}$ est le nombre de classes d'équivalences définies par la relation d'équivalence $op_i \sim op_j \Leftrightarrow dom(op_i) = dom(op_j)$.

Dans les expérimentations réalisées, ces critères sont vérifiés. Dans l'exemple de l'ascenseur, les valeurs obtenues sont (1) = 23, (2) $12/18 = 2/3 > 0.4$, (3) $70/23 > 12/6$ et (4) $5/11 > 1/3$.

7.6.2 Evaluation du résultat

Evaluation statique

Des critères permettent une évaluation de la qualité des classes obtenues : le nombre de classes terminales ne dépasse pas 1/3 du nombre d'états, moins de 10 sélecteurs de champ par classe, moins de 10 axiomes par extension, le nombre d'états est environ le quadruple de celui de classes, moins de 20 opérations définies par classe, moins de trois équations par condition d'axiome, etc. Une étude statistique de plus grande envergure est nécessaire.

Evaluation de la conformité TAG/CF

Le TAG et la classe formelle sont des représentation différentes du même type abstrait de donnée. Le TAG définit des sous-types par des états abstraits. Les classes formelles raffinent ces états en leur donnant une structure concrète et arrangent l'organisation des états de façon à mettre en commun un maximum de propriétés pour les rendre plus modulaires et donc plus réutilisables. Deux moyens existent pour prouver la conformité : la validation du raffinement et les tests. Etant donnés la différence de structure et la participation du concepteur dans le processus de raffinement, la mise au point de méthodes de vérification de conformité entre le TAG et les classes formelles nécessite une étude à part entière. En effet, il est laborieux de montrer qu'à chaque étape la même sémantique est conservée, bien que ce soient toujours les mêmes opérations qui soient manipulées. Une meilleure alternative semble donc de passer par des tests fonctionnels. Les termes engendrés par l'automate doivent être équivalents à ceux issus des classes formelles. Cette vérification peut être effectuée par l'évaluateur des classes formelles.

7.7 Récapitulatif

Nous avons distingué quatre traitements majeurs pour construire des classes formelles à partir des TAG. Ces traitements sont :

1. construction d'une hiérarchie d'héritage entre classes formelles,
2. structuration d'une classe formelle,
3. restructuration des classes et de la hiérarchie, et
4. écriture des extensions pour chaque classe.

Nous avons aussi donné quelques heuristiques pour limiter la complexité de ces traitements

– (a) regroupement d'états, automate réduit,

– (b) restriction à l'ensemble des opérations primitives,

– (c) sélection des profils généraux pour les méthodes,

– (d) écriture des méthodes en commençant par les classes abstraites.

La complexité des traitements varie selon les paramètres choisis :

option	complexité ou nombre							
opérations	toutes				primitives			
graphe	initial		réduit (a)		initial		réduit (a)	
profils	tous	généraux (c)	tous	généraux	tous	généraux	tous	généraux
phase (i)	-	-	****	****	-	-	****	****
phase (ii)	*****	****	***	**	***	**	**	*
nbcl†	*****	****	***	**	****	***	**	*
phase (iii)	****	*****	****	*****	***	****	**	****
nbch‡	****	****	***	***	**	***	**	***

† : nombre de classes abstraites
‡ : nombre de sélecteurs de champ par classe abstraite

En résumé, si le concepteur prend en compte toutes les opérations et/ou tous les états, les algorithmes sont coûteux et les résultats sont médiocres. S'il choisit de diminuer le nombre de classes intermédiaires, la structure les classes terminales est plus difficile à calculer car elle contient des redondances de typage, par contre la restructuration est plus facile car il y a moins d'héritage multiple et de classes à comparer.

La complexité des traitements varie aussi selon l'ordre des traitements :

ordre	traitements							
0	a-b				-			
1	1				1			
2	2				2			
3	3		4		3		4	
4	4		3		4		3	
5	d	-	d	-	d	-	d	-
commentaires	i	ii	iii	iv	v	vi	vii	viii

– (i) (ii) c'est la méthode préconisée : elle réduit les tâches à réaliser, mais les heuristiques nous éloignent du TAG de départ et la vérification de cohérence entre les deux est plus difficile. La restructuration est moins lourde mais aussi moins automatique. Pour (i), l'écriture est plus rapide mais moins rigoureuse. Pour (ii), les méthodes communes sont généralisées par comparaison et fusion, ce qui est rigoureux mais lourd.

- (iii) (iv) le fait de calculer les extensions avant de restructurer facilite l'écriture des méthodes et la comparaison des classes (plus petites et plus simple). La restructuration se fait par remontée des méthodes dans les super-classes.
- (v) (vi) l'accent est mis sur la restructuration verticale (cohérence modulaire). C'est la moins intéressante des solutions, car la démarche reste lourde mais pas toujours automatisable.
- (vii) (viii) le résultat avant restructuration est calqué sur la spécification algébrique du TAG. La vérification est donc plus facile et l'écriture semi-automatique. La comparaison des méthodes est plus facile. La restructuration se fait par comparaison sur toutes les méthodes et remontée de comportement commun.

7.8 Implantation des classes formelles

Les phases précédentes nous fournissent une hiérarchie des classes et des opérations, ainsi qu'une description des classes formelles (aspect et méthodes secondaires). Nous avons tout ce qu'il nous faut pour réaliser une mise en œuvre du système voulu. Le codage peut se faire facilement dans un langage à classes du genre Eiffel, C++, CLOS, Smalltalk.

La traduction d'une CF en classe concrète est assez directe ceci permet d'obtenir un prototype rapidement du système. La traduction n'est en général pas complètement automatisable, cela dépend de la forme des axiomes. Il est facile de produire la déclaration de la structure de la classe et de l'interface des méthodes. La production automatique du code des méthodes est possible si les axiomes sont orientables en règles de réécriture.

Dans les sections suivantes, nous abordons les principes généraux de la traduction et de son optimisation, puis nous développons cette traduction dans le cas d'Eiffel et de Smalltalk. La traduction est guidée par une grammaire des classes formelles et des actions sémantiques dans cette grammaire. Nous verrons l'implantation des traducteurs dans le chapitre B.

7.8.1 Optimisation du prototype

Bien qu'à ce stade du développement, le prototype ne soit pas astreint à une contrainte d'efficacité, le critère d'efficacité n'est pas négligeable pour les raisons suivantes :

- le prototype doit pouvoir fonctionner sous des configurations matérielles réduites,
- un temps d'exécution trop long est rédibitoire pour la validation du logiciel par le commanditaire,
- la définition fonctionnelle sur une architecture à objets entraîne des performances désastreuses :
  - conservation de tous les objets du calcul d'une fonction tant que celle-ci n'est pas évaluée,
  - création d'un objet à chaque appel fonctionnel et lien avec les objets précédents,
  - surcharge du système pour récupérer la place inutile (ramasse-miette),
  - multiplication des envois de message et donc des recherches de méthodes.

Un minimum d'efforts peut être fait pour améliorer les performances de la maquette. Un certain nombre d'optimisations rendent la description plus efficace. Nombre d'entre elles sont relatives soit à un changement de structure des classes, soit à une version impérative des classes formelles (voir section 6.6.3); d'autres sont liées à la sémantique même des classes. En voici un catalogue non exhaustif :

- Unifier deux représentations d'une même méthode. Par exemple, l'observateur `bottomLevel` défini dans la classe `ONFLOORLift` à la page 177 possède deux représentations. Elles sont unifiées par un sélecteur de champ non conditionnel (idem pour `topLevel` et `capacity`).

- Remplacer un sélecteur de champ par une extension. C'est le cas si le sélecteur est redondant, e.g. `isOverload` dans la classe `ONFLOORLift`). Coder une constante par une méthode d'instance ou de classe évite de surcharger la structure. Par contre, se pose le problème de la suffisance de l'aspect.
- Transformer une extension en sélecteur de champ. Par exemple, la récursivité engendrée par `oldState` est supprimée en recopiant les attributs qui manquent. Ceci diminuer le nombre d'objets d'une évaluation.
- Remonter des méthodes semblables ou grouper des parties de méthodes communes. Par exemple, `up` et `down` sont définies en fonction d'une méthode `move` ou inversement.
- Eliminer les préconditions triviales, inutiles ou redondantes. Par exemple, les trois sélecteurs de champ `top, bottom et cap` deviennent totaux, i.e. ils sont copiés à chaque fois (élimination de récursivité).
- Utiliser des effets de bord. Les optimisations concernent l'économie de création des objets : il faut transformer le maximum de copies en modifications sans perturber la sémantique initiale (attention aux partages de structure).
- Éliminer la recursivité des fonctions : des techniques de variables tampons ou de passage de continuations peuvent être adaptées ici.
- Changer la représentation par ajout de sélecteurs de champ privés pour optimiser l'appel d'une méthode utilisée fréquemment.
- Faire apparaître des fonctions auxilliaires privées qui permettent de partager un calcul coûteux entre plusieurs méthodes.
- Des optimisations sont possibles pour l'héritage : réduction de la hiérarchie, optimisation de la recherche, ... Le problème de l'efficacité dans la recherche des méthodes est supposé bien résolu au niveau du langage de mise en œuvre [AR92b].
- D'autres optimisations dépendent des langages d'implantation : réutilisation de classes de l'environnement, concepts propres, plus adaptés tels que les métaclasses, les démons, les variables de pool, les assertions, etc.

7.8.2 Traduction directe d'une classe formelle

Le principe de la traduction directe est de produire une classe concrète dans un langage donné qui soit une représentation opérationnelle de celle-ci. Si le langage cible est typé dynamiquement il suffit de ne pas tenir compte des informations de typage ou de les mettre en commentaire. Ce type de traduction est directe dans le sens où il y a un isomorphisme entre les structures abstraites et les structures concrètes. La démarche de traduction que nous proposons est la suivante :

1. Pour chaque classe formelle définir une classe concrète.
2. La structure de la classe concrète est un ensemble de champs variables d'instance, slots ou attributs (suivant le langage) privés correspondant aux sélecteurs de champ.
3. Définir les méthodes primitives non prévues par le langage concret, ceci est facile à partir de la structure et concerne les méthodes : `new, copy, =, describe`.
4. Pour chaque sélecteur une méthode de lecture (publique) est définie avec une éventuelle pré-condition.
5. Pour chaque méthode secondaire (extension) implanter une méthode concrète. Dans nos exemples nous avons une spécification quasi-opérationnelle ce qui rend la traduction immédiate. Ce n'est pas toujours le cas.

6. Si il existe une contrainte elle intervient comme un prédicat de définition du type. Il suffit d'en faire une préconditions du générateur `new`.

7. Suivant le langage des déclarations d'attributs exportables (privé, public, ...) sont à faire. Dans une classe formelle les méthodes primitives et secondaires sont exportables (exception faite de certaines méthodes auxilliaires).

8. L'accès à l'objet receveur et plus généralement le mode de sélection sont importants. Dans le modèle à classes formelles nous avons implicitement une sélection simple donc une traduction en CLOS doit être faite avec attention si il y a plusieurs arguments qui sont des objets. Dans les autres cas l'accès à l'objet receveur est important syntaxiquement et assez variable (`current, self, me` ...).

Illustrons cette méthode en traduisant la classe formelle `ONFLOORLift` en Eiffel et en Smalltalk. Cette classe formelle et ses traductions sont détaillées en annexe C.6 .

7.8.3 Eiffel

Eiffel propose un cadre de programmation rigoureux, où les classes sont des implantations de types abstraits de données. Une présentation détaillée du langage se trouve dans [Mey88].

A chaque classe formelle est associée une classe Eiffel écrite dans le fichier `<CFC>.e`. Si la classe formelle est abstraite, la classe Eiffel est DEFERRED. Il en est de même pour les méthodes. Si la classe est générique, la notation crochet `class[T]` est utilisée. Les commentaires sont ceux de la classe formelle.

```
-- Class for onfloor lift
-- straight translation without optimization
-- EIFFEL 2.3
-- 06/06/94

CLASS ONFLOORLift
```

La seconde tâche est d'écrire la clause d'exportation `EXPORT`. Elle contient les méthodes primitives et les caractéristiques exportées de la classe formelle. Chaque classe est libre d'exporter ou non les caractéristiques de la super-classe. La règle suivante est la même que dans les classes formelles : toute caractéristique exportée dans une super-classe est exportée dans la sous-classe. La routine spéciale `Create` est implicitement exportée et ne figure pas dans la liste.

```
EXPORT weight, bottomLevel, topLevel, capacity, chgCapacity, limits,
    isIn, oldstate, up, down, getIn, getOut, install, isOverloaded;
```

Ensuite la clause d'héritage `INHERIT` est construite. Le nom de chaque super-classe est indiqué et les méthodes qui sont redéfinies ou renommées par les clauses `RENAME` et `REDEFINE`. Les renommages sont utilisées pour résoudre les conflits de noms. Le calcul est réalisé par comparaison du comportement de la super-classe avec celui de la classe formelle en cours de traduction. Comme dans les classes formelles, la méthode d'instanciation `Create` n'est jamais héritée.

```
INHERIT ONLift
    REDEFINE weight, bottomLevel, topLevel, capacity, chgCapacity, isIn
```

Enfin, la partie la plus délicate est la définition des caractéristiques de la clause `FEATURE`. Des attributs Eiffel privés sont définis pour chaque nouveau sélecteur de champ et chaque sélecteur de champ spécialisé. Chaque sélecteur de champ devient une routine afin de pouvoir inclure la précondition du sélecteur de champ par la clause `REQUIRE`. Ce travail peut être affiné en ne considérant que les sélecteurs de champ conditionnels.

```
-- private fields
bottomLevel_private : Integer;
topLevel_private : Integer;
capacity_private : Real;
contents_private : ListWeightable;
isInstalled_private := Boolean;
oldState_private : INTERMEDIATELift;

-- field selectors
bottomLevel : Integer IS
    DO
    RESULT := bottomLevel_private;
    END; -- bottomLevel

oldState : INTERMEDIATELift IS
    REQUIRES not(isInstalled);
    DO
    RESULT := oldState_private;
    END; -- oldState
```

Viennent ensuite les méthodes primitives. La routine de création standard `Create` est la traduction de la primitive de création `new<CFC>`. Les paramètres sont les attributs de la classe (hérités, définis ou redéfinis). L'opération `new<CFC>` est un appel fonctionnel à `Create`.

```
-- creation
Create (Xbottom, Xtop : Integer; Xcap : Real; Xcontents : List;
   Xinst : Boolean; Xold : INTERMEDIATELift) IS
-- constraint translation
    REQUIRES Xbottom < Xtop and 0 <= contents.sum_weight
         and Xbottom <= Xlevel and ((Xinst and Xbottom <= Xtop)
         or (not(Xinst) and Xold.oldState.level + oldState.offset <= Xtp))
    DO
       bottomLevel_private := Xbottom;
       topLevel_private := Xtop;
       capacity_private := Xcap;
       contents_private := Xcontents;
       isInstalled_private := Xinst;
       oldState_private := Xold;
    END; -- Create
```

La contrainte est traduite par une précondition `REQUIRE` sur `new<CFC>` ou mieux par un invariant (`INVARIANT`) de classe. La méthode primitive `equal?` est implantée par `deep_equal` et `copy` est implantée par `deep_clone`.

A chaque extension est associée une routine Eiffel dont le profil est celui de l'extension hormis le receveur. La traduction des axiomes doit respecter la notation pointée d'Eiffel : `<selector>(<receiver><,args>*)` devient `<receiver>.<selector>(<,args>+)`$^{0/1}$. La précondition est donnée dans une clause `REQUIRE`. Les conditions des axiomes sont définies par les structures de contrôle conditionnelles `IF...THEN...ELSIF...END`. Le résultat de la méthode est affecté à la pseudo-variable `RESULT`. La variable `Self` s'écrit `Current` en Eiffel et un message de type `m(Self, args)` s'écrit `m(args)`.

```
-- overload
isOverloaded : Boolean IS
    DO
       RESULT := weight > capacity;
    END; -- isOverloaded
```

```
-- level
level : Integer IS
    REQUIRES not(isOverloaded);
    DO
    IF isInstalled
       THEN RESULT := bottomlevel;
       ELSE RESULT := oldState.level + oldState.offset;
    END;
    END; -- level

-- set a new capacity
chgCapacity (Xcap : Real) : ONFLOORLift IS
    DO
       -- copy is translated in creation
       IF isInstalled
          THEN RESULT.Create(bottomLevel, topLevel, Xcap, contents,
               isInstalled, undefined);
          ELSE RESULT.Create(bottomLevel, topLevel, Xcap, contents,
               isInstalled, oldState.chgCapacity(Xcap));
       END;
    END; -- chgCapacity

-- get up Xd levels
up (Xd : Integer) : INTERMEDIATELift IS
    REQUIRES Xd <= (toplevel - level) AND not(isOverloaded);
    DO
       RESULT.Create(current, Xd, false);
    END; -- up
```

Les méthodes de classes sont maintenant définies comme des méthodes d'instance en Eiffel.

```
-- it's a instance method in Eiffel, create a new lift
install (Xbottom, Xtop : Integer; Xcap : Real) : ONFLOORLift IS
    LOCAL undefined;
    DO
       -- it is an undefined value for oldState of Lift
       undefined.Create;
       RESULT.Create(bottomLevel, topLevel, capacity, empty, true, undefined);
    END; -- install
```

Le typage fort d'Eiffel est respecté dans la mesure où celui des classes formelles suit des règles similaires, hormis l'associattion de type **like**.

7.8.4 Smalltalk

Smalltalk-80 [LP90] est à la fois un langage, un système d'exploitation et un environnement de programmation. La richesse de la bibliothèque de classes prédéfinies et sa facilité d'utilisation en font un outil idéal pour le prototypage d'applications. A la différence d'Eiffel, toutes les classes sont accessibles directement dans l'environnement de travail. L'interface avec des fichiers se fait par la commande **fileIn** appliquée à des descriptions textuelles Smalltalk (fichier avec le suffixe **.st**). Il est conseillé de rentrer les classes dans l'ordre *super-classe* $\longrightarrow$ *sous-classe* pour éviter les liens indéfinis. Cette commande exécute aussi les méthodes d'initialisation de classe.

Toujours à la différence d'Eiffel, les noms de variables et de méthodes ne contiennent pas le caractère souligné, qui dans une version précédente de Smalltalk, désignait le destinaire du résultat d'une méthode (sorte d'affectation). Ils sont remplacés par des majuscules (manipulation de chaines de caractères). Certains noms ne doivent pas être interprétés, ils sont désignés par des symboles, ils sont préfixés par **#**.

Les différentes déclarations Smalltalk sont séparées par le point d'exclamation (`!`). La première chose à faire est l'écriture d'une entête de la description textuelle. Nous l'obtenons en lisant un fichier quelconque suffixé par `.st`.

```
'From Objectworks(r)\Smalltalk, Release 4 of 25 February 1991 on 17 October 1994 et\\
1:45:18 am'!
```

Viennent ensuite les déclarations de structure des instances de la classe et des liens d'héritage. Pour passer de l'héritage multiple des classes formelles à l'héritage simple de Smalltalk. Une politique simple est utilisée : un chemin d'héritage principal est choisi, puis les variables d'instance et les méthodes des chemins secondaires qui ne sont pas redéfinies dans la classe elle-même sont recopiées (règle de gestion des conflits des classes formelles). Autre particularité, en smalltalk les variables d'instance héritées ne sont pas déclarées. Un petit contrôle permet de savoir lesquelles sont héritées, lesquelles sont définies, lesquelles sont redéfinies (la redéfinition correspond à un sous-typage du type de la variable d'instance). Ces informations figurent en commentaire de la classe. Les paramètres de type n'apparaissent pas explicitement dans la déclaration, du fait de l'absence de types statiques. Ils figurent aussi en commentaire.

```
ONLift variableSubclass: #ONFLOORLift
   instanceVariableNames: 'bottomLevel topLevel capacity contents isInstalled oldState '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'TAGLift'!
```

Noter que le premier nom correspond à la super-classe (unique), et que le message `variableSubclass: .. category: ..` est envoyé à la classe `ONLift`, supposée exister. Le second nom ne désigne pas la classe mais uniquement son nom (c'est un symbole Smalltalk). Il n'y a pas de variables de classes ou de pool dans la traduction directe. La catégorie est le nom du TAG de départ. Les variables ne sont pas typées, pour ne pas perdre cette information, on la place en commentaire comme la contrainte ou la liste d'exportation.

```
ONFLOORLift comment:
'@Copyright P. Andre 1994
This class implements the ONFLOORLift class.

Here is the CF description :

bottomLevel   <Integer>
topLevel      <Integer>
capacity      <RealGt1>
contents      <ListWeightable>
isInstalled   <Boolean>
oldState      <INTERMEDIATELift>   requires: isInstalled(Self) == false

comment : class for onfloor lift

exports : limits, weight, level, getIn, getOut, isIn, up, down, install

constraints :
   bottomLevel(Self) < topLevel(Self) and
   0 <= weight(Self) and
   bottomLevel(Self) <= level(Self) <= topLevel(Self)'!
```

Les méthodes d'instance sont alors écrites. Quatre protocoles par défaut existent : **`field selectors`**, **`constructors`**, **`observers`** et **`private`**. Dans la description textuelle, chaque déclaration de protocole ou de méthode se termine par un `!`.

Tout d'abord, les méthodes correspondant aux sélecteurs de champ, en lecture et en écriture, sont créés automatiquement selon la convention usuelle de Smalltalk. La méthode de lecture est le nom du sélecteur de champ lui-même. La méthode d'écriture est le nom du sélecteur de champ suffixée par : et suivie du paramètre. Le nom du paramètre est celui du type correspondant, précédé du caractère `a` ou `an` selon la notation habituelle. La précondition est traduite automatiquement, lorsqu'elle est simple comme ici, par une alternative.

```
!ONFLOORLift methodsFor: 'field selectors'!

oldState
   "requires: isInstalled(Self) == false"

   self.isInstalled
      ifTrue: [^nil]
      ifFalse: [^oldState]!

oldState: anINTERMEDIATELift
   oldState := anINTERMEDIATELift!...!
```

Les autres méthodes primitives sont ainsi traduites :

– `equal` : l'égalité porte sur le contenu des objets (`=`) et non leur identité.

– `describe` : protocole `printing`, méthode `printOn:`. Soit la méthode de description textuelle par défaut est utilisée, soit elle est redéfinie.

```
!FCObject methodsFor: 'printing'!

describe
        "Textual representation of the object in a String"

        |aStream |
        aStream := WriteStream on: (String new: 16).
        self printOn: aStream.
        ^aStream contents! !
```

  d'autres méthodes sont possibles : `printString`, `storeOn:`, ...

– `copy` : implanté par les méthodes de copie simple `copy` (copie de pointeur par défaut), en profondeur `deepCopy`, de pointeur `shallowCopy`.

– d'autres sont propres à Smalltalk (`isNil`, `release`, `==`, `inspect`, `class`, `isKindOf:`, `respondsTo:`, `perform:`, `error`, `doesNotUnderstand`, `shouldNotImplement`, `subclassResponsability`).

Les extensions sont ensuite transformées en méthodes. Lorsque le traducteur ne peut traduire complètement les axiomes d'une méthode, il demande au concepteur Smalltalk de compléter.

Le profil est calculé automatiquement. Le commentaire correspond à la méthode de la classe formelle sous une forme textuelle. Les préconditions sont implantées par des alternatives, de même que les conditions des axiomes. Le résultat est le second terme de l'équation principale si le receveur de la méthode du premier terme est `Self`.

```
!ONLift methodsFor: 'constructors'!

getOOO
        ";; getOOO : the lift becomes out of order
        getOOO : ONLift --> OOOLift
        getOOO(Self) == new(OOOLift, oldState = Self)"

        ^OOOLift newXoldState: self! !
```

```
...
!Lift methodsFor: 'observers'!

bottomLevel
        ";; bottomLevel : bottom level of the lift
        bottomLevel : Lift --> Integer
        ABSTRACT"

        self subclassResponsability!...!
```

Les méthodes privées sont mises dans le protocole `private`. Les méthodes de création sont mises dans le protocole de la métaclasse selon les mêmes normes que la traduction des méthodes d'instance. Si la classe formelle comprend des valeurs par défaut pour ses sélecteurs de champ, ils sont décrits en commentaire et mis en oeuvre dans la méthode d'initialisation `initialize`.

Il n'y a pas de distinction entre classe abstraite et classe instanciable en Smalltalk. Cette distinction est forcée en redéfinissant la méthode `new`. Nous ne ferons qu'indiquer la différence en commentaire. Les méthodes abstraites sont implantées par `self subclassResponsability`. Le masquage est possible en Smalltalk par `self shouldNotImplement`.

Plusieurs méthodes d'instanciation sont possibles, faisant appel à la méthode de création par défaut `new`. Pour simplifier la présentation, nous pouvons définir une classe `CFObject` qui synthétise le comportement minimal d'un objet issu d'une classe formelle et dont hériteront toutes les classes produites par le traducteur.

7.9 Conclusion

Dans cette section, nous avons donné des éléments d'une démarche de conception. Nous avons décrit plus précisément le passage des TAG aux classes formelles et celui des classes formelles aux classes des langages de programmation.

Pour la conception des TAG en classes, nous proposons une solution en plusieurs étapes réalisant des tâches indépendantes, dont l'intérêt est d'être applicables à d'autres situations. L'automatisation n'est que partielle et le concepteur doit intervenir pour choisir entre plusieurs alternatives et doit proposer des idées de restructuration.

Le passage des classes formelles aux classes des langages de programmation est plus direct. Si les axiomes respectent une présentation fonctionnelle alors la traduction est entièrement automatisable. Il n'en reste pas moins que le programme n'est pas efficace et n'utilise pas au mieux les spécificités du langage cible. Le programmeur devra intervenir ensuite pour améliorer cette situation. Cette section démontre toutefois des avantages certains du modèle : réutilisabilité entre langages de programmation, rétro-conception de classes, portabilité d'applications entre langages et entre versions successives d'un même langage.

Des expérimentations plus poussées sont indispensables pour documenter le processus et fournir des conseils plus précis au concepteur. D'autres traductions sont envisagées, notamment celle vers un système à automates communicants pour concevoir des architectures distribuées.

Conclusion

Chapitre 8

Conclusion

"L'oubli de ses propres fautes est la plus sûre des absolutions."
Konrad ADENAUER.

Cette thèse est une contribution au développement du logiciel de qualité par utilisation de méthodes formelles à objets.

8.1 Bilan

Dans un premier temps, nous nous sommes attachés à mettre en évidence les problèmes du développement du logiciel et les réponses actuelles à ces problèmes. Un problème majeur est la maîtrise de la complexité du développement. Cette complexité est la conjonction de plusieurs facteurs : la multiplication des modèles de représentation et la difficulté de les agencer dans des spécifications, la variété des philosophies de résolution de problèmes et des cycles de développement, complexité des relations entre spécifications au cours du processus de développement, diversité et complexité des applications traitées. La réponse à ce problème de complexité passe par la définition de qualités des spécifications et du processus et de critères mesurables qui influencent ces qualités. Il en résulte de cette étude que les critères principaux pour une bonne méthode de développement sont la modularité, la cohérence et l'auto-documentation pour les spécifications et la rigueur, l'automatisation, le contrôle et la réutilisation pour les processus. Nous avons dès lors pu montrer en quoi les méthodes formelles et les méthodes à objets sont un apport incontestable dans le développement du logiciel. Les premières apportent la rigueur dans le développement : étude poussée des besoins diminuant les erreurs, automatisation du processus, cohérence et documentation des spécifications. Les secondes permettent la prise en compte de domaines volumineux par des outils de structuration puissants : modularité des spécifications, auto-documentation intrinsèque des objets, richesse des relations entre spécifications, distribution du processus.

Après avoir montré l'intérêt du modèle à objets et les méthodes formelles dans développement du logiciel. Nous avons constaté que ces deux approches ont des qualités complémentaires. Ce qui fait dire à certains auteurs, comme [CLLF93, LH93], que l'avenir passe par les méthodes formelles à objets. Beaucoup de travaux récents, notamment des travaux entre industriels et universitaires, sont des propositions dans ce domaine nouveau. Nous avons essayé de synthétiser cette approche dans un panorama, qui est à notre avis, représentatif, sans être exhaustif. Des exemples illustrent chacune des approches. Nous avons en particulier mis en valeur le fait que l'intégration des deux concepts n'est pas si simple et que des formalismes trop opérationnels ne permettent plus de distinguer le problème de sa solution. Certains problèmes difficiles comme le raffinement de spécification ou la concurrence, sont maintenant étudiés dans un cadre plus complet, et des propositions émergent. Quelques problèmes subsistent :

- aide à la construction de modèles, aide au raffinement, écriture des preuves,
- cohésion de modèles, séparation des différentes activités du développement, modèles sémantiques,
- liaison avec les systèmes existants, interopérabilité.

Nous avons alors décrit une proposition qui répond aux besoins généraux d'introduction de formalisme dans le développement à objets. Les modèles que nous proposons sont en général moins riches que ceux présentés dans la synthèse du chapitre 3. N'ayant pas un langage de spécification hôte, nous avons moins de problèmes d'intégration. Nous avons voulu définir un noyau fiable et extensible avec une sémantique algébrique. Les spécifications sont exécutables après transformations. Nous avons favorisé l'intégration transitionnelle pour la spécification et la définition d'un langage à objets haut niveau pour la conception abstraite. Nous avons insisté sur la lisibilité et la construction progressive et automatisée des spécifications.

Nous avons privilégié une démarche multi-langage : un langage pour la spécification abstraite de composants logiciels (TAG), un langage pour la conception abstraite des objets (CF) et des langages cibles pour la programmation. Chaque langage correspond à un niveau d'abstraction différent. Plusieurs avantages en sont retirés : le raisonnement se fait à des niveaux d'abstraction suffisants, le raffinement prend en compte progressivement les contraintes de conception et de réalisation en évitant la surspécification, la construction des composants et leur évolution est plus facilement automatisable.

La formalisation est progressive à l'intérieur même des langages : l'automate du TAG, plus lisible et plus facile à construire, est à la base de la construction du modèle algébrique; l'aspect d'une classe formelle est suffisant à sa description.

A l'intérieur d'un langage, les concepts que nous avons introduit ont une double interprétation : un type abstrait graphique est un automate et une spécification algébrique, une classe formelle est une spécification algébrique et une classe. Ceci permet de décrire des aspects différents sans risquer d'introduire d'incohérences comme dans les approches multi-modèles. La cohérence entre les différents niveaux s'en trouve également améliorée.

Nous avons montré que les transition entre langages étaient méthodiques et largement automatisables. Nous avons enfin insisté sur l'ouverture vers d'autres environnements de spécification et de preuve.

Le type abstrait graphique est un outil puissant, souple et bien adapté aux systèmes avec des préconditions complexes. Un des intérêts est de faire cohabiter une description dynamique et une spécification algébrique de type abstrait algébrique. L'utilisation d'un automate pour l'extraction d'une spécification algébrique nous semble donc un apport pratique intéressant qui ne se limite pas à notre cadre. Le résultat est plus lourd qu'avec les techniques usuelles d'écriture de spécifications mais il est plus sûr. Le traitement d'erreur est facilité et la compréhension générale des types de données complexes est facilitée par la notation graphique.

L'apport des classes formelles est de permettre une conception formelle à objets, de haut niveau, dans laquelle les contrôles stricts et les preuves sont une assurance qualité des implantations qui en découleront. L'utilisation de champs conditionnels augmente la puissance d'expression des classes et permet un contrôle plus efficace des relations entre classes (dépendance structurelle, héritage). Avoir un modèle formel des classes autorise les preuves abstraites, l'automatisation des contrôles et du codage. Le modèle nous a permis aussi d'étudier des aspects théoriques des langages à objets (héritage, typage, évaluation et sémantique).

Le formalisme dans l'un et l'autre des modèles est un gage dans la certification des spécifications mais aussi une aide importante dans le processus de développement. Le passage des types abstraits graphiques aux spécifications algébriques et celui des classes formelles aux langages de programmation est en grande partie automatisable. Le passage entre les types abstraits graphiques et les classes formelles nécessite quant à lui une véritable activité de conception. Nous en avons détaillé les étapes et les outils logiciels associés.

Il est préoccupant de positionner une proposition de développement à objets tel que la nôtre en fonction d'une norme. La norme pour l'analyse et la conception à objets de l'OMG n'est pas entièrement finalisée, le dernier document officiel produit étant [Hut94b]. Les aspects généraux suivant sont communs aux types abstraits graphiques, classes formelles et au modèle à objets de référence : pas de méta-classe, pas de fonction d'ordre supérieur, une sélection simple et des types abstraits .

Le concept central du modèle à objets de référence est le type comme dans les types abstraits graphiques et classes formelles et non la classe comme dans OMT. Les concepts de classe, d'attribut, d'association ... sont absents du modèle à objets de référence et font partie d'extensions dédiée à un domaine particulier.

L'autre concept central du modèle à objets de référence est l'héritage entre types, c'est-à-dire l'héritage d'interface et non l'héritage d'implantation, ce qui correspond dans notre démarche à l'héritage des classes formelles. Comme pour le modèle à objets de référence, les modèles types abstraits graphiques et classes formelles utilisent une relation de sous-typage. Notez que notre définition de classes formelles est suffisamment abstraite pour constituer un outil d'analyse et de conception de très haut niveau d'abstraction comme l'incite la norme et que peu d'approches de développement à objets suivent.

Un avantage des modèles types abstraits graphiques et classes formelles est qu'ils incluent un formalisme précis. D'autre part nous avons des contrôles de la redéfinition des méthodes plus généraux que ceux du modèle à objets de référence; il en est de même pour les règles d'exportation des opérations.

8.2 Perspectives

Cette thèse n'est pas la définition complète d'un langage de spécification ou d'une méthode de développement. Elle est plutôt une définition précise d'un cadre de travail, dans le domaine vaste du génie logiciel. Nous avons en effet mis en valeur l'intérêt des propositions, et présenté les bases de deux modèles de spécification et conception ainsi que des idées et algorithmes sur la construction des spécifications et leur raffinement. Pour être pleinement utilisables ces modèles et démarches doivent être étendus, validés sur des applications industrielles, simplifiés et outillés.

Les modèles types abstraits graphiques et classes formelles sont d'un usage assez strict, ils sont trop pauvres pour certaines applications. Ils sont utilisables pour la spécification de composants logiciels, et aussi a priori pour la spécification des transactions, dans les applications bancaires. Des extensions vers la concurrence et les communications asynchrones doivent être envisagées. Par ailleurs, la notion de métaclasse et surtout un typage plus puissant (au moins avec une forme de quantification bornée comme le `like` d'Eiffel) sont souhaitables. Pour l'aspect codage des études doivent préciser un certains nombre de règles pour l'introduction des effets de bords et des optimisations; autant que possible des outils doivent assister pendant ces phases fastidieuses et sources d'erreurs. Un des points sensibles, à notre avis, est la spécification naturelle d'un ensemble “modulaire” d'objets ayant des buts communs. Une solution nous semble être de passer par une définition abstraite des interfaces de ce groupe d'objet. Une étude complète sur ce sujet est indispensable pour gérer des ensembles complexes d'objets.

Un des problèmes majeurs des méthodes d'analyse et conception par objets est de trouver les objets. Nous l'avons omis ici car la spécification formelle est une étape suivant une analyse. Cependant, nous avons donné des idées de passage d'une méthode d'analyse à objets telle que OMT vers les types abstraits graphiques dans [ABR95]. Les types abstraits graphiques correspondent bien aux approches dans lesquelles le comportement dynamique de l'objet est privilégié par rapport à sa structure, ce qui est plus difficile mais plus cohérent.

Nous avons fait une proposition de méthode pour passer des types abstraits graphiques aux classes formelles, et nous l'avons expérimenté sur divers exemples. La démarche que nous proposons doit être étendue, diversifiée et validée sur des cas de tailles plus importantes. Par ailleurs, une généralisation des idées de cette méthode de raffinement devrait permettre de dégager des algorithmes applicables à d'autres méthodes que celle des TAG/CF.

Nous avons défini un environnement ouvert avec des passerelles vers d'autres systèmes. C'est une voie vers l'interopérabilité, via un modèle commun. C'est aussi une approche économique de développement d'outils complexes. Un des avantages des modèles types abstraits graphiques et classes formelles est la compatibilité d'outils. En effet, comme les deux formalismes sont basés sur les spécifications algébriques, des outils communs sont utilisés. Les perspectives actuelles sont de développer un outillage spécialisé dans certaines phases :

- contrôle plus approfondi dans l'édition des types abstraits graphiques et des classes formelles, basé sur la sémantique donnée dans cette thèse,
- améliorer la génération de la spécification algébrique et les traducteurs vers Eiffel et Smalltalk.
- définir précisément et implanter les interfaces vers d'autres systèmes de spécification et de preuve,
- réaliser un évaluateur plus performant pour exécuter les spécifications algébriques à objets sans passer par un environnement (restrictif) de spécification algébrique,
- améliorer le degré d'optimation de la conception des types abstraits graphiques en classes formelles, en donnant notamment des critères de restructuration d'un ensemble de classes,
- gérer efficacement une bibliothèque de classes et la traçabilité des objets d'un modèle à un autre.

Notre expérience de la programmation à objets et des bases de données à objets montre que l'évolution des programmes mène à un accroissement rapide du nombre de classes. Une dernière perspective est donc l'exploitation du formalisme dans la gestion de classe : recherche et comparaison de classes, restructuration d'un graphe d'héritage. L'automatisation partielle de cette gestion est un atout des systèmes à objets.

Annexes

Annexe A

Rappels sur les spécifications algébriques

Nous donnons ici brièvement quelques définitions et notations de base des spécifications algébriques, inspirées de [Bre91, EM85, Gau90, BB92, BG94a] et de diverses autres sources (cours, articles, [Gue94]). Dans un premier temps, nous ne considérons que les opérations totales.

A.1 Définitions de base

Soient $SORTES$, $OPER$, VAR des ensembles non-vides de symboles, tels que $OPER \cap VAR = \emptyset$. $SORTES$ est l'ensemble de tous les sortes. $OPER$ est l'ensemble de tous les noms d'opérations. VAR est l'ensemble de tous les noms de variables. Une **sorte** est un nom de type.

Définition A.1.1 (S-ensemble)
Etant donné un ensemble S de sortes, un $S-$ensemble est un ensemble A muni d'une partition indexée par S, $A = \sqcup_{s \in S} A_s$, où $\sqcup$ désigne l'union disjointe.

Définition A.1.2 (S-morphisme)
*Etant donné un ensemble S de sortes, un **S-morphisme** entre deux ensembles A et B est une application μ de A dans B telle que $\forall s \in S \bullet \mu(A_s) \subset B_s$.*

Le S-morphisme préserve les sortes : l'image d'une donnée de "type" s par μ est encore de type s.

Signature et algèbres

Définition A.1.3 (Signature)
*La **signature** d'un type abstrait est un couple (S, F) où S est un ensemble fini de **sortes** tel que $S \subset SORTES$ et une famille $S^* \times S$-indexée $F = (F_{w,s})_{w \in S^*, s \in S}$ de nom d'opération f, tel que $f \in OPER$.*

Une famille est un ensemble d'ensembles. Un **profil d'opération** est un élément de F, noté $f\ :\ s_1 ... s_n\ \rightarrow s$, $s_1 ... s_n$ et s sont appelés respectivement le **domaine** et le **co-domaine** de f. L'arité est égale à n. Les opérations ou fonctions d'arité nulle sont appelées **constantes**.
Remarques :
- pour certains auteurs, l'arité sur un ensemble de sortes S est un mot fini non vide sur S.
- la surcharge d'opérations est autorisée i.e. un même nom d'opération peut avoir plusieurs profils du moment que le domaine varie. Dans [Bre91] la surcharge est possible uniquement si les profils respectifs suivent une **condition de monotonicité** liée au sous-typage des sortes : $f \in F_{w,s} \cap F_{w',s'} \ \wedge\ w_1 \leq w_2 \Rightarrow s_1 \leq s_2$. Dans la suite, cette surcharge sera restreinte au

polymorphisme des opérations avec redéfinition (voir sous-typage et héritage).
- contrairement à certains auteurs [BB92, Gue94], les noms d'opérations ne sont pas nécessairement distincts deux à deux (i.e. F est un ensemble de profils et pas uniquement de noms d'opérations). Par abus de langage, un élément de $F_{w,s}$ est appellé **opération** (ou fonction).

La signature permet d'obtenir l'ensemble des expressions syntaxiques qu'on peut construire. En y ajoutant des variables, des propriétés plus générales sont exprimées.

Définition A.1.4 (terme)
Etant donnés une signature $\Sigma = (S, F)$ et un S-ensemble de variables X, on définit inductivement $T_{\Sigma[X]}$, le (plus petit) S-ensemble des Σ-termes avec variables qu'on peut former à partir des opérations de Σ, par

(i) $\forall f :\rightarrow s \in F \bullet f \in T_{\Sigma[X]_s}$,
(ii) $\forall x \in X_s \bullet x \in T_{\Sigma[X]_s}$,
(iii) $\forall f : s_1, ..., s_n \rightarrow s \in F,\ t_i \in T_{\Sigma[X]_{s_i}}, i = 1, ..., n \bullet f(t_1, ..., t_n) \in T_{\Sigma[X]_s}$.

Ces termes sont bien formés et conservent la compatibilité des sortes. $T_\Sigma \widehat{=} T_{\Sigma[\emptyset]}$ est l'ensemble des **termes clos** (i.e. sans variables), on dit aussi que c'est le langage engendré par la signature. Un terme quelconque t est de type T, $\texttt{typeOf}(t) = T$, si $t \in T_{\Sigma[X]_T}$.

S'il existe une relation d'ordre partielle entre les sortes (les types), alors l'ensemble des termes n'est pas un S-ensemble mais une famille, car un terme appartient à plusieurs types (ordonnés).

On s'intéresse à la classe des algèbres qui peuvent être interprétées comme des modèles de cette signature. Une interprétation est une fonction de l'algèbre des termes avec variables dans une algèbre.

Définition A.1.5 (Σ-algèbre)
Etant donnée une signature $\Sigma = (S, F)$, une Σ-algèbre A est donnée par deux familles: une famille $A_S = (A_s)_{s \in S}$ d'ensembles de valeurs de A, et une famille $F^A = (f^A)_{f \in F}$ de fonctions, telles que si $f : s_1\ s_2 \ ...s_n \rightarrow s$ est une opération de F alors f^A est une fonction de $A_{s_1} \times ... \times A_{s_n}$ dans A_s.

Remarques :
- A_s est le **support** de s dans A.
- f^A est l'interprétation de f dans A.
- l'algèbre triviale TRIV est celle qui associe à chaque terme sa sorte.
- T_Σ est l'algèbre des Σ-termes.
- si S n'est pas un singleton, alors les algèbres issues de Σ sont hétérogènes.
- $ALG(\Sigma)$ est la classe des Σ-algèbres.
- une algèbre possédant plusieurs ensembles supports est dite hétérogène (i.e. plusieurs sortes).

Les morphismes permettent de comparer et étudier les algèbres.

Définition A.1.6 (Σ-morphisme)
Etant donnée une signature Σ, un Σ-morphisme entre deux Σ-algèbres A et B est un S-morphisme μ de A vers B tel que pour chaque nom d'opération de la signature $f\ :\ s_1...s_n\ \rightarrow s$ et pour tout n-uplet $(a_1,\ ..., a_n) \in A_{s_1}\ x...x\ A_{s_n}$ on a :
$\mu(f_A(a_1,\ ..., a_n)) = f_B(\mu(a_1),\ ..., \mu(a_n))$

Un isomorphisme est un morphisme bijectif. L'intérêt des isomorphismes est de regrouper en classes les algèbres "égales".

Regardons les liens entre les termes produits et les algèbres qui les interprètent.

Définition A.1.7 (Valuation)
Etant donnés une signature $\Sigma = (S, F)$, un S-ensemble de variables (typées) $X = (X_s)_{s \in S}$ et

une Σ-algèbre A, une ***valuation*** *de V dans A est un S-ensemble d'applications $v = (v_s)_{s\in S}$ telles que $\forall s \in S \bullet v_s : X_s \rightarrow A_s$.*

Définition A.1.8 (Interprétation)
Etant donnés une signature $\Sigma = (S, F)$, un S-ensemble de variables $X = (X_s)_{s\in S}$, une Σ-algèbre A, et une valuation $v = (v_s)_{s\in S}$ de X dans A, la ***fonction d'interprétation****, notée $[\![\_]\!]_v^A$, est définie par: $[\![\_]\!]_v^A : T_{\Sigma[X]} \rightarrow A$*
$(1)(\forall s \in S), (\forall x \in X_s) \bullet [\![x]\!]_v^A = v_s(x)$
$(2)(\forall s \in S), (\forall c :\rightarrow s \in F) \bullet [\![c]\!]_v^A = c^A$
$(3)(\forall s_1, .., s_n, s_{n+1} \in S), (\forall f : s_1, .., s_n \rightarrow s_{n+1} \in F)(\forall t_1 \in T_{\Sigma[X]_{s_1}}), ..., (\forall t_n \in T_{\Sigma[X]_{s_n}}) \bullet$
$[\![f(t_1, .., t_n)]\!]_v^A = f^A([\![t_1]\!]_v^A, ..., [\![t_n]\!]_v^A)$

Soit t un terme de sorte s avec les variables $x_1, .., x_n$ de sortes $s_1, .., s_n$, t désigne une fonction de $A_{s_1} \times ... \times A_{s_n} \rightarrow A_s$ qui est la composition des différentes fonctions f^A apparaissant dans t. Cette fonction est notée t^A.
Lorsque t est un terme clos, son interprétation dans l'algèbre A est notée $[\![t]\!]^A$. Cette fonction est l'unique Σ-morphisme de T_Σ dans A.

Nous allons voir maintenant les propriétés de la fonction d'interprétation.

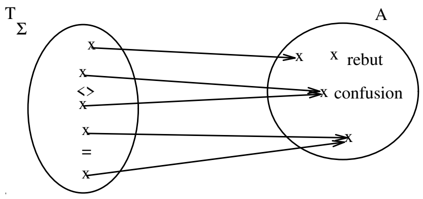


Figure 68 : Fonction d'interprétation

Définition A.1.9 (Confusion)
Il y a ***confusion*** *lorsque l'interprétation affecte la même valeur à des termes différents de la signature.*

Exemple: $I_{NAT\rightarrow I\!N} \widehat{=} \{zero \mapsto 0, succ(zero) \mapsto 0, ...\}$.

Définition A.1.10 (Rebut)
Un ***rebut*** *("junk") est une valeur de l'algèbre non calculable par la signature.*

Exemple: interpréter NAT dans $I\!N \cup \{-1\}$.
Remarque: il n'existe pas de valeurs de la signature absente de l'algèbre, sinon elles seraient non-interprétées.

Une signature est cohérente s'il n'y a pas de confusions. Elle est complète s'il n'y a pas de rebuts (i.e. si la fonction d'interprétation de la figure 68 est surjective). Dans la suite, nous nous s'intéressons uniquement aux algèbres dont les valeurs sont interprétation d'un terme de la signature, par les algèbres finiment engendrées.

Définition A.1.11 (Σ-algèbre finiment engendrée)
Etant donnée une signature $\Sigma = (S, F)$, une Σ-algèbre A est dite ***finiment engendrée*** *si et seulement si $(\forall s \in S), (\forall a \in A_s) \bullet \exists t \in T_\Sigma \bullet [\![t]\!]_v^A = a$*

Définition A.1.12 (Générateurs d'une algèbre)
Etant donnés une signature $\Sigma = (S, F)$ et une Σ-algèbre A, A est finiment engendrée par la sous-signature $\Sigma' = (S, C)$ de Σ si pour toute valeur v d'un ensemble support de A, il existe un terme t de $T_{\Sigma'}$ tel que $[\![t]\!]^A = v$. Les opérations de C sont appelées ***générateurs*** *de A.*

Une signature $\Sigma = (S, C, FS)$ sépare les générateurs (C) des autres opérations (FS).

Définition A.1.13 (signature raisonnable)
Une signature $\Sigma = (S, F)$ dite ***raisonnable*** *si son algèbre triviale est finiment engendrée.*

Pour qu'une signature soit raisonnable, il suffit que chaque sorte de la signature soit habitée [BB92].

Définition A.1.14 (sorte habitée)
*Une sorte s est **habitée** si et seulement si son algèbre des termes contient au moins un terme clos.*

Axiomes et présentation

Les axiomes sont des formules logiques définies sur les opérations, qui leur donnent leur propriétés. Soit un type booléen[1]. Toutes les opérations produisant un résultat de ce type sont appelées prédicat.

Définition A.1.15 (Equation, Equation conditionnelle positive)
*Une **équation** ou Σ-équation est une paire de termes avec variables de même sorte et notée $t == t'$[2]. Un Σ-axiome est une formule logique construite sur les Σ-équation avec les connecteurs du calcul des propositions : $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$. Une équation conditionnelle positive (appelée **axiome** par abus de langage) est un Σ-axiome de la forme $e_1 \wedge e_2 \wedge \ldots \wedge e_n \Rightarrow e$, où e_1, ..., e_n, e sont des Σ-équations.*

Une Σ-algèbre satisfait une Σ-équation $t == t'$ ayant ses variables dans X si pour toute valuation v de X dans A $[\![t]\!]_v^A = [\![t']\!]_v^A$. Cette définition se généralise facilement aux axiomes avec l'interprátion habituelle des connecteurs.

Définition A.1.16 (Présentation)
*Une **présentation** d'un TAA est un triplet $PRES = (\Sigma, X, E)$ où Σ est une signature, X un ensemble de variables de sorte $s \in \Sigma$ et E un ensemble d'axiomes sur Σ.*

Une Σ-algèbre A satisfait PRES signifie que A satisfait tous les axiomes de $PRES$. ALG(Σ, E) est la sous-classe de ALG(Σ) des Σ-algèbres qui valident (Σ, E) et GEN(Σ, E) le sous-ensemble des algèbres finiment engendrée de ALG(Σ, E). Une telle présentation est aussi appelée **spécification** ou spécification de base et notée $SPEC$. Nous verrons des présentations plus modulaires dans la section A.2.

Définition A.1.17 (cohérence)
Un ensemble E d'axiomes est cohérent ssi $GEN(\Sigma, E) \neq \{TRIV\}$, c'est-à-dire s'il existe au moins une algèbre qui valide les axiomes et telle qu'il y ait deux éléments différents dans un de ses domaines.

Le moyen le plus sûr et le plus fréquent pour montrer la cohérence est de donner un modèle non trivial de la spécification. C'est le cas de l'algèbre initiale quand elle existe.

Les prédicats d'égalité, inclusion, union et intersection sur les présentations sont définis en considérant ces dernières comme des produits cartésiens [BB92, Gue94].

Sémantique par approche initiale

Dans l'approche **initiale** (Goguen) il y a une seule Σ-algèbre correspondant à une présentation. Une algèbre initiale existe lorsque les axiomes de la présentation sont équationnels ou conditionnels positifs.

Théorème A.1.18
Soit $PRES = (\Sigma, E)$ une présentation positive conditionnelle,

[1] La définition de ce type est simple, et figure dans la plupart des documents sur les spécifications algébriques.
[2] La notation "==" est empruntée à Pluss et permet de ne pas confondre cette congruence avec l'égalité habituelle, utilisée dans les explications.

- il existe une plus petite congruence sur T_Σ compatible avec E, notée $\cong_E$;
- le quotient de T_Σ par $\cong_E$ est une Σ, E-Algèbre, qui est initiale dans $ALG(\Sigma, E)$.
Cette algèbre est notée $T_{\Sigma,E}$ ou $\Sigma_{T/E}$.

La sémantique initiale d'une spécification SPEC est la classe $ADT(SPEC) = \{A/A \cong T_{SPEC}\}$ [EM85]. L'algèbre initiale correspond à un type de données dans lequel les valeurs sont les classes d'équivalence de l'algèbre des termes clos. Comme ce sont les plus petites classes d'équivalence, $T_{\Sigma,E}$ satisfait uniquement les axiomes de E : *il n'y pas d'autres propriétés ni valeurs* [Gau90].

Théorème A.1.19
L'algèbre initiale n'a ni rebut ni confusion.

preuve : voir [Gau90].

Cette approche pose le problème de l'existence d'une algèbre initiale : les axiomes sont restreints aux équations simples ou conditionnelles positives; l'ajout dans le langage de spécification de mécanismes comme la généricité ou le traitement d'exception, est difficile [BB92].

Sémantique par classes de modèle

Dans les approches **classe de modèles**, la spécification exprime les conditions minimales que doit satisfaire le type de donnée. Plusieurs algèbres peuvent valider la spécification. [Gau90] donne l'exemple de la commutativité de l'addition des entiers naturels, qui est une propriété non-contradictoire avec les axiomes mais pas requise par les axiomes pour autant. La sémantique à classe de modèles d'une spécification SPEC est la classe $ALG(SPEC) = \{A/A \cong SPEC - alg\grave{e}bre\}$ [EM85]. Les algèbres de ALG(Σ, E) sont celles qui n'ont pas de rebuts (sauf pour les types génériques) et pas de confusion. C'est le cas de GEN(Σ, E).
Remarque : l'algèbre initiale (resp. triviale) est initiale (resp. finale) dans la catégorie où les objets sont des Σ-algèbre et les flèches des Σ-morphismes.

Un autre moyen de supprimer les algèbres inintéressantes est de construire des spécifications modulaires structurées, utilisant en général un type booléen. Puis de montrer que la construction préserve le propriétés, et notamment qu'on n'a jamais $vrai = faux$.

A.2 Spécification structurée

Jusqu'à maintenant, nous avons vu les définitions correspondant à des spécifications dans lesquelles sont définies plusieurs sortes avec leurs propriétés. Elles sont appelées spécifications **plates**. Nous allons voir maintenant des éléments de structuration modulaire des spécifications. Il s'agit de définir des opérateurs sur les signatures et sur les spécifications.

Afin d'introduire modularité, généricité, réutilisation, abstraction et sous-typage dans les spécifications algébriques de types abstraits, les différents auteurs ont ajouté des mécanismes pour combiner les spécifications :

- importation (ou enrichissement) : use (PLUSS), sum (ASL), protecting (OBJ),
- paramétrisation et instanciation,
- sous-sorte (ou sous-typage ou héritage de spécialisation) : subsort (OBJ),
- raffinage (ou héritage d'implantation) : opérateurs de structuration (Sacso),
- renommage,
- masquage (exportation/importation) : hidden sort, forget et restrict d'ASL,

Les foncteurs sont un outil pour définir la sémantique des spécifications algébriques combinées.

Les définitions et propriétés des spécifications algébriques structurée peuvent être trouvés dans [Bid82, Bre91, EM85, Gau90, GM87a, Gue94].

Spécification hiérarchique

Voyons la construction incrémentale de spécifications par inclusion de spécifications.

Définition A.2.1 (Spécification hiérarchique)
*Une spécification **hiérarchique** est un triplet* (Σ, E, P) *où* P *est une présentation* (Σ_P, E_P) *telle que* Σ_P *est inclu dans* Σ *et* E_P *est inclu dans* E. P *est appelée partie primitive de la spécification et peut être une spécification hiérarchique.*

Définition A.2.2 (Σ_P-réduction, Σ_P-extension)
Etant données deux signatures, Sigma et Σ_P *telles que* Σ_P *soit inclue dans* Σ, A *une* Σ_P*-algèbre et* B *une* Σ*-algèbre,* A *est une* Σ_P*-réduction de* B *si*

$$\forall s \in \Sigma_P \bullet A_s = B_s \ \wedge \ \forall f \in \Sigma_P \bullet f^A = f^B.$$

Inversement, B *est une* Σ*-extension de* A.

Définition A.2.3 (Modèle hiérarchique)
Etant donnée une spécification hiérarchique algébrique $HSPEC = (\Sigma, E, P)$, *une* Σ*-algèbre* A *est un modèle hiérarchique de* (Σ, E, P) *si elle appartient à* $GEN(\Sigma, E)$ [3] *et sa* Σ_P*-réduction est un modèle hiérarchique de* P. *La classe des modèles hiérarchiques de* (Σ, E, P) *est notée* ***HMOD(Σ, E,P)***.

Définition A.2.4 (complétude suffisante)
Une spécification hiérarchique (Σ, E, P) *est **suffisamment complète** si tout terme* t *de* T_Σ *de sorte primitive peut être prouvé égal à un terme* t_P *de* T_{Σ_P}.

Définition A.2.5 (cohérence hiérarchique)
Une spécification hiérarchique (Σ, E, P) *est **cohérente hiérarchiquement** si pour tout terme* t_P,t'_P *de* T_{Σ_P}, *si* $t_P = t'_P$ *peut être prouvé dans* (Σ, E, P) *alors il peut être prouvé dans* P.

Ces définitions générales sont simples mais assez difficiles à montrer. Comme beaucoup d'auteurs, nous préconisons une approche constructive par opérateurs de structuration.

Spécifications paramétrées

Définition A.2.6 (Spécifications paramétrées)
Une spécification paramétrée est définie par une paire $PSPEC =< SPEC, SPEC_1 >$ *où* $SPEC =< \Sigma, E >$ *est appelée paramètre formel et* $SPEC_1 = SPEC+ < \Sigma_1, E_1 >$ *est la spécification cible.*

Deux sémantiques existent pour les spécifications paramétrées : une par foncteur libre, l'autre par transformation calculable de spécifications, permettant des paramètres ayant des modèles non engendrés finiment [Gau90]. Cette dernière approche permet de simuler la généricité par l'héritage, en paramétrant par un plus grand type commun.

Enrichissement vertical de spécifications: vers l'implantation

L'enrichissement vertical de spécification consiste à donner une représentation plus concrète d'une signature avec axiomes par une autre signature. Par exemple : les entiers naturels peuvent être spécifiés par des listes de nombres naturels.

Une des difficultés est la compatibilité horizontale (avec la hiérarchisation et la paramétrisation). Ehrig a résolu ce problème dans le cadre initial par le triplet de foncteurs "*forget-restrict-identify*".

Dans Sacso, des opérateurs de restructuration (séquence, ensemble, table, produit cartésien) servent au raffinage des types abstraits [DLS87, Sou89].

[3] *est un modèle initial de* (Σ, E) pour la sémantique initiale

A.3 Spécification partiellement ordonnée

Les définitions ci-dessus peuvent être étendues aux **algèbres partielles**, algèbres pour lesquelles le domaine de définition des opérations est assujetti à un prédicat de définition (voir [Bre91, Gau90]).

Définition A.3.1 (Prédicat de définition)
Etant donnée une signature $\Sigma = (S, F)$, de type de données partiel, pour chaque sorte de S, il y a un prédicat D_s appelé ***prédicat de définition****.*

Si A est une Σ-algèbre et t un terme de $(T_\Sigma)_s$ alors $D_s(t)$ est satisfait par A si t est défini dans A ($\exists b : A_s \bullet [\![t]\!]^A = b$). Les égalités entre termes sont fortes : soit les deux termes sont indéfinis soit ils sont définis et égaux.

Il est aussi possible dans ce cas, de définir un ordre sur les spécifications : ordre portant sur les ensembles supports des algèbres. Le polymorphisme des opérations est ainsi accepté. Nous ne décrirons pas les modifications à apporter sur les définitions précédentes. Elles consistent principalement à ajouter des conditions de définitions pour les éléments. Consulter [EM85, Bre91, GM87a] pour des définitions précises sur les spécifications ordonnées.

A.4 Héritage et sous-typage

La relation de sous-typage est un ordre partiel sur les types. La principale propriété du sous-typage est la règle de substitution : chaque valeur du sous-type peut être utilisée dans un contexte du supertype. Cette propriété induit le polymorphisme des valeurs et des opérations. Une valeur d'un type donné `T` est aussi de type tous les super-types de `T`. Une opération polymorphe est applicable à plusieurs types (les sous-types de ses paramètres).

Dans [CW85], un type est un ensemble de valeurs et le sous-typage est défini comme une inclusion d'ensembles. Il existe un plus grand ensemble, contenant toutes les valeurs calculables, et un plus petit ensemble, l'ensemble vide. Cette vision est intéressante pour les types construits par des opérateurs (union, produit). Le sous-typage des opérations est définie par une règle contra-variante sur le résultat. Cette approche nous semble trop stricte. Elle ne prend pas en compte les opérations et des contraintes. Ainsi, un compte bancaire ayant l'opération, `découvertAdmis` reste un compte bancaire, mais il ne se comporte pas uniquement comme un compte bancaire.

Dans le langage OS [Bre91], un type est défini par une spécification algébrique (`class spec`). Il y a distinction entre sous-typage et héritage, comme dans GSBL [CO88] ou NDL [PPP91]. Une classe est à la fois un type et un module. Le sous-typage est un raffinage de type. Il induit une inclusion de signature, d'ensembles support et des modèles. L'héritage est un raffinage de module. L'héritage implique uniquement une inclusion des modèles, à renommage près (raffinement de modules). Le sous-typage implique l'héritage. Cette distinction permet de décrire des hiérarchies de sous-types raffinées ensuite par héritage. En effet, le sous-typage est utilisé principalement dans OS ou GSBL ou encore PLUSS, pour compléter des spécifications (correspondant à des classes abstraites). Mais on ne voit pas trop l'intérêt de cette distinction puisque les deux relations sont utilisées tant pour la spécification que pour l'implantation. Et un problème est que la règle de substitution n'est pas applicable à l'héritage, même si une propriété de composition horizontale garantie l'indépendance vis-à-vis des serveurs.

A.5 Concurrence

Les modèles ci-dessus s'appliquent dans un contexte séquentiel. Ils sont étendus aux systèmes concurrents dans [BS88, EO94, Jul83, Kna94, Gue94, Huf89]. Certains incluent une notion de temps dans la spécification algébrique, d'autres définissent un système de transition paramétré par des valeurs, elle-même définies par des spécifications algébriques.

A.6 Un exemple de spécification algébrique en ASSPEGIQUE

Une spécification ASSPEGIQUE+ possible de l'hôpital est la suivante:

```
" Hospital est une file de Patient avec urgence "
" version simplifiee de Hospital compte tenu  "
" des proprietes sur les gardes "
spec: Hospital;
    use: Patient;
    sort: Hospital;
    generated by:
        " creation de l'Hospital "
        init   _ :          Entier                           -> Hospital;
        " admission normale d'un patient "
        admit _  _ :        Hospital Patient                 -> Hospital;
        " entree d'une urgence "
        urgency  _ _ :      Hospital Patient                 -> Hospital;

    operations:
        " soigne le premier "
        cure _ :            Hospital                         -> Hospital;

        " nombre d'admission bornee "
        size _ :             Hospital                        -> Entier;
        " nombre d'entree normales"
        number _ :           Hospital                        -> Entier;
        " nombre d'urgences non borne"
        urgencyNumber _ :   Hospital                         -> Entier;
        " nombre total d'entrees "
        totalNumber _ :     Hospital                         -> Entier;
        " designe celui qui est sera soigne "
        first _ :            Hospital                        -> Patient;

        " un essai "
        essai :                                              -> Hospital;

    predicates:
        " hopital vide "
        empty? _ :         Hospital;
        " normal "
        middle? _ :        Hospital;
        " normal et plein "
        full? _ :          Hospital;
        " inter et urgences "
        murgency? _ :      Hospital;
        " plein et urgences "
        furgency? _ :      Hospital;
        " les gardes complementaires et exclusives "
        " A + B = 1; D + F = A; C + G = 1; I + H = 1"
        gardeA _ : Hospital ;
        gardeB _ : Hospital ;
        gardeC _ : Hospital ;
        gardeD _ : Hospital ;
        gardeF _ : Hospital ;
        gardeG _ : Hospital ;
        gardeH _ : Hospital ;
        gardeI _ : Hospital ;

    preconditions:

        " size Self is-defined;"
        " number Self is-defined;"
        " urgencyNumber Self is-defined;"
        " totalNumber Self is-defined;"
        " urgency Self is-defined;"
        (max > 1) => init max is-defined;
```

```
    (empty? Self) or (middle? Self) or (murgency? Self)
                    => (admit Self p) is-defined;
    not (empty? Self) => (cure Self) is-defined;
    not (empty? Self) => (first Self) is-defined;

axioms:
    " axiomes des predicats d'etat "

E1: empty? (init max) is-true;
E2: empty? (admit Self p) is-false;
E3: empty? (urgency Self p) is-false;

M1: middle? (init max) is-false;
M2: middle? (admit Self p) =
        (empty? Self) or ((middle? Self) and (gardeH Self));
M3: middle? (urgency Self p) is-false;

full1: full? (init max) is-false;
full2: full? (admit Self p) = (middle? Self) and (gardeI Self);
full3: full? (urgency Self p) is-false;

ms1: murgency? (init max) is-false;
ms2: murgency? (admit Self p) = (murgency? Self) and (gardeH Self);
ms3: murgency? (urgency Self p) =
        (empty? Self) or (middle? Self) or (murgency? Self);

fs1: furgency? (init max) is-false;
fs2: furgency? (admit Self p) = (murgency? Self) and (gardeI Self);
fs3: furgency? (urgency Self p) = (full? Self) or (murgency? Self) ;

    " les gardes "
    " gardes de CURE "
    A :  gardeA Self = egal (urgencyNumber Self) 1;
    B : gardeB Self = (urgencyNumber Self) > 1;
    C : gardeC Self = egal (number Self) 1;
    D : gardeD Self = (gardeA Self) and (number Self) > 0;
    F : gardeF Self = (gardeA Self) and (egal (number Self) 0);
    G : gardeG Self = (number Self) > 1;
    H : gardeH Self = (number Self) < (p (size Self));
    I : gardeI Self = egal (number Self) (p (size Self));

    " axiomes des observateurs "
    s1 : size (init max) = max;
    s2 : size (admit Self p) = size Self;
    s3 : size (urgency Self p) = size Self;

    n1 : number (init max) = 0;
    n2 : number (admit Self p) = s (number Self);
    n3 : number (urgency Self p) = number Self;

    u1 : urgencyNumber (init max) = 0;
    u2 : urgencyNumber (admit Self p) =   urgencyNumber Self;
    u3 : urgencyNumber (urgency Self p) = s (urgencyNumber Self);

    t1 : totalNumber Self = (number Self) + (urgencyNumber Self);

    " first"
    f1 : empty? Self is-true => first (admit Self p) = p;
    f2+3 : middle? Self is-true => first (admit Self p) = first Self;
    f4+8 : murgency? Self is-true => first (admit Self p) = first Self;
    f5 : empty? Self is-true => first (urgency Self p) = p;
    f6 : middle? Self is-true => first (urgency Self p) = p;
    f7 : murgency? Self is-true => first (urgency Self p) = first Self;
    f9 : full? Self is-true => first (urgency Self p) = p;
    f10 : furgency? Self is-true => first (urgency Self p) = first Self;

    " axiomes des constructeurs - version Pascal"
```

```
        c1a1+1b1 : empty? Self is-true => cure (admit Self p) = Self;
        c1a2+1b2+2a1 : middle? Self is-true =>
                cure (admit Self p) = admit (cure Self) p;

        c3a1+3b1 :  full? Self is-true => cure (urgency Self p) = Self;
        c3a2+3b2 : furgency? Self is-true =>
                cure (urgency Self p) = urgency (cure Self) p;
        c3a3+3b3 : murgency? Self is-true & gardeI Self is-true =>
             cure (urgency Self p) = urgency (cure Self) p;

        c4a1+4b1+4c1 : empty? Self is-true => cure (urgency Self p) = Self;
        c4a2+4b2+4c2 : middle? Self is-true => cure (urgency Self p) = Self;
        c4a3+4b3+4c3 : murgency? Self is-true =>
              cure (urgency Self p) = urgency (cure Self) p;
        c4a4+4b4+4c4 : murgency? Self is-true & gardeH Self is-true =>
                     cure (admit Self p) = admit (cure Self) p;

        " essai "
        essai : essai =  admit (urgency (admit (init 3) toto) tata) tutu;

        where:
                Self : Hospital;
                max : Entier;
                p, q : Patient;
end Hospital
```

Voici une évaluation symbolique permettant de montrer un théorème équationnel :

```
expression :
cure(essai)

 L'expression a evaluer est ...

cure_
  |
essai
    [ essai -->> admit (urgency (admit (init 3) toto) tata) tutu ] .

Cette regle nous permet d'obtenir :

    cure_
      |
  _admit___
 /         \
     urgency__    tutu
     /        \
  admit__    tata
  /      \
init_   toto
  |
  3

    [ 3 -->> s (s (s 0)) ] .

Cette regle nous permet d'obtenir :

    cure_
      |
  _admit___
 /         \
     urgency__    tutu
     /        \
  admit__    tata
  /      \
init_   toto
  |
  s_
  |
  s_
```

```
  |
  s_
  |
  0

 On essaie d'appliquer la regle Hospital.spec-c1a1+1b1-1
 SI empty? Self = true
 ALORS  [ cure (admit Self p) -->> Self ] .

 Resultat de l'evaluation des premisses:
 Les premisses ne sont pas verifiees, la regle ne s'applique pas

 On essaie d'appliquer la regle Hospital.spec-c1a2+1b2+2a1-1
 SI middle? Self = true
 ALORS  [ cure (admit Self p) -->> admit (cure Self) p ] .

 Resultat de l'evaluation des premisses:
 Les premisses ne sont pas verifiees, la regle ne s'applique pas

 On essaie d'appliquer la regle Hospital.spec-c4a4+4b4+4c4-1
 SI murgency? Self = true  &  gardeH Self = true
 ALORS  [ cure (admit Self p) -->> admit (cure Self) p ] .

 Resultat de l'evaluation des premisses:
 Les premisses sont verifiees, la regle va etre appliquee dans l'etape suivante

Cette regle nous permet d'obtenir :

  admit__
 /       \
      cure_    tutu
 |
    urgency__
    /       \
  admit__   tata
  /      \
init_   toto
  |
  s_
  |
  s_
  |
  s_
  |
  0

 On essaie d'appliquer la regle Hospital.spec-c3a1+3b1-1
 SI full? Self = true
 ALORS  [ cure (urgency Self p) -->> Self ] .

 Resultat de l'evaluation des premisses:
 Les premisses ne sont pas verifiees, la regle ne s'applique pas

 On essaie d'appliquer la regle Hospital.spec-c3a2+3b2-1
 SI furgency? Self = true
 ALORS  [ cure (urgency Self p) -->> urgency (cure Self) p ] .

 Resultat de l'evaluation des premisses:
 Les premisses ne sont pas verifiees, la regle ne s'applique pas

 On essaie d'appliquer la regle Hospital.spec-c3a3+3b3-1
 SI murgency? Self = true  &  gardeI Self = true
 ALORS  [ cure (urgency Self p) -->> urgency (cure Self) p ] .

 Resultat de l'evaluation des premisses:
 Les premisses ne sont pas verifiees, la regle ne s'applique pas
```

```
 On essaie d'appliquer la regle Hospital.spec-c4a1+4b1+4c1-1
 SI empty? Self = true
 ALORS  [ cure (urgency Self p) -->> Self ] .

 Resultat de l'evaluation des premisses:
 Les premisses ne sont pas verifiees, la regle ne s'applique pas

 On essaie d'appliquer la regle Hospital.spec-c4a2+4b2+4c2-1
 SI middle? Self = true
 ALORS  [ cure (urgency Self p) -->> Self ] .

 Resultat de l'evaluation des premisses:
 Les premisses sont verifiees, la regle va etre appliquee dans l'etape suivante

Cette regle nous permet d'obtenir :

      admit___
     /        \
  admit__    tutu
  /      \
init_   toto
  |
  s_
  |
  s_
  |
  s_
  |
  0

admit (admit (init (s (s (s 0)))) toto) tutu
expression :
```

Annexe B

Un environnement de développement

"L'expérience est un terrible maître d'ecole : elle vous fait passer l'examen d'abord et elle vous apprend la leçon ensuite."
Rivarol.

L'atelier ASFO, **A**telier de **S**pécification **F**ormelle et conception par **O**bjets, doit permettre de construire les différents modèles en assurant la prise en compte des tâches automatisables, la vérification de cohérence, et des facilités de réutilisation. Les principales fonctionnalités sont :

- fonctions d'interfaçage de type manipulation et visualisation de graphes.
- fonctions d'éditions et de manipulation de spécifications algébriques équationnelles.
- fonctions de preuves et validations de spécifications TAG et CF.
- fonctions d'aide à la transformation de TAGs en CFs.
- fonctions de traduction (semi-)automatique dans un langage cible à objets,
- fonctions de gestion de bibliothèque de TAGs et de CFs.
- fonctions d'interfaçage avec d'autres environnements de spécifications formelles.

L'interface générale a été réalisée par des étudiants du DESS de Génie Informatique [BBG$^+$94]. Cette interface graphique permet de manipuler des ensembles de TAG ou de CF. Une priorité est donnée dans ce projet à la manipulation interactive de composant en visualisant les liens entre ces composants. Nous nous sommes plutôt intéressés aux modèles eux-mêmes et nos travaux ne sont pas interfacés avec ceux cités ci-dessus. Les travaux que nous avons réalisé concernent l'édition des composants et les interactions entre niveaux d'abstraction :

- gestion simplifiée d'une bibliothèque de TAG, de CF,
- édition textuelle ou graphique des TAG (analyse syntaxique et sémantique),
- édition textuelle ou graphique des classes formelles (analyse syntaxique et sémantique),
- extraction d'une spécification algébrique plate des TAG,
- passage des TAG aux CF,
- traduction des CF en plusieurs langages de programmation,
- simplificateur d'expressions booléennes,
- contrôle et preuve sur les modèles (automates, contrôle de type, évaluateur symbolique
- de TAG ou de CF, simulation...).

Ce chapitre décrit ainsi quelques outils d'édition des modèles développés en Smalltalk version 4.0.

B.1 Syntaxe des langages

Notations BNF

Pour simplifier, nous donnons les grammaires, auxquelles on a ôté les actions sémantiques.

Les règles sont de type `left = right`. Les symboles non terminaux sont des suites de caractères. Les symboles terminaux sont soit des chaines de caractères (symboles au sens Smalltalk) préfixées par un `#` soit des caractères préfixés par un `$`. Un certain nombre de tokens sont déterminés par le parser Smalltalk (number, string (`".."`), word, etc.). Le signe `@` devant une règle indique que qu'un retour est possible dans l'évaluation en cas d'erreur. Comme dans les conventions usuelles, `|` désigne l'alternative, `*` l'itération positive ou nulle, `+` l'itération positive, `\$,` l'itération positive avec le caractère “,” comme séparateur.

B.2 Grammaire des expressions

```
constant =
    string
expr =
    union
    (
    @$=$> expr
    |
    exprNext
    )
exprNext =
    (
        ($< (
            $=
            |
            $>
            |
            )
        | $> (
            $=
            |
            )
        | $= (
            )
        )
        union
    |
        #in
        $[
        expr
        $,
        expr
        $]
    |
    )
factor =
    primitiveExpression
    (
        (
        $*
        |
        $/
        |
        $\$\
        |
        #AND
        )
        factor
    |
    )
functionCall =
    word
    $(
        (expr $,)
    $)
ident =
    word
primitiveExpression =
    (
    @functionCall
    |
    primitiveTerm
    |
    $- union
    |
    #not expr
    |
    $( expr $)
    )
primitiveTerm =
    (
    #true
    |
    #false
    |
    number
    |
    constant
    |
    ident
    )
union =
    factor
    (
        (
        $+
        |
        $-
        |
        #OR
        )
```

```
          union
     |
     )
comment =
     string
comments =
     (comment)*
point =
     number $@ number
predicate =
     expr
type =
     typeName
typedVariable =
     word
     $:
     type
typedVariables =
     ((word)\$,) $: type
typeName =
     word
types =
     (type*)
```

Grammaire des TAGs

```
axiomSpec =
    #TAG
         description
         genericParameters
         signature
         constraints
         automata
         properties
    #FINTAG
$.
description =
    #sort
         $: word
    #extended #name
         $: (word*)
    #authors
         $: (word*)
    #creation #date
         $: (number word number)
    #release #date
         $: (number word number)
    #release #number
         $: (number)
    #comments
         $: comments
genericParameters =
    #generic #types
         $: (((typeWithOperations) \$,)
         | )
         $.
    #generic #constants
         $: (((typedVariable) \$,)
         |)
         $.
typeWithOperations =
    type
    ($\{ ( operationProfile \$, ) $\} | )
signature =
    #operations
         $: ( operationProfile \$, )
         $.
operationProfile =
    word
    ($( (typedVariables \$,) $) | )
         $: type
constraints =
    #constraints
         $:
             (( guard ) \$,
             | )
         $.
guard =
    (word) $: predicate
automata =
    #fsm
         $:
         #states $:
             $\{ ( state \$,) $\}
         #initial #states $:
             $\{ ( (word) \$,) $\}
         #final #states $:
             $\{ ( ( (word ) \$,)) $\}
         guards
         #transitions $:
             (( $< transition $> ) \$,
             |)
         $.
guards =
    #predicates
         $:
             (( guard ) \$,
             | )
         $.
state =
    word
    ($[ word* $] | )
    ($( point $& number $) | )
transition =
    (word | ) $,
    (word | ) $,
    (word ) $,
    ((#not | ) word | )
         ($[ (signedNumber | )
             $, (signedNumber | ) $] | )
properties =
    #properties
         $: ((((word
             $: (string)* )
             )\$,)
         | )
```

```
        $.
```

Grammaire des CFs

```
axiomSpec =
     #CF
          header
          lines
          aspects
          lines
          secondaryM
          lines
          classM
     #FINCF
$.
header =
     fullName
     #instance #of
          (type) #inherits #from
          (types)
     #features
          $: features
     #comments
          $: comments
     (
     extendedHeader
     |
     )
fullName =
     word
($[ ((type) \$,) $] | )
     (#ABSTRACT | )
features =
          (((word )\$,) | )

lines =
     ($_)*
aspects =
          (((aspect $.)*) |)
aspect =
     #aspect
          $:  word
     #abstract #structure
          #fields $- #types $- #default
          fields
     #constraint $:
          constraint
fields =
     (@(field \$, ) | )
field =
     (
     (term $=$=$>)
     |
     )
     typedVariable
     ( $( primitiveTerm $) | )
constraint =
     (predicate | )
secondaryM =
          #secondary #methods
          (((secondaryMethodDef $.)*) | )
secondaryMethodDef =
     methodDef
methodDef =
     $;$; word
              $: comment
          word
              $: type* $-$-$> type
     (#requires $: predicate | )
     (#ABSTRACT | (axiom \$,) )

axiom =
     (@(equation \$^ $=$=$> | )
     equation

equation =
     term $=$= term
term =
     $[ expr $]
classM =
          #class #methods
          (((classMethodDef $.)* ) | )
classMethodDef =
     methodDef
```

B.3 Présentation rapide des éditeurs

Un éditeur permet de manipuler des spécifications avec les fonctions habituelles de manipulation de texte (*again/undo/copy/cut/paste/accept/cancel*) et des manipulations d'entrée/sortie avec les fichiers.

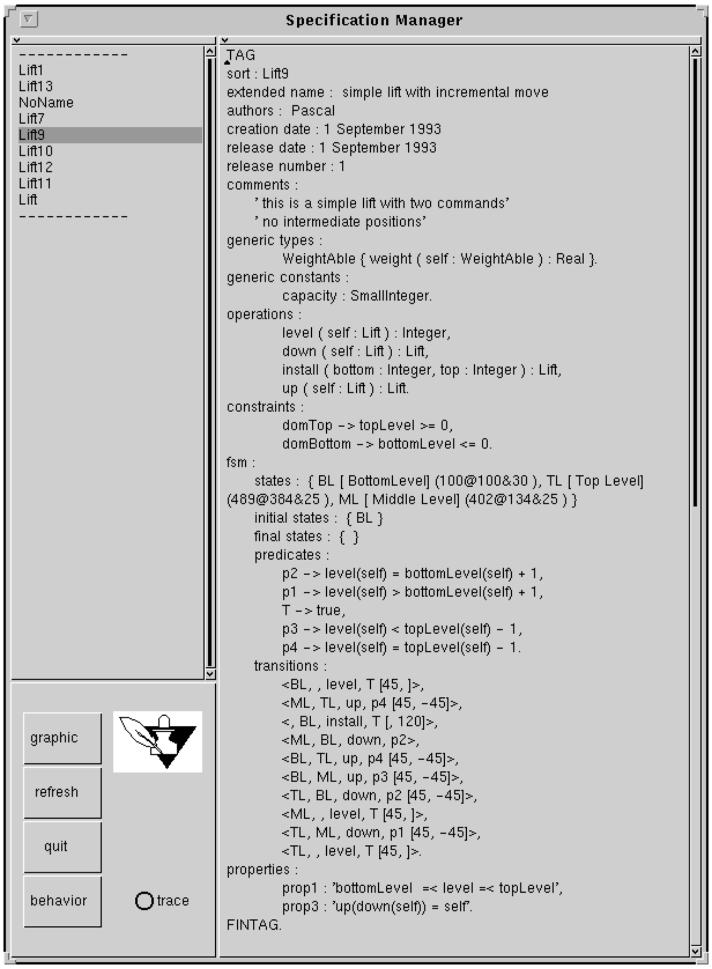


Figure 69 : Editeur textuel de TAG et gestion de la bibliothèque de TAGs

Le menu d'une liste est relativement standard :

- si aucune spécification n'existe

  - *inspect* : inspection Smalltalk du dictonnaire de spécification.
  - *new specification* : modèle vide.
  - *fileIn* : chargement d'une spécification.
  - *load* : chargement d'un ensemble de spécification à partir d'un fichier binaire.
  - *refresh* : raffraîchir la vue, surtout si d'autres vues en dépendent.

- si aucune spécification n'est sélectionnée

  - *inspect* : inspecter la liste de spécifications.
  - *new specification* : définir une nouvelle spécification.

- *fileIn*: lire une spécification à partir d'un fichier texte.
- *load*: charger la présente liste à partir d'un fichier binaire.
- *save*: sauver la présente liste dans un fichier binaire.
- *refresh*: raffraîchir la vue, surtout si d'autres vues en dépendent.
- *remove*: choisir puir supprimer une spécification de la liste.
- *removeAll*: supprimer toutes les spécifications de la liste.

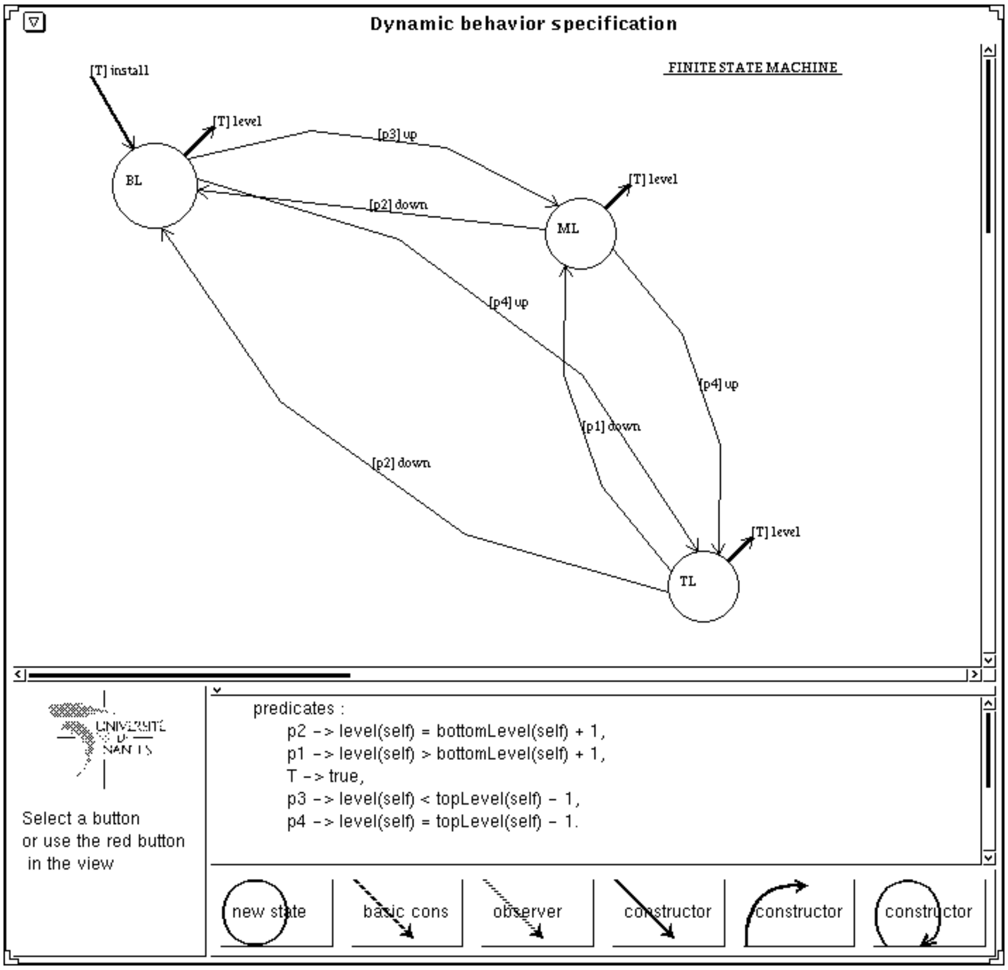


Figure 70 : Editeur graphique de comportement dynamique

- si une spécification n'est sélectionnée

- *inspect*: inspecter la spécification courante.
- *remove*: supprimer la spécification courante de la liste.
- *rename*: renommer la spécification courante.
- *save*: sauver la spécification courante sur fichier binaire.
- *fileOut*: sauver la spécification courante sur fichier texte.
- *new specification*: définir une nouvelle spécification.
- *fileIn*: lire une spécification courante sur fichier texte.
- *load*: lire une spécification courante sur fichier binaire.
- *refresh*: raffraîchir la vue, surtout si d'autres vues en dépendent.
- *removeAll*: supprimer toutes les spécifications de la liste.

Pour un éditeur de bibliothèque (figure 69 et figure 72), on a les options suivantes, accessibles par des boutons :

- *trace* : mettre l'option trace *On/OFF*, c'est-à-dire la visualisation des informations de l'analyseur dans la fenêtre `Transcript`.
- *refresh* : raffraîchir la vue.
- *graphic* : ouvrir l'éditeur graphique associé (figure 71 et figure 73).

quit : quitter l'éditeur.

l'option *behavior* permet d'ouvrir l'éditeur graphique de comportements dynamiques (partie du TAG) : voir la figure 70.

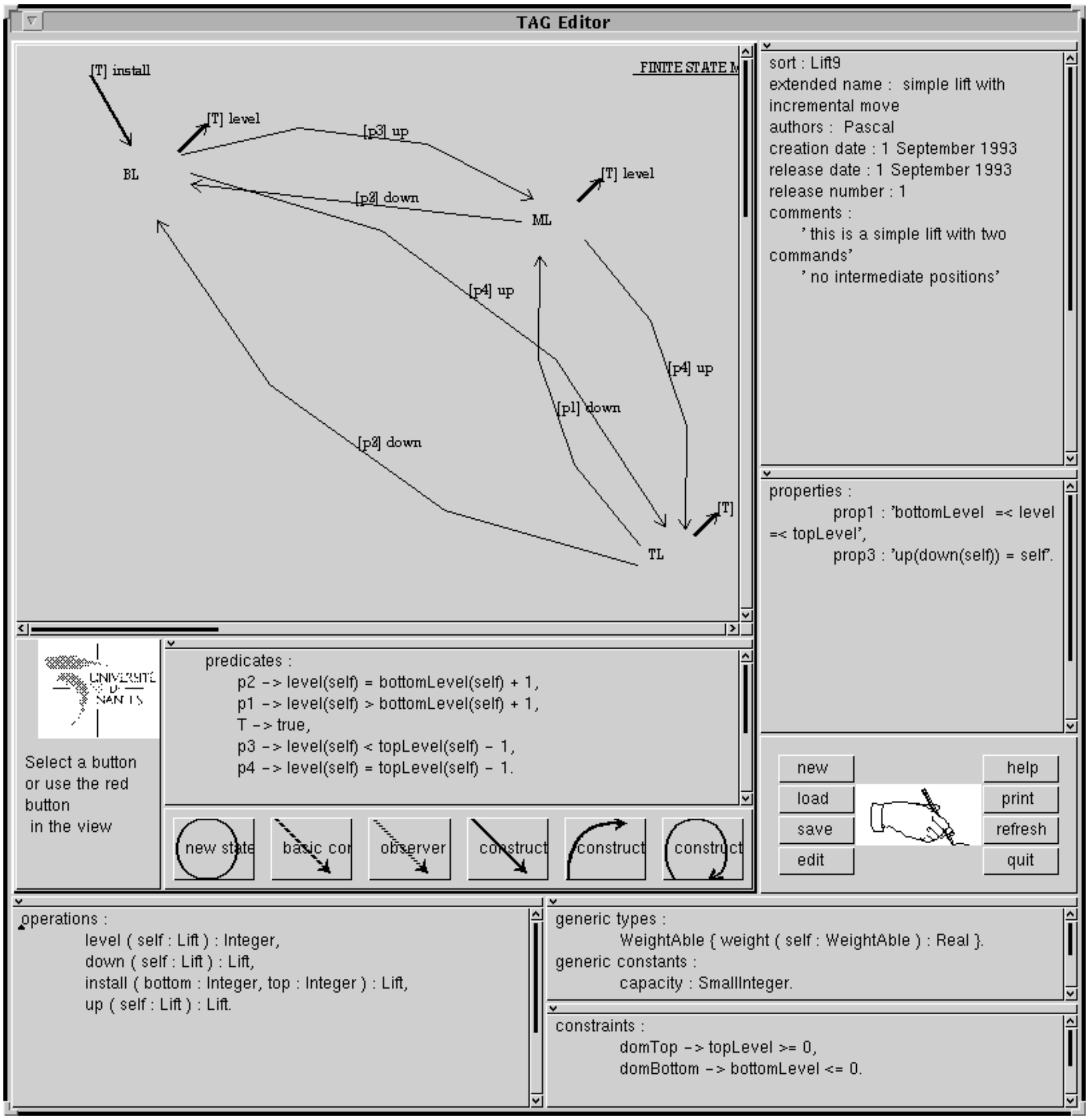


Figure 71 : Editeur graphique de TAG

Les boutons de créer de nouveaux états ou transitions :

- *new state* : ajoute un nouvel état dans l'automate (demande le nom),
- *basic cons*tructor : ajoute un constructeur de base (sélection de l'état d'arrivée),

- *observer*: ajoute un observateur (sélection de l'état où l'observateur est défini),
- *constructor* (direct) : ajoute un constructeur (sélectionne les deux états) vecteur,
- *constructor* (angular) : ajoute un constructeur (sélectionne les deux états states) arc,
- *constructor* (ring) : ajoute un constructeur (sélectionne l'état où le constructeur est défini) boucle.

La vue *predicates* affiche les prédicats liés aux transitions. Le menu de la vue graphique permet quatre types de manipulation :
• états (d'abord appel de l'option puis sélection de l'état)

- *move* : déplacer l'état,
- *grow*: agrandir le cercle représentant l'état,
- *lessen* : diminuer le cercle représentant l'état,
- *change name* : changer le nom de l'état,
- *wording*: *show/change* : afficher ou modifier le nom étendu de l'état,
- *delete* : détruire l'état.

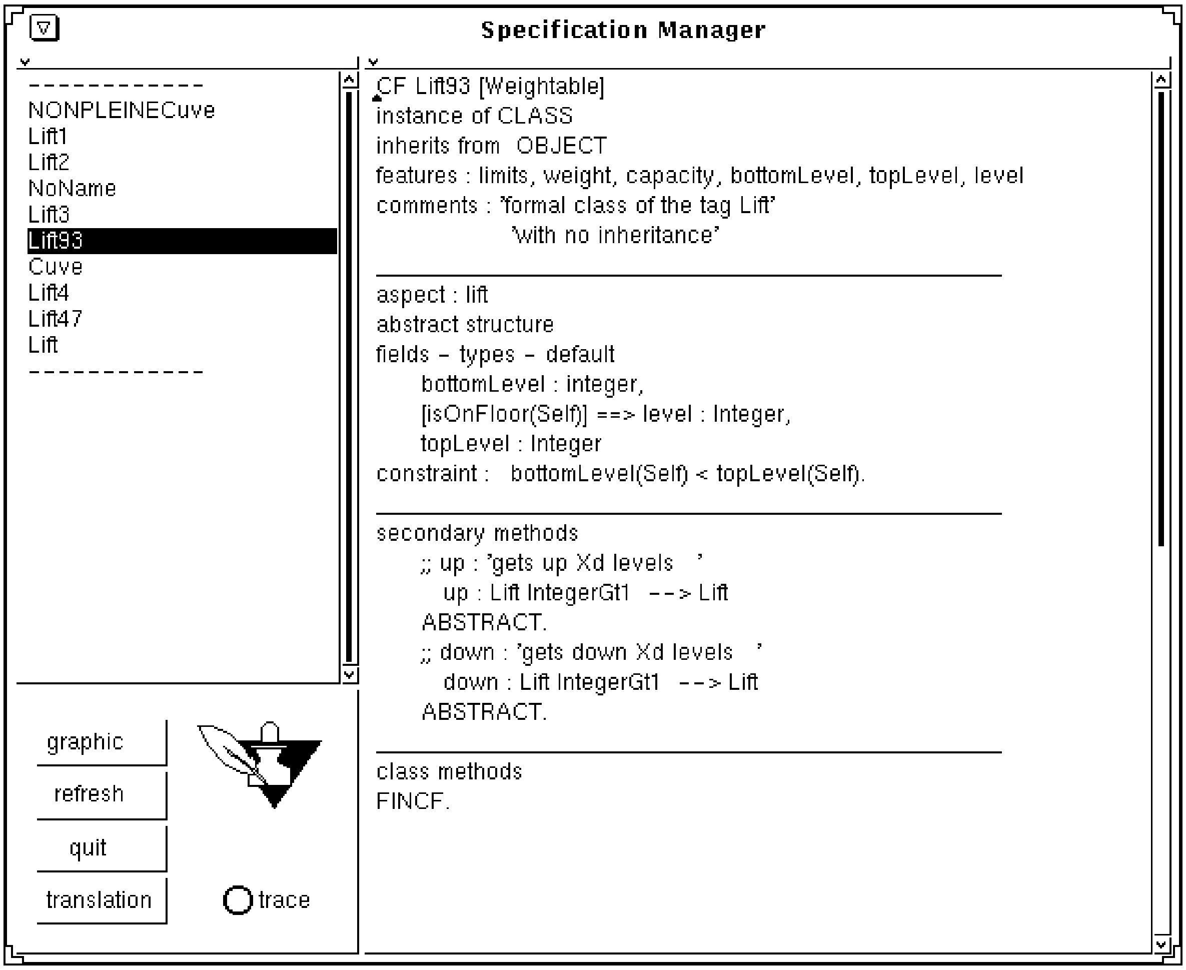


Figure 72 : Editeur textuel de classes formelles et gestion de la bibliothèque de CFs

• transitions (d'abord appel de l'option puis sélection d'un des états de la transition pour sélectionner la transition)

- *change origin* : modifier l'origine,

- *change destination*: modifier la destination,
- *change angles*: modifier les angles de la transition,
- *exchange states*: inverser l'état initial et l'état final,
- *label*: *change label/change guard*: changer le label (ici interprété comme un nom d'opération) ou la garde associée,
- *delete*: détruire la transition.

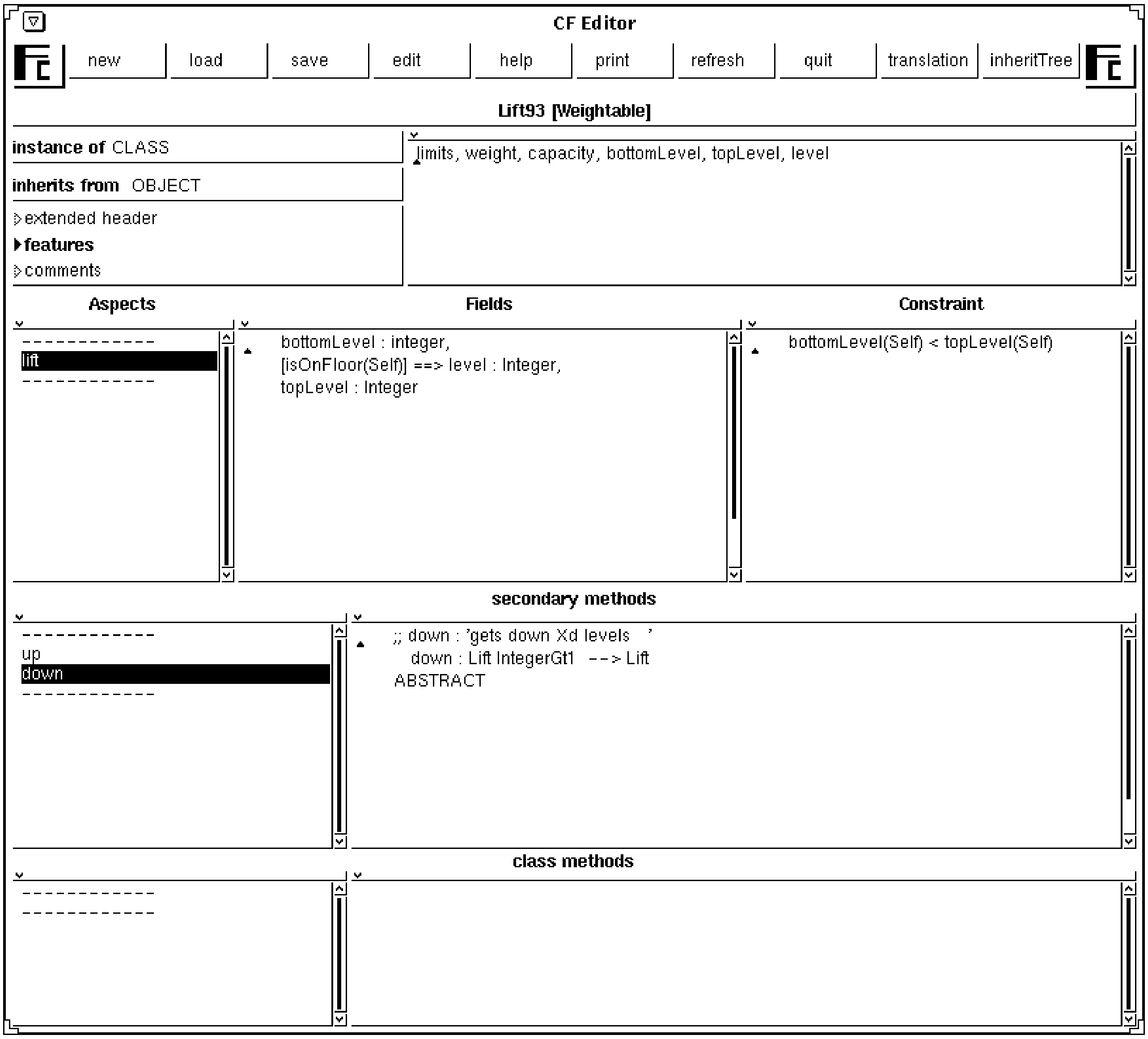


Figure 73 : Editeur graphique de classes formelles

- fichiers

  - *new*: nouvel automate,
  - *load*: lecture de l'automate à partir d'un fichier finaire,
  - *save*: sauvegarde de l'automate dans un fichier finaire,
  - *prePrint*: préparation d'impression postscript.

- vue spécification

- *inspect*: inspecter l'objet comportement dynamique,
- *refresh*: raffraîchir la vue,
- *quit*: quitter léditeur de comportement.

Le bouton *edit* permet de sélectionner dans la bibliothèque une spécification à éditer. Les autres boutons correspondent sensiblement aux options des menus des éditeurs textuels.

L'option *translation* permet d'ouvrir l'application de traduction des classes formelles dans un quelconque langage de programmation.

Les vues *name* (Lift93), *instance of*, *inherits from* sont des vues texte modifiables par appel du menu. La vue à droite des boutons *extended header*, *features*, *comments* est une vue texte classique, dont le contenu est fixé par le bouton actif de cette triade. Une fois de plus, les textes acceptés sont compilés par l'analyseur spécialisé et modifie l'objet Smalltalk correspondant à la spécification en cours d'édition.

La vue suivante représente les aspects de la classe formelle. Chaque aspect est considéré comme une spécification ayant deux parties : les champs *fields* et la contrainte. Consulter les exemples pour voir la syntaxe acceptée.

Les deux vues suivantes sont des éditeurs de méthodes. Le principe des éditeurs d'un ensemble de spécifications et les menus associés sont repris ici. Voir les explications ci-dessus.

L'option *translation* permet d'ouvrir l'application de traduction des classes formelles dans un quelconque langage de programmation. L'option *print* permet de donner la description LaTeXde la classe formelle. L'option *inheritTree* permet d'examiner les superclasses de la classe formelle en cours d'édition.

B.4 Architecture de l'application

Un effort a été founi pour documenter les classes et méthodes de l'application. Un certain nombres d'erreurs s'y sont sans doute glissées, dues à l'usage intensif de la fonction *couper/coller*. Un soucis de généralité nous a conduit à décrire mon application dans une triade `TC - TAG - CF`, où `TC` représente les caractéristiques ou objets communs aux deux modèles ou à leur manipulation (`TAG`CF).

On distingue trois grand types de manipulation, les éditeurs des spécification, les vues associées et les compilateurs associés, définis dans la section B.4. L'architecture choisie favoris l'indépendance entre les objets et le MVC qui les manipule, c'est pourquoi nous utilisons des éditeurs pour interfacer objet/compilateurs/vue d'édition.

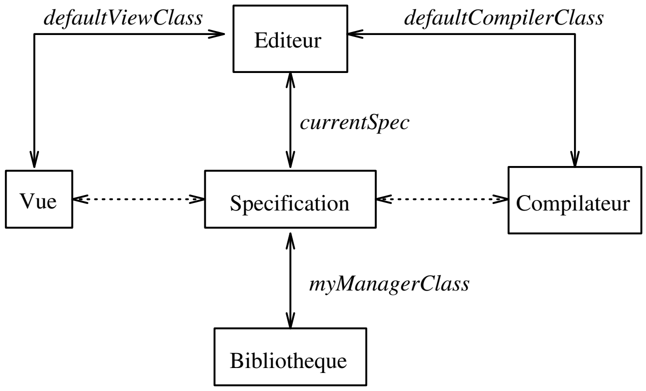


Figure 74 : Liens entre les différents composants du système d'édition

Chaque spécification est un objet smalltalk composé d'autres objets. A chacun de ces objets, correspond grossièrement une règle dans la grammaire du compilateur associé. Il y a une surcharge volontaire du terme spécification qui représente en fait tout objet nommé et auquel correspond un triplet `<règle de grammaire, compilateur, objet Smalltalk>`.

Les hiérarchies des sections suivantes ont été réalisées par réutilisation et adaptation d'un triplet MVC, de visualisation d'une hiérarchie de classes Smalltalk, aimablement mis à notre disposition par Nadir Yousfi.

Les outils support

La hiérarchie suivante représente les modèles des éditeurs de spécifications. On distingue deux sortes d'éditeurs :

- les éditeurs portant sur une spécification seule `TCEditor`,
- les éditeurs portant sur un ensemble de spécifications `TCAbstractListModel`.

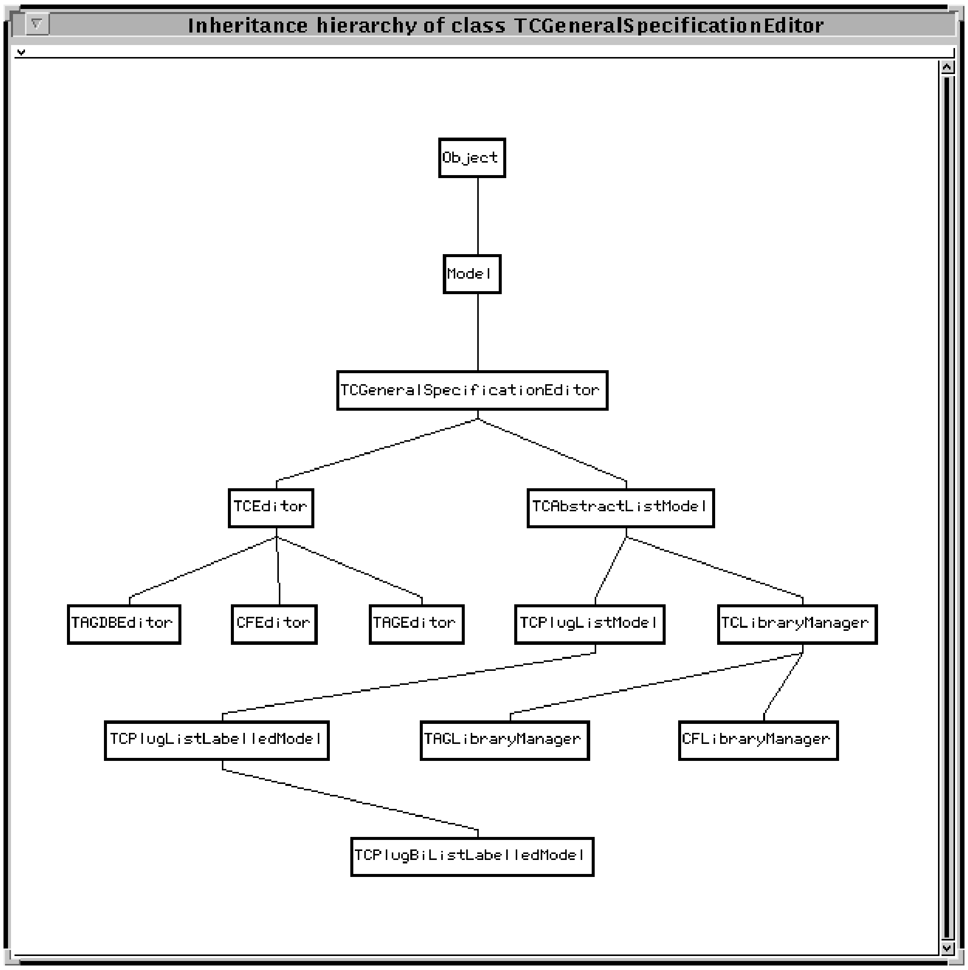


Figure 75 : Editeurs de spécifications

La hiérarchie suivante représente les vues associées aux modèles des éditeurs de spécifications de la figure 75.

```
TCGeneralSpecificationEditor ('analyzer' 'textMenu')
        TCAbstractListModel ('currentKey')
            TCLibraryManager ('currentGraphicEditor')
```

```
                CFLibraryManager ()
                TAGLibraryManager ('currentBehaviorEditor')
            TCPlugListModel ('specClass' 'suffixe' 'analyseAxiom' 'model' 'dicoMsg')
                TCPlugListLabelledModel ('label')
                    TCPlugBiListLabelledModel ('analyseAxiom2' 'label2' 'label3')
        TCEditor ('currentSpec')
            CFEditor ('currentHeaderNext')
            TAGDBEditor ('activeTool' 'stateKey' 'transitKey' 'helpMessage' 'labels')
            TAGEditor ('fsmEditor')
```

Les autres outils supports concernent l'interface (figure 76) et le compilateur générique (figure 77).

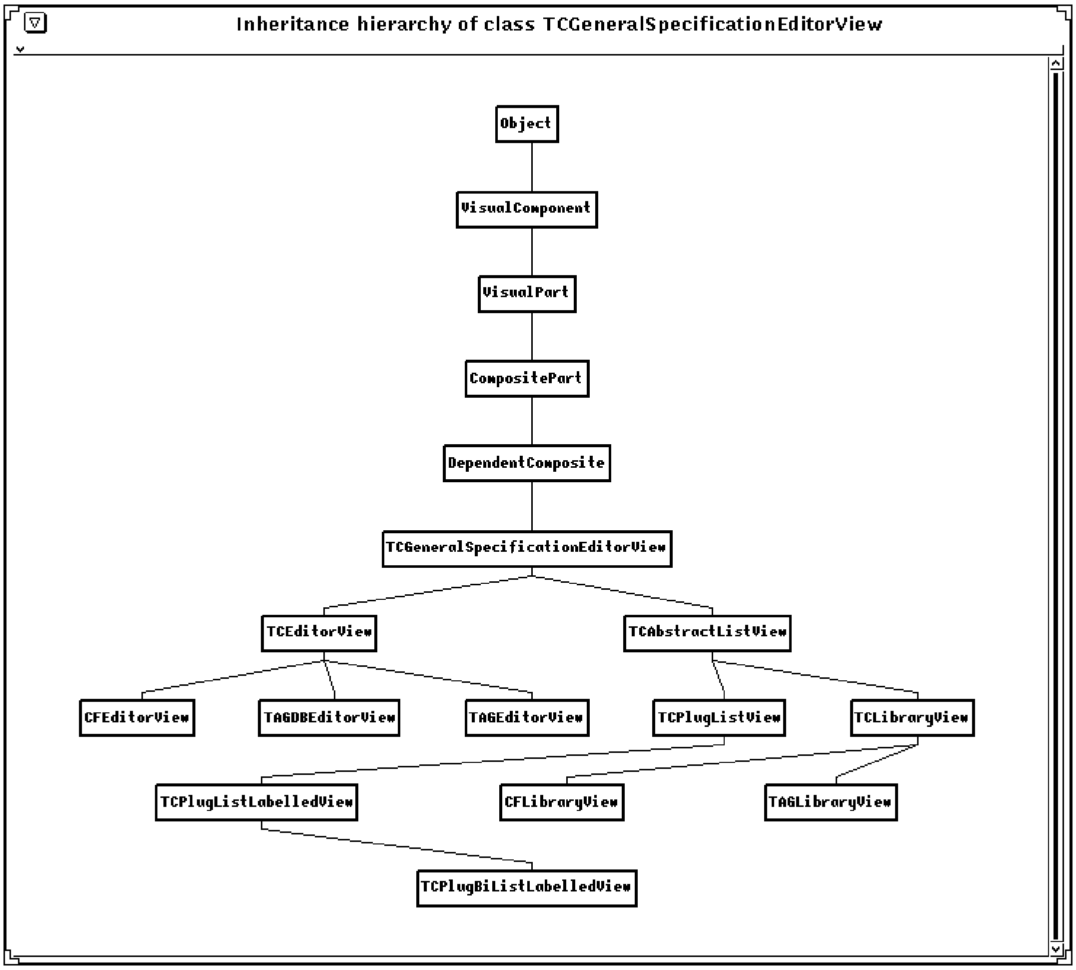


Figure 76 : Vues associées aux éditeurs de spécifications

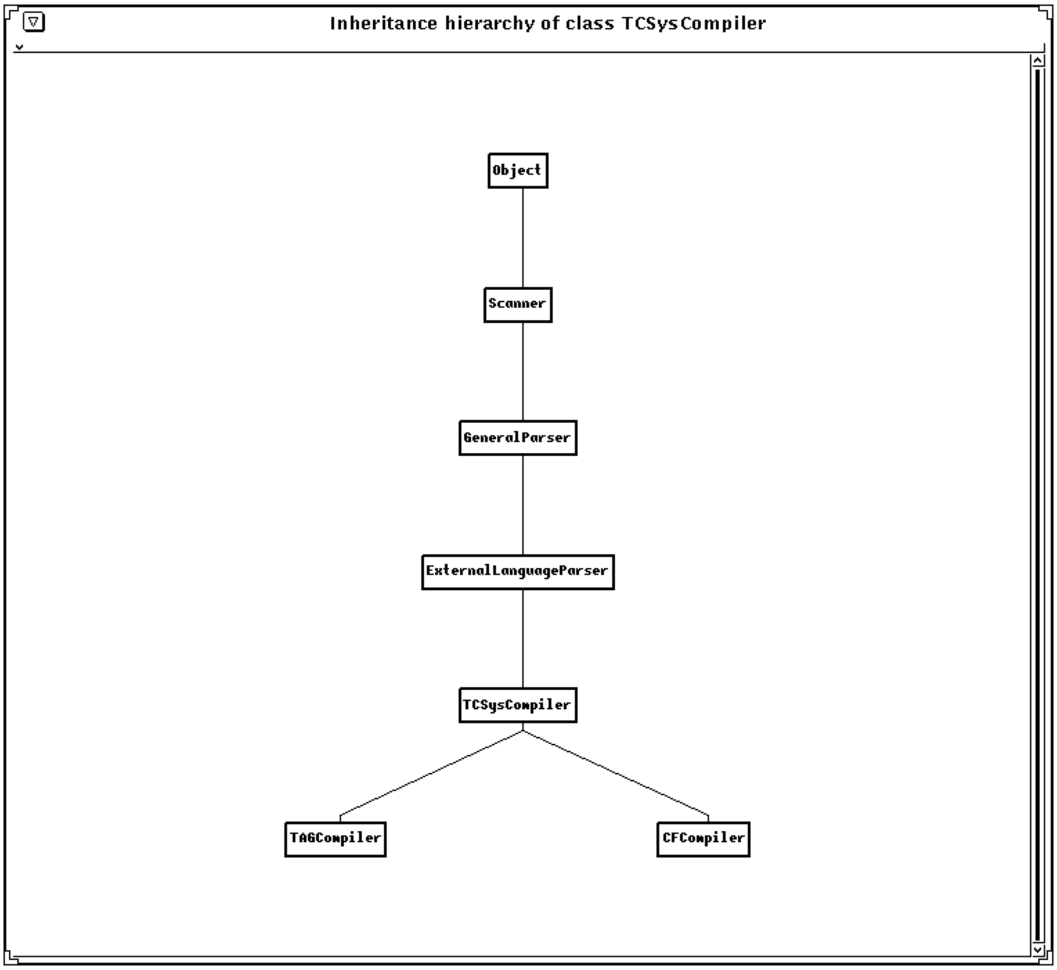


Figure 77 : Compilateurs de spécifications

Autres classes support

– modèle MVC (*@Copyright P. Andre 1994*)

1. `TCButtonView`: This class implements buttons, that are active when the mouse is down. This is defined to facilitate a button creation (`LabeledBooleanView`).
2. `TCTitleView`: This class implements a view on an elementary text (`View`).
3. `TCTitleMView`: This class implements a view with menu on an elementary text (`TCTitleView`) see `TCTitleView`.
4. `TCScrollingWrapper`: This class implements scrolling wrappers for large subview... The current ScrollingWrapper is blocked by the system Screen to 1152@900 pixel (primitive getScreenDimensions #(#(0 0 1152 900)) in class Screen) (`ScrollingWrapper`).
5. `TCDialogView`: This class implements additional immediate dialogs (`DialogView`).
6. `TCButtonCollectionView`: This class implements structured picture where components are buttons. This is defined to facilitate a button handling (`CompositePart`).
7. `TCTitleMController`: This class implements a a pluggable control with menu on an elementary text (`ControllerWithMenu`) see `TCTitleMView`.

– divers

1. `TCSysExamples`,`TAGExamples`,`CFExamples`: This class implements TC (TAG, CF) specifications that are Smalltalk methods. (`Object`).

2. `TCSet` : This class implements arithmetic sets (inter, union, diff). (`Set`).
3. `TCDictionary` : This class describes dictionaries of TCNamedObjects. No doubles are accepted in the new protocol (RedundancyErrorSignal raised). Rewring is possible throught the super protocol. (`Dictionary`).
4. `TCSysSemanticError`: This class implements a visual display for semantic errors. (`SyntaxError`).
5. `TCExternalReadWriteStream`: This class implements read/write streams that allow skip and append writing. Method flush is redefined in order to keep alive the write limit. (`ExternalReadWriteStream`).
6. `TCIcons` : This class implements graphic icon constants. (`CachedImage`).

Les objets

```
Object ()
  Model ('dependents')

    TCObject ()
      CFObject ()
        CFAxiom ('condition' 'equation')
        CFEquation ('term1' 'term2')
        CFHeader ('name' 'generics' 'isAbstract' 'isa' 'ako' 'comments' 'features')
          CFFullHeader ('tagName' 'extendedName' 'authors' 'creationDate' 'releaseDate'
              'releaseNumber')
        CFTerm ('expr')
          CFPredicate ()
      TAGObject ()
        TAGDescription ('sort' 'extendedName' 'authors' 'creationDate' 'releaseDate'
              'releaseNumber' 'comments')
        TAGProperty ('name' 'comments')
      TCExpression ('type')
        TCFunctionCall ('functionName' 'param')
        TCParenthesedExpression ('expr')
        TCUnaryExpression ('operator' 'expr')
          TCBinaryExpression ('arg')
        TCValuedExpression ('value' 'name')
          TCConstant ()
          TCVariable ()
      TCNamedObject ('name')
        CFNamedObject ()
          CFAspect ('fields' 'constraint')
        TAGNamedObject ()
          TAGPredicate ('value')
          TAGState ('wording')
            TAGDBState ('center' 'radius')
            TAGSuperState ('states')
              TAGDBSuperState ('polygon')
          TAGTransition ('label' 'signe' 'guard' 'init' 'final')
            TAGDBTransition ('initPoint' 'finalPoint')
              TAGDBDirectLink ()
              TAGDBOrientedLink ('angori' 'angdest')
                TAGDBEdge ()
                TAGDBInTransition ()
                TAGDBOutTransition ()
                TAGDBRing ()
          TAGType ('operations')
```

```
            TAGTypeParam ()
        TCTypedObject ('type')
            TAGGenConstant ()
            TCFunction ('param' 'result')
                CFMethod ('condition')
                    CFAxiomaticMethod ('comment' 'axioms')
                        CFClassMethod ()
                        CFField ('default')
                TAGOperation ()
            TCTypedVariable ()
    TCSpec ()
        CFSpec ('header' 'aspects' 'secondary' 'classMethods')
        TAGFSM ('states' 'initialStates' 'finalStates' 'guards' 'labels' 'transitions')
        TAGSpec ('description' 'typeParam' 'genericConstants' 'signature' 'constraints'
                'behavior' 'properties')
```

B.5 Perspectives

Il reste encore beaucoup de codage à réaliser. Des outils doivent être ajoutés. Certains sont en cours, surtout concernant les contrôles, vérifications de validité et preuve tandis que d'autres sont à concevoir :

- extraction d'une spécification algébrique plate des TAG,
- passage des TAG aux CF,
- traduction des CF en d'autres langages de programmation (C++, Pascal Objet...),
- simplificateur d'expressions booléennes.
- contrôle et preuve sur les modèles (automates, contrôle de type, évaluateur symbolique de TAG ou de CF, simulation...) à réaliser.

Par ailleurs, l'aspect méthode de conception doit être approfondi et la convivialité de l'atelier doit être confirmée par différents utilisateurs. Un travail théorique sur la gestion de classe est prévu.

Annexe C

Un exemple plus complexe : l'ascenseur

C.1 Spécification informelle

Une fois installé, un ascenseur peut monter et descendre sauf quand il est en panne. Un ascenseur en déplacement peut être stoppé puis repartir. A l'arrêt, l'ascenseur surchargé ne pourra se mettre en mouvement. On ne s'occupe pas de la gestion concurrente des demandes de déplacement, ni des contraintes temporelles, telles que les délais avant fermeture des portes, durées de trajet. On ne s'occupe pas non plus de l'optimisation des déplacements. La spécification d'un contrôleur plus complet avec politique de gestion de file d'attente prioritaire est possible par extension.

C.2 Spécification TAG

behavior description through a finite state machine

(see figure Lift Behavior)

install (bottom: Integer, top: Integer, cap: RealGt1) : Lift
down (Self : Lift, d : IntegerGt1) : Lift
up (Self : Lift, d : IntegerGt1) : Lift
level (Self : Lift) : Integer
topLevel (Self : Lift) : Integer
bottomLevel (Self : Lift) : Integer
limits (Self : Lift) : Integer x Integer
capacity (Self : Lift) : RealGt1
chgCapacity (Self : Lift, cap : RealGt1) : Lift
getIn (Self : Lift, w : Weightable) : Lift
getOut (Self : Lift, w : Weightable) : Lift
weight (Self : Lift) : Real
arrived (Self : Lift) : Lift
stop (Self : Lift) : Lift
restart (Self : Lift) : Lift
getOOO (Self : Lift) : Lift
repaired (Self : Lift) : Lift
isIn (Self : Lift, w : Weightable) : Boolean

Type : Lift
Description : shifts people, animals or luggage
Author : <Andre, univ nantes>
<Febrary 1994>
Release : 1.6
Properties : destination level not changed

constants:
bottom, top

formal parameters:
Weightable [weight (Self: Weightable): Real]

constraints:
bottomLevel(Self) <= level(Self) <= topLevel(Self)
bottomLevel(Self) < topLevel(Self)
0 <= weight(Self)

Figure 78 : Profils du TAG `Lift`

La figure 79 propose un comportement dynamique pour l'ascenseur, où `ML` (resp. `BL`, `TL`, $\mathtt{S}_i$, `MMU`, `MMD`, `MB`, `BM`, `MT`, `TM`, `TB`, `BT`, `BO`, `MO`, `TO`, `OOO`) est l'abbréviation de *MiddleLevel* (resp. *BottomLevel*, *TopLevel*, *Stopped*$_i$, *MiddletoMiddleUp*, *MiddletoMiddleDown*, *MiddletoBottom*, *BottomtoMiddle*, *MiddletoTop*, *ToptoMiddle*, *ToptoBottom*, *BottomtoTop*, *BottomOverload*, *Midd-*

leOverload, TopOverload, OutOfOrder)

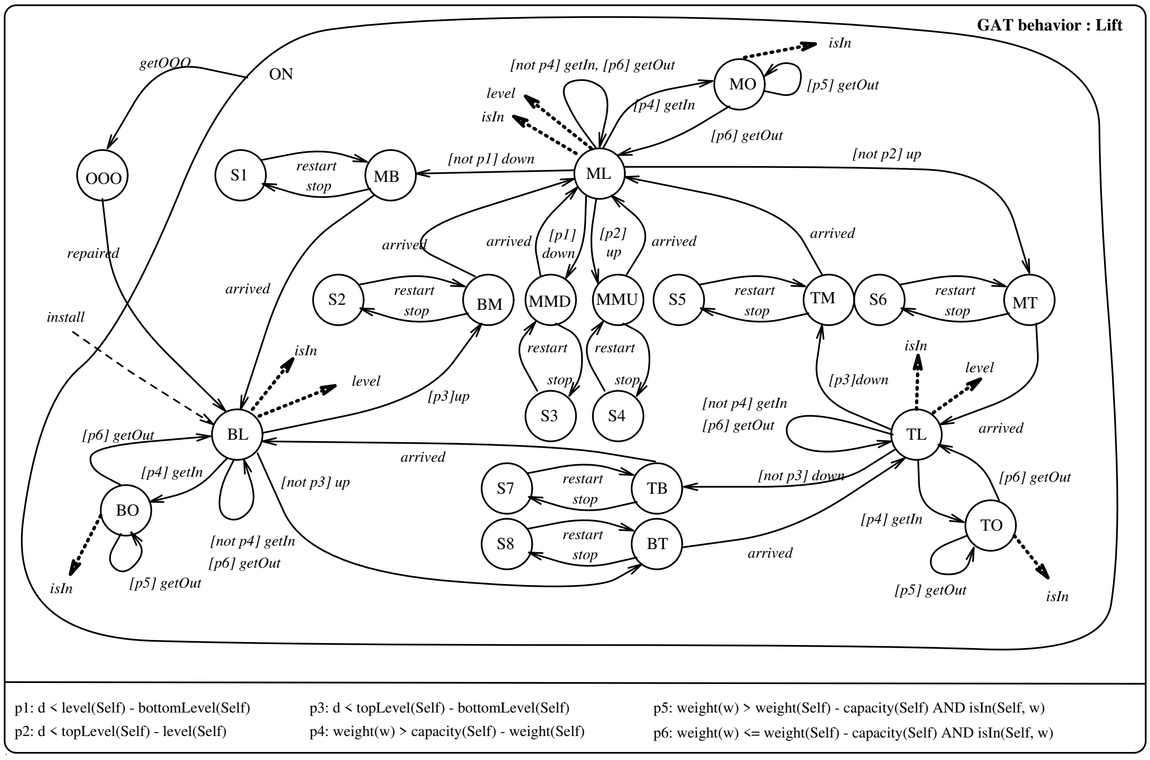


Figure 79 : Comportement dynamique du TAG `Lift`

C.3 Spécification algébrique

Voici la spécification algébrique issue de l'analyse de la spécification graphique.

- Signature of the `Lift` ADT:

$$
\left.\begin{array}{ll}
install & : Integer\ Integer\ RealGt1 \rightarrow\ Lift \\
up & : Lift\ IntegerGt1\ \rightarrow\ Lift \\
down & : Lift\ IntegerGt1\ \rightarrow\ Lift \\
arrived & : Lift\ \rightarrow\ Lift \\
stop & : Lift\ \rightarrow\ Lift \\
restart & : Lift\ \rightarrow\ Lift \\
repaired & : Lift\ \rightarrow\ Lift \\
chgCapacity & : Lift\ RealGt1\ \rightarrow\ Lift \\
getOOO & : Lift\ \rightarrow\ Lift \\
getIn & : Lift\ Weightable\ \rightarrow\ Lift \\
getOut & : Lift\ Weightable\ \rightarrow\ Lift
\end{array}\right\} constructors
$$

$$
\left.\begin{array}{ll}
level & : Lift\ \rightarrow\ Integer \\
topLevel & : Lift\ \rightarrow\ Integer \\
bottomLevel & : Lift\ \rightarrow\ Integer \\
limits & : Lift\ \rightarrow\ Integer\ \times\ Integer \\
weight & : Lift\ \rightarrow\ Real \\
capacity & : Lift\ \rightarrow\ RealGt1 \\
isIn & : Lift\ Weightable\ \rightarrow\ Boolean
\end{array}\right\} observers
$$

$$
\left.\begin{array}{ll}
ML & : Lift\ \rightarrow\ Boolean \\
MO & : Lift\ \rightarrow\ Boolean \\
 & \ldots \\
S8 & : Lift\ \rightarrow\ Boolean \\
OOO & : Lift\ \rightarrow\ Boolean \\
ON & : Lift\ \rightarrow\ Boolean
\end{array}\right\} state\ observers
$$

- General predicates :

```
guards
        p1(Self, Xd) == Xd < level(Self) - bottomLevel(Self)
        p2(Self, Xd) == Xd < topLevel(Self) - level(Self)
        p3(Self, Xd) == Xd < topLevel(Self) - bottomLevel(Self)
        p4(Self, Xw) == weight(Xw) > capacity(Self) - weight(Self)
        p5(Self, Xw) == weight(Xw) > weight(Self) - capacity(Self) AND isIn(Self, Xw)
        p6(Self, Xw) == weight(Xw) <= weight(Self) - capacity(Self) AND isIn(Self, Xw)

invariant(Self) == bottomLevel(Self) ≤ level(Self) ≤ topLevel(Self) AND
                bottomLevel(Self)) < topLevel(Self) AND
                (0 ≤ weight(Self)).
```

- Preconditions of operations (axioms are not simplified).

```
precondition(install(Xbottom, Xtop, Xcap)) == true
precondition(topLevel(Self)) == true
precondition(bottomLevel(Self)) == true
precondition(limits(Self)) == true
precondition(capacity(Self)) == true
precondition(isIn(Self, Xw)) == true
precondition(chgCapacity(Self,Xcap)) == true
precondition(weight(Self)) == true
precondition(up(Self,Xd)) == [(p3(Self, Xd) OR NOT p3(Self, Xd)) AND BL(Self)] OR [(p2(Self,
Xd) OR NOT p2(Self, Xd)) AND ML(Self)][1].
precondition(down(Self,Xd)) == [(p3(Self, Xd) OR NOT p3(Self, Xd)) AND TL(Self)] OR [(p1(Self,
Xd) OR NOT p1(Self, Xd)) AND ML(Self)]
precondition(level(Self)) == BL(Self) OR ML(Self) OR TL(Self)
precondition(getIn(Self,Xw)) == [(p4(Self, Xw) OR NOT p4(Self, Xw)) AND BL(Self)] OR [(p4(Self,
Xw) OR NOT p4(Self, Xw)) AND ML(Self)]] OR [(p4(Self, Xw) OR NOT p4(Self, Xw)) AND TL(Self)]
```

[1] Note that, `Xd` is constrained by the invariant

```
precondition(getOut(Self,Xw)) == [p6(Self, Xw) AND BL(Self)] OR [p6(Self, Xw) AND ML(Self)]]
OR [p6(Self, Xw) AND TL(Self)] OR [(p5(Self, Xw) OR p6(Self, Xw)) AND BO(Self)] OR [(p5(Self,
Xw) OR p6(Self, Xw)) AND MO(Self)]] OR [(p5(Self, Xw) OR p6(Self, Xw)) AND TO(Self)]
precondition(arrived(Self)) = BM(Self) OR MB(Self) OR MMD(Self) OR MMU(Self) OR TM(Self)
OR MT(Self) OR TB(Self) OR BT(Self)
precondition(stop(Self)) == BM(Self) OR MB(Self) OR MMD(Self) OR MMU(Self) OR TM(Self) OR
MT(Self) OR TB(Self) OR BT(Self)
precondition(restart(Self)) == S1(Self) OR S2(Self) OR S3(Self) OR S4(Self) OR S5(Self) OR
S6(Self) OR S7(Self) OR S8(Self)
precondition(getOOO(Self)) == ON(Self)
precondition(repaired(Self)) == OOO(Self)
```

- State observers on generators

```
BL1: BL(install(Xbottom, Xtop, Xcap)) == true
BL2: BL(up(Self, Xd)) == false
BL3: BL(down(Self, Xd)) == false
BL4: BL(arrived(Self)) == MB(Self) OR TB(Self)
BL5: BL(stop(Self)) == false
BL6: BL(getOOO(Self)) == false
BL7: BL(getIn(Self, Xw)) == NOT p4(Self, Xw) AND BL(Self)
...
MT1: MT(install(Xbottom, Xtop, Xcap)) == false
MT2: MT(up(Self, Xd)) == NOT p2(Self, Xd) AND ML(Self)
MT3: MT(down(Self, Xd)) == false
MT4: MT(arrived(Self)) == false
MT5: MT(stop(Self)) == false
MT6: MT(getOOO(Self)) == false
MT7: MT(getIn(Self, Xw)) == false
...
```

- primitive observers

```
bottomLevel1: bottomLevel(install(Xbottom, Xtop, Xcap)) == Xbottom
bottomLevel2: bottomLevel(up(Self, Xd)) == bottomLevel(Self)
bottomLevel3: bottomLevel(down(Self, Xd)) == bottomLevel(Self)
bottomLevel4: bottomLevel(arrived(Self)) == bottomLevel(Self)
bottomLevel5: bottomLevel(stop(Self)) == bottomLevel(Self)
bottomLevel6: bottomLevel(getOOO(Self)) == bottomLevel(Self)
bottomLevel7: bottomLevel(getIn(Self, Xw)) == bottomLevel(Self)
topLevel1: topLevel(install(Xbottom, Xtop, Xcap)) == Xtop
topLevel2: topLevel(up(Self, Xd)) == topLevel(Self)
topLevel3: topLevel(down(Self, Xd)) == topLevel(Self)
topLevel4: topLevel(arrived(Self)) == topLevel(Self)
topLevel5: topLevel(stop(Self)) == topLevel(Self)
topLevel6: topLevel(getOOO(Self)) == topLevel(Self)
topLevel7: topLevel(getIn(Self, Xw)) == topLevel(Self)
weight1: weight(install(Xbottom, Xtop, Xcap)) == 0
weight2: weight(up(Self, Xd)) == weight(Self)
weight3: weight(down(Self, Xd)) == weight(Self)
weight4: weight(arrived(Self)) == weight(Self)
weight5: weight(stop(Self)) == weight(Self)
weight6: weight(getOOO(Self)) == weight(Self)
weight7: weight(getIn(Self, Xw)) == weight(Self) + weight(Xw)
capacity1: capacity(install(Xbottom, Xtop, Xcap)) == Xcap
capacity2: capacity(up(Self, Xd)) == capacity(Self)
capacity3: capacity(down(Self, Xd)) == capacity(Self)
capacity4: capacity(arrived(Self)) = capacity(Self)
```

```
capacity5: capacity(stop(Self)) = capacity(Self)
capacity6: capacity(getOOO(Self)) = capacity(Self)
capacity7: capacity(getIn(Self, Xw)) = capacity(Self)
// partial observers //
isIn1: isIn(install(Xbottom, Xtop, Xcap), Xw) == false
isIn2: BL(Self) == true & p4(Self, Xw1) == false ==>
      isIn(getIn(Self, Xw1), Xw) == equal(Xw, Xw1) OR isIn(Self, Xw)
isIn3: ML(Self) == true & p1(Self, Xd) == false ==>
      isIn(arrived(down(Self, Xd))) == isIn(Self)
isIn4: TL(Self) == true & p3(Self, Xd) == false ==>
      isIn(arrived(down(Self, Xd))) == isIn(Self)
isIn5: BL(Self) == true & p4(Self, Xw1) == true ==>
      isIn(getIn(Self, Xw1), Xw) == equal(Xw, Xw1) OR isIn(Self, Xw)
isIn6: TL(Self) == true & p4(Self, Xw1) == false ==>
      isIn(getIn(Self, Xw1), Xw) == equal(Xw, Xw1) OR isIn(Self, Xw)
isIn7: ML(Self) == true & p2(Self, Xd) == false ==>
      isIn(arrived(up(Self, Xd))) == isIn(Self)
isIn8: BL(Self) == true & p3(Self, Xd) == false ==>
      isIn(arrived(up(Self, Xd))) == isIn(Self)
isIn9: TL(Self) == true & p4(Self, Xw1) == true ==>
      isIn(getIn(Self, Xw1), Xw) == equal(Xw, Xw1) OR isIn(Self, Xw)
isIn10: ML(Self) == true & p4(Self, Xw1) == false ==>
      isIn(getIn(Self, Xw1), Xw) == equal(Xw, Xw1) OR isIn(Self, Xw)
isIn11: ML(Self) == true & p1(Self, Xd) == true ==>
      isIn(arrived(down(Self, Xd))) == isIn(Self)
isIn12: ML(Self) == true & p2(Self, Xd) == true ==>
      isIn(arrived(up(Self, Xd))) == isIn(Self)
isIn13: TL(Self) == true & p3(Self, Xd) == true ==>
      isIn(arrived(down(Self, Xd))) == isIn(Self)
isIn14: BL(Self) == true & p3(Self, Xd) == true ==>
      isIn(arrived(up(Self, Xd))) == isIn(Self)
isIn15: ML(Self) == true & p4(Self, Xw1) == true ==>
      isIn(getIn(Self, Xw1), Xw) == equal(Xw, Xw1) OR isIn(Self, Xw)
level1: BL(Self) == true & p3(Self, Xd) == true ==>
      level(arrived(up(Self, Xd))) == level(Self) + Xd
level2: TL(Self) == true & p3(Self, Xd) == true ==>
      level(arrived(down(Self, Xd))) == level(Self) - Xd
level3: ML(Self) == true & p1(Self, Xd) == true ==>
      level(arrived(down(Self, Xd))) == level(Self) - Xd
level4: ML(Self) == true & p2(Self, Xd) == true ==>
      level(arrived(up(Self, Xd))) == level(Self) + Xd
level5: ML(Self) == true & p4(Self, Xw) == false ==>
      level(getIn(Self, Xw)) == level(Self)
level6: BL(Self) ==> level(Self) == bottomLevel(Self)
level7: TL(Self) ==> level(Self) == topLevel(Self)
```

• secondary constructors

```
chgCapacity1: chgCapacity(install(Xbottom, Xtop, Xcap1), Xcap) ==
      install(Xbottom, Xtop, Xcap)
chgCapacity2: chgCapacity(up(Self, Xd), Xcap) == up(chgCapacity(Self, Xcap), Xd)
chgCapacity3: chgCapacity(down(Self, Xd), Xcap) == down(chgCapacity(Self, Xcap), Xd)
chgCapacity4: chgCapacity(arrived(Self), Xcap) == arrived(chgCapacity(Self, Xcap))
chgCapacity5: chgCapacity(stop(Self), Xcap) == stop(chgCapacity(Self, Xcap))
chgCapacity6: chgCapacity(getOOO(Self), Xcap) == getOOO(chgCapacity(Self, Xcap))
chgCapacity7: chgCapacity(getIn(Self, Xw), Xcap) == getIn(chgCapacity(Self, Xcap), Xw)
restart1: MB(Self) == true ==> restart(stop(Self)) == Self
restart2: BM(Self) == true ==> restart(stop(Self)) == Self
restart3: MMD(Self) == true ==> restart(stop(Self)) == Self
```

```
restart4: MMU(Self) == true ==> restart(stop(Self)) == Self
restart5: TM(Self) == true ==> restart(stop(Self)) == Self
restart6: MT(Self) == true ==> restart(stop(Self)) == Self
restart7: TB(Self) == true ==> restart(stop(Self)) == Self
restart8: BT(Self) == true ==> restart(stop(Self)) == Self
repaired1: [OOO(Self)] ==> repaired(Self) ==
      install(bottomLevel(Self), topLevel(Self), capacity(Self))
--         getOut : G-derivation on BL
getOut1: p6(install(Xbottom, Xtop, Xcap), Xw) == true ==>
      getOut(install(Xbottom, Xtop, Xcap), Xw) == PRECONDITION ALWAYS FALSE
getOut2: MB(Self) == true & p6(arrived(Self), Xw) == true ==>
      getOut(arrived(Self), Xw) == arrived(getOut(Self, Xw))
getOut3: TB(Self) == true & p6(arrived(Self), Xw) == true ==>
      getOut(arrived(Self), Xw) == arrived(getOut(Self, Xw))
getOut4: p4(Self, Xw) == false & BL(Self) == true & p6(getIn(Self, Xw), Xw1)
      == true ==> getOut(getIn(Self, Xw), Xw1) ==
           IF Xw1 = Xw THEN Self ELSE getIn(getOut(Self, Xw1), Xw)
--             G-derivation on BO
getOut5: p4(Self, Xw) == true & BL(Self) == true & p6(getIn(Self, Xw), Xw1)
      == true ==> getOut(getIn(Self, Xw), Xw1) ==
           IF Xw1 = Xw THEN Self ELSE getIn(getOut(Self, Xw1), Xw)
getOut6: p4(Self, Xw) == true & BL(Self) == true & p5(getIn(Self, Xw), Xw1)
      == true ==> getOut(getIn(Self, Xw), Xw1) ==
           IF Xw1 = Xw THEN Self ELSE getIn(getOut(Self, Xw1), Xw)
--         getOut : G-derivation on ML
getOut7: BM(Self) == true & p6(arrived(Self), Xw) == true ==>
      getOut(arrived(Self), Xw) == arrived(getOut(Self, Xw))
getOut8: MMD(Self) == true & p6(arrived(Self), Xw) == true ==>
      getOut(arrived(Self), Xw) == arrived(getOut(Self, Xw))
getOut9: MMU(Self) == true & p6(arrived(Self), Xw) == true ==>
      getOut(arrived(Self), Xw) == arrived(getOut(Self, Xw))
getOut10: TM(Self) == true & p6(arrived(Self), Xw) == true ==>
      getOut(arrived(Self), Xw) == arrived(getOut(Self, Xw))
getOut11: p4(Self, Xw) == false & ML(Self) == true & p6(getIn(Self, Xw), Xw1)
      == true ==> getOut(getIn(Self, Xw), Xw1) ==
           IF Xw1 = Xw THEN Self ELSE getIn(getOut(Self, Xw1), Xw)
--             G-derivation on MO
getOut12: p4(Self, Xw) == true & ML(Self) == true & p6(getIn(Self, Xw), Xw1)
      == true ==> getOut(getIn(Self, Xw), Xw1) ==
           IF Xw1 = Xw THEN Self ELSE getIn(getOut(Self, Xw1), Xw)
getOut13: p4(Self, Xw) == true & ML(Self) == true & p5(getIn(Self, Xw), Xw1)
      == true ==> getOut(getIn(Self, Xw), Xw1) ==
           IF Xw1 = Xw THEN Self ELSE getIn(getOut(Self, Xw1), Xw)
--         getOut : G-derivation on TL
getOut14: MT(Self) == true & p6(arrived(Self)) == true ==>
      getOut(arrived(Self), Xw) == arrived(getOut(Self, Xw))
getOut15: BT(Self) == true & p6(arrived(Self)) == true ==
      getOut(arrived(Self), Xw) == arrived(getOut(Self, Xw))
getOut16: p4(Self, Xw) == false & TL(Self) == true & p6(getIn(Self, Xw), Xw1)
      == true ==> getOut(getIn(Self, Xw), Xw1) ==
           IF Xw1 = Xw THEN Self ELSE getIn(getOut(Self, Xw1), Xw)
--             G-derivation on TO
getOut17: p4(Self, Xw) == true & TL(Self) == true & p6(getIn(Self, Xw), Xw1)
      == true ==> getOut(getIn(Self, Xw), Xw1) ==
           IF Xw1 = Xw THEN Self ELSE getIn(getOut(Self, Xw1), Xw)
getOut18: p4(Self, Xw) == true & TL(Self) == true & p5(getIn(Self, Xw), Xw1)
      == true ==> getOut(getIn(Self, Xw), Xw1) ==
           IF Xw1 = Xw THEN Self ELSE getIn(getOut(Self, Xw1), Xw)
```

• secondary observers

```
limits1: limits(Self) = Tuple(bottomLevel(Self), topLevel(Self))
```

C.4 Un exemple de théorème démontré en ASSPEGIQUE

Démontrons en Asspegique+ le théorème équationnel suivant :

```
BL(arrived(down(arrived(up(install( 0, 3,Xr), 2)), 2)) == true
   ------------------------------------------------------------
   The expression to be evaluated is ...
     BL (arrived (down (arrived (up (install 0 3 Xr) 2)) 2))

              BL_
               |
            arrived_
               |
            _down__
           /       \
       arrived_   2
           |
       __up____
      /        \
   install___   2
   /   |   \
   0   3   Xr
   ------------------------------------------------------------
       [ 3 -->> s (s (s 0)) ] .
   This rule let us get the expression :

              BL_
               |
            arrived_
               |
            _down__
           /       \
       arrived_   2
           |
       __up____
      /        \
   install___   2
   /   |    \
   0   s_   Xr
       |
       s_
       |
       s_
       |
       0
   ------------------------------------------------------------
   ... apply [2 -->> s (s 0)) ] two times ...
   ------------------------------------------------------------
       [ BL (arrived Self) -->> (MB Self) or (TB Self) ] .
   This rule let us get the expression :

                  __________or__________
                 /                      \
               MB_                      TB_
                |                        |
            __down___                __down___
           /         \              /         \
        arrived_     s_          arrived_     s_
           |         |              |         |
       ___up____     s_         ___up____     s_
      /         \    |         /         \    |
   install___    s_   0     install___    s_   0
```

```
/   |    \    |        /   |    \    |
0   s_   Xr   s_       0   s_   Xr   s_
    |         |            |         |
    s_        0            s_        0
    |                      |
    s_                     s_
    |                      |
    0                      0
------------------------------------------------------------
    [ MB (down Self Xd) -->> (not (p1 Self Xd)) and (ML Self) ] .
This rule let us get the expression :

                        ________________or______________
                       /                                \
               _______and________                      TB_
              /                  \                      |
             not_               ML_                 __down___
              |                  |                 /         \
          ___p1____           arrived_          arrived_     s_
         /         \             |                  |        |
      arrived_     s_        ___up____          ___up____    s_
         |         |        /         \        /         \   |
     ___up____     s_   install___    s_   install___    s_  0
    /         \    |    /   |    \    |    /   |    \    |
install___    s_   0    0   s_   Xr   s_   0   s_   Xr   s_
/   |    \    |             |         |        |         |
0   s_   Xr   s_            s_        0        s_        0
    |         |             |                  |
    s_        0             s_                 s_
    |                       |                  |
    s_                      0                  0
    |
    0
------------------------------------------------------------
... after more than 100 conditionnal rewriting ...
------------------------------------------------------------
    [ b or false -->> b ] .
This rule let us get the expression :

true
------------------------------------------------------------
```

C.5 Spécification des classes formelles terminales structurées

MMDLift	
inherits from INTERMEDIATELift	
aspect : lift	
field selectors	**constraint**
oldState : MMDLift ⟶ MLLift	INVT
isStopped : MMDLift ⟶ Boolean	AND
aXd : MMDLift ⟶ IntegerGt1	p1(oldState(Self), aXd(Self))

avec `p1(Self, Xd) = Xd < level(Self) - bottomLevel(Self)`.

MMULift	
inherits from INTERMEDIATELift	
aspect : lift	
field selectors	**constraint**
oldState : MMULift ⟶ MLLift	INVT
isStopped : MMULift ⟶ Boolean	AND
aXd : MMULift ⟶ IntegerGt1	p2(oldState(Self), aXd(Self))

avec `p2(Self, Xd) = Xd < topLevel(Self) - level(Self)`.

MBLift	
inherits from INTERMEDIATELift	
aspect : lift	
field selectors	**constraint**
oldState : MBLift ⟶ MLLift	INVT
isStopped : MBLift ⟶ Boolean	AND
aXd : MBLift ⟶ IntegerGt1	NOT p1(oldState(Self), aXd(Self))

MTLift	
inherits from INTERMEDIATELift	
aspect : lift	
field selectors	**constraint**
oldState : MTLift ⟶ MLLift	INVT
isStopped : MTLift ⟶ Boolean	AND
aXd : MTLift ⟶ IntegerGt1	NOT p2(oldState(Self), aXd(Self))

BMLift	
inherits from INTERMEDIATELift	
aspect : lift	
field selectors	**constraint**
oldState : BMLift ⟶ BLLift	INVT
isStopped : BMLift ⟶ Boolean	AND
aXd : BMLift ⟶ IntegerGt1	p3(oldState(Self), aXd(Self))

avec `p3(Self, Xd) = Xd < topLevel(Self) - bottomLevel(Self)`.

BTLift	
inherits from INTERMEDIATELift	
aspect : lift	
field selectors	**constraint**
oldState : BTLift ⟶ BLLift	INVT
isStopped : BTLift ⟶ Boolean	AND
aXd : BTLift ⟶ IntegerGt1	NOT p3(oldState(Self), aXd(Self))

TMLift	
inherits from INTERMEDIATELift	
aspect : lift	
field selectors	**constraint**
oldState : TMLift ⟶ TLLift	INVT
isStopped : TMLift ⟶ Boolean	AND
aXd : TMLift ⟶ IntegerGt1	p3(oldState(Self), aXd(Self))

TMLift	
inherits from INTERMEDIATELift	
aspect : lift	
field selectors	**constraint**
oldState : TMLift ⟶ TLLift	INVT
isStopped : TMLift ⟶ Boolean	AND
aXd : TMLift ⟶ IntegerGt1	NOT p3(oldState(Self), aXd(Self))

BLLift	
inherits from BMLLift	
aspect : lift	
field selectors	**constraint**
bottomLevel : BLLift ⟶ Integer	bottomLevel(Self) < topLevel(Self)
topLevel : BLLift ⟶ Integer	AND 0 <= weight(Self)
capacity : BLLift ⟶ RealGt1	AND bottomLevel(Self) <= level(Self)
contents : BLLift ⟶ List[Weightable]	AND level(Self) <= topLevel(Self)
isInstalled : BLLift ⟶ Boolean	
oldState : BLLift ⟶ INTERMEDIATELift	
requires: isInstalled(Self) == false	

MLLift	
inherits from BMLLift TMLLift	
aspect : ML lift	
field selectors	**constraint**
isOverload : MLLift ⟶ Boolean	CONT
oldState : MLLift ⟶ INTERMEDIATELift	
contents : MLLift ⟶ List[Weightable]	AND INVT

CONT = [isOverload(Self) = weight(oldState(Self)) + map(contents(Self), weight) > capacity(oldState(Self))]

TLLift	
inherits from TMLLift	
aspect : TL lift	
field selectors	**constraint**
isOverload : TLLift ⟶ Boolean	CONT
oldState : TLLift ⟶ INTERMEDIATELift	
contents : TLLift ⟶ List[Weightable]	AND INVT

C.6 Spécification des classes formelles après restructuration

Lift ABSTRACT
inherits from OBJECT
comments: class for abstract lift
features: bottomLevel, topLevel, capacity, weight, chgCapacity, limits
extensions
;; weight : weight of the lift weight : Lift ABSTRACT ⟶ Real ABSTRACT
;; bottomLevel : bottom level of the lift bottomLevel : Lift ABSTRACT ⟶ Integer ABSTRACT
;; topLevel : top level of the lift topLevel : Lift ABSTRACT ⟶ Integer ABSTRACT
;; capacity : capacity of the lift capacity : Lift ABSTRACT ⟶ RealGt1 ABSTRACT
;; chgCapacity : set a new capacity for the lift chgCapacity : Lift ABSTRACT Lift RealGt1 ⟶ Lift ABSTRACT
;; limits : provides the lift limits (top and bottom levels) limits : Lift ABSTRACT ⟶ Tuple(Integer Integer) limits(Self) == newTuple(bottomLevel(Self), topLevel(Self))

ONLift ABSTRACT
comments: class for on lift
features: bottomLevel, topLevel, capacity, weight, chgCapacity, limits, getOOO
inherits from Lift
extensions
;; getOOO : the lift becomes out of order getOOO : ONLift ABSTRACT ⟶ OOOLift getOOO(Self) == newOOOLift(oldState = Self)

OOOLift

inherits from Lift

comments: class for out of order lift

features: bottomLevel, topLevel, capacity, weight, chgCapacity, limits, repaired

aspect : lift

field selectors	constraint
oldState : OOOLift ⟶ ONLift	INVT

extensions

```
;; weight : weight of the lift
   weight : OOOLift    ⟶  Real
weight(Self) == weight(oldState(Self))
```

```
;; bottomLevel : bottom level of the lift
   bottomLevel : OOOLift    ⟶  Integer
bottomLevel(Self) == bottomLevel(oldState(Self))
```

```
;; topLevel : top level of the lift
   topLevel : OOOLift    ⟶  Integer
topLevel(Self) == topLevel(oldState(Self))
```

```
;; capacity : capacity of the lift
   capacity : OOOLift    ⟶  RealGt1
capacity(Self) == capacity(oldState(Self))
```

```
;; chgCapacity : set a new capacity for the lift
   chgCapacity : OOOLift RealGt1  ⟶  OOOLift
chgCapacity(Self, Xcap) == copy(Self, oldState = chgCapacity(oldState(Self), Xcap))
```

```
;; repaired : the lift is being repaired
   repaired : OOOLift    ⟶  ONFLOORLift
repaired(Self) == install(ONFLOORLift, bottomLevel(Self), topLevel(Self), capacity(Self))
```

ONFLOORLift

inherits from ONLift

comments: class for onfloor lift

features: limits, weight, level, getIn, getOut, isIn, up, down, install

aspect : lift

field selectors	constraint
bottomLevel : ONFLOORLift ⟶ Integer	bottomLevel(Self) < topLevel(Self)
topLevel : ONFLOORLift ⟶ Integer	AND 0 <= weight(Self)
capacity : ONFLOORLift ⟶ RealGt1	AND bottomLevel(Self) <= level(Self)
contents : ONFLOORLift ⟶ List[Weightable]	AND level(Self) <= topLevel(Self)
isInstalled : ONFLOORLift ⟶ Boolean	
oldState : ONFLOORLift ⟶ INTERMEDIATELift	
requires: isInstalled(Self) == false	

extensions

```
;; weight : weight of the lift
   weight : ONFLOORLift    ⟶  Real
weight(Self) == sum(weight, contents(Self))
```

```
;; isOverloaded : there are too many weightable in the cage
   isOverloaded : ONFLOORLift    ⟶  Boolean
isOverloaded(Self) == weight(Self) > capacity(Self)
```

```
;; level : indicates the current level of the lift
   level : ONFLOORLift    ⟶  Integer
requires: isOverloaded(Self) == false
isInstalled(Self) == true ==> level(Self) == bottomLevel(Self)
isInstalled(Self) == false AND isUp(oldState(Self)) == true ==> level(Self) ==
     level(oldState(oldState(Self))) + offset(oldState(Self))
isInstalled(Self) == false AND isUp(oldState(Self)) == false ==> level(Self) ==
     level(oldState(oldState(Self))) - offset(oldState(Self))
```

```
;; chgCapacity : set a new capacity for the lift
   chgCapacity : ONFLOORLift RealGt1  ⟶  ONFLOORLift
var: Xcap : RealGt1
isInstalled(Self) == true ==>
     chgCapacity(Self, Xcap) == copy(Self, capacity = Xcap)
isInstalled(Self) == false ==>
     chgCapacity(Self, Xcap) == copy(Self, capacity = Xcap,
                                oldState = chgCapacity(oldState(Self), Xcap))
```

ONFLOORLift
;; isIn : indicates if the lift contains a weightable isIn : ONFLOORLift ⟶ Boolean isIn(Self, Xw) == belongsTo(contents(Self), Xw)
;; up : get up Xd levels up : ONFLOORLift IntegerGt1 ⟶ INTERMEDIATELift **var:** Xd : IntegerGt1 **requires:** Xd <= topLevel(Self) - level(Self) == true & isOverloaded(Self) == false up(Self, Xd) == newINTERMEDIATELift(oldState = Self, offset = Xd, isStopped == false, isUp == true)
;; down : gets down Xd levels down : ONFLOORLift IntegerGt1 ⟶ INTERMEDIATELift **var:** Xd : IntegerGt1 **requires:** Xd <= level(Self) - bottomLevel(Self) == true & isOverloaded(Self) == false down(Self, Xd) == newINTERMEDIATELift(oldState = Self, offset = Xd, isStopped == false, isUp == false)
;; getIn : a weightable gets inside the cage getIn : ONFLOORLift Weightable ⟶ ONFLOORLift **var:** Xw : Weightable **requires:** isOverloaded(Self) == false getIn(Self, Xw) == copy(Self, contents = add(contents(Self),Xw))
;; getOut : a weightable gets outside the cage getOut : ONFLOORLift Weightable ⟶ ONFLOORLift **var:** Xw : Weightable **requires:** isIn(Self, Xw) == true getOut(Self, Xw) == copy(Self, contents = remove(contents(Self),Xw))
class methods
;; install : build an initial lift install : Integer Integer RealGt1 ⟶ ONFLOORLift **var:** Xbot : Integer,Xtop : Integer,Xcap : RealGt1 install(Xbottom, Xtop, Xcap) == newONFLOORLift(bottomLevel = Xbottom, topLevel = Xtop, capacity = Xcap, contents = newEmptyList(), isInstalled = true)

INTERMEDIATELift	
inherits from ONLift	
comments: class for moving lift	
features: bottomLevel, topLevel, capacity, weight, chgCapacity, limits, restart, stop, arrived	
aspect : lift	
field selectors	**constraint**
oldState : INTERMEDIATELift ⟶ ONFLOORLift	bottomLevel(Self) < topLevel(Self) AND 0 <= weight(Self)
offset : INTERMEDIATELift ⟶ IntegerGt1	
isUp : INTERMEDIATELift ⟶ Boolean	
isStopped : INTERMEDIATELift ⟶ Boolean	
;; weight : weight of the lift weight : INTERMEDIATELift ⟶ Real weight(Self) == weight(oldState(Self))	
;; bottomLevel : bottom level of the lift bottomLevel : INTERMEDIATELift ⟶ Integer bottomLevel(Self) == bottomLevel(oldState(Self))	
;; topLevel : top level of the lift topLevel : INTERMEDIATELift ⟶ Integer topLevel(Self) == topLevel(oldState(Self))	
;; capacity : capacity of the lift capacity : INTERMEDIATELift ⟶ RealGt1 capacity(Self) == capacity(oldState(Self))	
;; chgCapacity : set a new capacity for the lift chgCapacity : INTERMEDIATELift RealGt1 ⟶ INTERMEDIATELift **var:** Xcap : RealGt1 chgCapacity(Self, Xcap) == copy(Self, oldState = chgCapacity(oldState(Self), Xcap))	
;; restart : restarts the lift restart : INTERMEDIATELift ⟶ INTERMEDIATELift **requires:** isStopped(Self) == true restart(Self) == copy(Self, isStopped = false)	

INTERMEDIATELift
;; stop : stops the lift ()   stop : INTERMEDIATELift   ⟶ INTERMEDIATELift **requires:** isStopped(Self) == false stop(Self) == copy(Self, isStopped = true)

;; arrived : the lift reaches the required level
   arrived : INTERMEDIATELift   ⟶ ONFLOORLift
requires: isStopped(Self) == false
arrived(Self) == newONFLOORLift(bottomLevel = bottomLevel(Self), topLevel = topLevel(Self),
        capacity = capacity(Self), contents = contents(oldState(Self)), isInstalled = false,
        oldState = Self)

C.7 Implantation de la classe ONFLOORLift

C.7.1 Eiffel

```
-- Class for onfloor lift
-- straight translation without optimization
-- EIFFEL 2.3
-- 06/06/94

CLASS ONFLOORLift

EXPORT weight, bottomLevel, topLevel, capacity, chgCapacity, limits,
    isIn, oldstate, up, down, getIn, getOut, install, isOverloaded;

INHERIT ONLift
    REDEFINE weight, botomlevel, topLevel, capacity, chgCapacity, isIn

FEATURE
    -- private fields
    bottomLevel_private : Integer;
    topLevel_private : Integer;
    capacity_private : Real;
    contents_private : ListWeightable;
    isInstalled_private := Boolean;
    oldState_private : INTERMEDIATELift;

    -- creation
    Create (Xbot, Xtop : Integer; Xcap : Real; Xcontents : List;
        Xinst : Boolean; Xold : INTERMEDIATELift) IS
-- constraint translation
        REQUIRES Xbot < Xtop and 0 <= contents.sum_weight
             and Xbot <= Xlevel and
((Xinst and Xbot <= Xtop)
or (not(Xinst) and
   Xold.oldState.level + oldState.offset <= Xtp))
        DO
        bottomLevel_private := Xbot;
        topLevel_private := Xtop;
        capacity_private := Xcap;
        contents_private := Xcontents;
        isInstalled_private := Xinst;
        oldState_private := Xold;
        END; -- Create

    -- field selectors
    bottomLevel : Integer IS
        DO
           RESULT := bottomLevel_private;
        END; -- bottomLevel

    topLevel : Integer IS
        DO
           RESULT := topLevel_private;
        END; -- topLevel
```

```
capacity : Real IS
    DO
       RESULT := capacity_private;
    END; -- capacity

contents : ListWeightable IS
    DO
       RESULT := contents_private;
    END; -- bottomLevel

isInstalled : Boolean IS
    DO
       RESULT := isInstalled_private;
    END; -- isInstalled

oldState : INTERMEDIATELift IS
    REQUIRES not(isInstalled);
    DO
       RESULT := oldState_private;
    END; -- oldState

-- the weight
weight : Real IS
    DO
       RESULT := contents.sum("weight");
    END; -- weight

-- overload
isOverloaded : Boolean IS
    DO
       RESULT := weight > capacity;
    END; -- isOverloaded

-- level
level : Integer IS
    REQUIRES not(isOverloaded);
    DO
    IF isInstalled
       THEN RESULT := bottomlevel;
       ELSE IF oldState.isUp
               THEN RESULT := oldState.level + oldState.offset;
               ELSE RESULT := oldState.level - oldState.offset;
            END;
    END;
    END; -- level

-- set a new capacity
chgCapacity (Xcap : Real) : ONFLOORLift IS
        DO
    -- copy is translated in creation
    IF isInstalled
       THEN RESULT.Create(bottomLevel, topLevel, Xcap, contents,
            isInstalled, undefined);
       ELSE RESULT.Create(bottomLevel, topLevel, Xcap, contents,
            isInstalled, oldState.chgCapacity(Xcap));
    END;
    END; -- chgCapacity

-- a weightable is in the lift
isIn (Xw : Weightable) : Boolean IS
    DO
       RESULT := contents.belongs(Xw);
    END; -- isIn

-- get up Xd levels
up (Xd : Integer) : INTERMEDIATELift IS
```

```
        REQUIRES Xd <= (toplevel - level) AND not(isOverloaded);
        DO
           RESULT.Create(current, Xd, false);
        END; -- up

    -- get down Xd levels
    down (Xd : Integer) : INTERMEDIATELift IS
        REQUIRES Xd <= (level - bottomLevel) AND not(isOverloaded);
        DO
           RESULT.Create(current, - Xd, false);
        END; -- up

    -- a weightable gets inside the lift
    getIn (Xw : Weightable) : ONFLOORLift IS
        REQUIRES not(isOverloaded);
        DO
           RESULT.Create(bottomLevel, topLevel, capacity, contents.add(Xw),
                isInstalled, oldState);
        END; -- getIn

    -- a weightable gets outside the lift
    getOut (Xw : Weightable) : ONFLOORLift IS
        REQUIRES isIn(Xw);
        DO
           RESULT.Create(bottomLevel, topLevel, capacity, contents.remove(Xw),
                isInstalled, oldState);
        END; -- getOut

    -- it's a instance method in Eiffel, create a new lift
    install (Xbot, Xtop : Integer; Xcap : Real) : ONFLOORLift IS
        LOCAL undefined;
              emptyList : EmptyListWeightable;
        DO
-- it is an undefined value for oldState of Lift
undefined.Create;
           RESULT.Create(Xbot, Xtop, Xcap, emptyList.Create, true, undefined);
        END; -- install

END --class ONFLOORLift
```

C.7.2 Smalltalk

```
'From Objectworks(r)\Smalltalk, Release 4 of 25 February 1991 on 17 October 1994 at 1:45:18 am'!

ONLift variableSubclass: #ONFLOORLift
    instanceVariableNames: 'bottomLevel topLevel capacity contents isInstalled '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'TAGLift'!
ONFLOORLift comment:
'@Copyright P. Andre 1994
This class implements the ONFLOORLift abstract class.

Here is the CF description :

inherited :
---------

redefined :
---------

defined :
---------
bottomLevel   <Integer>
topLevel      <Integer>
capacity      <RealGt1>
contents      <ListWeightable>
```

```
isInstalled   <Boolean>
oldState      <INTERMEDIATELift>   requires: isInstalled(Self) == false

comment : class for onfloor lift

exports : limits, weight, level, getIn, getOut, isIn, up, down, install

constraints :
   bottomLevel(Self) < topLevel(Self) and
   0 <= weight(Self) and
   bottomLevel(Self) <= level(Self) <= topLevel(Self)'!

!ONFLOORLift methodsFor: 'private'!

isOverloaded
    ";; isOverloaded : there are too many weightable in the cage
    isOverloaded : ONFLOORLift --> Boolean
    isOverloaded(Self) == weight(Self) > capacity(Self)"

    ^self weight > self capacity! !

!TMLLift methodsFor: 'field selectors'!

oldState
    "requires: isInstalled(Self) == false"

    self isInstalled ifTrue: [^nil]
            ifFalse: [^oldState]!

oldState: anINTERMEDIATELift
    oldState := anINTERMEDIATELift!

contents
    ^contents!

contents: aListOfWeightable
    contents := aListOfWeightable!

isInstalled
    ^isInstalled!

isInstalled: aBoolean
    isInstalled := aBoolean!

bottomLevel
    ^bottomLevel!

bottomLevel: anInteger
    bottomLevel := anInteger!

capacity
    ^capacity!

capacity: aRealGt1
    capacity := aRealGt1!

topLevel
    ^topLevel!

topLevel: anInteger
    topLevel := anInteger! !

!ONFLOORLift methodsFor: 'constructors'!

chgCapacityXcap: aRealGt1
    self isInstalled
        ifTrue:
```

```
                [capacity := aRealGt1.
                ^self].
        self isInstalled
            ifFalse:
                [oldState := self oldState chgCapacityXcap: aRealGt1.
                 capacity := aRealGt1.
                ^self].
        ^nil!

    upXd: anIntegerGt1
        ((anIntegerGt1 <= (self topLevel - self level)) and: [self isOverloaded not])
            ifTrue: [^INTERMEDIATELift newXoldState: self offset: anIntegerGt1 isStopped: False new,
isUp: True new].
        ^nil!

    downXd: anIntegerGt1
        ((anIntegerGt1 <= (self level - self bottomLevel)) and: [self isOverloaded not])
            ifTrue: [^INTERMEDIATELift newXoldState: self offset: anIntegerGt1 isStopped: False new,
isUp: False new].
        ^nil!

    getInXw: aWeightable
        self isOverloaded
            ifFalse:
                [contents := self contents add: aWeightable.
                ^self].
        ^nil!

    getOutXw: aWeightable
        self isInXw: aWeightable
            ifTrue:
                [contents := self contents remove: aWeightable.
                ^self].
        ^nil! !

    !ONFLOORLift methodsFor: 'observers'!

    isInXw: aWeightable
        ^self contents belongsTo: aWeightable!

    level
        self isOverloaded
            ifFalse:
                [self isInstalled ifTrue: ^self bottomLevel.
                 self isInstalled ifFalse:
                    [self oldState isUp ifTrue:
                           [^(self oldState oldState level + self oldState offset)]].
                 self isInstalled ifFalse:
                    [self oldState isUp ifFalse:
                           [^(self oldState oldState level - self oldState offset)]]].
        ^nil! !
    "-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

    ONFLOORLift class
        instanceVariableNames: ''!

    !ONFLOORLift class methodsFor: 'instance creation'!

    installXbottom: anInteger1 Xtop: anInteger2 Xcap: aRealGt1
        ^self newXbottomLevel: anInteger1 topLevel: anInteger2 Xcapacity: aRealGt1
             contents: ListOfWeightable empty isInstalled: True new!

    newXbottomLevel: anInteger1 topLevel: anInteger2 capacity: aRealGt1 contents:
       aListOfWeightable isInstalled: aBoolean1

        ^self new bottomLevel: anInteger1; topLevel: anInteger2; capacity: aRealGt1;
             contents: aListOfWeightable; isInstalled: aBoolean1! !
```

Annexe D

Glossaire

Il est très difficile de définir exhaustivement et objectivement (i.e. sans retenir le contexte) le vocabulaire utilisé. Par ailleurs, les définitions varient selon les auteurs et selon les branches de l'informatique. Un même terme peut évoquer des concepts différents (polysème). Par exemple, un type sera parfois consid'eré comme un type de donnée, comme un type abstrait, comme un regroupement d'éléments, comme une classe, comme un type d'entité... Et plusieurs termes désignent le même concept, avec parfois des petites variantes. Ainsi une entité est un objet, mais sa présentation ne contient pas de mêthodes.

action [action]: opération instantanée (opération).

activité [activity]: opération interruptible (opération).

agrégation [aggregation]: relation entre deux objets telle que l'objet contenu n'a pas d'existence propre en dehors de son objet contenant (partie-de, voir aussi importation).

algèbre initiale [initial]: *cf sématique algébrique.*

analyse [analysis, user requirement specification, requirements engineering]: construction d'un modèle du système à partir de l'analyse des besoins. Ce modèle peut faire l'objet d'une étude de faisabilité pour plusieurs scenarii (spécification des besoins, analyse préalable, étape conceptuelle, analyse conceptuelle, conception préliminaire, modélisation conceptuelle).

analyse des besoins [user requirements analysis]: définit les services du système, ses contraintes et ses buts en consultant les utilisateurs du système. Une étude d'opportunité peut être menée pour savoir si le système est réalisable et donner une approximation de la rentabilité de ce système (analyse préalable).

application [1/application, 2/mapping]: 1/utilisation pour, par extension utilisation possible de l'informatique à un problème quelconque 2/relation orientée entre deux ensembles telle que tout élément de l'ensemble de d'épart ait un correspondant dans l'ensemble d'arrivée (1/problème, 2/fonction).

attribut [attribute]: variable locale à un composant, de type un type de base (champ, variable).

classe [class]: modèle d'un ensemble d'objets appelés instances de la classe. Elle définit la structure et le comportement des instances.

classe de modèle [loose]: *cf sématique algébrique.*

classe formelle [formal class]: définition d'une classe sous forme d'un type abstrait algébrique et dont les méthodes sont classées en partie primitive et partie secondaire (CF).

classe concrète [concrete class]: classe d'un langage de programmation à objets.

clientèle [clientship]: *cf importation.*

cohérence [consistency]: une spécification est dite cohérente s'elle ne contient pas de contradictions.

cohésion [cohesion]: un système est dit cohérent si ses parties forment un tout uni et harmonisé (voir aussi cohérence).

contradiction [inconsistency]: chose à la fois vraie et fausse. Le terme est utilisé en général pour une interprétation d'une formule dans un système logique contenant vrai et faux (voir [Lau86, WL88]). Une contradiction est alors une formule dont l'interprétation est à la fois vraie et fausse..

comportement [behavior]: ensemble des méthodes d'un objet (signature, interface).

comportement dynamique [dynamic behavior]: façon dont un objet réagit aux événements extérieur. Il peut être exprimé par un automate, des traces, des diagrammes d'état, des expressions temporelles ou algébriques, etc (contrôle).

composant [component]: un composant est un concept, une abstraction, ou une chose ayant une existence propre dans le système étudié. Un composant logiciel est une partie cohérente et relativement autonome du logiciel (composant, entité, forme, objet, sujet, chose).

conception [global design, system design, detailed design, design engineering]: proposition de solution au problème spécifié dans l'analyse tenant compte des caractéristiques logiques d'usage du futur système d'information et des moyens de réalisation, humains, techniques et organisationnels. Conception système (architecture globale) et conception détaillée sont parfois séparées (étape fonctionnelle, analyse fonctionnelle, analyse organique, étape logique, conception technique).

concrétisation [concretisation]: consiste à fixer certains paramètres d'une spécification (voir aussi raffinement, instanciation).

délégation [délégation]: relation orientée entre deux objets, par laquelle un objet peut faire exécuter une opération par un autre objet.

démarche [method]: *cf processus de développement.*

dépendance structurelle [structural dependency]: relation orientée entre deux types de données telle que si A dépend structurellement de B alors toute définition complète et cohérente de A n'est possible que s'il existe une définition complète et cohérente de B (attribut).

enrichissement vertical [vertical enrichment]: *cf raffinement.*

entité [entity]: composant définit par un ensemble d'attributs, c'est une unité de connaissance et non de traitement (composant).

envoi de message [message send]: invocation d'une méthode d'un objet appelé le receveur par un autre objet appelé l'expéditeur. Un envoi de message est un événement (event).

événement [event]: fait que quelque chose se passe dans le système. Les événements peuvent être classés dans des hiérarchies de spécialisation. Un événement est soit interne soit externe à un système.

héritage [inheritance]: relation orientée entre deux classes, telle que une instance de la sous-classe soit aussi une instance de la superclasse. Mécanisme qui permet de définir incrémentalement une classe à partir d'une autre (héritage de spécialisation, héritage d'implantation).

héritage de spécialisation [implementation inheritance]: *cf héritage, sous-typage.*

héritage d'implantation [specialization inheritance]: *cf héritage, raffinement.*

identifiant [identifier]: attribut ou un ensemble d'attributs permettant de distinguer clairement deux composants (clé, identité).

implantation [implementation]: *cf réalisation.*

importation [import, enrichment]: relation générale orientée entre type de composants ou types de données qui indique qu'un composant utilise les services d'un autre composant (clientèle, enrichissement, utilisation, voir aussi agrégation, délégation).

installation [installation]: règle les problèmes de mise en place du logiciel dans l'organisation.

instanciation [instanciation]: consiste à fixer tous les paramètres d'une spécification. Par extension, l'instanciation est une relation entre un objet et sa classe ou encore entre une spécification sans paramètres et la spécification générique correspondante (voir aussi instanciation).

logiciel [software]: résultat de la phase de codage.

maintenance [maintenance]: processus itératif dont l'activité répétée est un cycle de développement complet qui adapte la solution conceptuelle aux erreurs, aux changements organisationnels et aux évolutions technologiques.

méthode [method]: opération applicable à un objet et définie dans sa classe (opération).

méthode de résolution de problème [method]: manière d'aborder un problème. Une méthode comporte un formalisme pour exprimer le problème, une philosophie, une démarche et des outils (méthode, méthode d'analyse, méthode de conception, modélisation, voir processus de développement).

modèle [model]: représentation d'une situation dans une notation et avec des concepts donnés.

objet [object]: composant défini par une structure et un comportement, c'est à la fois une unité de connaissance et de traitement (composant, entité).

objet complexe [composite object]: *cf agrégation.*

opération [operation]: traitement appliqué à des paramètres d'un type donné et rendant un résultat d'un type donné par le profil de l'opération. Dans une sémantique fonctionnelle, une opération est une fonction appliquée à des valeurs et rendant une valeur.

partie-de [part-of]: *cf agrégation.*

processus [process]: entité qui transforme des données.

processus de développement [development process]: ensemble d'activités visant à construire et maintenir du logiciel (démarche, cycle de vie).

profil d'opération [operation profile]: produit cartésien des paramètres et du type résultat de l'opération (arité).

qualité [quality]: qualité de spécification, qualité du logiciel, qualité du processus de développement.
propriété souhaitable d'une spécification, un composant

raffinement [refinement]: méthode de structuration qui consiste à donner une représentation plus proche des langages de programmation cible à un type abstrait de donnée. Raffinement des structures de données et raffinement des opérations sont distingués (voir aussi concrétisation).

réalisation [coding, implementation, program testing]: raffine la conception en termes de programmes validés par des tests unitaires et par **intégration** des parties et tests systèmes (codage, implantation, mise en oeuvre, programmation).

réification [reification]: *cf raffinement.*

sématique algébrique [loose]: interprétation d'une spécification algébrique en termes d'une classe d'algébres de même propriétés. Lorsqu'elle existe, l'algèbre initiale est unique et correspond à la plus petite classe d'algèbres satisfaisant les axiomes de la spécification. La sémantique à classe de modèles correspond à un ensemble de classes satisfaisant les propriétés de l'axiomatique (modèle algébrique).

signature [signature]: ensemble des opérations et des types utilisés pour définir un type abstrait de donnée (interface).

sous-typage [subtyping]: relation entre deux types, telle que toute valeur du sous-type est une valeur du type.

spécification [specification, product]: ensemble de modèles résultant d'une étape de développement du logiciel (spécification des besoins, spécification détaillée, spécification fonctionnelle, spécification formelle,logiciel).

spécification algébrique [algebraic specification]: spécification d'un type abstrait de données par des profils d'opérations et des axiomes (type abstrait algébrique).

spécification du logiciel [software specification]: description plus ou moins détaillée du comportement attendu du logiciel.

structuration [structuration]: découpage cohérent et organisé d'un élément quelconque. La structuration d'un type de donnée ou d'un composant se fait généralement par des opérateurs de structuration (union ou produit de types, généricité, héritage ou sous-typage, récurrence, structures de données abstraites (listes, arbres, tables...) (structure, raffinement).

structure [structure]: organisation interne d'un composant. La structure d'un type de donnée est une représentation particulière de ce type de donnée. La structure d'un objet est l'ensemble des objets et des attributs qui le composent. (aspect).

système [system]: ensemble structuré de composants. Les liens entre composants peuvent être de plusieurs types, selon la méthode de modélisation utilisée (agrégation, clientèle, association, héritage, délégation...).

type [type]: ensemble d'éléments appelés occurrence ou valeurs du type. Des variantes existent : types de données, des types d'attributs, des types d'entité, des types d'objets(classe) (occurrence,valeur).

type de base [basic type]: type prédéfini du langage de spécification (entier, réel, caractère, booléen,...) (valeur de base).

type de donnée [data type, abstract data type]: type caractérisé par un ensemble de valeurs et un ensemble d'opérations applicables sur ces valeurs (type abstrait de donnée).

type abstrait de donnée [type]: définition formelle des opérations et des propriétés d'un type de donnée indépendamment d'une représentation particulière. Plusieurs styles existent, plus ou moins abstraits: équation de type (on donne la structure), modèle abstrait (une structure mathématique ou algorithmique permet définir axiomatiquement les opérations par pre/post condition), spécification algébrique (définition algébrique des opérations) (abstraction de donnée).

type abstrait algébrique [algebraic specification of a data type]: *cf spécification algébrique.*

utilisation [use]: *cf importation.*

valeur [value]: occurrence d'un type (terme).

variable [variable]: triplet $<nom, valeur, type>$ où *valeur* est une occurrence de *type*.

Annexe E

Notations

Un certain nombre de conventions syntaxiques ont été adoptées dans les langages décrivant les modèles. Nous les rappelons ici.

Notations générales

Les classes sont considérées comme des implantations particulières de types de données s. Les noms de classes et de types commencent par une majuscule. Les noms d'opérations commencent par une minuscule et peuvent contenir plusieurs mots séparés par le caractère souligné "_". Les noms de variables sont commencent par une lettre. Une variable spéciale désignant le receveur est notée `Self`.

On utilise des termes, équations et axiomes comme dans les spécifications algébriques ou les langages fonctionnels. Un terme est soit une variable soit l'application d'une opération à des termes. Une équation est une paire de termes `t == u`. Un axiome est une équation conditionnelle `cond ==> t == u` où `cond` est une conjonction d'équations.

L'ensemble des opérations d'un TAG est appelé sa signature. L'ensemble des méthodes d'une classe formelle est appelé son comportement. Le **profil** d'une opération (resp. une méthode) est noté par :

$$\texttt{oper: } \texttt{A}_1 \ \texttt{...} \ \texttt{A}_n \ \longrightarrow \texttt{R}$$

où A_i et R sont des noms de types. Le **type d'intérêt** (`TI`) est le type défini par le TAG. La **classe formelle courante** (`CFC`) est la classe en cours de définition. Si le type d'intérêt fait partie des paramètres, alors c'est le premier et la variable associée est `Self`. Chaque opération contient au moins une fois le type d'intérêt (resp. la classe formelle courante) dans son profil.

Les opérations (resp. méthodes) qui ont le type d'intérêt (resp. la classe formelle courante) en résultat sont appelées **constructeurs**. Les autres sont des **observateurs**. Comme dans les spécifications algébriques, nous utilisons un principe d'induction ([LG90]). Un noyau d'opérations (resp. de méthodes) est suffisant pour décrire entièrement le type (resp. la classe). Les unes sont dites primaire les autres secondaires. Les constructeurs primaires sont aussi appelés générateurs. Ils suffisent pour obtenir toutes les valeurs du type (resp. toutes les instances de la classe).

L'instanciation, notée `isa`, est une relation entre un objet et sa classe. L'héritage, noté `subclass` (resp. `ako` est une relation entre TAGs (resp. classes formelles). L'héritage entraine le sous-typage dont la propriété principale est la règle de substitution.

Notations TAG

Voici les notations utilisées

DPF	fermeture transitive de l'importation types
USE	fermeture transitive de la dépendance entre types
$spec(S)$	spécification algébrique du type S
TAD	type abstrait de données
TAG	type abstrait graphique de données
TAA	type abstrait algébrique de données
TI	type d'intérêt
$PRES$	présentation d'un type de données $PRES = (\Sigma, X, E)$
Σ	signature d'un type de données $\Sigma = (S, F)$
F	ensemble de noms d'opérations munies d'un profil (ou arité) $F = (C, FS)$
C	ensemble de générateurs d'une signature
FS	ensemble d'opérations secondaires d'une signature
PFC	ensemble des paramètres formels constants
PFT	ensemble des paramètres formels de type
S	ensemble de noms de types (sortes)
X	ensemble de variables
E	ensemble d'axiomes
$axiome_{op}$	axiome relatif à l'opération op
Γ	ensemble des constructeurs, $\Gamma \subseteq F$
Ω	ensemble des générateurs, famille génératrice, $\Omega \subseteq \Gamma$
$\widehat{\Omega_T}$	ensemble des familles génératrices du type T, $\Omega \in \widehat{\Omega_T}$
Γ_b	ensemble des constructeurs de base $\Gamma_b \subseteq \Gamma$, on n'a pas forcément $\Gamma_b \subseteq \Omega$
φ	ensemble des observateurs, $\varphi \subset F$
φ_p	ensemble des observateurs primaires $\varphi_p \subseteq \varphi$
∂	ensemble des opérations partielles
τ	ensemble des fonctions totales, $\partial \cup \tau = F$
η	ensemble des fonctions neutres, $\eta \subseteq \tau$
M	automate de comportement, $M = (F, K, \Delta, \delta, K_0, K_f)$
K	ensemble des états
Δ	ensemble des gardes
K_0	ensemble d'états initiaux
K_f	ensemble des états finaux
δ	fonction de transition $\delta\ : K \times \Sigma \times \Delta \rightarrow K$
σ	fonction de substitution
$\sim$	transition similaire
Dom_{op}	ensemble des états ou l'opération op est définie

Figure 80 : La notation du modèle TAG

Classification des opérations

<table>
<tr><td colspan="10">Classification de l'opération op</td></tr>
<tr><td>Dom$_{op}$</td><td>$Dom_{op} = \{\varepsilon\}$</td><td colspan="4">$Dom_{op} = K$</td><td colspan="4">$Dom_{op} \subset K$</td></tr>
<tr><td>transition$^{(\dagger)}$</td><td></td><td colspan="2">non$^{(*)}$</td><td colspan="2">oui</td><td colspan="4">oui$^{(*)}$</td></tr>
<tr><td>observateur?</td><td></td><td>oui</td><td>non</td><td colspan="2"></td><td colspan="2">non</td><td colspan="2">oui</td></tr>
<tr><td>partition$^{(\ddagger)}$</td><td></td><td></td><td></td><td>oui</td><td>non</td><td>oui</td><td>non</td><td>oui</td><td>non</td></tr>
<tr><td>classe A</td><td>$\Gamma_b \in \Gamma$</td><td>φ</td><td colspan="5">Γ</td><td colspan="2">φ</td></tr>
<tr><td>classe B</td><td>τ</td><td colspan="2">$\eta \subseteq \tau$</td><td>τ</td><td colspan="5">∂</td></tr>
</table>

$(*)$: une transition gardée est forcément sur l'automate.
$(\dagger)$: il existe une transition non vide supportant cette opération : $\exists e, e' \in K \bullet e \neq e' \ \wedge \ \delta(e, op, g) = e'$
$(\ddagger)$: les gardes de l'opération forment une partition : $\forall e \in Dom_{op} \bullet \cup_{g_i \in G} g_i = true$ avec $g_i \in G \equiv \exists e' \in K \bullet \delta(e, op, g_i) = e'$

Figure 81 : Classification des opérations TAG

Voyons maintenant l'influence de cette classification sur les différents calculs. La seconde colonne indique la classe A concernée.

<table>
<tr><td>catégorie</td><td>classe A</td><td>totale non neutre $(\tau - \eta)$</td><td>neutre (η)</td><td>partielle (∂)</td></tr>
<tr><td>calcul de la signature</td><td></td><td colspan="3">calcul de la classe A : Γ, φ</td></tr>
<tr><td rowspan="2">calcul des préconditions</td><td rowspan="2">Γ φ</td><td>calcul de τ</td><td>calcul de η</td><td>calcul de ∂</td></tr>
<tr><td colspan="2">précondition vraie</td><td>prédicat dérivé automatiquement</td></tr>
<tr><td>calcul des générateurs</td><td>Γ</td><td>$\Omega \subseteq (\tau - \eta)$</td><td></td><td>$\Omega \subset \partial$</td></tr>
<tr><td>détermination des observateurs primaires</td><td>φ</td><td>φ_p est une partition de $(\tau - \eta)$</td><td>φ_p est une partition de τ</td><td>φ_p est une partition de ∂</td></tr>
<tr><td>calcul des observateurs d'état</td><td>φ
(Ω)</td><td>fonction des transitions entrantes</td><td>pas de changement d'état</td><td>fonction des transitions entrantes</td></tr>
<tr><td>calcul des constructeurs secondaires</td><td>Γ
(Ω)</td><td>Γ_b : G-dérivation terminale sur $(K_0 - K_\Omega)$
$\Gamma \cap (\tau - \eta)$: G-dérivation sur K</td><td>G-dérivation sur K</td><td>G-dérivation sur Dom_op (états de définition</td></tr>
<tr><td>action des observateurs sur les générateurs</td><td>φ_p
(Ω)</td><td colspan="2">application à chaque générateur</td><td>fonction des états G-dérivation</td></tr>
<tr><td>calcul des observateurs secondaires</td><td>φ
(φ_p, Γ)</td><td colspan="2">- équations directe
- G-dérivation sur K</td><td>G-dérivation sur Dom_{op}</td></tr>
</table>

Figure 82 : Utilisation de la classification

Bibliographie

[ABDS95] Eric Amiel, Marie-Jo Bellosta, Eric Dujardin, and Eric Simon. Type-Safe Relaxing of Schema Consistency Rules for Flexible Modelling in OODMBS. *VLDB Journal*, 1995.

[Abr84] Jean-Raymond Abrial. Spécifier ou comment matérialiser l'abstrait. *Technique et Science Informatique*, 3(3):201–219, 1984.

[ABR95] Pascal André, Frank Barbier, and Jean-Claude Royer. Une expérimentation de développement formel à objets. *Techniques et Sciences Informatique*, 1995. A paraître.

[ACR94] Pascal André, Dan Chiorean, and Jean-Claude Royer. The Formal Class Model. In *Joint Modular Languages Conference*, Germany, September 1994. GI, SIG and BCS.

[ADL91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static Type Checking of Multi-Methods. *OOPSLA'91 Proceedings*, pages 113–128, October 1991.

[AG91] Antonio J. Alencar and Joseph A. Goguen. OOZE: An Object-Oriented Z Environment. In Pierre America, editor, *Proceedings of ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 180–199, Geneva, Switzerland, 1991. Springer Verlag.

[AK86] Hassan Ait-Kaci. An Algebraic Semantics Approach to the Effective Resolution of Type Equations. *Theoretical Computer Science*, 45:293–351, 1986.

[Ame89] Pierre America. A behavioral Approach to Subtyping in Object-Oriented Programming languages. *Proceedings of the wokshop on inheritance hierarchies in knowledge representation and programming languages*, pages 141–156, 1989. Viareggio Italy, february 6-8.

[AR92a] Egil P. Andersen and Trygve Reenskaug. System Design by Composing Structures of Interacting Objects. In *Proceedings of ECOOP'92, Lehrman Madsen (Ed.)*, volume 615 of *Lecture Notes in Computer Science*, pages 133–152. Springer Verlag, 1992.

[AR92b] Pascal André and Jean-Claude Royer. Optimizing method search with lookup caches and incremental coloring. In Andreas Paepcke, editor, *OOPSLA'92 Proceedings*, pages 110–126, Vancouver, October 1992. ACM, ACM Press.

[AR92c] Michel Augeraud and Jean-Claude Royer. Une interprétation du concept de classe en termes de type abstrait. *Journées du GDR Programmation avancée et outils pour l'intelligence artificielle*, pages 13–27, March 1992.

[AR94a] Pascal André and Jean-Claude Royer. Introduction de concepts formels dans le développement objet. In *Journées du GDR Programmation*, Lille, 22-23 septembre 1994.

[AR94b] Pascal André and Jean-Claude Royer. La modélisation des listes en programmation par objets. In Pierre Cointe, Christian Queinnec, and Bernard Serpette, editors, *Journées Francophones des Langages Applicatifs (JFLA'94)*, number 11 in Collection Didactique, pages 259–285, Noirmoutier, 31 janvier - 1 février 1994. INRIA.

[AR95] Pascal André and Jean-Claude Royer. Formal Concepts and Tools for Object-Oriented Analysis and Design. In *5th International Computing Congress*, Hyderabad, India, January 1995. Tata McGraw-Hill Publishers. Theme : Object-Oriented Technology: Methods and Applications.

[Ari90] M. Aristide. Modéliser un logiciel à l'aide de VDM : une expérience. *Technique et Science Informatique*, 9(4):313–330, 1990.

[Arn92] André Arnold. *Systèmes de transitions finis et sémantique des processus communiquants.* Collection Etudes et recherches en informatique. Masson, 1992.

[Bai89] Sidney C. Bailin. An Object-Oriented Requirements Specification Method. *Communication of the ACM*, 32(5):608–623, May 1989.

[Bar92] Moncef Bari. *Une méthode d'analyse et de conception orientée objet de systèmes d'information actif.* Thèse de doctorat, Université Paris 6, 6 février 1992 1992.

[BB92] Gilles Bernot and Michel Bidoit. Introduction aux spécifications algébriques. In *Seconde école des jeunes chercheurs*, Bordeaux, April 1992. Greco de Programmation.

[BBC+92] Didier Bert, Michel Bidoit, Christine Choppy, Rachid Echahed, Jean-Pierre Finance, Marie-Claude Gaudel, Jean-Michel Hufflen, Jean-Pierre Jacquot, Michel Lemoine, Nicole Lévy, Jean-Claude Reynaud, Clément Rocques, and Frédéric Voisin. Définition de la forme interne des spécifications algébriques dans la structure d'accueil SALSA. Technical report, Greco de Programmation du CNRS, February 1992.

[BBG+94] Bruno Bonnin, J.C. Broquaire, Raymond Gueguen, Isabelle Mainguet, and Vincent Sevel. ASFO : un Atelier de Spécification Formelle et conception à Objets. Rapport de projet industriel de dess génie informatique, Université de Nantes, March 1994.

[BCC+87] Michel Bidoit, Francis Capy, Christine Choppy, Nicole Choquet, Christian Gresse, Stéphane Kaplan, Françoise Schlienger, and Frédéric Voisin. ASSPRO : un environnement de programmation interactif et intégré. *Technique et Science Informatique*, 6(1):21–41, 1987.

[BCC90] Michel Bidoit, Francis Capy, and Christine Choppy. The design and specification of the ASSPEGIQUE data base. In *Proc. of the Int. Symp. on Design and Implementation of Symbolic Computation Systems (DISCO)*, pages 205–214. Springer-Verlag L.N.C.S. 429, 1990.

[Ber82] Didier Bert. Construction de spécifications en LPG. In *Compte-rendu des journées Gréco-GROSPLAN*, March 1982.

[BG94a] Paulo Borba and Joseph A. Goguen. An operational Semantics for FOOPS. Technical Report PRG-TR-16-96, Programming Research Group - Oxford University, November 1994.

[BG94b] Paulo Borba and Joseph A. Goguen. On Refinement and FOOPS. Technical Report PRG-TR-17-96, Programming Research Group - Oxford University, November 1994.

[BGM87] Michel Bidoit, Marie-Claude Gaudel, and Anne Mauboussin. How to make Algebraic Specifications More Understandable. Technical Report 3, Rapport Gréco de Programmation, CNRS, 1987.

[BH94] Jonathan P. Bowen and Michael G. Hinchley. Seven More Myths of Formal Methods. In LNCS Springer Verlag, editor, *Proceedings FME'94 Symposium*, Wolfson Building, Parks Road, Oxford OX1 3QD, 1994. PRG-TR-7-94, report of Oxford University Computing Laboratory.

[BHS91] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL with Applications from Protocol Specification*. The BCS Practitioner. Prentice Hall, 1991.

[Bid82] Michel Bidoit. Types abstraits algébriques : spécifications structurées et présentations gracieuses. In *Colloque AFCET, Les mathématiques de l'informatique*, pages 347–357, Paris, March 1982.

[Bid89] Michel Bidoit. *Pluss, un langage pour le développement de spécifications algébriques modulaires*. Thèse d'etat, Université de Paris Sud, Orsay, May 1989.

[Boe82] B.W. Boehm. Les facteurs du coût logiciel. *Technique et Science Informatique*, 1(1):5–24, 1982.

[Boo92] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison Wesley France, 1ere edition, 1992.

[BP94] Andrew Black and Jens Palsberg. Foundations of Object-Oriented Languages. *ACM SIGPLAN Notices*, 29(3):3–11, 1994.

[BR85] Barry Boehm and Rony Ross. La gestion de projets logiciels selon la théorie W : une étude de cas. *Revue Génie Logiciel et Systèmes Experts*, 15:4–20, September 1985.

[Bra88] Gilles Bracon. La spécification de logiciels dans l'industrie. *BIGRE + GLOBULE*, 58:12–19, January 1988. IRISA-AFCET.

[Bre91] Ruth Breu. *Algebraic Specification Techniques in Object-Oriented Programming Environments*, volume 562 of *Lecture Notes in Computer Science*. Springer Verlag, June 1991. Thesis, Universität Passau, Germany.

[Bri93] Frédéric Brissaud. *Contribution à la conceptionn logicielle d'applications : la méthode MOSAIC dans le projet Aristote*. Thèse de doctorat, Université Joseph Fourier, Grenoble, May 1993.

[BS88] Manfred Broy and Pierre-Yves Schobbens. Une notation pour la specification algébrique de systèmes concurrents. In *4th Conference-Exhibition of Software Engineering*, pages 141–153, Paris, October 1988. AFCET.

[BW85] Richard Bird and Philip Wadler. *Functional Programming*. Prentice Hall International, 1985.

[Cal90] Jean-Paul Calvez. *Spécification et conception des systèmes : une méthodologie*. Masson, May 1990.

[Car84] Luca Cardelli. A Semantics of Multiple Inheritance. *LNCS 173 "semantics of data type"*, 173:131–144, June 1984.

[Car88] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1988.

[Car91] Denis Caromel. *Programmation parallèle asynchrone et impérative : études et propositions.* Thèse de doctorat, Université de Nancy I, February 1991. 16 février.

[Cas91] Xavier Castellani. L'implémentation de concepts fondamentaux du modèle de la méthode MCO d'analyse et de conception des systèmes d'objets. In *Actes des quatrièmes journées internationales sur le Génie logiciel et ses applications*, pages 743–759, Toulouse, December 1991.

[Cas94a] Giuseppe Castagna. *Overloading, Subtyping and Late Binding: Functional Foundation of Object-Oriented Programming.* PhD thesis, Université Paris VII, January 1994.

[Cas94b] Pierre Castéran. Construction de programmes CPS sous `Coq`. In *Journées Francophones des Langages Applicatifs*, volume 11 of *Collection didactique*, pages 171–183. INRIA, January 1994. Pierre Cointe, Christian Queinnec and Bernard Serpette (Eds.).

[Cas95] Giuseppe Castagna. Covariance and contravariance: conflict without cause. Technical report, Laboratoire d'Informatique de l'École Normale Supérieure, 1995.

[CBB+93] Christine Choppy, Didier Bert, Michel Bidoit, Rachid Echahed, Jean-Claude Reynaud, Clément Rocques, and Frédéric Voisin. Rapid prototyping with algebraic specifications: A case study. Technical Report 844, LRI - Université de Paris Sud, May 1993.

[CG88] Christine Choppy and Marie-Claude Gaudel. Impact des spécifications formelles sur le développement de logiciel. *BIGRE + GLOBULE*, 58:3–11, January 1988. IRISA-AFCET.

[CGL92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. In *Journées GDR Programmation et outils pour l'intelligence artificielle*, pages 28–37, March 1992. Nancy 18-20 mars Rapport GRECO de Programmation.

[CHC90] William Cook, W. L. Hill, and Peter S. Canning. Inheritance Is Not Subtyping. In *Proceedings of the 1990 ACM Seventeenths Annual Symposium on POPL*, pages 125–135, 1990.

[Cho87] Christine Choppy. Formal specifications, prototyping and integration tests. In *Proceedings of the 1st European Software Engineering Conference*, pages 185–192, Strasbourg, September 1987. AFCET.

[Cho88] Christine Choppy. *ASSPEGIQUE User's Manual.* LRI, Orsay, August 1988. Rapport Greco de Programmation.

[CI88] CRI A-S CISI Ingénierie. *HOOD Manual Issue 2.2.* Matra Espace, 1988.

[CL94] Craig Chambers and Gary T. Leavens. Type Checking and Modules for Multi-Methods. *OOPSLA'94 Proceedings*, October 1994.

[CLLF93] Eduardo Casais, Claus Lewerentz, Thomas Lindner, and Weber Franz. Formal Methods and Object-Orientation. In Boris Magnusson, Bertrand Meyer, and Jean-François Perrot, editors, *TOOLS EUROPE '93*, Versailles, March 1993. International Conference & Exibition, Prentice Hall. Tutorial T7.

[CO88] S. Clerici and F. Orejas. GSBL: An Algebraic Specification Language Based on Inheritance. In *Proceedings of ECOOP '88*, volume 322 of *LNCS*, pages 78–92, Oslo, August 1988. Springer Verlag.

[Coa92] Peter Coad. Object-Oriented Patterns. *Communications of the ACM*, 35(9):152–159, September 1992. Special Issue on Analysis and Modelling in Software Development.

[Coi87] Pierre Cointe. Metaclasses Are First Classes : the ObjVlisp Model. In *ACM OOPSLA'87 proceedings*, pages 156–167. ACM, October 1987. October 4-8.

[Coo89] William R. Cook. A Proposal for Making Eiffel Type-safe. *The computer Journal*, 32(4):305–311, 1989.

[CSM94] Virginia A.O. Cordeiro, Augusto Sampaio, and Silvio L. Meira. From MooZ to Eiffel - A Rigorous Approach to System Development. In Naftalin. Maurice, Tim Denvir, and Miquel Bertran, editors, *FME'94 Symposium*, volume 973 of *LNCS*, pages 306–325, Barcelona, Spain, October 1994. Springer Verlag.

[Cus91] Elspeth Cusack. Inheritance In Object Oriented Z. In Pierre America, editor, *Proceedings of ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 167–179, Geneva, Switzerland, 1991. Springer Verlag.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[CY91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Prentice Hall, 2 edition, 1991.

[D̈94] Eugene Dürr. VDM++ Language Reference Manual. Technical report afro/cg/ed/lrm/v9, Cap Gemini - Utrecht University, May 1994. Plat, Nico (Ed) - Esprit III AFRODITE project.

[Dah90] Ole-Johan Dahl. Object-Orientation and Formal Techniques. In *Proceedings of VDM'90, D. Bjorner, C.A.R. Hoare, H. Langsmaack (Eds)*, pages 1–11. Springer Verlag, 1990.

[DCAF92] Dennis De Champeaux, Al Anderson, and Ed Feldhousen. Case Study of Object-Oriented Software Development. In *Proceedings of OOPSLA'92 (Andreas Paepcke Ed)*, pages 377–391, Vancouver, Canada, October 1992. ACM Press.

[DCF92] Dennis De Champeaux and Penelope Faure. A comparative Study of Object-Oriented Analysis Methods. *JOOP*, pages 21–33, March 1992.

[DD90] D. Duke and Roger Duke. Towards a Semantics for Object-Z. In D. Bjorner, C.A.R. Hoare, and H. Langsmaack, editors, *Proceedings of VDM'90*, pages 244–261. Springer Verlag, 1990.

[DDHL94] Hervé Dicky, Christophe Doby, Marianne Huchard, and Thérèse Libourel. ARES, un algorithme d'Ajout avec REStructuration dans les hiérarchies de classes. In *LMO'94*, pages 125–136, Grenoble, October 1994.

[Dec88] Olivier Declerfayt. Software Through Pictures de IDE : Outils et Méthodes. *BIGRE + GLOBULE*, 58:57–67, January 1988. IRISA-AFCET.

[Des92] Philippe Desfray. *Ingénierie des objets, approche classe-relation, application à C++*. Collection MIPS. Masson, 1992.

[DGI90] R. Di Giovanni and P.L. Iachini. HOOD and Z for the Development of Complex Software Systems. In *Proceedings of VDM'90, D. Bjorner, C.A.R. Hoare, H. Langsmaack (Eds)*, pages 262–289. Springer Verlag, 1990.

[DGP94] Eugene Dürr, Stephen Goldsack, and Nico Plat. Rigorous Development of Concurrent Object-oriented Systems. Technical report afro/cg/ed/tools9sgnp/tools94/v2, Cap Gemini - Utrecht University, May 1994. Esprit III AFRODITE project.

[DHH+95] Roland Ducournau, Michel Habib, Marianne Huchard, Marie-Laure Mugnier, and Amedeo Napoli. Le point sur l'héritage multiple. *T.S.I*, 14(3):109–345, 1995.

[Dia92] Michel Diaz. Conception formelle des protocoles et des services dans les systèmes distribués. *Revue Génie Logiciel*, 26:57–69, March 1992.

[DLS87] Eric Dubois, Nicole Lévy, and Jeanine Souquières. Formalising Restructuring Operators in a Specification Process. Technical Report 2, Rapport Gréco de Programmation, CNRS, 1987.

[DM91] Pierre Dauchy and Béatrice Marre. Test data selection from the algebraic specification of a module of an automatic subway. Rapport de recherche 638, LRI, Orsay, January 1991.

[DS92] Roger Duke and Graeme Smith. Temporal Logic and Z Specifications. *Autralian Computer Journal*, 21(2):62–66, May 1992.

[DT88] Scott Danforth and C. Tomlison. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):–, March 1988.

[Duk90] Roger Duke. Object-Z. In *Tutorial of TOOLS PACIFIC '90*, 1990.

[EC90] Hartmut Ehrig and Ingo Classen. Overview of Algebraic Specification Languages, Environment and Tools, and Algebraic Specifications of Software Systems. *EATCS Bulletin*, 39, 40, 41, 1990. Part 1 in number 39, Part 2 in number 40, Part 3 in number 41.

[EHS93] Julian M. Edwards and Brian Henderson-Sellers. A graphical notation for object oriented and design. *JOOP*, pages 53–74, February 1993.

[EM85] Harmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, volume 6 of *EATCS Monograph on Theorical Computer Science*. Springer-verlag, 1985.

[EO94] Hartmut Ehrig and Fernando Orejas. Dynamic Abstract Data Types: An Informal Proposal. *EATCS Bulletin*, 53, June 1994.

[FGJM85] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985. SIGACT, SIGPLAN.

[FJ92] L. M. G. Feijs and F. B. M. Jonkers. *Formal Specification and Design*, volume 35 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[FKV94] Martin D. Fraser, Kuldeep Kumar, and Vijay K. Vaishnavi. Strategies for Incorporating Formal Specifications in Software Development. *Communications of the ACM*, 37(10):74–86, October 1994.

[FLSV90] Jean-Pierre Finance, Nicole Lévy, Jeanine Souquières, and Agnès Valdenaire. SACSO : un environnement d'aide à la spécification. *Technique et Science Informatique*, 9(3):245–261, 1990.

[Fuc92] Norbert E. Fuchs. Specifications Are (Preferably) Executable. *Software Engineering Journal*, September 1992.

[Gal87] Jean. H. Gallier. *Logic for Computer Science : Fondation of Automatic Theorem Proving.* J. Wiley & Sons, 1987.

[Gau90] Marie-Claude Gaudel. *Algebraic Specifications*, chapter 22, pages –. Software Engineers Reference Book. Butterworths, Mc Dermid, J. (Ed.), March 1990.

[GG88] Stephen J. Garland and John V. Guttag. Inductive Methods for Reasoning about Abstract Data Types. In *Fifteenth Annual ACM Symposiom on PoPL*, pages 219–228, San Diego, California, January 1988.

[GH78] J. V. Guttag and J. J. Horning. The algebraic Specification of Abstract Data Types. *Acta informatica*, 10:27–52, 1978.

[Gir91] Xavier Girod. *Conception par objets, MECANO : une Méthode et un Environnement de Construction d'ApplicatioNs par Objets.* Thèse d'université, Université Joseph Fourier (Grenoble I), June 1991.

[GLO91] Souheil Gallouzi, Luigi Logrippo, and Abdellatif Obaid. Le LOTOS : théorie, outils, applications. In *CFIP'91*, pages 385–404, 1991.

[GM85] Joseph A. Goguen and José Meseguer. Initiality, Induction and Computability. *in Algebraic Methods in Semantics, Maurice Nivat and John C. Reynolds, Eds. Cambridge University Press,*, pages 459–541, 1985.

[GM87a] Joseph A. Goguen and Jose. Meseguer. Ordered-Sorted Algebra Solves the Constructor-Selector, Multiple Representation, and Cœrcions Problems. In *IEEE*, pages 18–29, 1987.

[GM87b] Joseph A. Goguen and José Meseguer. *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics*, pages 417–477. Computer Systems Series. Research Directions in Object-Oriented Programming, 1987. Shriver, Bruce and Wegner, Peter (Eds).

[GM92] Carlo Ghezzi and Dino Mandrioli. *On Eclectic Specification Environments*, Advances in Object-Oriented Software Engineering 5, pages 115–146. Object-Oriented Series. Prentice Hall, Dino Mandrioli and Bertrand Meyer (Eds), 1992.

[Gog90] Joseph A. Goguen. An Algebraic Approach to Refinement. In *Proceedings of VDM'90, D. Bjorner, C.A.R. Hoare, H. Langsmaack (Eds)*, pages 12–28. Springer Verlag, 1990.

[Gri89] A. Griffault. *Contribution à l'étude des systèmes communicants et des algorithmes d'exclusion mutuelle.* PhD thesis, Université de Bordeaux I, Janvier 1989.

[GTE+90] Simon Gibbs, Dennis Tsichritzis, Casais Eduardo, Oscar Nierstrasz, and Xavier Pintado. Class Management for Software Communities. *Communication of the ACM*, 33(9):90–103, September 1990.

[GTW78] Joseph A. Goguen, James Thatcher, and E. Wagner. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In R. Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Englewood Cliffs, Prentice Hall, 1978.

[Gue94] Nicolas Guelfi. *Les réseaux algébriques hiérarchiques : un formalisme de spécifications structurées pour le développement de systèmes concurrents.* Phd thesis, Université Paris XI Orsay, September 1994.

[Gut80] John Guttag. Formal Specifications as a Design Tool. *ACM*, 1980.

[Hab93] Henri Habrias. *Introduction à la spécification*. Collection Méthodologies du logiciel. Editions Masson, 1993.

[Hal90] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–20, 1990.

[Hay92] Ian Hayes, editor. *Specification Case Studies*. C.A.R Hoare Series. Prentice-Hall International, 2 edition, 1992.

[HC91] Fiona Hayes and Derek Coleman. Coherent Model for Object-Oriented Analysis. In *Proceedings of OOPSLA'91*, pages 171–183, Phoenix Arizona, October 1991.

[HD89] Michel Habib and Roland Ducournau. La multiplicité de l'héritage dans les langages à objets. *T.S.I*, 8(1):41–61, 1989.

[HH82] Gérard Huet and Jean-Marie Hullot. Proofs by Induction in Equational Theories with Constructors. *Journal of Computer and System Sciences*, 25:239–266, 1982.

[HJW+92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Arvind, Brian Boutel, Jon Fairbain, Joseph Fasel, Maria M Guzman, Kevin Hammond, John Hughes, Thomas Jonhnsson, Richard Kierburtz, Rishiyur S Nikhil, Will Partain, and John Peterson. Report on the Functional Programming Languages Haskell, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[HLNP90] D. Harel, H. Lachover, A. Naomad, and A. Pnuelli. Statemate: a working environment for the development of complex reactive systems. *IEEE transactions on software engineering*, 16(4), April 1990.

[Hoa72] C.A.R. Hoare. Proof of correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. C.A.R Hoare Series. Prentice-Hall International, 1985.

[Hoa90] C.A.R. Hoare. preface of VDM'90. In D. Bjorner, C.A.R. Hoare, and H. Langsmaack, editors, *Proceedings of VDM'90*, Kiel, Germany, 1990. Springer Verlag.

[Hod91] Ralph Hodgson. The X-Model: A Process Model for Object-Oriented Software Development. In *Actes des quatrièmes journées internationales sur le Génie logiciel et ses applications*, pages 713–728, Toulouse, December 1991.

[HSE90] Brian Henderson-Sellers and Julian M. Edwards. The Object-Oriented Systems Life Cycle. *Communication of the ACM*, 33(9):143–159, September 1990.

[Hue87] Gérard Huet. A Uniform Approach to Type Theory. Rapport technique 12, Gréco de Programmation, CNRS, 1987.

[Huf89] Jean-Michel Hufflen. *Fonction et généricité dans un langage de programmation parallèle*. Phd thesis, Intitut National Polytechnique de Grenoble, July 1989.

[Hut94a] Andrew Hutt. *Object Analysis and Design, Comparison of Methods*. John Wiley & Sons eds., OMG edition, 1994.

[Hut94b] Andrew Hutt. *Object Analysis and Design, Description of Methods*. John Wiley & Sons eds., OMG edition, 1994.

[JCJO91] Ivar Jacobson, Magnus Christensen, Patrik Jonsson, and Gunmar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Prentice Hall, 1991.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *'mural': A Formal Development Support System.* Springer-Verlag, 1991. ISBN: 3-540-19651-X.

[JKKM89] Jean-Pierre Jouannaud, Claude Kirchner, Hélène Kirchner, and A. Megrelis. Programming with Equalities, subsorts, overloading and parameterization in OBJ. Technical Report 89-R-224, CRIN, 1989.

[JL86] Jean-Pierre Jouannaud and Pierre Lescanne. La réécriture. *Technique et Science Informatique*, 5(6):433–452, 1986.

[Joh86] Ralph E. Johnson. Type-Checking Smalltalk. In *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 315–321, Portland, OR, November 1986. ACM.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM.* C.A.R Hoare Series. Prentice-Hall International, 1986.

[Jon91] Cliff B. Jones. Does the OO-Community Need Formal Methods. In Bertand Meyer and Jean Bezivin, editors, *TOOLS'91*, pages 15–18, Paris, March 1991. Invited Lectures.

[Jon93] Cliff B Jones. *VDM, une méthode rigoureuse pour le développement du logiciel.* Masson, Paris, 2e edition, 1993.

[JS90] Cliff B. Jones and Roger C. Shaw. *Case Studies Systematic Software Development.* C.A.R Hoare Series. Prentice-Hall International, 1990.

[Jul83] Jacques Julliand. Spécification algébrique de la communication entre processus parallèles. *Technique et Science Informatique*, 3(4):257–269, 1983.

[Kah92] Gilles Kahn. Sémantique des langages de programmation. In *Seconde école des jeunes chercheurs*, Bordeaux, April 1992. Greco de Programmation.

[Kap86] Stéphane Kaplan. Simplifying Conditional Term Rewriting Systems: Unification, Termination and Confluence. Rapport de Recherche 316, Laboratoire de Recherche en Informatique d'Orsay, Université de Paris-Sud, Btiment 490, 91405 ORSAY, December 1986.

[KKM87] Claude Kirchner, Hélène Kirchner, and Jose Meseguer. Operational Semantic of OBJ-3. Technical Report 87-R-87, CRIN, 1987.

[KM90] Tim Korson and John D. McGregor. Understanding Object-Oriented : a Unifying Paradigm. *Communication of the ACM*, 33(9):40–60, September 1990.

[Kna94] Teodor Knapik. Concurrency and Real-time Specification with Many-Sorted Logic and Abstract Data Types: an Example. In *2nd African Conference on Research in Computer Science*, Ouagadougou, October 1994.

[KR90] Emmanuel Kounalis and Michael Rusinowitch. Automatic proof methods for algebraic specifications. In *10th International Conference on Fundamentals of Computation Theory*, volume 527 of *LNCS*, 1990.

[KR91] Samuel N. Kamin and Uday S. Reddy. Two Semantic Models of Object-Oriented Languages. Technical report, University of Illinois, Urbana-Champaign, 1991.

[KS90] Paul King and Graeme Smith. Formalisation of Behavioral and Structural Concepts for Communication Systems. *Protocol Specification, Testing and Verification*, 1990. L. Logrippo, R.L. Probert & H. Ural (Editors), Elsevier Science Publisher B.V. (North-Holland).

[Lau86] Jean.-L. Laurière. *Résolution de problèmes par l'Homme et la machine.* Eyrolles, 1986.

[LB79] Barbara H. Liskov and V. Berzins. An appraisal of program specifications. -, pages 276–301, 1979.

[LG90] Barbara Liskov and John Guttag. *La Maîtrise du Développement de Logiciel: abstraction et spécification.* Collection "Ingénierie des systèmes d'information". Les Editions d'Organisation, 1990.

[LH92] Kevin Lano and Howard Haughton. Reasoning and Refinement in Object-Oriented Specification Languages. In O. Lehrman Madsen, editor, *Proceedings of ECOOP '92*, volume 615 of *Lecture Notes in Computer Science*, pages 78–97. Springer Verlag, 1992.

[LH93] Kevin Lano and Howard Haughton, editors. *Object-Oriented Specification Case Studies.* Object Oriented Series. Prentice Hall, 1993.

[LH94] Kevin Lano and Howard Haughton. The Z++ Manual. Technical report, Imperial College, London, October 1994.

[LLS91] Christine Lafontaine, Yves Ledru, and Pierre-Yves Schobbens. An Experiment in Formal Software Development: using the B Theorem Prover on a VDM Case Study. *Communications of the ACM*, 34(5):62–71, May 1991.

[LM90] Jean-Louis Le Moigne. *La modélisation de systèmes complexes.* Dunod, 1990.

[LMV87] Valérie Lecompte, Eric Madelaine, and Didier Vergamini. AUTO : un système de vérification de processus parallèles et communicants. Technical Report 83, INRIA, March 1987. french.

[LP90] Wilf R. Lalonde and John R. Pugh. *Inside Smalltalk*, volume I and II. Prentice Hall International Editions, 1990.

[MC92] R. L. Meira, Silvio and Ana Lùcia C. Cavalcanti. The `MooZ` Specification Language. Technical Report ES/1.92, Universidade Federal de Pernambuco, Janeiro 1992. version 0.4.

[Mes90] Jose Meseguer. A Logical Theory of Concurrent Objects. In *Proceedings of OOPSLA '90*, Ottawa Canada, October 1990.

[Mey85] Bertrand Meyer. On Formalism in Specifications. *IEEE Software*, 2(1):6–26, January 1985.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall International, 1988.

[Mey89a] Bertrand Meyer. Static Typing For Eiffel. Technical report, Interactive Software Engineering Inc., July 1989.

[Mey89b] Bertrand Meyer. The New Culture of Software Development: Reflections on the Practice of OOD. In Jean Bezivin and Bertrand Meyer, editors, *Proceedings of TOOLS'89*, Paris, November 1989.

[Mey90a] Bertrand Meyer. *Introduction to the Théorie of Programming Languages.* C.A.R. Hoare Series in Computer Science. Prentice Hall International, 1990.

[Mey90b] Bertrand Meyer. Lessons from the Design of the Eiffel Libraries. *Communication of the ACM*, 33(9):68–88, September 1990.

[Mey92] Bertrand Meyer. *Design by Contract*, Advances in Object-Oriented Software Engineering 1, pages 1–50. Object-Oriented Series. Prentice Hall, Dino Mandrioli and Bertrand Meyer (Eds), 1992.

[Mil89] Robin Milner. *Communication and Concurrency*. C.A.R Hoare Series. Prentice-Hall International, 1989.

[MNC+90] Gérald Masini, Amedeo Napoli, D. Colnet, D. Léonard, and Karl Tombre. *Les langages à objets*. Collection IIA. InterEditions, 1990.

[MP92] David E. Monarchi and Gretchen I. Puhr. A Research Typology for Object-Oriented Analysis and Design. *Communications of the ACM*, 35(9):35–47, September 1992. Special Issue on Analysis and Modelling in Software Development.

[Mus80] David R. Musser. On proving Inductive Properties of Abstract Data Types. In *Proceedings of the Seventh Annual ACM Symposium on PoPL*, pages 154–162, Las Vegas, January 1980.

[Ner90] Jean-Marc Nerson. Case Studies in Object-Oriented Analysis. In Jean Bezivin and Bertrand Meyer, editors, *TOOLS'90*, June 1990. Tutorial.

[Ner92] Jean-Marc Nerson. Applying Object-Oriented Analysis and Design. *Communications of the ACM*, 35(9):63–74, September 1992. Special Issue on Analysis and Modelling in Software Development.

[OPS92] Nicholas Oxhoj, Jens Palsberg, and Michael I. Schartzbach. Making Type Inference Practical. *ECOOP'92 Proceedings, Springer Verlag*, 615:329–349, July 1992.

[Pal91] Mario Paludetto. *Sur la commande de procédés industriels : une méthodologie basée objets et réseaux de Petri*. Thèse de doctorat, Université Paul Sabatier, Toulouse, 10 décembre 1991 1991.

[PPP91] F. Parisi Presicce and A. Pierantonio. An Algebraic View of Inheritance and Sybtyping in Object Oriented Programming. In *3rd European Software Engineering Conference, ESEC'91*, volume 550 of *LNCS*, pages 364–379, Milan, Italy, 1991. A. van Lamsweerde and A. Gugetta (Eds.), Springer Verlag.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behavior Modelling with Automata. In Naftalin. Maurice, Tim Denvir, and Miquel Bertran, editors, *FME'94 Symposium*, volume 973 of *LNCS*, pages 306–325, Barcelona, Spain, October 1994. Springer Verlag.

[PS92] Jens Palsberg and Michael I. Schartzbach. Three Discussions on Object-Oriented Typing. *OOP Messenger*, 3(2):31–38, April 1992. Report on ECOOP'91 Workshop W5.

[PT93] Benjamin Pierce and David N. Turner. Simple Type-Checking Fondations For Object-Oriented Programming. *Journal of Functional Programming*, 1993.

[RB93] Arnold Rochfeld and Mokrane Bouzeghoub. From Merise to OOM. *Ingénierie des systèmes d'information*, 1(2):151–176, February 1993.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall International, 1991.

[RCCE92] Olivier Roux, Franck Cassez, Denis Creusot, and Jean-Pierre Elloy. Le langage réactif asynchrone Electre et ses systèmes de compilation et d'exécution. *Technique et Sciences Informatique*, 11(5), 1992.

[R.D91] R.D.Tennent. *Semantics of Programming Languages.* Prentice Hall International Series in Computer Science, 1991.

[Roc91] Arnold Rochfeld. Modèle Externe de Données et Modèle Externe Objet. In *Actes des quatrièmes journées internationales sur le Génie logiciel et ses applications*, pages 775–789, Toulouse, December 1991.

[Rol86] Colette Rolland. Introduction à la conception des systèmes d'information et panorama des méthodes disponibles. *Revue Génie Logiciel*, 4:7–62, June 1986.

[Roq92] Clément Roques. L'environnement ASSPEGIQUE+: Le valideur. L.R.I. Research Report 727, 1992.

[Ros77] D.T. Ross. Structured analysis (SA). A language for communicating ideas. *IEEE Trans. Software Engineering*, 3(1):16–34, 1977.

[Roy92] Jean-Claude Royer. A New Set Interpretation for the Inheritance Relation and its Checking. *ACM OOPS MESSENGER*, 3(3):22–40, July 1992. Rapport N^o 91-10 LIST, Université de Nantes.

[Roy93] Jean-Claude Royer. Un exercice de spécification formelle de preuve et de conception à objets. Rapport de recherche 30, IRIN, Faculté des Sciences et des Techniques, Université de Nantes, October 1993.

[Roy94] Jean-Claude Royer. Type Checking Formal Class. In *Rapport ERTO, Université de Nantes*, 1994.

[San90] Donald Sanella. A survey of formal software development methods. Technical Report ECS-LFCS-88-56, University of Edinburgh, Laboratory for Foundations of Computer Science, July 1990.

[SD92] Graeme Smith and Roger Duke. Specifying Concurrent Systems Using Object-Z. In *15th Autralian Computer Science Conference*, January 1992.

[SM92] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles.* Yourdon Press Computing. Prentice Hall, 1992. Modeling the World in States.

[SM93] Sally Shlaer and Stephen J. Mellor. A deeper look at the transition from analysis to design. *JOOP*, pages 16–21, February 1993.

[Smi93] Graeme Smith. A Fully-Abstract Semantics of Classes for Object-Z. *Formal Aspects of Computing*, 3(1), 1993. BCS.

[Som92] Ian Sommerville. *Le génie logiciel.* Addison-Wesley France, 4e edition, October 1992.

[Sou89] Jeannine Souquières. Quelques éléments d'un langage de Construction de Spécifications. Technical Report 11, Rapport Gréco de Programmation, CNRS, May 1989.

[Spi] Mike Spivey. A guide to the `zed` style option. free LaTeX style file - ftp <archive-management@comlab.ox.ac.uk>.

[Spi89] Mike Spivey. *The Z notation : a Reference Manual.* C.A.R Hoare Series. Prentice-Hall International, 1989.

[ST90] Donald Sanella and Andrewj Tarlecki. Algebraic Specification and Formal Methods for Program Development : what are the real problems? *EATCS bulletin*, 41:134–137, June 1990.

[Sun91] Young Sun. Equational Logics. In *BIGRE*, volume 74, pages 44–52, October 1991.

[Szy92] Clemens A. Szyperski. Import is Not Inheritance Why we need both: modules and classes. In Lehrman Madsen (Ed.), editor, *Proceedings of ECOOP '92*, volume 615 of *Lecture Notes in Computer Science*, pages 19–32. Springer Verlag, 1992.

[TRC83] Hubert Tardieu, Arnold Rochfeld, and R. Coletti. *La méthode Merise*. Editions d'Organisation, 1983. tomes 1, 2 et 3.

[Vau85] Jacques Vautherin. *Un modèle algébrique basé sur les réseaux de Petri, pour l'étude des systèmes parallèles*. These de docteur ingénieur, Université Paris XI, Orsay, 1985.

[Wal93] Kim Walden. O-O Analysis and Design. In *Tutorial T2 of TOOLS EUROPE'93*, Versailles, March 1993. International Conference & Exibition.

[WBJ90] Rebecca J. Wirfs-Brock and E. Johnson, Ralph. Surveying Current Research in Object-Oriented Design. *Communication of the ACM*, 33(9):104–124, September 1990.

[WBWW90] Rebecca Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[Weg90] Peter Wegner. Concepts and paradigms of OOP. *OOP Messenger*, 1(1), August 1990. ACM.

[Wil91] Alan Wills. Capsules and types in Fresco : Smalltalk meets VDM,. In Pierre America, editor, *Proceedings of ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*. ECOOP '91,, Springer Verlag, 1991. Geneva, Switzerland.

[Wil93] Lloyd G. Williams. Integrating Formal Methods and Object-Oriented Development. Technical Report SERM-020-93, Software Engineering Research, Boulder, Colorado USA, December 1993.

[Wir93] Martin Wirsing. Développement de logiciel et spécification formelle. *Technique et Science Informatique*, 12(4):413–431, 1993.

[WL88] J. Woodcock and M. Loomes. *Software Engineering Mathematics*. Addison Wesley, 1988.

[WL93] Pierre Weis and Xavier Leroy. *Le langage CAML*. Collection IIA. InterEditions, 1993.

[1]

[1] Les outils utilisés, notamment LaTeXet BibTex, ne m'ont pas permis de créer une bibliographie qui soit automatique et par thème. Veuillez m'en excuser.

Index

équation, 194
    Σ-équation, 194
état, 101
    état composite, 120
    destination, 101
    final, 101
    origine, 101
    super-état, 120
état, 28

abstraction, 9, 91
    niveau d', 11, 90
abstrait, 99
actualisation, 101, 126
algèbre
    des termes, 192
    finale, 195
    finiment générée, 193
    initiale, 195
    multi-sorte, 25
    partielle, 197
    triviale, 192
application, 9, 13, 17, 31, 56
approche descendante, 8
arité, 191
aspect, 129, 131, 132
    multi-aspect, 155
association, 29
attribut, 28
automate, 92, 101
    minimal, 117
axiome
    équation conditionnelle positive, 194

bisimulation, 64
booléen, 194

cœrcition, 140
capsule, 75
caractéristique, 130
catégorie
    théorie des, 195
classe, 28, 131
    abstraite, 29, 31, 132, 135, 138, 162
    bien définie, 136
    concrète, 91, 95
    formelle, 91, 95, 131, 132
    habitée, 135
cluster, 24, 31
co-domaine, 191
cohérence, 10, 118, 133, 148
    hiérarchique, 104, 196
complétude, 118
    suffisante, 104, 196
comportement, 28, 130, 131, 141, 162
    complet, 141
    dynamique, 30, 101, 125
composant, 12, 14, 17, 32, 33, 38
    logiciel, 14, 37
conception
    ordonnée, 96, 143, 150, 154, 161
    plate, 96, 143, 150, 159
concrétisation, 10, 11
condition de monotonicité, 191
confluence, 118
confusion, 193
constante, 101, 191
constructeur, 92, 100
    de base, 100
contrainte, 101, 131, 132
convergence, 118
covariance, 142
    covariant, contravariant, novariant, 148
    multi-covariance, 148
    simple covariance, 129, 143, 148
cycle de vie, 7, 8

dépendance, 109
dépendance structurelle, 134
développement, 5
développement du logiciel, 5
dictionnaire
    de méthodes, 133
domaine, 31, 100, 191
    codomaine, 100

encapsulation, 28
envoi de message, 28
extension, 111, 130, 131, 133
    G-dérivée, 111, 125

opérationnelle, 111

famille, 191
génératrice, 107
fonction, 191, 192
fonction d'abstraction, 123, 125, 169, 170

G-dérivation, 94, 111
générateur, 107, 193
mono-, 129
multi-, 153
génie logiciel, 6, 14
garde, 101

héritage, 29, 132, 140
critère d', 142
de structure, 140
extension, 123, 125
héritage dynamique, 161
hypothèses TAG, 102

identité d'objet, 29
implantation, 10
induction, 135
instance, 28, 131
instanciation, 28, 130
interface, 104
interprétation
fonction de, 193
invariant, 101, 106, 133

langage, 192
engendré, 104
logiciel, 5

métaclasse, 31
méthode, 6, 28, 130, 133
virtuelle, 138
abstraite, 29, 138
cartésienne, 6
d'instance, 28, 131
définie, 141
de classe, 28, 131
formelle, 15
à objets, 43, 89
héritée, 141
multi-méthodes, 155
primitive, 130
privée, 154
protégée, 154
sélecteur, 133
secondaire, 130, 137
systémique, 6
virtuelle, 29
visible, 141
méthodologie, 6
maquettage, 8
modèle, 5, 6, 129
classe de, 60
d'interprétation, 192
de représentation, 6
dynamique, 92
formel à objets, 90
modularité, 12
modularité, 12
module, 12, 31, 52
de spécification, 12
morphisme
Σ-extension, 196
Σ-morphisme, 192
Σ-réduction, 196
isomorphisme, 192
S-morphisme, 191, 192

noethérien, 118
noyau d'une spécification, 107

objet, 28, 130
actif, 30
passif, 30
observateur, 92, 100
primaire, 107
opération, 100, 191, 192
extension, 111, 123
neutre, 105
partielle, 104, 105
primaire, 107
profil de, 100
secondaire, 107
totale, 105

paramètre
formel constant, 101
formel de type, 101
paramètre de type, 101
polymorphisme, 148, 197
précondition, 106, 132
prédicat, 194
de définition, 110, 197
prédicat de définition, 104
présentation, 194
G-dérivée, 111
primaire, 107
secondaire, 107
TAG, 103
présentation sûre, 153
processus de développement, 6, 7
profil d'opération, 191
projection structurelle, 140

promotion, 86, 158
prototypage, 8
prototype, 179

qualité, 12
  du processus de développement, 13

réécriture, 118
  conditionnelle, 145
  systèmes de, 118
réalisation, 15
rétraction, 150
raffinement, 10, 11, 30, 158, 159
raisonnement équationnel, 118
rebut, 193
receveur, 28
redéfinition, 129, 137, 142, 148, 151
  sémantique voisine, 143
  simple covariance, 148
  systématique, 141
relation
  complémentaire, 9
  d'agrégation, 29
  d'héritage, 29
  d'implantation, 10
  d'importation, 9, 102
  d'inclusion, 9
  d'utilisation, 29
  de clientèle, 29
  de concrétisation, 10
  de dépendance, 102
  de raffinement, 10
  de spécialisation/généralisation, 9, 29
  horizontale, 9, 29
  tout-partie, 29
  verticale, 9, 29
relaxation de type, 166
représentation, 5, 136, 143, 169
restructuration, 168

sélecteur, 28, 130, 132, 133, 141
  d'état, 132, 166, 170
  de champ, 131, 132
  sel, 133
sélection
  mono-, 28
  multi-, 28
  multiple, 28, 155
  simple, 28, 129, 146, 151
sémantique
  classe de modèles, 195
  des langages, 45
  initiale, 195
  opérationnelle, 145
satisfaction, 194
schéma, 31, 61, 96, 154
  initial, 161
service, 28
signature, 104, 191
  cohérente, 193
  complète, 193
  raisonnable, 104, 117, 193
sorte, 191
  habitée, 117, 194
sous-aspect, 166
sous-classe, 29
sous-spécification, 123
sous-structure, 66
sous-système, 31
sous-typage, 29, 30, 123, 141, 148, 197
spécification, 8, 9
  étape de spécification formelle, 15
  abstraite, 102
  du logiciel, 8
  formelle, 15, 18
  générique, 101
spécification algébrique, 134
  de base, 194
  hiérarchique, 196
  ordonnée, 96, 197
  partie primitive de la, 196
  plate, 61, 96, 195
  présentation, 103, 194
spécification TAG, 103
stratégie de développement, 11
structuration, 9, 166
structure abstraite, 131
substitution, 148
  principe de, 123
sujet, 31
super-spécification, 123
support
  ensemble, 192
système d'information, 5
système formel, 25

terme, 94, 104, 110, 192
  clos, 192
  erroné, 105
terme douteux, 150
terminaison, 118
théorie, 53
traduction, 180
trait
  de contrôle, 59
transition, 101, 120, 121
typage, 119, 147
type, 13

d'intérêt, 99, 103
de base, 13, 130, 146
de données, 13
dynamique, 146
fonction de, 192
type abstrait
algébrique, 25, 93, 129
de données, 13
graphique TAG, 91, 92, 99
type concret
de données, 13
type structurant, 132

validation, 8
valuation, 193
variables d'instance, 28, 132
vecteur caractéristique, 135
vue, 53

Résumé : Les spécifications formelles et le modèle à objets sont considérés comme deux champs ayant un fort potentiel d'influence sur l'avenir du génie logiciel. Dans un premier temps, nous étudions cette affirmation en dégageant les caractéristiques essentielles du développement du logiciel. L'intersection des deux champs est un domaine nouveau et prometteur. Nous en étudions les alternatives et nous synthétisons les choix faits dans les méthodes formelles à objets actuelles. Dans ce contexte, cette thèse propose une méthode de spécification et de conception de composants logiciels destinée à faciliter la formalisation, automatiser partiellement le processus de conception, permettre le contrôle et la réutilisation. L'accent est mis sur la séparation entre les niveaux de description pour clarifier le processus de conception et sur l'uniformisation à l'intérieur des niveaux de description pour assurer la cohérence. Deux modèles sont décrits. Le premier modèle, dit des types abstraits graphiques, définit les composants suivant un axe dynamique et suivant un axe fonctionnel. La modélisation dynamique, naturelle pour des objets est donnée en termes d'automates d'états finis gardé. Outre son accessibilité à un utilisateur non spécialiste, le modèle dynamique a pour intérêt majeur la construction guidée de spécifications algébriques, support de l'axe fonctionnel. Le second modèle, dit des classes formelles, est un modèle général, formel et abstrait pour la conception à objets. Basé sur les types abstraits algébriques, il permet le raisonnement abstrait et l'implantation dans différents langages de programmation à objets. Les modèles présentés sont indépendants et sont adaptables dans d'autres méthodes de développement. Nous proposons une méthode de transition entre ces deux modèles, qui favorise le contrôle et la réutilisation des spécifications. Plusieurs outils d'écriture et de preuve sont communs aux deux modèles et nous favorisons l'ouverture vers d'autres environnements de spécification.

Mots-clés : spécification formelle, analyse et conception, modèle à objets, type abstrait graphique, classe formelle, spécifications algébriques.

Abstract: Formal methods and object orientation are widely regarded as two fields with significant potential for influence on the future of software engineering. We first study this statement by isolating the essential features of the software development. The intersection of these two fields is a new and promising domain. We study its issues and synthesize the choices adopted in the current object-oriented formal methodes. In this context, this thesis proposes a software component specification and design method, aimed to facilitate formalization, partially automate the design process and enable control and reuse. The separation between the description levels and the uniformity inside the levels are emphasized first to clarify the design process and second to ensure consistency. Two models are described. The first one, called the graphic abstract type model, defines the components through a dynamic and a functional aspect. The dynamic modelling is natural for objects and expressed through guarded finite state machines. Beyond its accessiblility to non-expert users, the main interest of the dynamic model is to support the construction of the algebraic specifications, which constitutes the functional axis. The second model, called the formal class model, is a general, formal and abstract model for object oriented design. Based on algebraic data types, it permits abstract reasoning and the implementation in several object oriented programming languages. The proposed model are independent and adaptable in other development methods. We propose a transition methods between these models, that enhances specification control and reuse. Several writing and proof tools are common to both models. It has also been kept in mind to open our system to other specification environments.

Keywords: formal specification, analysis and design, object model, graphic abstract type, formal class, algebraic specifications.