Conception d'algorithmes répartis et de protocoles réseaux en approche objet

Lionel Seinturier

▶ To cite this version:

Lionel Seinturier. Conception d'algorithmes répartis et de protocoles réseaux en approche objet. Réseaux et télécommunications [cs.NI]. Conservatoire National des Arts et Métiers - CNAM Paris, 1997. Français. NNT : . tel-00439136

HAL Id: tel-00439136
https://theses.hal.science/tel-00439136

Submitted on 6 Dec 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Conservatoire National des Arts et Métiers

- Paris -

THESE DE DOCTORAT

Spécialité : Informatique

présentée
par Lionel SEINTURIER

pour obtenir le grade de :
Docteur du Conservatoire National des Arts et Métiers

CONCEPTION D'ALGORITHMES REPARTIS ET DE PROTOCOLES RESEAUX EN APPROCHE OBJET

soutenue le 3 décembre 1997 devant le jury :

Président	Mr	C. Kaiser
Rapporteurs	Mr	E. Najm
	Mr	M. Riveill
Directeur	Mr	G. Florin
Co-Directeur	Mme	L. Duchien
Examinateurs	Mr	C. Carrez
	Mr	P. Estraillier
	Mr	I. Lavallée

Remerciements

Le travail présenté dans ce document a été réalisé au sein du laboratoire CEDRIC (Centre d'Etudes et de Recherche en Informatique du CNAM) sous la direction de Monsieur Gérard Florin, Professeur des Universités au CNAM, et la co-direction de Madame Laurence Duchien, Maître de Conférences au CNAM.

Je tiens à exprimer à Monsieur Gérard Florin, Professeur des Universités au CNAM, toute ma gratitude pour m'avoir accueilli au sein du laboratoire qu'il dirige. Je tiens également à le remercier pour les conseils qu'il m'a prodigués et pour les nombreuses corrections qu'il a bien voulu apporter à mon travail, me permettant ainsi d'effectuer cette thèse dans les meilleures conditions. Qu'il sache que j'ai beaucoup appris sous sa direction.

Je tiens à exprimer toute ma reconnaissance à Madame Laurence Duchien, Maître de Conférences au CNAM, pour l'aide précieuse qu'elle m'a apportée, ainsi que pour le soutien et la compréhension qu'elle a montrés au cours de nos discussions. Sa disponibilité, son aide aux moments opportuns et ses nombreuses relectures et corrections, ont été essentielles à la réalisation de cette thèse.

Je remercie Monsieur Claude Kaiser, Professeur du CNAM, titulaire de la chaire Informatique-Programmation et responsable de l'équipe Systèmes Répartis Tolérant les Pannes du CEDRIC, de m'avoir accueilli au sein de son équipe, et de me faire l'honneur de présider le jury de cette thèse.

Je remercie Monsieur Michel Riveill, Professeur à l'université de Savoie, et Monsieur Elie Najm, Maître de Conférences à l'ENST, pour avoir accepté avec tant d'enthousiasme d'être rapporteurs de cette thèse.

Je remercie Monsieur Pascal Estraillier, Professeur à l'université Pierre et Marie Curie, d'avoir accepté de juger cette thèse en siégeant à ce jury, malgré ses nombreuses occupations et responsabilités.

Je remercie Monsieur Christian Carrez, Professeur des Universités au CNAM, pour sa présence parmi ce jury, ainsi que pour l'intérêt qu'il porte à ce travail.

Mes remerciements les plus vifs à Monsieur Ivan Lavallée, Professeur à l'université Paris VIII, pour avoir accepté de participer à ce jury.

Je remercie Monsieur Daniel Enselme, Maître de Conférences au CNAM, et Madame María-Virginia Aponte, Maître de Conférences au CNAM, pour le travail ingrat de relecture des premières épreuves de ce document.

Je remercie toute l'équipe du CEDRIC, et en particulier Eric Gressier, pour leurs conseils et leurs nombreux encouragements. Je tiens également à exprimer à Viviane Gal toute ma reconnaissance pour sa disponibilité et les nombreuses informations qu'elle m'a fournies sur les arcanes administratives du CNAM.

Enfin, j'ai une pensée amicale pour l'ensemble de mes collègues de bureau, passés et présents, avec lesquels j'ai partagé ces années de thèse. A tout seigneur, tout honneur, merci donc à Jean-Paul, Christian, Laurent, Pascal, Jean-Marc, Thierry, Ludovic et aux deux Frédéric Weis et Bergdolt.

Résumé

Dans cette thèse, nous nous intéressons à la conception d'algorithmes répartis, de protocoles réseaux et d'applications coopératives avec une approche objet. Cette activité, souvent complexe, nécessite la mise en œuvre de nombreux mécanismes systèmes et réseaux. Nous proposons donc différents outils et formalismes permettant de mener à bien cette tâche.

L'originalité de notre approche est, d'une part, d'étendre le point de vue local des méthodologies de conception existantes afin d'intégrer les comportements de groupes d'objets distribués, et d'autre part, de proposer les premiers éléments d'une démarche systématique d'algorithmique répartie. Ainsi, nous proposons un processus de développement en trois niveaux méthodologiques : groupe, objet et méthode. Ce processus de type descendant permet d'introduire, par raffinements successifs, de plus en plus de détails dans les modèles de comportements. Ces trois niveaux traitent, respectivement, des aspects liés à la distribution, à la concurrence et aux traitements séquentiels. Dans cette thèse, nous nous intéressons aux deux premiers niveaux.

Le niveau groupe concerne la coordination des comportements de groupes d'objets distribués. Nous modélisons les interactions au sein de tels groupes en terme d'échanges de connaissances. Ceux-ci peuvent être vus comme la conséquence d'actions globales entreprises par l'ensemble des objets du groupe. Nous avons mis en évidence quatre actions globales, ou structures de contrôle de groupe, qui apparaissent dans de nombreuses applications distribuées : le schéma de phasage, la conditionnelle distribuée, l'itération distribuée et la récursion distribuée. Elles peuvent être vues comme l'extension, à un niveau réparti, des structures algorithmiques de base que sont, respectivement, la séquence, les instructions de type *case* ou *if*, les boucles *while* et le parcours récursif. Nous utilisons une logique épistémique pour décrire les différents niveaux de connaissance atteints lors de l'exécution de ces structures. Nous proposons une notation appelée programme à base de connaissances de niveau groupe pour exprimer les actions globales et les prédicats épistémiques utilisés par une application.

Le niveau objet s'intéresse à la coordination des comportements internes à un objet. C'est un raffinement du niveau précédent, au sens où la coordination inter-objets est implantée par des objets dont les activités concurrentes nécessitent une synchronisation. Pour mener à bien la description de cette coordination intra-objet, nous proposons le langage CAOLAC. Nous en avons réalisé une implantation au-dessus du langage objet du système distribué GUIDE. Le langage CAOLAC se présente sous la forme d'un protocole méta-objet et sépare les aspects de synchronisation des traitements effectifs. Les premiers sont définis dans des méta-classes, tandis que les seconds le sont dans des classes. L'originalité du langage CAOLAC est d'utiliser conjointement un modèle à base d'états et de transitions et du code objet habituel pour l'écriture des méta-classes. Ainsi, chaque objet est associé à un méta-objet qui intercepte les invocations de méthodes et les coordonne avant de les délivrer à l'objet. L'avantage de cette approche est de séparer clairement les différents fonctionnalités et de faciliter la réutilisation des politiques de synchronisation. La sémantique du langage CAOLAC est définie, partiellement, par une logique temporelle, la logique temporelle d'actions de Lamport.

Enfin, nous illustrons notre propos par deux études de cas. Nous présentons la conception d'un algorithme réparti de calcul d'arbres couvrants et d'un protocole transactionnel de validation à deux phases.

Mots-clés : objets distribués, coordination inter-objets, logique épistémique, structures de contrôle distribuées, coordination intra-objet, langage CAOLAC, protocoles méta-objets, modèles états/transitions, système distribué GUIDE

Abstract

This thesis deals with the design of distributed algorithms, network protocols, and cooperative applications with an object-oriented approach. This process requires the use of numerous system and network software components. We propose several tools and formalisms to carry out this activity.

The originality of our approach is, first, to extend the local point of view of existing object-oriented methods in order to integrate the behaviors of groups of distributed objects, and second, to set up a distributed programming language. Thus, we propose a development process with three methodological levels: group, object, and method. This top-down process introduces, with successive refinement steps, more and more details in the behavioral models. These three levels deals, respectively, with distribution, concurrency, and sequentiality. In this thesis, we are interested in the first two levels.

The group level deals with the inter-objects coordination of distributed objects. The interactions inside such groups are defined in terms of knowledge exchanges. They can be seen as the consequences of the run of some global actions performed by all the objects of the group. We put forward four global actions, called distributed control structures, frequently used in distributed applications: the phase, the distributed condition, the distributed iteration, and the distributed recursion. They can be seen as the extension, at a distributed level, of the four following basic algorithmic statements: the sequence, the *case* and the *if* statements, the *while* loop, and the back-track traversal. We use an epistemic logic to describe the knowledge levels reached by the run of these structures. We propose a syntax, called group level knowledge based program, to describe the global actions and the epistemic predicates used by an application.

The object level deals with intra-object coordination. According to the fact that inter-objects coordination is performed by objects whose concurrent activities need to be synchronized, the object level is a refinement of the group level. To carry out the description of intra-object coordination, we propose the CAOLAC language. We implemented it on top of the object language of the GUIDE distributed system. The CAOLAC language is a meta-object protocol that separates synchronization tasks from sequential ones. The former are defined in meta-classes, and the latter in classes. The originality of the CAOLAC language is to use both some state/transition models and some usual object-oriented code to write meta-classes. Each object is associated with a meta-object that traps method calls and that performs some coordination before delivering them to the object. The advantage of this approach is to clearly separate the different functionalities, and to facilitate the reuse of synchronization policies. The semantics of the CAOLAC language is partially defined with Lamport temporal logic of actions.

We illustrate our approach with two examples. We present the design of a spanning tree construction distributed algorithm, and of a transactional two phases commit protocol.

Key words : distributed objects, inter-objects coordination, epistemic logic, distributed control structures, intra-object coordination, CAOLAC language, meta-object protocols, state/transition models, GUIDE distributed system

Table des matières

Introduction 1

I Etat de l'art 7

1 Approche objet 7
1.1 Les langages orientés objet 8
1.1.1 Introduction 8
1.1.2 Concurrence 9
1.1.3 Synchronisation 11
1.1.3.1 Sémaphores 12
1.1.3.2 Moniteurs 12
1.1.3.3 Expressions de chemin 12
1.1.3.4 Commandes gardées 13
1.1.3.5 Utilisation de modèles états/transitions 14
1.1.3.6 Approches réflexives 15
1.1.3.7 Conclusion sur la synchronisation 16
1.1.4 Relations de réutilisation 17
1.1.4.1 Héritage 17
1.1.4.2 Composition 19
1.1.4.3 Délégation 20
1.1.4.4 Classes paramétrées 20
1.1.4.5 Conclusion sur les relations de réutilisation 20
1.1.5 Héritage et synchronisation 21
1.1.5.1 Historique des invocations de méthodes 21
1.1.5.2 Partitionnement des états acceptables 22
1.1.5.3 Modification des états acceptables 22
1.1.5.4 Conclusion sur l'héritage et la synchronisation 22
1.1.6 Conclusion sur les langages orientés objet 23
1.2 Environnements pour les objets distribués 23
1.2.1 Introduction 24
1.2.2 CORBA 25
1.2.2.1 Architecture 25
1.2.2.2 Langage de définition d'interface 26
1.2.2.3 Protocoles d'interopérabilité 27
1.2.3 Conclusion sur les environnements pour les objets distribués 27

1.3 Méthodologies de conception orientées objet 28
1.3.1 Analyse et conception orientées objet 29
1.3.2 La méthode Booch 30
1.3.2.1 Notation 30
1.3.2.2 Processus de développement 33
1.3.3 Comparaison avec d'autres méthodologies 33
1.3.3.1 OMT 33
1.3.3.2 UML 34
1.3.4 Conclusion sur les méthodologies orientées objet 37
1.4 Conclusion sur l'approche objet 38

2 Logiques pour l'approche objets répartis 41
2.1 Logique modale 42
2.2 Logique temporelle 42
2.2.1 Différentes catégories de logiques temporelles 42
2.2.2 TLA 46
2.2.3 Conclusion sur la logique temporelle 50
2.3 Logique épistémique 51
2.3.1 Connaissance 51
2.3.2 Différents degrés de connaissance 52
2.3.3 Croyance 55
2.3.4 Conclusion sur la logique épistémique 56
2.4 Conclusion sur la logique modale 57

3 Programmation à base de connaissances 59
3.1 Notion d'état global d'un système réparti 60
3.1.1 Relation de causalité (arrive avant ou happened before) 60
3.1.2 Estampillage des événements 61
3.1.3 Notion de coupe cohérente 62
3.1.4 Conclusion sur la notion d'état global 63
3.2 Logique épistémique et temporelle 64
3.2.1 Définition 65
3.2.2 Sémantique 65
3.3 Programmes à base de connaissance de niveau agent 66
3.3.1 Définition 67
3.3.2 Exemple 69
3.3.3 Raffinement 69
3.3.4 Conclusion sur les programmes de niveau agent 71
3.4 Conclusion sur la programmation à base de connaissances 71

II Coordination inter-objets 73

4 Conception de comportements de groupe 73
4.1 Processus de développement 74
4.1.1 Niveau groupe 75

4.1.2 Niveau objet 76
4.1.3 Niveau méthode 77
4.2 Comportement de groupe 77
4.3 Structure de données de groupe 78
4.4 Opérateurs de connaissance de groupe 80
4.4.1 Opérateurs retenus 80
4.4.2 Notation 82
4.4.3 Gradation des niveaux de connaissance 83
4.4.4 Niveaux de connaissance et structures de données réparties 84
4.5 Programmes à base de connaissances de niveau groupe 85
4.5.1 Définition 85
4.5.2 Notation 86
4.5.3 Exemple 86
4.5.4 Raffinement 89
4.5.5 Conclusion sur les programmes de niveau groupe 92
4.6 Conclusion sur la conception de niveau groupe 92

5 Gabarits de conception de niveau groupe 95
5.1 Le schéma de phasage 96
5.1.1 Dimension temporelle 96
5.1.2 Dimension épistémique 97
5.1.2.1 Variables et groupes 97
5.1.2.2 Statut des variables 98
5.1.2.3 Mondes possibles 98
5.1.3 Approche observationnelle 99
5.1.4 Spécialisation 101
5.1.5 Composition 101
5.1.5.1 Séquence 101
5.1.5.2 Constructeur de parallélisme 101
5.1.5.3 Phases gardées 103
5.2 La conditionnelle distribuée 104
5.2.1 Définition 104
5.2.2 Raffinement 105
5.2.2.1 Structure if then else 105
5.2.2.2 Schéma transactionnel 105
5.3 L'itération distribuée 106
5.3.1 Définition 107
5.3.2 Raffinement 107
5.3.2.1 Itération synchrone 108
5.3.2.2 Itération asynchrone 108
5.4 La récursion distribuée 108
5.4.1 Définition 109
5.4.1.1 Description informelle 109
5.4.1.2 Programme de niveau groupe 109
5.4.1.3 Description du programme 111

5.4.2 Raffinement . . . 112
5.4.2.1 Raffinement de niveau groupe . . . 112
5.4.2.2 Raffinement de niveau objet . . . 112
5.5 Conclusion sur les gabarits de niveau groupe . . . 114

III Coordination intra-objets 115

6 Langage CAOLAC 115
6.1 Introduction . . . 116
6.1.1 Présentation de l'approche . . . 116
6.1.2 Comparaison avec d'autres approches . . . 117
6.2 Concepts de base . . . 118
6.2.1 Définition d'un comportement . . . 118
6.2.2 Exemple de comportement . . . 119
6.2.3 Relation méta-comportementale . . . 119
6.2.3.1 Association avec une classe . . . 119
6.2.3.2 Tour méta . . . 121
6.3 Comportements . . . 122
6.3.1 Variables . . . 122
6.3.2 Interfaces . . . 123
6.3.3 Etats . . . 123
6.3.3.1 Représentation des états . . . 123
6.3.3.2 Sémantiques d'état . . . 125
6.3.4 Transitions . . . 127
6.3.4.1 Définition . . . 128
6.3.4.2 Sémantique . . . 128
6.3.5 Désignation des éléments d'un comportement . . . 129
6.3.6 Compteurs et historiques d'événements . . . 129
6.3.6.1 Compteurs d'événements . . . 129
6.3.6.2 Historiques d'événements . . . 130
6.3.6.3 Attendus sur les compteurs et les historiques d'événements . . . 131
6.3.6.4 Exemple d'utilisation . . . 131
6.3.7 Héritage de comportements . . . 134
6.3.7.1 Définition . . . 135
6.3.7.2 Héritage d'états . . . 135
6.3.7.3 Héritage de transitions . . . 137
6.3.7.4 Anomalie d'héritage . . . 137
6.4 Evaluation du langage CAOLAC . . . 141
6.4.1 Approche états/transitions . . . 141
6.4.2 Approche compteurs d'événements . . . 142
6.4.3 Résumé . . . 144
6.5 Implantation du compilateur CAOLAC . . . 144
6.5.1 Volume de développement . . . 144
6.5.2 Chaîne de production . . . 145

6.5.3 Performances . . . 145
6.5.4 Présentation du compilateur . . . 146
6.5.4.1 Analyse lexicale . . . 146
6.5.4.2 Analyse syntaxique . . . 146
6.5.4.3 Analyse sémantique . . . 147
6.5.4.4 Génération du code GUIDE . . . 148
6.5.4.5 Librairie de composants . . . 149
6.6 Conclusion . . . 150

7 Sémantique du modèle de synchronisation 153
7.1 Eléments de base . . . 154
7.1.1 Comportement . . . 154
7.1.2 Variables propositionnelles . . . 155
7.2 Sémantiques d'état . . . 157
7.2.1 Sémantiques entrantes . . . 158
7.2.2 Sémantiques sortantes . . . 158
7.2.2.1 Séquentielle . . . 158
7.2.2.2 Parallèle . . . 159
7.2.2.3 Tant que . . . 159
7.3 Modèle états/transitions . . . 160
7.3.1 Formule générale . . . 160
7.3.2 Exemple . . . 162
7.3.2.1 Comportement buffer synchronisé . . . 162
7.3.2.2 Raffinement . . . 162
7.3.2.3 Eléments de preuve . . . 164
7.4 Conclusion . . . 167

IV Etudes de cas 169

8 Calcul d'arbres couvrants 169
8.1 Rappel de la démarche . . . 169
8.2 Présentation des algorithmes . . . 170
8.2.1 Algorithmes avec un seul initiateur . . . 170
8.2.2 Algorithmes avec plusieurs initiateurs . . . 171
8.2.2.1 Version à vagues inondantes . . . 171
8.2.2.2 Version à régions . . . 173
8.2.2.3 Version par régions et retournement de branches . . . 173
8.3 Spécification de niveau groupe . . . 175
8.3.1 Hypothèses . . . 175
8.3.2 But global . . . 177
8.3.3 Raffinement du but global . . . 178
8.3.4 Premier programme de niveau groupe . . . 178
8.3.5 Premier raffinement du programme de niveau groupe . . . 178
8.3.6 Deuxième raffinement du programme de niveau groupe . . . 180
8.3.6.1 Méthode *Terminer* . . . 180

8.3.6.2 Méthode *Construire* . . . 180
8.3.7 Résumé . . . 183
8.4 Spécification de niveau objet . . . 183
8.4.1 Variables . . . 183
8.4.2 Invocations . . . 185
8.4.3 Comportement CAOLAC . . . 185
8.4.4 Synchronisations additionnelles pour la phase de construction . . . 187
8.4.4.1 Méta-niveau 2 : contrôle des numéros d'époque . . . 187
8.4.4.2 Méta-niveau 1 : contrôle des initiateurs de vagues . . . 188
8.4.5 Résumé . . . 189
8.5 Spécification de niveau méthode . . . 190
8.6 Conclusion . . . 191

9 Protocole transactionnel **193**
9.1 Présentation du protocole . . . 193
9.1.1 Protocoles de validation atomique . . . 193
9.1.2 Protocole de validation à deux phases . . . 194
9.2 Spécification de niveau groupe . . . 194
9.2.1 Hypothèses . . . 194
9.2.2 But global . . . 195
9.2.3 Programme de niveau groupe . . . 195
9.2.4 Modèle états/transitions . . . 195
9.3 Spécification de niveau objet . . . 196
9.4 Spécification de niveau méthode . . . 198
9.5 Conclusion . . . 200

Conclusion **201**

Annexe **209**

A Langage CAOLAC **209**
A.1 Présentation du langage CAOLAC . . . 209
A.1.1 Définition de comportements . . . 209
A.1.1.1 Concepts de base . . . 209
A.1.1.2 Association entre un comportement CAOLAC et une classe GUIDE 212
A.1.1.3 Fonctionnement d'un comportement . . . 213
A.1.1.4 Concepts avancés . . . 215
A.1.1.4.1 Réutilisation de comportements . . . 216
A.1.1.4.2 Invocations génériques . . . 216
A.1.1.4.3 Compteurs d'événements . . . 218
A.1.1.4.4 Historiques d'événements . . . 218
A.1.2 Syntaxe . . . 219
A.1.2.1 Eléments lexicaux . . . 219
A.1.2.2 BNF . . . 220

A.2 Traduction du langage CAOLAC en GUIDE 224
A.2.1 Traduction des éléments de base 224
A.2.2 Traduction des sémantiques d'états 232
A.2.2.1 Sémantique sortante 232
A.2.2.2 Sémantique entrante 232
A.2.3 Classes auxiliaires pour la traduction CAOLAC vers GUIDE 239

Bibliographie **241**

Table des figures

1.1 Classe tampon synchronisé dans le langage ACT++ . . . 15
1.2 Sous-typage et héritage . . . 19
1.3 Partitionnement et modification des états acceptables . . . 22
1.4 Architecture d'un environnement selon la norme CORBA . . . 26

2.1 Deux interprétations de la modalité $\Diamond$ dans une logique du temps ramifié . . . 45
2.2 Forme algorithmique et formule TLA du programme Φ . . . 48
2.3 Interprétation en terme de mondes possibles de la notion de connaissance . . . 52
2.4 Interprétation en terme de mondes possibles de la notion de connaissance distribuée 53
2.5 Interprétation en terme de mondes possibles de la notion de connaissance de tous . 54
2.6 Interprétation en terme de mondes possibles de la notion de connaissance commune 55

3.1 Relations de dépendance et histoire causale . . . 61
3.2 Coupes non cohérente et cohérente . . . 63
3.3 Treillis des états globaux cohérents et observation possible de l'exécution . . . 63
3.4 Sémantique de la logique épistémique et temporelle . . . 66
3.5 Ensemble des mondes possibles pour l'agent P_3 au point (r, m) . . . 67

4.1 Niveaux groupe, objet et méthode . . . 76
4.2 Formalisme états/transitions pour les trois niveaux groupe, objet et méthode . . . 76
4.3 Le fait φ est seulement vrai dans un sous-ensemble des mondes possibles . . . 81
4.4 Méta-classe de gestion de la variable var . . . 82
4.5 Modèle d'un protocole de validation à deux phases . . . 83
4.6 Version impérative d'un programme à base de connaissances de niveau groupe . . . 87
4.7 Programme de niveau groupe TB_G de transmission de bit . . . 87
4.8 Version impérative d'un programme à base de connaissance de niveau objet . . . 90
4.9 Automates des programmes TB_E et TB_R pour les objets émetteur et récepteur . . 91
4.10 Programmes TB_E et TB_R pour les objets émetteur et récepteur . . . 92

5.1 Observation d'une exécution et états globaux inévitables . . . 100
5.2 Phase d'élévation de la connaissance . . . 102
5.3 Boucle avec phase de révocation de connaissance . . . 102
5.4 Boucle sans phase de révocation de connaissance . . . 102
5.5 Composition parallèle de phases . . . 103
5.6 Phases gardées . . . 103
5.7 Gabarit conditionnelle distribuée . . . 104

5.8 Structure if then else pour le gabarit conditionnelle distribuée 105
5.9 Schéma transactionnel 106
5.10 Gabarit itération distribuée 107
5.11 Treillis des états globaux pour une itération distribuée 108
5.12 Programme de niveau groupe RD_G pour le gabarit récursion distribuée 110
5.13 Etapes intermédiaires d'une récursion distribuée 111
5.14 Récursion distribuée terminale transformée en itération 113

6.1 Modèle états/transitions du comportement BufferDeTailleFixe 116
6.2 Définition d'un comportement CAOLAC et d'une classe GUIDE associée 120
6.3 Double héritage du code de synchronisation et du code effectif 121
6.4 Association entre un objet et son méta-objet 121
6.5 Trois configurations, Vide, Partiel et Plein, d'une instance de BufferDeTailleFixe . 124
6.6 Sémantiques d'état du langage CAOLAC 127
6.7 Algorithme d'élection sur un anneau par vagues contra-rotatives 132
6.8 Modèle états/transitions de l'algorithme d'élection 132
6.9 Modèle CAOLAC de l'algorithme d'élection 133
6.10 Relation d'héritage sous-comportementale 136
6.11 Partitionnement d'état 136
6.12 Prise en compte des historiques d'exécution pour l'invocation Gget 139
6.13 Partitionnement des états acceptables par l'invocation Get2 139
6.14 Modification des états acceptables par les invocations Lock et Unlock 141
6.15 Classe du langage ACT++ 142
6.16 Traduction littérale d'une synchronisation GUIDE en CAOLAC 143
6.17 Chaîne de production d'un programme CAOLAC 145
6.18 Principe de la représentation mémoire des programmes CAOLAC 147
6.19 Traduction des tours méta CAOLAC en schémas d'héritage de classes GUIDE . . . 149
6.20 Traduction d'un comportement CAOLAC en classe GUIDE 149

7.1 Variables propositionnelles associées au comportement bdtf 156
7.2 Formule TLA associée à un comportement CAOLAC 161
7.3 Formule TLA associée au comportement bdtf 163
7.4 Comportement bdtf2 raffinement du comportement bdtf 165
7.5 Formule TLA associée au comportement bdtf2 166

8.1 Exécution de la vague avec un seul initiateur 172
8.2 Deuxième exécution possible de la vague de la figure 8.1 172
8.3 Exécution partielle de la vague avec deux initiateurs 172
8.4 La vague issue de 10 recouvre celle issue de 1 172
8.5 Algorithme de calcul d'arbre couvrant par régions 174
8.6 Régions adjacentes après une phase de propagation/reflux 174
8.7 Arbre couvrant après deux phases de propagation/reflux 174
8.8 Algorithme de calcul d'arbre couvrant par régions et retournement de branches . . 176
8.9 Retournement de branches 176
8.10 Premier programme de niveau groupe 179
8.11 Méthode Arbre du programme de niveau groupe 180

8.12 Programme de niveau groupe de la phase Terminer 181
8.13 Traduction en terme de comportements CAOLAC des programmes de groupe . . . 184
8.14 Comportement CAOLAC implantant l'algorithme sur un objet 186
8.15 Méta-niveaux pour la synchronisation de la phase de construction 187
8.16 Contrôle des numéros d'époque 188
8.17 Contrôle des initiateurs de vague 189
8.18 Diagramme de réutilisation des comportements CAOLAC 190
8.19 Classe implantant l'algorithme sur chaque site 192

9.1 Protocole de validation à deux phases avec deux participants 194
9.2 Programme de niveau groupe de la validation à deux phases 196
9.3 Modèle états/transitions CAOLAC du programme de niveau groupe 197
9.4 Modèle états/transitions CAOLAC du comportement de niveau objet 198
9.5 Niveau de base pour le protocole de validation à deux phases 199

A.1 Concept de base d'un comportement CAOLAC 210
A.2 Définition d'un comportement CAOLAC 211
A.3 Définition d'une classe GUIDE associée à un comportement CAOLAC 214
A.4 Fonctionnement du comportement BufferDeTailleFixe pour la transition Partiel.TPut 215
A.5 Relations méta entre comportements CAOLAC 216
A.6 Héritage de comportements 217
A.7 Invocations génériques 218
A.8 Schéma de traduction d'une invocation CAOLAC en GUIDE 225

Liste des tableaux

1.1 Apports des différentes méthodologies à la notation UML 37

2.1 Différentes écritures des modalités temporelles 46

6.1 Sémantiques d'état du langage CAOLAC 127
6.2 Compteurs d'événements 130
6.3 Comparaison de différentes approches de la synchronisation en univers objet 144
6.4 Temps moyens de compilation CAOLAC 146

Introduction

L'amélioration des performances des matériels ouvre de nouveaux débouchés à l'informatique distribuée. Néanmoins, la conception et la mise en place d'applications réparties est une tâche difficile pour laquelle peu d'outils existent. Ces applications sont, en général, complexes et de taille importante. Par rapport à celles conçues pour des environnements centralisés, elles présentent un certain nombre de caractéristiques originales. Par exemple, elles introduisent de nombreuses opérations de communication entre sites distants. De même, elles comprennent de multiples activités s'exécutant en parallèle qu'il est nécessaire de gérer et de synchroniser. Enfin, elles s'exécutent dans des environnements systèmes et réseaux comportant de nombreuses sources d'indéterminisme. Ainsi, des réseaux de communication peu fiables et l'entrelacement des actions dans les systèmes multi-tâches sont deux sources d'indéterminisme qu'il est, presque toujours, indispensable de prendre en compte dans ces applications. Ces caractéristiques, parmi d'autres, introduisent des mécanismes complexes qu'il faut intégrer aux phases de conception et de développement.

De plus en plus les applications distribuées reposent sur le paradigme objet.

Dans ce paradigme un objet est une entité qui encapsule une structure de données dont l'accès est contrôlé par un ensemble de méthodes. La définition de la sémantique de ces primitives forme la spécification externe, appelée interface, de l'objet. Son implantation est définie, quant à elle, dans une classe dont l'objet est une instance. La classe fournit l'ensemble des algorithmes permettant de réaliser la sémantique externe. Les objets interagissent les uns avec les autres par invocations de méthodes. L'intérêt du mode de programmation objet est qu'il fournit un découpage clair entre, d'une part, l'interface des objets, et d'autre part, la façon dont est implantée cette interface. De plus, c'est un choix naturel pour les applications distribuées de type client/serveur. Par exemple, les objets clients utilisent les services de gestion d'une ressource partagée fournis par des objets serveurs.

Le paradigme objet introduit un modèle d'applications qui diffère légèrement de celui obtenu en utilisant des interfaces de programmation réseau (comme par exemple les interfaces de transport de type TCP). Dans ce cas, une application répartie est, en général, modélisée par un ensemble de processus communicant par envoi de messages asynchrones. Le paradigme objet propose un niveau d'abstraction supérieur et considère qu'une application répartie est un ensemble d'objets qui interagissent par appels de méthodes synchrones ou asynchrones. Le terme synchrone ne signifie pas ici le fait qu'il existe forcément une borne supérieure aux délais de communication. Il désigne le fait que l'appelant est bloqué jusqu'au moment de l'envoi des paramètres de retour de la méthode. Le terme asynchrone signifie que l'appelant n'est pas bloqué et continue son exécution immédiatement après son appel. Notons, tout d'abord, que ces deux modèles ont un pouvoir d'expression équivalent : l'appel de méthodes synchrones peut être réalisé à l'aide de plusieurs échanges de messages asynchrones (deux, au moins, et éventuellement plus selon la qualité de service demandé) et, réciproquement, l'envoi de message peut être considéré comme un appel de méthode asynchrone. Néanmoins, il est très largement admis que l'approche objet fournit une plus grande souplesse de structuration. Par exemple, la notion d'encapsulation permet de créer des composants logiciels autonomes. On peut ainsi, d'une part, diminuer les temps de développement et de maintenance en créant des éléments de taille réduite testables indépendemment du reste de l'application et, d'autre part, augmenter le taux de réutilisation du code. De plus, l'appel de méthodes fournit un mécanisme de communication puissant qui permet de masquer la distribution et qui recouvre l'appel procédural local et l'appel procédural distant ou RPC [BN84]. Lorqu'émetteur et récepteur sont localisés sur un même site, l'appel de méthodes emploie un appel procédural classique et un passage de paramètres par pile, tandis que lorsqu'ils sont localisés sur des sites différents un mé-

canisme à base de RPC est utilisé. Il convient, néanmoins, d'intégrer les différences de sémantique importantes qui existent entre l'appel local et l'appel distant. Les environnements de programmation objets répartis tels que CORBA [OMG95] ou les systèmes tels que Amoeba [MvRT+90], Chorus/COOL [LJP93] ou GUIDE [BBD+91], offrent tous un tel mécanisme d'invocation de méthodes qui masque la distribution et qui permet d'appeler une méthode d'un objet quel que soit sa localisation.

Néanmoins, malgré ces mécanismes d'invocation de méthode à distance, la conception d'applications distribuées reste une tâche complexe. Les méthodes orientée objet actuelles, comme par exemple UML [BRJ97], commencent à intégrer certains concepts liés à la distribution. Cependant, il n'en existe pas, à notre connaissance, qui présentent une démarche systématique d'algorithmique répartie. Elles se contentent, dans la plupart des cas, d'aider l'utilisateur à fournir une vision statique en lui permettant de décrire la répartition des composants logiciels d'une application sur les différents nœuds physiques d'un système. L'une des raisons principales de cet état de fait est que les méthodologies orientée objet sont encore essentiellement adaptées à la manipulation des ensembles de données. Il existe, par ailleurs, de très nombreux travaux sur des modèles d'expression du contrôle (tels que les automates synchronisés ou encore les différentes catégories de réseaux de Pétri). Ces méthodes sont surtout développées pour l'expression de la concurrence et tendent, actuellement, à intégrer les concepts objets et la distribution. Malgré l'existence de très nombreux travaux précédents, l'objectif de notre thèse reste donc celui de mettre en place une démarche algorithmique répartie prenant en compte la concurrence et la distribution. En particulier, il nous faut :

- proposer un processus de développement guidant les développeurs dans la définition des différentes caractéristiques de leurs applications,
- proposer des schémas de collaboration et de coopération standards entre entités distantes d'une application distribuée (que l'on appelle structures de contrôle distribuées),
- décrire les aspects liés à la distribution en fournissant un modèle pour analyser et décrire de façon synthétique les informations échangées au cours d'une exécution répartie,
- décrire les aspects liés à la concurrence en définissant un modèle et un langage pour synchroniser l'exécution des objets concurrents d'une application distribuée.

En partant de la constatation que les méthodes de développement actuelles ne sont que peu adaptées aux environnements distribués, notre objectif est donc d'améliorer la qualité des logiciels développés en intégrant à une démarche de conception des apports de différents domaines, et en apportant, si possible, un certain nombre de concepts originaux. De ce fait, ce travail s'appuie sur différentes approches existantes. En particulier, notre but est d'essayer de faire converger, dans un contexte objets répartis, les approches de programmation issues des systèmes répartis et des systèmes multi-agents.

Positionnement

Nous nous plaçons dans le cadre d'un environnement de programmation réparti orienté objet. Nous nous intéressons aux applications distribuées, c'est à dire à des programmes qui mettent en relation différentes entités logicielles s'exécutant sur différentes machines. Nous envisageons

plus particulièrement des environnements dans lesquels les applications sont structurées selon le paradigme objet. Le type d'application auquel nous nous intéressons concerne aussi bien les applications client/serveur de l'informatique de gestion, que les algorithmes répartis du domaine des réseaux, ou que les applications coopératives pour les collectitiels.

Ce travail a été influencé par plusieurs domaines. Ainsi, les méthodes de conception, les méthodes formelles, les systèmes répartis et les systèmes multi-agents ont fourni, à des degrés divers, des concepts que nous avons repris dans notre démarche. Tout d'abord, les méthodes de conception telles que OMT [RBP+91], Booch [Boo94] ou UML [BRJ97], fournissent un cadre méthodologique pour le développement d'applications. Elles proposent des cycles de vie, des notations et des démarches, pour l'analyse d'un problème et pour la conception d'architectures informatiques. Bien qu'elles soient universelles, leur domaine de prédilection reste celui de l'informatique de gestion. La seconde influence provient des méthodes formelles telles que B [Abr96], VDM [Jon86] ou CCS [Mil80]. Celles-ci définissent un cadre rigoureux pour l'écriture de programmes. A partir d'une théorie, comme par exemple la théorie des ensembles pour B, elles permettent de construire des modèles d'un système informatique et de vérifier mathématiquement des propriétés sur ces modèles. Ces méthodes, qui permettent d'atteindre un haut niveau de qualité, sont couramment utilisées pour les systèmes critiques. Par ailleurs, notre travail a également subit l'influence des développements théoriques et pratiques réalisés dans le domaine des systèmes répartis (comme Amoeba [MvRT+90], Chorus/COOL [LJP93] ou GUIDE [BBD+91]) et des standards d'interopérabilité (comme CORBA [OMG95]). Ces environnements orientés objet fournissent, entre autres, des mécanismes systèmes de désignation, de gestion mémoire et de gestion de la concurrence. Ces environnements sont bien adaptés aux applications réparties de petites tailles déployées sur, au maximum, quelques dizaines de sites. Finalement, les modèles de systèmes multi-agents [FHMV95][Sin94][AH86] fournissent un paradigme de programmation décentralisé. Ils suggèrent de concevoir une application distribuée comme un ensemble d'agents autonomes poursuivant des buts locaux. La résolution du problème global émerge alors de la superposition des comportements locaux et de leurs interactions.

Dans cette thèse, les outils que nous proposons pour la conception d'applications distribuées s'appuient sur les caractéristiques majeures évoquées dans chacun des domaines précédents.

Objectifs

L'objectif principal de cette thèse est de fournir des outils pour aider les concepteurs à développer et à mettre en place des applications à base d'objets distribués. Notre but n'est pas de décrire les aspects statiques de ces applications tels que la répartition des classes et des objets sur les différents nœuds physiques du système. Notre travail se concentre sur les aspects dynamiques de ces applications. Ainsi, nous sommes intéressés par la description des comportements, des collaborations et des synchronisations entrepris par les différentes entités composant une application distribuée. Cet objectif principal recouvre les points suivants.

Le premier concerne la définition d'un démarche pour l'algorithmique répartie. Nous voulons fournir un niveau d'abstraction suffisamment élevé permettant de raisonner sur la coordination des comportements de plusieurs entités, sur les stratégies globales de résolution entreprises par des groupes d'objets distribués ou sur les connaissances échangées lors d'une exécution répartie.

Le second aspect abordé définit un cadre méthodologique pour guider les concepteurs dans

leurs développements. Il est important, par exemple, d'identifier clairement les différentes étapes du développement d'une application distribuée et de mettre en valeur les différents points de vue adoptés au cours de chacune de ces étapes. C'est pourquoi, nous proposons de définir un processus de développement fondé sur le point de vue de la distribution et qui intègre la notion de comportement pour des groupes d'objets.

La réutilisation, que ce soit celle des composants logiciels ou des schémas de conception, est un aspect majeur qui permet de réduire les temps de développement et de maintenance. En liaison avec le processus de développement évoqué ci-dessus, nous proposons de définir un certain nombre de schémas de comportement et de collaboration type pour des groupes d'objets distribués.

Le quatrième aspect présent dans cette thèse concerne la description des informations échangées lors d'une exécution répartie. Les interactions d'informations mutuelles entre sites sont, en général, complexes et délicates à décrire, particulièrement dans la phase de spécification. Dans ce cadre, plutôt que de raisonner comme on le fait habituellement en terme de messages échangés ou de procédures invoquées, il est souhaitable d'abstraire les mécanismes d'informations pour se concentrer sur la sémantique des interactions d'informations. Nous proposons donc d'inclure au processus de développement une démarche épistémique permettant de raisonner en terme de connaissances échangées.

Finalement, la gestion de la synchronisation entre méthodes (appelée synchronisation intra-objet) est une des tâches difficiles de la conception d'applications distribuées à base d'objets concurrents. Elle est souvent occultée par les méthodes de conception qui la relègue au niveau de la programmation. Or, les comportements distribués engendrent un nombre important d'activités qu'il est nécessaire de gérer correctement afin de préserver la cohérence des objets. Nous proposons donc un langage de synchronisation qui permet de faciliter la tâche des développeurs d'applications distribuées.

Organisation de la thèse

Ce document comprend huit chapitres regroupés en quatre parties. La première, qui s'étend sur les trois premiers chapitres, est un état de l'art du domaine. Les chapitres quatre et cinq constituent la seconde partie et présentent notre approche pour la conception de comportements pour des groupes d'objets distribués. La troisième partie comprend les chapitres six et sept. Elle présente le langage CAOLAC de synchronisation d'objets concurrents. Finalement, les chapitres huit et neuf forment la quatrième partie qui est consacrée à des études de cas. Enfin, nous donnons une conclusion sur le travail réalisé ainsi que les perspectives envisagées.

Première partie : état de l'art

Le premier chapitre est consacré à un état de l'art de la programmation et de la conception en environnement objet. Nous rappelons les caractéristiques principales des langages de programmation orientés objets ainsi que les différents concepts, comme la concurrence, la synchronisation et l'héritage, qui viennent s'y greffer. Nous présentons ensuite les caractéristiques principales des environnements distribués en prenant l'exemple du standard d'interopérabilité CORBA. Finalement, nous présentons en les comparant, les différents apports des méthodes de conception existantes.

Le second chapitre est consacré à une présentation d'outils mathématiques permettant de décrire la concurrence et la distribution. Nous nous intéressons plus spécialement aux logiques dites

modales, et en particulier, aux logiques temporelles et aux logiques épistémiques. Les logiques temporelles, que nous envisageons plus particulièrement sous l'angle de la logique temporelle d'actions [Lam94] de Lamport, fournissent un modèle de la concurrence fondé sur des règles de précédence ou de concurrence entre actions ou entre séquences d'actions. Les logiques épistémiques sont, quant à elles, des logiques de la connaissance. Elles permettent de décrire la répartition de l'information entre les différentes entités d'un environnement distribué et l'échange d'information entre sites.

Le chapitre trois présente, plus en détail, un des modèles de programmation qui a influencé ce travail. C'est la notion, proposée par Fagin, Halpern, Moses et Vardi, de programme à base de connaissances [FHMV95] pour les agents d'un système multi-agents. A partir d'une logique incorporant des modalités épistémiques et temporelles, cette approche a pour but de décrire le comportement d'un agent à l'aide de règles de décision de type système expert. Ces règles testent les connaissances et les possibilités d'évolution d'un agent, puis décident des actions à entreprendre. Un processus de raffinement permet d'implanter concrètement ces règles en transformant les tests de connaissances en tests sur des variables.

Deuxième partie : coordination inter-objets

Nous proposons, au chapitre quatre, un ensemble de concepts pour la description de comportements pour des groupes d'objets distribués et pour la description de la coordination de ces objets. En particulier, nous proposons un cycle de développement en trois niveaux méthodologiques fondé sur une démarche incrémentale de raffinement. Le premier niveau, dit de groupe, suggère de décrire les buts globaux et l'évolution globale du groupe d'objets prenant part à la réalisation de l'application distribuée. Le second niveau, dit niveau objet, est un raffinement du précédent et prend le point de vue d'un objet au sein du groupe. Finalement, le niveau méthode raffine le niveau objet et étudie le comportement d'une méthode au sein d'un objet. Nous précisons alors plus particulièrement les points clés qui constituent l'originalité de cette approche. Ainsi, nous expliquons les notions de comportement de groupe, de structures de données de groupe et de niveaux de connaissance pour ces structures. Finalement, nous introduisons la notion de programme à base de connaissances de niveau groupe qui regroupe l'ensemble de ces concepts et qui généralise la notion introduite par Fagin, Halpern, Moses et Vardi, de programme à base de connaissances de niveau agent présentée au chapitre précédent.

Le chapitre cinq présente différentes structures de contrôle pouvant être utilisées dans les programmes à base de connaissance de niveau groupe. Nous en proposons quatre : la phase, la conditionnelle distribuée, l'itération distribuée et la récursion distribuée. Ces structures constituent des gabarits de conception qui peuvent être réutilisés, raffinés et composés dans de nombreuses applications. Elles peuvent être vues comme la généralisation à un niveau réparti, de la séquence, des alternatives de type *if* ou *case*, des boucles *while* et des parcours arborescents que l'on retrouve habituellement dans l'algorithmique séquentielle. Ces structures définissent des schémas de coordination type entre les membres d'un groupe d'objets distribués.

Troisième partie : coordination intra-objet

Le chapitre six présente le langage CAOLAC de synchronisation d'objets concurrents que nous avons défini. Ce langage a été implanté sous la forme d'un protocole de niveau méta-objet pour le langage du système réparti orienté objet GUIDE. Néanmoins, il est suffisamment général pour

pouvoir être appliqué à d'autres langages et à d'autres plateformes distribuées. En partant de la constatation que l'implantation des schémas de coordination du chapitre précédent requiert des comportements d'objets complexes, nous avons ressenti le besoin d'offrir un outil puissant permettant de faciliter la tâche des développeurs. Le langage CAOLAC permet donc, de découpler dans un langage objet concurrent la synchronisation des méthodes, des traitements de base réalisés par ces méthodes. La synchronisation est définie sous la forme de systèmes états/transitions dont la sémantique prend en compte le parallélisme intra-objet. Ces politiques de synchronisation peuvent alors être réutilisées et composées.

Quatrième partie : étude de cas

Le chapitre sept définit la sémantique du langage CAOLAC. Nous proposons pour chaque politique de synchronisation une formulation équivalente en terme de logique temporelle d'actions. Cette démarche permet d'introduire formellement les propriétés de sûreté, de vivacité et de raffinement pour une politique de synchronisation.

Finalement, les chapitres huit et neuf illustrent la démarche de conception présentée tout au long de cette thèse à l'aide d'études de cas. Le chapitre huit propose l'étude complète d'un algorithme de calcul d'arbre couvrant pour des groupes d'objets distribués, tandis que le chapitre neuf s'intéresse à un protocole transactionnel de validation à deux phases. Nous appliquons, pour chacun de ces exemples, le processus de développement en trois niveaux méthodologiques présenté au chapitre quatre. Nous définissons les structures de données réparties manipulées et les connaissances échangées. Parmi les gabarits de conception définis au chapitre cinq, trois (la phase, l'itération distribuée et la récursion distribuée) sont utilisés par l'algorithme de calcul d'arbres couvrants, tandis que le protocole de validation à deux phases en utilise deux (la phase et la conditionnelle distribuée). Finalement, ces programmes sont implantés à l'aide du langage de synchronisation CAOLAC présenté au chapitre six, et du langage objet du système réparti GUIDE.

Première partie

Etat de l'art

Chapitre 1

Approche objet

L'approche objet entraîne un certain nombre de changements dans la façon de concevoir et d'implanter les applications. Que ce soit dans le domaine de l'informatique distribuée ou dans celui de l'informatique centralisée, la complexité inhérente aux applications amène un besoin important de structuration auquel l'approche objet essaie de répondre simplement et de façon plus intuitive que les approches procédurales. Ainsi, l'approche objet consiste, avant toute chose, à modéliser un système en fonction des objets du monde réel qu'il contient. Dans ce chapitre, nous nous intéressons plus particulièrement à la programmation en approche objet, aux environnements distribués supportant le concept d'objets et aux méthodes de conception orientées objet.

Booch [Boo94] justifie l'adéquation de l'approche objet pour la modélisation des systèmes complexes en dégageant les caractéristiques suivantes. Tout d'abord, dans la plupart des cas, les systèmes complexes sont organisés naturellement de façon hiérarchique. Les systèmes sont décomposés en sous-systèmes qui sont eux-mêmes décomposés en sous-sous-systèmes. Cette organisation naturelle est essentiellement liée au fait que la nature humaine n'est en mesure d'appréhender et de comprendre les mécanismes complexes qu'en deçà d'une taille limite relativement faible. L'utilisation, par exemple, de diagrammes de classes dans les modélisations objets participe donc à ce besoin de décomposition hiérarchique des systèmes. Dans un second temps, il apparaît que, dans une telle hiérarchie, les liens au sein d'un même composant sont, en général, plus forts que ceux entre composants distincts. De ce fait, il est plus facile de réunir tous les éléments définissant un composant au sein d'une même et unique structure. C'est le principe d'encapsulation des langages objet. En isolant ainsi les éléments participant à une même fonctionnalité, il est alors plus simple d'en étudier les liens. Dans un troisième temps, Booch fait remarquer que les systèmes hiérarchiques sont arrangés et combinés à partir d'un petit nombre de sous-systèmes différents. Cela justifie, par exemple, les différentes possibilités de réutilisation de code des approches objet.

Les trois caractéristiques précédentes de décomposition, d'encapsulation et de réutilisation constituent l'originalité principale des langages et des méthodes de conception orientées objet. Ainsi, le concept de classe permet de traduire la décomposition d'un système en différents composants. Les dépendances entre composants sont exprimées, par exemple, par des liens dans les diagrammes de classes des méthodes de conception. Chaque objet est alors vu comme l'instance d'une classe. Il encapsule des données accédées par l'intermédiaire de méthodes. Celles-ci constituent l'interface de communication mise à la disposition du monde extérieur (c'est à dire des autres objets). Ces interfaces sont utilisées dans les environnements de programmation distribuée

comme CORBA [OMG95], pour définir les services offerts par chaque objet serveur. Les objets communiquent alors par invocation de méthode. Dans le cas d'un environnement distribué, ce mécanisme permet d'abstraire les communications inter-objets. Ainsi, selon que les objets émetteur et destinataire sont ou non localisés sur le même site, l'invocation peut emprunter, soit un mécanisme d'appel local, soit un mécanisme protocolaire d'appel à distance. Finalement, les relations de réutilisation prennent, aussi bien dans les langages que dans les méthodes de conception, différentes formes qui vont de l'héritage à la délégation en passant par la composition.

Sans vouloir être exhaustif, ce chapitre a donc pour but d'exposer les concepts principaux liés à l'approche objet. Ainsi, le paragraphe 1.1 s'intéresse aux langages de programmation orientés objet. Le paragraphe 1.2 illustre les concepts introduits par les environnements de programmation orientés objet en prenant l'exemple du standard d'interopérabilité CORBA. Le paragraphe 1.3 concerne, quant à lui, les méthodes de conception orientées objet. Finalement, le paragraphe 1.4 conclut ce chapitre.

1.1 Les langages orientés objet

Cette partie propose une introduction à l'approche objet et aux langages associés. Dans un premier temps, nous nous intéressons aux concepts de base de ce domaine. En particulier, nous présentons les notions d'encapsulation et d'héritage. Elles permettent, respectivement, de rendre modulaire l'écriture des programmes et de faciliter la réutilisation des composants logiciels. Puis, nous nous intéressons au paragraphe 1.1.2 à l'introduction de la concurrence dans les langages orientés objet. La possibilité d'exécuter simultanément plusieurs activités est l'un des aspects majeurs des environnements distribués. On imagine mal, en effet, qu'un serveur moderne ne traite qu'une seule requête ou qu'un objet n'exécute qu'une méthode à la fois. Nous analysons donc différents modèles de concurrence qui ont été proposés pour les langages orientés objet. La gestion de plusieurs activités au sein d'un objet va nécessairement de pair avec le contrôle de leur exécution. Par exemple, dans certains cas, il est nécessaire de limiter l'accès aux données afin de préserver leur cohérence. Le paragraphe 1.1.3 présente donc différents techniques et outils existants pour synchroniser les objets. La notion d'héritage est un des arguments majeurs des partisans de l'approche objet. Elle permet de réduire les temps de développement d'une classe d'objets en ajoutant des nouvelles fonctionnalités à une classe existante. Néanmoins, ce n'est pas la seule relation de réutilisation introduite par l'approche objet. On peut citer également les relations de composition, de délégation et de classes paramétrées. Ces différentes relations sont examinées au paragraphe 1.1.4. Cependant, il a été établi qu'un certain nombre de limitations apparaissent lorsqu'on introduit l'héritage dans un langage concurrent. Ces problèmes connus sous le terme d'anomalies d'héritage sont présentés au paragraphe 1.1.5.

1.1.1 Introduction

L'augmentation constante de la taille et de la complexité des programmes informatiques amène, aussi bien dans les phases d'analyse et de conception que dans les phases de codage, un besoin de techniques et d'outils de structuration et d'organisation des applications. Dans ce paragraphe nous nous intéressons plus particulièrement à l'approche objet dans le cadre de la programmation.

Une étape majeure dans l'avènement de la programmation structurée a été franchie par les langages procéduraux de type Algol. Ceux-ci ont permis de s'affranchir de la programmation par

branchement en permettant de factoriser des parties de code en procédures réutilisables. De plus, le renforcement du typage dans les langages comme Pascal permet de définir pour chaque procédure une signature qui comprend un ensemble de paramètres identifiés et typés.

Un objet vise à modéliser le comportement d'une entité du monde physique et peut être vu dans un certain sens, comme une implantation d'un type abstrait de données. Ceux-ci définissent de façon formelle les opérations permettant de manipuler une structure de données (une pile, une file, un arbre, etc ...) et les propriétés associées à ces opérations. Par exemple, une pile comprend des opérations pour *empiler* et *dépiler* des éléments et pour tester si elle est vide. Une propriété de cette structure est qu'une opération *empiler* suivie d'une opération *dépiler* sur une pile non vide et non pleine laisse la pile dans le même état. La notion d'objet poursuit un but identique et permet d'encapsuler, dans un même entité logique, des données et des opérations les manipulant. L'objet présente ainsi au monde extérieur une interface qui abstrait la façon dont est réalisée l'implantation.

En plus du concept d'encapsulation, l'approche objet introduit les notions d'instanciation et d'héritage. L'instanciation désigne la capacité à générer des objets à partir d'un modèle commun appelé classe. L'héritage permet, quant à lui, de définir de nouvelles classes en dérivant des classes existantes. A partir de ces définitions, Wegner dans [Weg87] propose la classification suivante : les langages offrant un mécanisme d'encapsulation sont dits à base d'objets ou *object-based*, ceux introduisant en plus la notion de classe sont dits à base de classes ou *class-based* et, finalement, ceux proposant en plus l'héritage sont dit orientés objet ou *object-oriented*. Le concept de *package* Ada [DoD80, Bar89] et de module Modula [Wir82] permet, par exemple, de classer ces deux langages dans la première catégorie. Simula [BDMN73] entre, quant à lui, dans la catégorie des langages à base de classes. Finalement, C++ [Sou86] et Smalltalk [GR83] sont des langages orientés objet.

1.1.2 Concurrence

La concurrence désigne la possibilité d'exécuter simultanément plusieurs tâches au sein d'un même système. Les langages dit concurrents, orientés objet ou non, offrent un ensemble de primitives permettant de décrire de façon explicite ou implicite, ces différentes tâches. L'idée de base suivie par ces langages est d'essayer de tirer parti au maximum du parallélisme inhérent à de nombreuses applications afin d'améliorer les performances du système ou de permettre d'implanter des systèmes de contrôle pour des applications présentant des caractéristiques de parallélisme intrinsèque. Ces langages présentent un nombre important de points communs avec les langages utilisés dans les architectures parallèles, mais sont plutôt destinés à être utilisés dans des environnements mono-processeurs fonctionnant en pseudo-parallélisme ou dans des environnements distribués. Guerraoui dans [Gue95] propose cinq critères pour classer les langages concurrents à objets : leur origine, la façon dont est exprimée la concurrence, la granularité de la concurrence, la sémantique des invocations de méthodes et le contrôle de la concurrence. Cette dernière notion désigne la façon dont sont coordonnées et synchronisées les activités simultanées au sein d'un objet. Nous traitons ce point plus en détail au paragraphe 1.1.3. Dans ce paragraphe, nous nous intéressons plus spécialement aux quatre premiers critères.

Le premier critère concerne l'origine des langages concurrents à objets : il distingue les langages concurrents à objets classiques, des extensions concurrentes des langages à objets et des langages concurrents. Les premiers n'offrent pas de mécanismes propres pour la gestion des activités concur-

rentes. Ils permettent, au travers d'un ensemble de librairies ou d'interfaces de programmation, d'accéder aux primitives du système. C'est le cas, par exemple, des langages C++, Objective-C [Cox86] ou Smalltalk. D'autres langages se présentent comme des extensions de langages à objets existants et offrent leur propre gestion d'activités. On peut citer par exemple les langages Eiffel// [Car94] qui est une extension d'Eiffel [Mey88], Concurrent Smalltalk [YT87] qui est une extension de Smalltalk, Argus [LS83] qui est une extension de CLU [LS79], ACT++ [KL89a, KL89b] et Arjuna [PS88] qui sont des extensions de C++. Finalement, les langages concurrents ont été conçus afin d'intégrer dès le départ la notion de concurrence. C'est le cas, par exemple, des langages GUIDE [BBD+91], Hybrid [Nie87], ABCL [YSTH87, Yon90] et POOL [Ame87].

Le second critère de classification des langages à objets concerne la façon dont est exprimée la concurrence, c'est à dire l'association entre une activité et un objet. On distingue principalement deux modèles qui sont le modèle à objets passifs et le modèle à objets actifs. Il existe également un troisième modèle qui se situe à mi-chemin des deux précédents et qui est dit hybride. Dans un modèle à objets passifs, la notion d'activité est indépendante de la notion d'objet. Les activités ne sont pas liées à un objet particulier et peuvent se propager sur plusieurs objets au gré des invocations de méthodes. Ces langages fournissent des constructeurs de parallélisme explicites de type *fork/join*, *cobegin/coend* ou *parbegin/parend* pour créer ou détruire des activités concurrentes. Parmi les systèmes ou les langages qui utilisent ce paradigme on peut citer C++, Arjuna, GUIDE, Smalltalk-80, Clouds [LA88], Amoeba [MvRT+90] et SOS [SGH+89]. Les objets actifs possèdent, quant à eux, leur propre activité qui réceptionne et traite les invocations de méthodes. Les langages ACT++, Eiffel//, POOL, Ada, Argus, DRAGOON [AGMB91, Atk91] et Emerald [BHJ+87] implantent un système d'objets actifs. La tâche de fond qui exécute l'activité principale associée à l'objet peut être implicite comme dans le langage ACT++ ou peut être redéfinie explicitement par les programmeurs comme dans les langages Ada (procédure *task body*), Eiffel// (méthode *Live*) et POOL. Les modèles à objets passifs et actifs sont à peu près équivalent en terme de fonctionnalités offertes à l'utilisateur final. Il apparaît néanmoins que les systèmes à objets actifs sont plutôt adaptés pour des objets à gros grain de type serveurs d'applications, tandis que les objets passifs, de part leur plus faible consommation en ressources systèmes, sont plus adaptés aux environnements comprenant un nombre élevé d'instances. Finalement, certains systèmes proposent des modèles d'exécutions intermédiaires qui ne sont ni complètement passifs, ni complètement actifs. Les objets sont en général rassemblés en grappes. Chaque cluster est associé à une activité et gère les objets qui en sont membres. Ce modèle est utilisé par exemple, dans le système DIAMONDS [BCSL94].

Le troisième critère pour classer les langages à objets concurrents est la granularité de la concurrence. Celle-ci peut être de type inter-objets ou intra-objet. Dans les langages à concurrence inter-objets, plusieurs objets différents peuvent s'exécuter simultanément au sein de l'environnement. La concurrence intra-objet désigne quant à elle, la possibilité d'exécuter plusieurs activités simultanément au sein d'un même objet. On distingue en général trois niveaux de concurrence intra-objet : séquentiel, quasi-concurrent et concurrent. Un modèle séquentiel désigne un objet qui ne peut traiter qu'une seule invocation à la fois (c'est donc plutôt une absence de concurrence). Les objets quasi-concurrents peuvent commuter d'une exécution à l'autre. C'est par exemple le cas des objets des systèmes Hybrid et ABCL. Finalement un objet complètement concurrent peut traiter plusieurs invocations simultanément. La majorité des langages concurrents supporte à la fois un niveau de parallélisme inter-objets et un niveau de parallélisme intra-objet. Certains ne supportent qu'un niveau de parallélisme inter-objets (par exemple Ada et POOL).

Finalement, le quatrième critère concerne la sémantique de l'invocation de méthode. Il peut paraître assez peu rationnel d'introduire la concurrence comme un effet de bord de l'invocation de méthode (c'est, par exemple, l'usage qu'en font les langages ACT++ et Eiffel//). Cependant, il est incontestable que la sémantique des invocations est très liée à la concurrence. Ainsi, celles-ci peuvent être synchrones, asynchrones ou semi-synchrones. Avec une invocation synchrone, l'émetteur reste bloqué en attente du résultat de l'invocation. Avec une invocation asynchrone, l'émetteur envoie son message et continue son exécution tout de suite après. Finalement, pendant une invocation semi-synchrone, l'émetteur continue son exécution tant qu'il n'a pas besoin d'utiliser le résultat de l'invocation. La partie de code qui se trouve entre l'invocation et l'utilisation du résultat est donc exécutée en concurrence avec l'invocation elle-même. On distingue deux modes d'utilisation de l'invocation semi-synchrone : celle à futur explicite et celle à futur implicite. Dans les langages à futur explicite, les programmeurs manipulent une structure de données qui reçoit le résultat de l'invocation. Par exemple, le langage ACT++ définit une classe appelée *Cbox* qui est une boîte à lettre de réponse. Une instance de cette classe est alors transmise lors de toute invocation de type semi-synchrone. Le programmeur interroge cette instance lorsqu'il a besoin du résultat. Inversement, dans une approche à futur implicite, le compilateur détermine, par analyse statique du code, la première instruction utilisant le résultat d'une invocation semi-synchrone. Il génère alors automatiquement le code permettant d'effectuer un rendez-vous entre le retour d'invocation et l'activité de l'objet.

La concurrence est l'un des aspects quasi incontournable pour de très nombreuses classes d'applications et de manière certaine pour des applications distribuées. On imagine mal, en effet, un serveur moderne fournissant un seul type de service à un seul client. De ce fait, les applications distribuées évoluent vers une prise en compte d'un double niveau de concurrence : inter et intra-objet. Cet essor a également favorisé le développement de modèles d'exécution variés. Dans le domaine des objets systèmes, le critère dominant est sans doute, la dichotomie entre objets actifs et objets passifs. Les premiers encapsulent leurs activités, tandis que dans le modèle passif, celles-ci sont orthogonales aux objets. Néanmoins, quel que soit le modèle retenu, des mécanismes de contrôle de la concurrence sont nécessaires à une utilisation cohérente des objets partagés.

1.1.3 Synchronisation

Les mécanismes de synchronisation ont pour but de contrôler le séquencement et le parallélisme des comportements réalisés par les objets. Par exemple, les méthodes des objets concurrents peuvent être synchronisées afin :

- d'autoriser la concurrence entre des exécutions n'ayant aucun lien entre elles.
- d'assurer la cohérence de leurs variables d'instance. Par exemple, il peut être nécessaire de garantir qu'une section accédée en exclusion mutuelle ou qu'une variable partagée ne soit accédée simultanément que par une seule méthode de type écrivain.
- d'imposer un ensemble de contraintes de précédence dans les exécutions des méthodes. Par exemple, le concepteur spécifie qu'une donnée doit être initialisée avant d'être utilisée (une méthode de type écrivain doit être invoquée et exécutée avant toute autre méthode).

Dans la suite de ce paragraphe, nous présentons sept mécanismes qui sont couramment employés pour synchroniser des objets concurrents. Ce sont les sémaphores, les moniteurs, les expressions de chemin, les commandes gardées, les compteurs d'événements, les ensembles acceptables

et les solutions à base d'approches réflexives. Ces mécanismes sont, pour la plupart, très anciens et très étudiés. Nous n'en présentons que les grandes lignes.

1.1.3.1 Sémaphores

Les sémaphores [Dij65] constituent le mécanisme de synchronisation de base de l'algorithmique concurrente. Ils ont été définis par Dijkstra en 1965. Un sémaphore est une variable entière associée à deux opérations de base P et V. Dans son utilisation de base, un sémaphore permet de protéger l'accès à une section critique, c'est à dire à une zone de code ne pouvant être accédée que par une seule activité simultanément. Chaque processus souhaitant entrer dans cette zone commence par acquérir le sémaphore. Il invoque pour cela l'opération P. Si aucun autre processus n'est présent dans la zone critique, le sémaphore est attribué au processus demandeur. Sinon, ce dernier doit être mis en attente jusqu'à ce que le sémaphore se libère. Cette opération est implantée par l'opérateur V. Il existe plusieurs autres modes d'utilisation des sémaphores permettant de construire n'importe quel schéma de synchronisation. Dans le cas d'un langage de programmation orienté objet, les mécanismes système de gestion des sémaphores peuvent facilement être encapsulés dans une classe offrant une méthode d'initialisation et deux méthodes P et V.

Les sémaphores offrent certainement un mécanisme de base pour des politiques de synchronisation simples. Néanmoins, leur utilisation devient rapidement délicate pour des politiques complexes. En effet, lorsque celles-ci nécessitent plus de deux ou trois niveaux de synchronisation imbriqués, on estime en général qu'ils sont une source d'erreurs importante, et qu'il est préférable de les abandonner au profit de mécanismes de plus haut niveau. On estime, par ailleurs, qu'il est peu structurant de mêler dans le même programme le code de base et le code de synchronisation.

1.1.3.2 Moniteurs

Le concept de moniteur [Hoa74] a été introduit par Hoare en 1974. Ce mécanisme est employé dans de nombreux systèmes et langages. C'est, par exemple, la solution retenue pour synchroniser les flots d'exécution concurrents dans le langage Java [GM95]. Un moniteur fournit deux primitives : *Wait* et *Notify*. La première permet de suspendre un fil d'activité si une condition (en général une garde) n'est pas vérifiée. La seconde permet de réactiver un fil d'activité suspendu. Le mécanisme de synchronisation par moniteur est employé, de façon transparente, dans les objets protégés d'Ada 95 [Bar95, BW95]. Un objet protégé ne peut être accédé simultanément que par une seule instance de méthode. Cette instance possède ainsi un accès exclusif aux données privées de l'objet qu'elle peut mettre à jour de façon cohérente.

Les moniteurs offrent un mécanisme pratique et transparent pour synchroniser l'accès à un objet concurrent. Leur pouvoir d'expression est en général suffisant pour traiter de nombreux cas simples. Néanmoins, ils sont relativement inadaptés à des gestions du contrôle plus complexes mettant en jeu de nombreuses conditions.

1.1.3.3 Expressions de chemin

Les expressions de chemin [CH73] permettent de spécifier tous les enchaînements valides d'opérations qui peuvent être réalisés pour un ensemble d'actions. Quatre opérateurs sont disponibles pour décrire ces enchaînements : la séquence notée par un point-virgule, le choix noté par une barre verticale, la répétition notée par une étoile et la simultanéité notée entre accolades. Ainsi,

l'expression *m;n* autorise une seule exécution de la méthode m, suivie d'une exécution de la méthode n. De la même façon, l'expression $m|n$ permet, soit une seule exécution de la méthode m, soit une seule exécution de la méthode n. L'expression $m*$ autorise la répétition de la méthode m entre 0 et n fois. Finalement, plusieurs instances d'une expression entre accolades peuvent être exécutées simultanément. Par exemple, $(\{lire\}|écrire)*$ traduit une politique d'exécution de type lecteurs/écrivain dans laquelle, soit plusieurs instances de la méthode *lire* s'exécutent simultanément, soit une seule instance de la méthode *écrire* s'exécute. Un certain nombre d'extensions au mécanisme de base des expressions de chemin ont rendu ce modèle plus expressif. Par exemple, les langage PROCOL [vdBL89] et PathPascal [CK80] introduisent des gardes pour chaque élément d'une expression de chemin. Les langages de définition de protocoles [AG94b, AG94a, Bok96] suivent une démarche similaire mais s'intéressent à la synchronisation des interactions entre sites distants plutôt qu'à la synchronisation locale d'un objet. Ils permettent de définir toutes les séquences valides de messages échangés par deux ou plusieurs entités au cours de l'exécution d'un protocole. Cette technique peut permettre l'implantation d'une spécification à base de logique temporelle d'actions (Cf. paragraphe 2.2.2) qui utilise, lui-aussi, la notion de séquences d'opérations.

Comme leur nom l'indique, les expressions de chemin décrivent donc un ensemble de chemins pouvant être suivis par les exécutions de méthodes au sein d'un objet. Cette description exhaustive peut néanmoins se révéler fastidieuse lorsque le nombre de méthodes est important et que les possibilités d'entrelacements sont nombreuses. Ceci est particulièrement le cas lorsque le code est découpé en morceaux très courts devant, néanmoins, faire l'objet de règles de synchronisation. Dans ce cas, on leur préfère en général des approches orientées par les données de type commandes gardées.

1.1.3.4 Commandes gardées

Version de base

La notion de commande gardée [Dij75, Dij76] a été définie par Dijkstra en 1975. Dans cette approche, chaque commande, opération ou méthode est associée à une expression booléenne appelée garde. A chaque invocation, la garde associée à la méthode est évaluée. Si l'évaluation est positive, l'invocation est acceptée, sinon elle est mise en attente et sa garde est évaluée ultérieurement. Plusieurs politiques de réévaluation des gardes peuvent être mises en place. Les réévaluations peuvent par exemple être périodiques. Elles peuvent également être déclenchées lors de l'occurrence d'un événement particulier. Cet événement peut être la modification d'un élément de la garde, la modification de l'état de l'objet, le début ou la fin d'une activité au sein de l'objet. L'ordre de réévaluation des gardes en attente peut être quelconque, suivre un schéma de type première arrivée, première évaluée, ou être fondé sur un système de priorités.

Version avec utilisation de compteurs d'événements

La notion de compteur d'événements a été définie conjointement par Robert et Verjus [RV77] et Gerber [Ger77]. Ce mécanisme est employé par exemple dans les systèmes GUIDE [BBD+91] et DRAGOON [AGMB91, Atk91]. Il permet d'introduire un premier niveau dans la gestion des historiques d'exécution. Un compteur d'événement est une variable entière gérée par le système qui comptabilise le nombre d'occurrences d'un événement particulier. Par exemple, le système

GUIDE définit cinq compteurs : *invoked*, *started*, *completed*, *pending* et *current*. Ils comptabilisent pour chaque méthode respectivement, le nombre d'invocations arrivées, le nombre d'invocations commencées, le nombre d'invocations terminées, le nombre d'instances bloquées dans la file d'attente et le nombre d'instances en cours d'exécution. Ces compteurs peuvent être utilisés dans les gardes associées aux méthodes. On spécifie ainsi, en fonction de l'état interne de l'objet et de ses activités, les méthodes qui peuvent ou non être exécutées. Par exemple, dans une politique de type lecteurs/écrivain, une méthode *écrire* ne peut être exécutée que si aucune autre instance de la méthode *écrire*, ni aucune instance de la méthode *lire* ne sont en cours d'exécution. Avec les compteurs précédents, la condition d'activation de la méthode *écrire* s'écrit : *current(écrire)+current(lire) = 0*.

Le pouvoir d'expression d'une synchronisation à base de commandes gardées et de compteurs d'événements est nettement supérieur à celui des moniteurs ou des sémaphores. Néanmoins, lorsque la politique de synchronisation comporte un nombre élevé de configurations atteignables, chaque garde doit indiquer, pour toutes les configurations, si la méthode qui lui est associée est exécutable ou non. Il faut alors gérer un nombre très important de variables auxiliaires et le schéma de synchronisation est très difficile à comprendre par une lecture de conditions booléennes très longues. Cette technique est alors facilement sujette à erreurs. Pour des schémas de synchronisation complexes, on lui préfère en général une démarche à base d'états qui permet d'exprimer directement au niveau du langage les différentes configurations de la politique de synchronisation.

1.1.3.5 Utilisation de modèles états/transitions

De façon schématique, la démarche suivie par de nombreux langages orientés objets concurrents (comme ACT++ [KL89a, KL89b] et Rosette [TS89]) consiste à définir un ensemble d'états pour chaque classe. Ces états traduisent les différentes configurations possibles d'une politique de synchronisation. Chaque état accepte un ensemble de méthodes et les transitions entre états sont déclenchées par les exécutions de ces méthodes. Le contrôle d'une classe d'objet par un automate ou par un modèle à base d'états est employé dans de nombreuses approches : par exemple, [SB94] propose d'utiliser des réseaux de Pétri, tandis que les automates de Harel [Har87, Har88, HG96] sont employés par de nombreuses méthodologies de conception orientées objet comme Booch, OMT ou UML.

A titre d'exemple un peu plus détaillé, nous traitons ici le cas des ensembles acceptables. La notion d'ensemble acceptable ou *accept set* a été introduite par Kafura et Lee [KL89a, KL89b] dans le cadre de leur langage ACT++. Elle a été étendue par Tolimson et Singh [TS89] dans le langage Rosette. Un ensemble acceptable est un ensemble de méthodes qui peuvent être exécutées par un objet à un instant donné. Le langage ACT++ se présente comme une extension du langage C++. Chaque classe synchronisée définit les états valides et l'ensemble des méthodes acceptables pour chacun de ses états à l'intérieur d'une section appelée abstraction de comportement ou *behavior abstraction*. Les changements d'états sont spécifiés au sein du corps des méthodes par la primitive *become*. La figure 1.1 présente l'exemple simple d'une classe gérant un tampon de taille fixe. Cette classe hérite de la classe *ACTOR*, qui dans la terminologie ACT++, désigne une classe active[1]. Trois états sont définis : *vide*, *partiel* et *plein*. Dans le premier, on peut seulement ajouter un élément, donc l'ensemble acceptable comprend la méthode *put*. Dans l'état *plein*, on

1. Les objets ACT++ sont actifs et séquentiels. Ils possèdent leur propre flot de contrôle mais n'exécutent qu'une seule instance de méthode simultanément. Le parallélisme du langage ACT++ est donc uniquement de type inter-objets.

peut seulement en retirer un, donc la seule méthode acceptable est *get*. Finalement, dans l'état *partiel*, les deux méthodes sont acceptables.

```
class buf: ACTOR {
 int in, out, buf[SIZE];
behavior:
 vide    = {put};
 partiel = {put, get};
 plein   = {get};
public:
 void buf() {
  in = out = 0;
  become vide;
 }
void put() {
 in++;
 // Ajout
 if ( in == out+size ) become plein;
 else                  become partiel;
}
void get() {
 out++;
 // Retrait
 if ( in == out ) become vide;
 else             become partiel;
}}
```

FIG. 1.1 – *Classe tampon synchronisé dans le langage ACT++*

Dans l'approche de Kafura et Lee, les ensembles acceptables sont des éléments structurels du langage de programmation. L'extension introduite par Tolimson et Singh dans Rosette permet de traiter ces ensembles qu'ils appellent ensembles sensibilisés ou *enabled sets*, comme des objets. Les opérateurs ensemblistes habituels d'union et d'intersection sont alors disponibles pour manipuler les ensembles de méthodes acceptables.

Bien que la technique des ensembles acceptables apporte une grande souplesse dans l'écriture des politiques de synchronisation, son principal inconvénient est de ne pas séparer clairement les codes de synchronisation et de celui des traitements (ils sont définis au sein d'une seule et même classe). Cette approche limite la réutilisation de ces deux codes et entraîne un certain nombre de problèmes désignés dans la littérature sous le terme d'anomalie d'héritage (Cf. paragraphe 1.1.4). Ce problème est en général résolu par l'emploi d'une programmation dite réflexive pour la définition du code de synchronisation.

1.1.3.6 Approches réflexives

La notion de réflexivité est issue des travaux en intelligence artificielle. Une étude complète de ses différentes caractéristiques est en dehors du cadre de ce mémoire. Nous nous contentons donc d'en citer les aspects qui concernent la synchronisation d'objets concurrents.

La réflexivité consiste à distinguer dans une application le niveau des données de celui des programmes. Ainsi, les variables sont des données tandis que les types, modules, procédures, fonctions et méthodes sont au niveau des programmes. Ils en constituent la structure. Dans la plupart des langages procéduraux (par exemple C, Pascal ou Ada), ces éléments ne peuvent pas faire l'objet de traitements au même titre que des données. L'approche objet introduit, avec des langages comme Smalltalk [GR83], CLOS [BDB+88] ou ABCL/R [WY88, Yon90], une démarche réflexive dans laquelle les programmes peuvent interroger leur propre structure ou celle d'autres programmes, pour pouvoir répondre à des questions comme :

- quels sont les identificateurs des champs d'un enregistrement de type t?
- quelles sont les procédures qui sont en cours d'exécution ?

– quel est le type de l'appelant de la méthode m qui est en cours d'exécution?

La programmation réflexive traite donc les types, modules, procédures, classes et méthodes comme des données et l'interrogation et/ou la modification de ces données sont considérées comme des activités d'un niveau différent de celui programme et sont dites de niveau méta. Les systèmes et les langages qui supportent ce paradigme sont dits réflexifs. Ces systèmes peuvent être classés selon le degré de réflexivité qu'ils offrent. En LISP, tous les programmes sont considérés comme des données. Leur structure peut être interrogée et modifiée dynamiquement. Ainsi, de nouveaux programmes peuvent être créés et exécutés par un programme. En Smalltalk, la structure d'une classe est décrite dans une méta-classe dont la classe est une instance. La méta-classe possède alors des méthodes pour gérer les création, destruction et migration des instances de la classe de base et pour contrôler l'accès à ces instances. On peut donc redéfinir tous les mécanismes de base associés à cette classe, et en particulier le mécanisme de prise en compte des invocations. Il est alors possible d'introduire la politique de synchronisation de la classe concurrente dans cette redéfinition.

Le pouvoir d'expression des langages réflexifs est élevé. Ils permettent de traiter de façon élégante de nombreux problèmes algorithmiques complexes. Néanmoins, leur mise en place est souvent coûteuse et leurs performances médiocres par rapport à des langages comme C++. De ce fait, un certain nombre de ces langages limitent leur degré de réflexivité à un certain type de fonctionnalités. Par exemple, les langages Open C++ [Chi95], MetaJava [GK97a, GK97b], PC++ [WSMB95] ou CodA [McA95] permettent seulement de redéfinir le mécanisme de prise en compte des méthodes. Plutôt que réflexifs, ces langages sont en général désignés sous le terme de protocoles de niveau méta-objet [KdRB91]. C'est l'approche que nous avons retenue pour le modèle de synchronisation présenté au chapitre 6. Ce modèle utilise le concept de méta-objet et une approche à base d'états pour la synchronisation d'objets concurrents. Chaque invocation arrivant à destination de l'objet est prise en compte par le méta-objet. Celui-ci lui applique le schéma de synchronisation puis la délivre à l'objet de base. L'avantage d'un tel système est que l'on sépare le code de synchronisation du code de base. De ce fait, on facilite la modularité et l'héritage. L'inconvénient majeur de cette approche réside dans le surcoût imposé au traitement de chaque appel.

1.1.3.7 Conclusion sur la synchronisation

La synchronisation a pour but de coordonner les flots d'exécutions simultanés accédant à une donnée partagée. Dans le cadre de l'approche objet, elle concerne par exemple, le contrôle des exécutions de méthodes au sein d'un objet concurrent. Dans cette partie, nous avons présenté différents mécanismes permettant d'assurer une telle fonctionnalité. On peut tout d'abord distinguer les mécanismes comme les sémaphores et les moniteurs qui ont été mis au point en premier. Ils sont faciles à utiliser, leur emploi est largement répandu mais ils sont peu expressifs. Les expressions de chemin sont une solution alternative mais semble, pour l'instant, être délaissées au profit des approches à base de commandes gardées et de compteurs d'événements. Ces dernières permettent une synchronisation plus fine que celles permises par les approches précédentes, mais restent confinées à des projets de recherche et n'ont pas encore été adoptées par les langages de programmation grand public. La situation est la même pour les solutions à base d'ensembles acceptables. Finalement, les solutions réflexives offrent un pouvoir d'expression important qui permet d'exprimer la synchronisation de façon élégante. Elles permettent de définir de façon complète-

ment autonome des schémas de synchronisation, mais on peut, sans encourir trop de difficultés, y ajouter aussi des aspects intéressants des approches antérieures (identification d'états, ensembles acceptables, commandes gardées, gestion des historiques par les compteurs d'événements, ...). C'est la solution retenue pour le modèle de synchronisation que nous présentons au chapitre 6. On peut même conjecturer que les solutions réflexives, avec des langages tels que Open C++ ou MetaJava [GK97a, GK97b], qui ajoutent une programmation de type méta aux langages C++ et Java, sont destinées à un avenir prometteur.

1.1.4 Relations de réutilisation

Les bénéfices attendus de l'approche objet concernent essentiellement les domaines de la sûreté de fonctionnement, de la maintenance et de la réutilisation du code. A partir du découpage d'une application en classes définies vis à vis du monde extérieur (c'est à dire des autres classes) par des interfaces et encapsulant des données et des comportements permettant de répondre à la sémantique de ces interfaces, on espère ainsi faciliter la validation, la correction, l'évolution et la réutilisation du code. Dans ce paragraphe, nous n'examinons que les mécanismes de réutilisation. Les aspects et les politiques de maintenance ne sont pas abordés dans ce mémoire. Différentes techniques de réutilisation de code cohabitent et sont employées dans les langages objets. L'héritage est certainement la relation la plus courante, mais on trouve également les notions de composition, de délégation et de classe paramétrée. Bien que ce soit la réutilisation de comportements qui nous intéresse principalement dans cette thèse, nous l'étudions ici de façon générale.

1.1.4.1 Héritage

L'héritage permet de créer une nouvelle classe par dérivation d'une classe existante. De façon plus précise, l'héritage peut être vu comme:

> Une relation entre classes dans laquelle une classe partage la structure ou le comportement définis dans une (héritage simple) ou plusieurs (héritage multiple) autres classes. L'héritage définit une relation "est-un" entre classes dans laquelle une sous-classe hérite d'une ou plusieurs super-classes; typiquement une sous-classe spécialise sa super-classe en augmentant ou en redéfinissant sa structure ou son comportement.

Cette définition donnée par Booch [Boo94], est retenue dans des termes identiques par la quasi-totalité des auteurs. Si l'héritage de classe permet de réutiliser de façon pratique les comportements définis par des classes existantes, il est important de distinguer ce concept de celui de sous-typage, parfois désigné sous le terme d'héritage d'interface. Le type d'un objet définit l'ensemble des invocations qu'il accepte. La classe définit, quant à elle, la façon dont sont implantées ces invocations. Certains langages (par exemple C++, Eiffel, Smalltalk) ne font pas de différence entre une classe et son type. La classe est alors utilisée pour définir à la fois l'interface et l'implantation. D'autres langages (par exemple GUIDE, Emerald) séparent explicitement les hiérarchies de sous-typage et d'héritage. Dans la suite de ce paragraphe, nous définissons plus précisément ces deux notions.

Sous-typage (héritage d'interface)

Un type définit l'ensemble des attributs d'un objet et l'ensemble des opérations qui peuvent lui être appliquées. Chaque attribut possède un identificateur et est lui-même typé. Chaque opération (c'est à dire chaque méthode) possède une signature. Un sous-type T2 d'un type T1 fournit au moins toutes les opérations et tous les attributs du type T1 et peut éventuellement en ajouter.

Héritage de classe

Une classe définit une implantation particulière d'un type. Cette implantation est commune à tous les objets instances de cette classe. Un type peut être implanté par plusieurs classes. Réciproquement, selon le degré de visibilité que l'on désire attribuer à une classe, celle-ci peut être l'implantation de plusieurs interfaces, c'est à dire de plusieurs types. La classe fournit un ensemble de variables et le code des méthodes. Une classe C2 implantant un type T2 peut être une sous-classe de la classe C1 implantant le type T1. Dans ce cas :

- C2 hérite toutes les variables d'instance de C1 et en ajoute éventuellement,
- C2 hérite toutes les méthodes de C1 et les méthodes héritées peuvent être surchargées. Des méthodes peuvent être ajoutées dans C2 (en l'occurrence celles déclarées dans T2 et non présentes dans T1).

Héritage et sous-typage

Bien que dans l'exemple précédent, la classe C2 soit une sous-classe de la classe C1, le type T2 n'est pas nécessairement un sous-type de T1. En effet, la sous-classe réutilise certaines méthodes héritées de la super-classe, peut en ajouter mais peut également redéfinir certaines méthodes héritées. Il se peut alors que l'interface de la sous-classe ne contiennent plus l'interface de la super-classe et que donc, T2 ne soit pas un sous-type de T1. La différence entre héritage et sous-typage ne peut être levée que par l'emploi de la notion de conformité [CW85] pour les types. Cette notion définit les conditions qui permettent à un type d'être employé de façon sûre en lieu et place d'un autre type. On peut alors établir que lorsque le type de la sous-classe est conforme à celui de la super-classe, alors l'interface de la sous-classe contient l'interface de la super-classe et les instances de la sous-classe peuvent être utilisées à la place de celles de la super-classe.

La différence entre héritage et sous-typage a fait l'objet de nombreuses études (Cf. par exemple [CHC90] et [Lal91]) et apparaît en général lorsqu'une opération doit prendre un argument qui est de même type que `SELF`. Il peut être illustré par l'exemple de la figure 1.2 tiré de [BLR94] et exprimé dans la syntaxe du langage GUIDE. Dans cet exemple, la méthode `Concat` de la classe `Chapitre` redéfinit la méthode héritée de la classe `Document` et introduit une restriction supplémentaire : seul un chapitre peut être concaténé au chapitre courant (ceci se justifie par le fait que l'on a besoin, par exemple, de modifier le chapitre à ajouter). De ce fait, un `Chapitre` ne peut pas être passé en argument à une méthode qui s'attend à recevoir un `Document`. En effet, cette méthode pourrait essayer d'ajouter à cet argument un `Document`, ce qui ne marcherait pas puisqu'on ne peut pas ajouter un `Document` à un `Chapitre`. De ce fait, la classe `Chapitre` est bien une sous-classe de `Document`, mais le type `Chapitre` n'est pas un sous-type de `Document`, c'est à dire qu'il ne peut pas être employé en lieu et place d'un `Document`.

```
TYPE Document IS
 METHOD Concat
   (IN d:REF Document);
 ...
END Document.
```

```
TYPE Chapitre IS
 METHOD Concat( IN REF Chapitre );
 METHOD Modifier;
 ...
END Chapitre.
```

```
CLASS Document
IMPLEMENTS Document IS
 l: List OF REF Top;
 METHOD Concat
   (IN REF Document);
 BEGIN
  l.Append(d);
 END Concat;
 ...
END Document.
```

```
CLASS Chapitre
SUBCLASS OF Document
IMPLEMENTS Chapitre IS
 METHOD Concat( IN c:REF Chapitre );
 BEGIN
  c.Modifier;
  l.Append(c);
 END Concat;
 METHOD Modifier;
 ...
END Chapitre.
```

FIG. 1.2 – *Sous-typage et héritage*

Conclusion sur l'héritage

L'héritage présente un certain nombre d'avantages et d'inconvénients. Il est défini statiquement au moment de la compilation et s'exprime directement au niveau du langage de programmation. Néanmoins, il ne permet pas de changer les implantations héritées à l'exécution. Bien que la redéfinition des méthodes héritées puisse sembler être un avantage, elle expose la structure de la super-classe à ses sous-classes. Certains auteurs, comme par exemple [Sny86], ont donc fait remarquer qu'elle cassait d'une certaine façon le principe d'encapsulation.

1.1.4.2 Composition

Alors que l'héritage est un mécanisme de type boîte blanche dans lequel une sous-classe a accès à la structure interne d'une super-classe, la composition est un mécanisme de type boîte noire. La composition d'objets (on parle également d'agrégation) permet d'ajouter de nouvelles fonctionnalités à un objet en lui adjoignant de nouveaux composants. L'objet composé possède alors un ensemble de variables qui sont, soit les composants, soit des références à ces composants. Il n'a pas accès à la structure interne des composants et se contente de communiquer avec eux au travers de leurs interfaces.

Contrairement à l'héritage qui est statique, les références entre un composé et ses composants peuvent être résolues dynamiquement à l'exécution. Par ailleurs, chaque composé doit respecter les interfaces de ses composants. Cela impose une certaine discipline de programmation et donc, *in fine*, un code plus clair et plus lisible. La relation de composition favorise donc l'encapsulation et la clarté et est donc préférée à l'héritage par un certain nombre d'auteurs.

1.1.4.3 Délégation

Le mécanisme de délégation permet à une classe de déléguer le traitement d'une requête à une autre classe. De la même façon qu'une sous-classe délègue à une super-classe le traitement de méthodes héritées, le mécanisme de délégation est, en général, utilisé conjointement à la relation de composition pour aiguiller une requête d'un objet composé vers un de ses composants.

Dans le cas de l'héritage, ce mécanisme est traité automatiquement au niveau du langage. De plus, le traitant a automatiquement accès à l'objet récepteur par l'intermédiaire de la référence `self` (ou du pointeur `this` dans le cas de C++). Dans le cas de la délégation associée à la composition, cette fonctionnalité ne peut être obtenue qu'en transmettant explicitement la référence du composé aux composants au moment de la délégation. L'inconvénient de cette technique est qu'elle complexifie les codes et nuit à leur lisibilité. Néanmoins, elle semble plus souple que l'héritage, elle respecte le principe d'encapsulation, et elle permet de composer plus facilement les comportements à l'exécution.

1.1.4.4 Classes paramétrées

La notion de classe paramétrée est un autre mécanisme qui permet, comme l'héritage, la composition ou la délégation, de réutiliser des comportements. Contrairement à ces derniers, ce n'est pas à proprement parler une notion orientée objet. En effet, ce concept a d'abord été utilisé pour les éléments génériques dans les langages comme Ada ou Eiffel.

Cette technique permet de définir des patrons de classe. Chaque patron définit des données et des méthodes qui comportent un certain nombre d'éléments dont les types ne sont pas précisés. Par exemple, une classe *liste* peut être paramétrée en ne spécifiant pas le type de ses éléments. Celui-ci est alors un paramètre générique renseigné à l'instanciation de la classe. Ce mécanisme de liaison est réalisé statiquement lors de la compilation. Une classe générique ne peut pas systématiquement être instanciée avec n'importe quel type. En effet, chaque type candidat doit nécessairement implanter toutes les opérations utilisées dans la définition de la classe paramétrée. Par exemple, si la classe paramétrée *liste* définit une méthode qui affiche tous ses éléments en invoquant une méthode *afficher* sur chacun d'eux, il faut que le type servant à l'instanciation fournisse une telle méthode. Cela interdit par exemple l'utilisation d'un type entier. On dit alors que la généricité est contrainte par un type de base qui définit une interface minimale. L'instanciation ne peut alors se faire qu'avec un sous-type de ce type de base.

1.1.4.5 Conclusion sur les relations de réutilisation

Nous avons vu dans ce paragraphe que plusieurs techniques sont disponibles pour réutiliser des comportements : l'héritage, la composition, la délégation et les classes paramétrées. Chacune de ces relations présente des avantages et des inconvénients. En résumé, l'héritage est directement intégré au niveau langage mais viole le principe d'encapsulation. La composition associée à la délégation respecte l'encapsulation mais requiert, de la part des programmeurs, un certain nombre d'additions manuelles. Les classes paramétrées sont pratiques mais ont un champ d'application limité. Le choix d'une de ces relations dépend donc des objectifs des concepteurs et des programmeurs. On peut néanmoins dire que la composition est la relation qui respecte le plus les principes de l'approche objet tandis que l'héritage, de part sa souplesse et sa facilité d'utilisation, est la relation la plus couramment employée.

1.1.5 Héritage et synchronisation

Dans ce paragraphe, nous examinons les liens entre les notions d'héritage et les propriétés de synchronisation d'un langage à objets concurrents. Nous présentons essentiellement les problèmes désignés dans la littérature sous le terme d'anomalie d'héritage (*inheritance anomaly* en anglais). Ceux-ci désigne les différentes limitations qui apparaissent lorsque l'on introduit la concurrence dans un langage à objets. De nombreux auteurs ont proposé des solutions plus ou moins complètes à ces problèmes. Sans être exhaustif, on peut citer [KL89a, TS89, AvdL90, GW91, Frø92]. De façon extrême, ces conflits ont amené certains auteurs à supprimer complètement l'héritage de leur langage à objets concurrents. C'est par exemple le cas de la famille des langages d'acteurs comme Act/1 [Lie87] et des langages POOL [Ame87], PROCOL [vdBL89] et ABCL/1 [YSTH87, Yon90].

Deux catégories de code peuvent être distingués dans un objet concurrent : le code des traitements et le code de synchronisation. La première désigne les traitements effectifs associés à une méthode tandis que la seconde désigne les conditions qui autorisent ou interdisent la prise en compte d'une méthode concurrente. L'anomalie d'héritage désigne alors les différentes redéfinitions du code de synchronisation qu'il est nécessaire d'opérer lorsqu'on souhaite hériter d'une classe concurrente. Matsuoka et Yonezawa dans [MY93] ont décelé trois causes principales qui entraînent l'anomalie d'héritage. Elles sont désignées respectivement sous les termes d'historique des invocations de méthodes, de partitionnement des états acceptables et de modification des états acceptables. Ils montrent qu'en employant des mécanismes de synchronisation à base d'expressions de chemin ou d'ensembles acceptables (Cf. paragraphe 1.1.3), une part importante, voire la totalité, du code de synchronisation doit être redéfini (et donc, ne peut pas être hérité). Une solution réflexive permet, selon eux, d'optimiser le taux de redéfinition. Le modèle de synchronisation que nous proposons au chapitre 6 est fondé sur une approche partielle réflexive (seul le mécanisme de prise en compte d'invocations peut être géré à un niveau méta) et nous montrons au paragraphe 6.3.7.4 qu'il résout les anomalies d'héritage de façon satisfaisante.

1.1.5.1 Historique des invocations de méthodes

Avec des mécanismes de synchronisation simple tels que les sémaphores, les moniteurs, les commandes gardées ou les ensembles acceptables, les conditions d'activation des méthodes sont fondées sur l'état interne des objets. Celui-ci est une fonction entre un ensemble de variables d'instances (celles déclarées dans l'objet et celles héritées) et un ensemble de valeurs. Les compteurs d'événements étendent cette approche en fournissant un résumé très simplifié du passé causal de l'objet. Néanmoins, une simple analyse de l'état courant de l'objet n'est pas toujours suffisante pour accepter ou non une méthode. Par exemple, un tampon synchronisé possédant un méthode *Put* pour ajouter un élément et une méthode *Get* pour en retirer un, peut être étendu avec une méthode *Gget* pour retirer un élément seulement si la précédente méthode exécutée est de type *Get*. L'état interne doit alors être raffiné, en ajoutant par exemple une variable booléenne *après-get*, afin de pouvoir distinguer la dernière méthode exécutée. Cette solution n'est néanmoins pas transparente pour les méthodes héritées *Put* et *Get*. Leur code de synchronisation doit être redéfini afin, d'une part, de faire en sorte que la nouvelle méthode s'exécute en exclusion mutuelle avec les anciennes et, d'autre part, de gérer la nouvelle variable *après-get*. Matsuoka et Yonezawa montrent que cette anomalie peut être levée à l'aide d'une approche réflexive. Le niveau méta gère le point de vue de la dernière méthode exécutée tandis que la synchronisation initiale est conservée telle quelle dans l'objet de base.

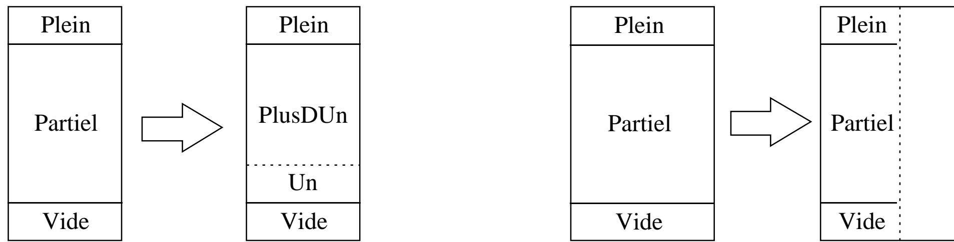


FIG. 1.3 – *Partitionnement et modification des états acceptables*

1.1.5.2 Partitionnement des états acceptables

L'ensemble des configurations d'un objet peut être découpé en états. Par exemple, le tampon synchronisé accepte trois états : *Vide*, *Partiel* et *Plein*. Ces états s'expriment directement au niveau langage dans les solutions à base d'ensembles acceptables, ou par l'intermédiaire de variables dans le cas des gardes, des moniteurs ou des sémaphores. La seconde anomalie apparaît lorsque l'ajout d'une méthode supplémentaire entraîne le partitionnement d'un état. Par exemple, on peut souhaiter ajouter une méthode *Get2* au tampon pour retirer deux éléments simultanément. Cette méthode n'étant acceptable que si le tampon contient plus d'un élément, l'état *Partiel* se retrouve divisé en deux sous-états : *Un* et *PlusDUn*. Le nouveau partitionnement doit être répercuté dans toutes les conditions d'activation qui utilisaient précédemment l'état *Partiel*. En pratique, cela impose de redéfinir le code de synchronisation des méthodes *Put* et *Get*. Nous montrons au paragraphe 6.3.7.4 que cette anomalie peut être résolue en intégrant directement au niveau du langage la notion de sous-état et en définissant les formules de partitionnement d'un sur-état. De cette façon, les changements d'états du modèle hérité peuvent être redéfinis automatiquement par le compilateur.

1.1.5.3 Modification des états acceptables

La troisième anomalie apparaît lorsque l'ajout d'une méthode modifie les états d'une politique de synchronisation. Par exemple, l'ajout de deux méthodes *Lock* et *Unlock* pour bloquer et autoriser l'accès au tampon entraîne une modification des états *Vide*, *Partiel* et *Plein*. En effet, après un *Lock*, plus aucune méthode n'est acceptable dans l'un quelconque de ces trois états. Une solution standard à base de conditions d'activation impose donc une réécriture de toutes les gardes utilisant ces états. En pratique, cela impose une redéfinition des méthodes *Put* et *Get*. Cette anomalie peut être résolue de la même façon que la première par l'emploi d'une approche réflexive. Une synchronisation de niveau méta gère le point de vue des méthodes *Lock* et *Unlock* et autorise ou interdit tout autre méthode. Ce niveau peut alors être associé à la synchronisation initiale qui n'a pas lieu d'être modifiée.

1.1.5.4 Conclusion sur l'héritage et la synchronisation

Les concepts d'héritage et de synchronisation ne sont pas complètement orthogonaux. Les auteurs de l'étude majeure dans ce domaine [MY93], ont montré que l'introduction de la concurrence dans un langage de programmation orienté objet ne se faisait pas sans entraîner un certain nombre de limitations dans le volume de code hérité. Ces limitations, désignées dans la littérature sous le terme d'anomalies d'héritage, concernent le code de synchronisation (par opposition au code des

traitements) qui détermine les conditions d'acceptation d'une invocation de méthode par un objet concurrent. Il ressort des nombreuses études publiées que la solution la plus efficace pour résoudre ce problème repose sur l'utilisation d'un niveau de programmation réflexif. Le code des traitements constitue un niveau de base qui est manipulé à un niveau méta par le code de synchronisation. Au chapitre 6, nous proposons une solution de ce type pour les objets concurrents du système distribué GUIDE.

1.1.6 Conclusion sur les langages orientés objet

Dans cette partie, nous avons présenté les caractéristiques principales des langages de programmation orientés objets. La principale notion apportée par cette approche est celle d'encapsulation : chaque objet encapsule une structure de données qui est accédée par un ensemble de méthodes.

Puis, nous avons présenté l'introduction de la concurrence dans les langages de programmation orientés objets. Deux grandes catégories peuvent être dégagées : les langages à objets actifs et ceux à objets passifs. Dans le premier cas, les flots de contrôle qui exécutent les invocations de méthodes sont propres aux objets, tandis que, dans le second cas, les flots d'exécution sont des structures indépendantes qui se propagent d'objet en objet au fil des invocations. On classe également les langages objet selon le degré de concurrence qu'ils offrent. On distingue ainsi la concurrence intra-objet, de la concurrence inter-objets. Dans le premier cas, chaque objet de l'environnement peut exécuter plusieurs activités, tandis que dans le second, plusieurs objets peuvent s'exécuter concurremment dans l'environnement. Notons enfin, que dans la plupart des cas, les langages modernes offrent à la fois un degré de concurrence intra et inter-objets.

La présence de plusieurs flots d'activité au sein d'une même application requiert, dans certains cas, un contrôle de leur exécution. Par exemple, il peut être nécessaire de protéger une zone à accès exclusif ou de préserver la cohérence de données partagées. Nous avons donc présenté sept techniques de synchronisation d'activités concurrentes. Ce sont des techniques de base, comme par exemple les sémaphores ou les moniteurs. Nous avons également examiné des techniques plus évoluées comme les expressions de chemin, les commandes gardées ou les compteurs d'événements. Finalement, nous avons envisagé des approches de haut niveau comme celles à base d'états acceptables ou celles à base de réflexivité.

L'un des bénéfices attendus de l'approche objet concerne la réduction des temps de développement. Nous avons donc présenté l'héritage, la composition, la délégation et le paramétrage. Ces différentes relations permettent de réutiliser les composants logiciels.

Finalement, nous avons évoqué l'impact de la concurrence sur un langage orienté objet. En effet, les notions d'héritage et de concurrence ne sont pas orthogonales. Nous avons présenté les différentes causes, désignées dans la littérature sous le terme d'anomalies d'héritage, qui imposent une redéfinition des objets concurrents hérités.

1.2 Environnements pour les objets distribués

Dans cette partie nous présentons les principaux concepts et mécanismes systèmes et réseaux des environnements de programmation distribués. Nous illustrons notre propos par une présentation du standard CORBA.

1.2.1 Introduction

Par rapport aux environnements centralisés, les environnements distribués présentent un certain nombre de caractéristiques spécifiques qui influencent l'architecture des applications. Parmi les principales, on peut citer l'absence de mémoire commune, l'absence d'horloge globale et des canaux de communication non nécessairement fiables. Les environnements distribués prennent en compte ces spécificités en fournissant par exemple, des mécanismes de désignation, d'horloge logique, d'invocation, ainsi que des services de persistance de données et d'exécution transactionnelle.

Deux tendances majeures peuvent être dégagées pour ces environnements. C'est, d'une part, l'adoption de l'approche objet et d'autre part, la percée des normes d'interopérabilité pour les systèmes ouverts. Depuis quelques années, le paradigme le plus couramment employé pour les applications distribuées est celui d'un ensemble de composants autonomes qui échangent des informations à l'aide d'opérations de communication. Chaque composant exécute un comportement qui lui est propre. La résolution du problème (c'est à dire le service) provient de la mise en commun de cet ensemble de comportements locaux à l'aide d'interactions (c'est à dire d'un protocole). On retrouve donc la définition de l'approche objet. En outre, les principes d'encapsulation des données et des traitements dans une instance, et les principes d'accès à cette instance au travers d'une interface, fournissent un cadre naturel qui s'adapte bien à ce type de paradigme. La seconde tendance, qui apparaît dans le domaine des environnements distribués, est l'essor des normes d'interopérabilité telles que CORBA ou DCE. Il semblerait que l'engouement qui accompagne l'adoption de ces standards par les principaux acteurs du marché de l'informatique et des réseaux, se fasse au détriment des systèmes répartis issus de projets de recherche tels que Amoeba, Spring, Chorus ou GUIDE. Bien que ces derniers possèdent, dans certains cas, une avance technologique incontestable et apportent des concepts novateurs que l'on ne retrouve pas encore dans les normes d'interopérabilité, ils présentent l'inconvénient d'être bien souvent liés à une architecture précise et, donc, d'être relativement fermés. En diffusant largement les spécifications de leurs mécanismes de base et en favorisant l'intégration de différentes architectures et de différents langages, les standards d'interopérabilité s'adressent à une audience plus large. Elles s'adaptent parfaitement à la réalité des réseaux locaux et globaux actuels, qui se caractérisent par une diversité des architectures, des systèmes et des langages interconnectés. De plus, elles s'intègrent bien au modèle de référence pour les systèmes distribués ODP [UIT95, FH94]. Ce modèle fournit un cadre conceptuel pour la spécification d'une architecture de systèmes répartis en environnement hétérogène. Il définit cinq points de vues pour mener à bien cette tâche :

- le point de vue de l'entreprise exprime les objectifs et les obligations des entités qui composent l'application,
- le point de vue de l'information définit la sémantique de l'application,
- le point de vue des traitements traduit une vision fonctionnelle de l'application,
- le point de vue de l'ingénierie décrit les services de base des infrastructures,
- le point de vue de la technologie exprime les contraintes technologiques.

Il est couramment admis que les standards d'interopérabilité tels que CORBA fournissent un cadre naturel pour le point de vue des traitements. Dans la suite de cette partie, nous présentons les principales caractéristiques des environnements CORBA.

1.2.2 CORBA

Le standard CORBA (*Common Object Request Broker Architecture*) [OMG95, Sie96, OHE96] définit une architecture, des fonctionnalités et des services qui permettent d'intégrer selon un mode client/serveur les objets d'une application distribuée dans un environnement hétérogène. Son développement est assurée par l'OMG (*Object Management Group*). Depuis 1989, ce consortium de plus de 400 membres, regroupe les acteurs principaux du marché de l'informatique et des réseaux (à l'exception de Microsoft). Deux versions majeures (1.1 et 2.0) de la norme ont été développées. Les deux caractéristiques fondamentales de CORBA sont constituées par le langage de définition d'interface IDL qui permet de spécifier de façon universelle les interfaces des services offerts par les objets, et par le protocole IIOP qui fournit une interopérabilité entre les différentes plateformes. Plus d'une vingtaine d'implantations de CORBA sont, à ce jour, proposées par des sociétés informatiques ou par des organismes de recherche.

1.2.2.1 Architecture

L'architecture générale proposée par CORBA (Cf. figure 1.4) est désignée sous le terme OMA (*Object Management Architecture*). Elle s'organise autour de quatre catégories de fonctionnalités fournies respectivement par : un bus à objets, un ensemble de services, un ensemble de facilités communes et des objets de métiers.

- Le bus à objets : fournit les mécanismes systèmes et protocolaires qui permettent à des objets distribués d'interopérer dans un environnement hétérogène. Il permet de masquer la localité des différentes méthodes invoquées. Ainsi, les utilisateurs émettent, comme dans les versions centralisées des langages objet, des appels de méthodes sur des instances dont ils connaissent la référence. Le bus se charge de déterminer, à partir de cette référence, la localisation de l'objet et d'effectuer soit un appel local, soit un appel distant.

- Les services : définissent un ensemble de fonctionnalités système pour les objets prenant part à la réalisation d'une application distribuée. Seize services sont proposés par l'OMG. Ils se répartissent en quatre catégories : gestion d'objets, gestion de concurrence, gestion de base de données et gestion système. Les services de gestion d'objets comprennent les services de désignation, de courtier, d'événements et de cycle de vie. Ils permettent respectivement d'organiser les objets distribués en annuaires de type pages blanches ou de type pages jaunes, de gérer la diffusion d'événements et de gérer les opérations de copie et de déplacement d'objets. Les services de gestion de concurrence comprennent le service de transaction pour mettre à jour de façon cohérente des données avec un protocole de validation à deux phases et un service de concurrence pour gérer des verrous d'accès à des zones protégées. Les services de gestion de base de données comprennent les services de persistance, de base de données objets, de requêtes et de relations. Ils fournissent les fonctionnalités que l'on trouve habituellement dans une base de données. Les services de gestion système comprennent les services d'externalisation, de gestion de licence, de propriétés, de temps, de sécurité et de numéros de version. Ces services permettent de gérer l'environnement des objets distribués.

- Les facilités communes : définissent un ensemble de fonctionnalités pour la manipulation de composants à un niveau applicatif. Ce sont des ensembles de classes qui fournissent des services prêts à l'emploi pour les applications. L'OMG définit deux catégories de facilités : les

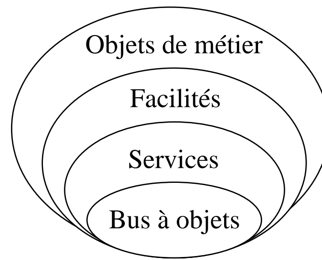


FIG. 1.4 – *Architecture d'un environnement selon la norme CORBA*

facilités verticales et les facilités horizontales. Les premières proposent des services pour des segments de marchés particuliers tel le commerce, la santé ou encore les télécommunications. Les facilités horizontales définissent, quant à elles, des services communs à tout type d'applications. Quatre catégories de facilités horizontales ont été mises en place. Elles s'adressent respectivement aux problèmes d'interface utilisateur, de gestion d'information, de gestion de tâches et de gestion système. A ce jour, l'OMG a retenu la solution OpenDoc [CI 97], pour les facilités interface utilisateur et gestion d'information. Cette technologie, issue de travaux communs entre Apple et IBM, définit un ensemble de primitives et de concepts permettant de gérer, entre autre choses, la création, la mise à jour et la sauvegarde de documents composites. Les deux autres facilités sont en cours de définition.

– Les objets de métier : décrivent des comportements standards que l'on retrouve dans de nombreuses applications. Ils concernent essentiellement l'informatique de gestion. Les objets de métier envisagés sont par exemple une commande, un règlement, un article ou un fournisseur. L'OMG propose de standardiser leur définition afin de faciliter le développement d'applications distribuées orientées objet. A ce jour, la plupart des objets d'applications n'ont pas encore été complètement normalisés.

1.2.2.2 Langage de définition d'interface

Le langage de définition d'interface de CORBA permet de séparer les interfaces des composants logiciels de leurs implantations. Chaque objet possède donc une interface écrite en langage IDL et une implantation écrite dans l'un des langages supportés par CORBA. L'interface décrit les types des services exportés par l'objet, tandis que l'implantation définit la façon dont ils sont réalisés. Le langage IDL permet de spécifier les attributs (c'est à dire les variables publiques) d'un composant, les classes dont il hérite la structure, les exceptions qu'il retourne et les méthodes qu'il fournit. Il permet également de déclarer des types et des constantes. Sa syntaxe est proche de celle de C++. Néanmoins, certains mots clés sont différents et d'autres ont été ajoutés. Il supporte les fonctions habituelles d'inclusion de fichiers, de compilation conditionnelle et de définition de macro-instructions. Il est organisé autour de la notion de modules et d'interfaces. Chaque module est un espace de nommage et permet de définir des types de données utilisateur, des constantes, des exceptions, des interfaces et d'autres modules. Les interfaces décrivent, quant à elles, les opérations et attributs des objets serveurs.

Le langage IDL est purement déclaratif. Il ne fournit aucun détail sur l'implantation des objets serveurs. Il permet ainsi d'intégrer des objets issus de langage de programmation différents. Une traduction (un *mapping* dans la terminologie CORBA) de l'IDL est définie pour un certain nombre

de langages. Actuellement, CORBA supporte les langages C, Ada, C++, Smalltalk, Java, Ada 95 et Object-Cobol.

1.2.2.3 Protocoles d'interopérabilité

Le but principal d'un bus à objets tel que CORBA, est de masquer la localité des différentes méthodes invoquées. Ainsi, les utilisateurs émettent, comme dans les versions centralisées des langages objet, des appels de méthodes sur des instances dont ils connaissent la référence. Le bus se charge de déterminer, à partir de cette référence, la localisation de l'objet et d'effectuer soit un appel local, soit un appel distant. L'utilisateur n'a donc pas à se préoccuper de la façon dont le système réalise cette invocation. Néanmoins, il peut être intéressant de décrire ces mécanismes afin de cerner clairement les différents possibilités d'interopérabilité offerts par les environnements CORBA.

Les mécanismes protocolaires servant de support aux invocations de méthodes sont répartis en trois types d'éléments : le protocole général inter-ORBs ou *General Inter-ORB Protocol* (GIOP), le protocole Internet inter-ORBs ou *Internet Inter-ORB Protocol* (IIOP) et les protocoles spécifiques inter-ORBs ou *Environment-Specific Inter-ORB Protocols* (DCE/ESIOP).

- le protocole général inter-ORBs : définit un ensemble de messages et un format de représentation des données pour les communications entre ORBs. Il a été conçu de façon à fonctionner au-dessus de n'importe quel protocole de transport en mode connecté. Sept types de messages différents permettent de réaliser les envois de requêtes et de réponses selon différentes sémantiques. Les paramètres des invocations sont codés à l'aide de la syntaxe de représentation *Common Data Representation* (CDR). Cette représentation définit une structure de message pour chaque type de donnée du langage IDL.

- le protocole Internet inter-ORBs : spécifie la façon dont sont échangés les messages GIOP sur un réseau de type TCP/IP. Ce protocole permet d'utiliser de façon simple tout réseau local ou global proposant ces protocoles comme support pour un environnement CORBA. Il permet également de se servir de l'Internet comme d'une épine dorsale pour la communication entre différents ORBs.

- les protocoles spécifiques inter-ORBs : sont des protocoles qui peuvent être utilisés en lieu et place d'IIOP. Par exemple, le protocole DCE/ESIOP est l'un des protocoles envisagé par CORBA pour remplir ce rôle. Les structures de messages définies par le format CDR sont alors traduites dans le format de ce protocole (NDR ou *Network Data Representation* dans le cadre de DCE). D'autres protocoles comme les RPCs ONC ou HTTP sont également envisageables.

1.2.3 Conclusion sur les environnements pour les objets distribués

Les systèmes répartis offrent un ensemble de services et de mécanismes qui permettent à des entités localisées sur des sites distants de communiquer. Ils ont pour but de masquer, autant que faire se peut, la distribution. Ils disposent, par exemple, de services de nommage et de localisation qui permettent de fournir une seule et même sémantique pour les communications locales et pour les communications distantes. Les principes d'encapsulation des données et d'interface de communication font que le paradigme objet est maintenant presque systématiquement retenu

comme base de programmation pour de tels environnements. De plus, les systèmes répartis dédiés tels que Amoeba, Spring ou GUIDE semblent céder la place à des standards ouverts tels que DCE ou CORBA dans lesquels les constructeurs informatiques et les sociétés de logiciels se sont massivement investis.

Dans cette partie, nous avons présenté les principales caractéristiques de ce dernier. Ses deux points forts sont constitués par la définition d'un langage de définition d'interfaces ou IDL, et par la spécification d'un protocole d'interopérabilité entre environnements hétérogènes (mais utilisant une pile de protocoles TCP/IP) ou IIOP. Le premier permet de masquer l'hétérogénéité des langages de programmation. Il définit, indépendamment de la façon dont ils sont implantés, l'interface des services mis à disposition par les objets serveurs sur le réseau. On peut alors être en présence, par exemple, d'un objet développé en C++ qui sert des requêtes pour le compte d'un objet client écrit en Smalltalk. Le second point fort du standard CORBA est constitué par le protocole de communication IIOP. Comme tout protocole de ce type, il permet de masquer l'hétérogénéité des systèmes mis en commun dans l'environnement distribué. Il spécifie un ensemble de messages et un format de représentation qui permet d'assurer les communications entre plateformes CORBA issues de vendeurs différents.

Néanmoins, la norme CORBA présente un certain nombre d'inconvénients qui pourraient freiner son développement. Par exemple, le volume des spécifications fournit par l'OMG est tel qu'à ce jour, il n'existe pas, à notre connaissance, d'implantation complète de la norme. Les vendeurs de solutions CORBA se contentent d'implanter les fonctionnalités principales du bus à objets (en laissant souvent de côté des fonctionnalités non essentiellement comme l'invocation dynamique de méthode ou l'interface dynamique de serveur) et quelques services. Peu d'implantations CORBA s'intéressent à la totalité des seize services évoqués au paragraphe 1.2.2.1 et encore moins, aux facilités communes ou aux objets de métiers. Le second frein au développement de CORBA provient de l'approche DCOM promue par Microsoft. Bien que par certains côtés cet environnement de programmation distribuée, qui emprunte de nombreux aspects à l'architecture DCE, soit moins avancé que CORBA, l'assise de Microsoft en fait un concurrent de poids. Finalement, le succès de CORBA ne peut provenir, certainement, que de la qualité des implantations proposées par les différents vendeurs ainsi que d'un effort de rationalisation des spécifications de la part de l'OMG.

1.3 Méthodologies de conception orientées objet

Les environnements de programmation présentés précédemment, tels que CORBA, offrent un ensemble d'outils qui facilitent la mise en place d'applications distribuées. Néanmoins, ils ne couvrent que les aspects liés à la programmation et n'abordent ni la phase d'analyse, ni celle de conception. L'OMG a d'ailleurs commencé à se préoccuper de ce problème en émettant un appel à soumission pour une notation orientée objet couvrant les phases d'analyse et de conception. La notation UML [BRJ97], présentée dans la suite de ce paragraphe, fait partie des principaux candidats à cette soumission.

De nombreuses méthodologies de conception ont été mises au point, tant dans le domaine procédural que dans le domaine objet (par exemple Graham dans [Gra93] recense et expose pas moins d'une trentaine de méthodologies orientées objet). Un certain nombre d'entre elles abordent la programmation distribuée en permettant, par exemple, de répartir un ensemble d'objets sur différents nœuds physiques d'un système, de définir différentes sémantiques d'invocation de méthodes ou de

concevoir des objets multi-tâches. Néanmoins, il semble que des problèmes plus pointus, tels que la gestion du parallélisme inter-objets, la conception de comportement globaux ou le contrôle des interactions relèvent encore pour l'instant du domaine de la recherche. Cette thèse va chercher à présenter, aux chapitres 4 et 5, des propositions dans ces domaines, donc nous présentons ici une synthèse des principaux concepts introduits par les méthodologies orientées objet existantes.

1.3.1 Analyse et conception orientées objet

Le but principal d'une méthodologie de conception et d'analyse est de permettre la construction de modèles d'applications informatiques. La démarche de modélisation, qui est tout aussi courante dans d'autres disciplines scientifiques que l'informatique, est motivée par la constatation simple que, plus la taille d'un système ou d'une application augmente, plus il est difficile pour une seule personne ou pour un groupe de personnes d'en maîtriser la complexité, d'y ajouter de nouvelles caractéristiques ou d'en corriger les erreurs. Les méthodologies proposent donc, entre autres, des étapes de développement, des cycles de vie, des notations, des outils de tests, des outils de preuve et des outils pour construire des modèles abstraits. Ces modèles représentent indépendamment du langage de programmation dans lequel ils sont destinés à être implantés, le comportement de l'application. Le but de cette modélisation est de faire apparaître de façon aussi claire que possible la structure et l'architecture (c'est à dire l'emboîtement des différents éléments) de l'application et de permettre la preuve mathématique de propriétés, comme par exemple la terminaison d'un programme ou le non débordement d'un indice.

Bien que certains points de détail puissent être spécifiques à telle ou telle méthodologie, il est possible d'extraire un ensemble de concepts fondamentaux repris sous une forme ou une autre par l'ensemble des méthodologies orientées objet existantes. Dans la suite de cette partie, nous en présentons un certain nombre. Nous illustrons notre propos à l'aide de la méthode Booch 93 [Boo94] et nous proposons une brève comparaison avec les méthodes OMT [RBP+91] et UML 1.0 [BRJ97]. Bien que ce choix puisse paraître réducteur vu le nombre élevé de propositions existantes, les méthodes Booch et OMT sont parmi les plus complètes du domaine et ont été éprouvées par de nombreux concepteurs. Finalement, la méthode UML, bien que récente, est certainement promise à un certain succès puisqu'elle se présente comme la synthèse des deux méthodes précédentes et de la méthode OOSE [JCJO92].

L'approche objet peut être déclinée selon trois axes complémentaires : l'analyse, la conception et la programmation. Le développement d'un logiciel comprend en général, en plus de ces trois aspects, les étapes suivantes :

1. établir un cahier des charges (conceptualiser),
2. développer un modèle des comportements à mettre en œuvre (analyser),
3. créer une architecture (concevoir),
4. implanter cette architecture (coder),
5. vérifier son comportement (tester),
6. faire évoluer l'application (maintenir).

Différents cycles de vie peuvent alors être mis en place autour de ces six axes tels que les cycles en cascade ou en spirale (Cf. entre autres [Des94]). L'approche objet n'introduit pas d'apport

notoire dans les étapes 1, 5 et 6. Nous nous intéressons donc exclusivement dans cette partie aux étapes d'analyse et de conception et nous n'évoquons que brièvement la phase de codage. Graham dans [Gra93] définit la phase d'analyse comme "la spécification des exigences des utilisateurs" tandis que Booch précise dans [Boo94] que l'analyse orientée objet doit permettre de répondre aux deux questions suivantes : "quel est le comportement attendu du système", c'est à dire quels sont les services qu'il rend, et "quels sont les rôles et les responsabilités des objets qui remplissent ce comportement". La conception, quant à elle, consiste à construire une architecture du système et, dans le cas de l'approche orientée objet, à identifier les objets du système, leurs attributs, leurs méthodes, à définir les interfaces de chaque objet et à établir les relations de visibilité entre les objets. Finalement, la programmation orientée objet consiste à implanter les programmes en terme d'ensembles d'objets coopérant entre eux. Comme ces définitions le suggèrent, l'analyse et la conception sont étroitement liées. La majorité des méthodes abordent d'ailleurs ces deux aspects sans parfois les distinguer vraiment. La programmation est, quant à elle, un domaine plus indépendant. On peut noter que bien qu'un langage objet soit un choix naturel pour implanter une analyse et une conception orientées objet, cette étape peut être conduite, au prix d'une certaine discipline d'écriture, à l'aide d'un langage non orienté objet (l'inverse étant moins courant et quelque peu anachronique). La plupart des méthodes comprennent deux volets principaux : elles définissent tout d'abord un ensemble de conseils méthodologiques qui servent de fil conducteur pour gérer le développement d'une application, puis elles proposent un ensemble de notations textuelles et/ou graphiques pour exprimer de façon concise la structure de l'application. Ces conseils et ces notations couvrent, dans la plupart des cas, les phases d'analyse et de conception. Puis, selon le degré de précision des notations, des squelettes de programmes plus ou moins complets peuvent être générés automatiquement. Dans la suite de cette partie, nous développons les notations et les conseils méthodologiques mis en place par Booch.

1.3.2 La méthode Booch

Booch [Boo94] organise les phases d'analyse et de conception orientées objet autour d'une notation et d'un processus de développement. La notation comprend essentiellement un ensemble de diagrammes qui présentent différentes facettes d'une application objet. Le processus de développement fournit un ensemble de conseils et de guides pour mener à bien la construction d'une application.

1.3.2.1 Notation

La notation proposée par Booch comprend quatre diagrammes pour décrire les aspects statiques d'une application : le diagramme de classes, le diagramme d'objets, le diagramme de modules et le diagramme de processus. Les deux premiers présentent une vue logique de l'application tandis que les deux suivants s'intéressent aux aspects physiques, c'est à dire par exemple, à la façon dont les différents composants sont répartis dans un système. Booch définit deux diagrammes supplémentaires pour décrire plus spécifiquement les aspects dynamiques d'une application : le diagramme états/transitions et le diagramme d'interactions.

Diagramme de classes

Le diagramme de classes définit les classes existantes, leurs structures et leurs relations de dépendance mutuelle. Chaque classe possède un nom, des attributs et des opérations. Trois types d'association sont disponibles pour relier les classes entre elles : l'héritage (relation de type "est-un"), l'agrégation (relation de type "a-un") et l'utilisation (relation de type "client-serveur"). La première permet de spécialiser les attributs et les opérations d'une sur-classe dans une sous-classe. La relation d'agrégation permet de construire une classe à partir de plusieurs autres classes composants. Finalement, la relation d'utilisation est employée lorsqu'une classe utilise les services d'une autre classe. Chaque classe peut être associée à une méta-classe qui est vue alors comme une classe de classes. Les classes peuvent être concurrentes. Quatre degrés de concurrence ont été définis par Booch : séquentiel, gardé, synchrone et actif. Néanmoins, l'absence de sémantique formelle pour ces modèles, rend difficile le passage d'un modèle conceptuel à un système ou un langage implantant ses propres mécanismes de gestion de la concurrence.

Diagramme d'objets

Le diagramme d'objets est une instanciation du diagramme de classes. Il présente les différentes entités qui participent à la réalisation de l'application et la façon dont elles interagissent. Chaque objet a un nom et des attributs. Les interactions sont représentées à l'aide de messages d'appels et de retours. Booch propose quatre sémantiques d'interactions : synchrone (le client attend indéfiniment que le serveur accepte le message), "balking" (le client abandonne le message si le serveur ne peut pas exécuter la requête immédiatement), à délai de garde (le client abandonne le message si le serveur ne peut pas exécuter la requête dans un intervalle de temps donné) et asynchrone (le client envoie le message au serveur et poursuit son exécution sans attendre la réponse). Ces différentes sémantiques permettent d'aborder la modélisation de systèmes temps réel réactifs.

Diagramme de modules

Le diagramme de modules présente la répartition des objets et des classes dans les différents modules du système. Un module est par exemple un fichier source. Booch distingue quatre types de modules : ceux contenant le programme principal, ceux contenant les interfaces des classes, ceux contenant le corps des classes et ceux contenant des agrégats d'interfaces ou d'implantation de classes. Les dépendances entre modules traduisent alors les liens entre les différentes parties de code et fournissent, entre autres, l'ordre des compilations.

Diagramme de processus

Le diagramme de processus présente l'allocation des processus sur les différents processeurs du système. Booch inclut, dans ce diagramme, une notation permettant de représenter des unités de traitements ainsi que des dispositifs physiques (modems, terminaux, ...). Ce diagramme n'offre qu'une vue statique de la répartition. Il ne propose aucune méthode permettant d'optimiser la répartition des différentes entités de l'application en fonction, par exemple, du nombre d'interactions ou de la taille des messages échangés. Le fait que ce diagramme s'intéresse avant tout aux processus, rend ce concept orthogonal à celui d'objet. Finalement, bien que ce diagramme concerne la répartition, il n'aborde pas vraiment le domaine de la conception ou de la coordination de comportements distribués.

Diagramme états/transitions

Les aspects dynamiques du diagramme de classes sont déterminés par le diagramme états/transitions. Il définit les états légitimes d'une classe, les événements qui déclenchent les transitions entre deux états et les actions qui sont entreprises lors d'un changement d'état. La notation utilisée pour ce diagramme est issue des automates ou *statecharts*, de Harel [Har87, Har88]. Bien que le but principal de ce diagramme soit de définir l'espace d'états d'une classe, Booch précise qu'il peut également être utilisé pour définir l'espace d'états du système dans sa totalité (on retrouve alors l'approche originelle définie par Harel). Seules les classes comportant un nombre élevé d'états et/ou des transitions ou un comportement complexe justifient la définition d'un tel diagramme.

Chaque état représente une configuration des variables de la classe. A chaque état sont associées deux actions *entry* et *exit* entreprises, respectivement, en entrant et en quittant l'état. Une transition est déclenchée par l'occurrence d'un événement interne ou externe, et/ou par l'évaluation d'une condition booléenne. Le mécanisme de déclenchement des transitions s'interprète de la façon suivante :

- si l'événement survient et si la condition s'évalue à vrai, alors l'action de sortie de l'état puis l'action associée à la transition sont entreprises.
- si l'événement survient et si la condition s'évalue à faux, alors la transition n'est réexaminée que lorsque l'événement se produit à nouveau.
- l'action de sortie de l'état n'influence pas le tir de la transition. Si elle introduit un effet de bord qui modifie la valeur de vérité de la condition, la transition est tout de même déclenchée et son action exécutée.
- l'événement et/ou la condition peuvent être omis. Si les deux le sont, alors la transition est déclenchée dès la fin de l'action d'entrée associée à l'état source.

Plusieurs transitions peuvent être issues du même état et un état peut posséder une transition vers lui-même. Néanmoins, une seule transition est déclenchée simultanément à partir d'un même état. La concurrence est représentée dans les automates de Harel par une décomposition de type “et” d'un état en plusieurs sous-états. De ce fait, chaque partition ne traite qu'une seule transition simultanément, mais le système évolue simultanément dans plusieurs partitions. Alors que Booch stipule clairement que certaines classes peuvent être concurrentes, c'est à dire peuvent exécuter simultanément plusieurs méthodes, il dit paradoxalement (Cf. page 208 de [Boo94]) que les décompositions de type “et” ne sont pas nécessaires dans les diagrammes états/transitions. De ce fait, la façon dont Booch gère le parallélisme et la synchronisation intra-objet ne nous semble pas claire. Le modèle états/transitions de synchronisation que nous proposons au chapitre 6 s'affranchit de cette limitation en abandonnant le postulat selon lequel une seule transition s'exécute concurremment au sein d'un modèle états/transitions et en autorisant l'activation concurrente de plusieurs états.

Diagramme d'interactions

Le diagramme d'interactions traduit l'aspect dynamique du diagramme d'objets. Il décrit les appels de méthodes entre objets et les différents scénarios d'exécutions possibles. Par rapport au diagramme d'objet, il n'introduit ni concepts nouveaux, ni notations nouvelles. Il reformule juste sous une forme plus temporelle la séquence des invocations échangées par les objets.

1.3.2.2 Processus de développement

Le processus de développement suggéré par Booch s'appuie sur ces six diagrammes. Il n'est ni complètement descendant, ni complètement ascendant. C'est un processus incrémental qui prend en compte des aller-retours entre différents niveaux d'abstractions et de raffinement. Les étapes suggérées consistent à :

1. identifier les classes et leurs instances,
2. définir leur sémantique (leur interface, leurs attributs),
3. définir les relations (d'association, d'agrégation, d'héritage) entre les classes,
4. implanter un prototype de cette architecture
5. tester la cohésion et la cohérence de ce prototype,
6. raffiner les classes, les instances, leurs sémantiques et leurs structures en fonction des différents retours sur expérience des phases précédentes.

Ce processus doit être itéré jusqu'à ce que le concepteur juge que les modèles proposés définissent de façon claire et satisfaisante les fonctionnalités essentielles de l'application. Booch insiste sur le fait que les six étapes évoquées ci-dessus ne sont pas strictement linéaires et peuvent s'influencer mutuellement. Par exemple, la définition des relations peut faire apparaître de nouvelles classes. De même, la définition des interfaces peut amener une modification de la hiérarchie d'héritage. Finalement, le raffinement suggéré au point six ci-dessus, n'implique pas une révision systématique de tous les éléments mis en place. Selon les cas, seules la structure ou la sémantique des classes seront par exemple réexaminées. De plus, cette notion n'est pas définie de manière formelle. Aucun cadre n'est proposé pour établir de façon mathématique que le raffinement est une implantation correcte du modèle proposé à un niveau supérieur.

1.3.3 Comparaison avec d'autres méthodologies

Dans ce paragraphe nous proposons une comparaison entre la méthode Booch présentée précédemment, et les méthodes OMT et UML. Nous indiquons les différences majeures entre ces méthodes.

1.3.3.1 OMT

La méthode OMT (*Object Modeling Technique*) [RBP+91], issue des travaux de Rumbaugh et de son équipe, couvre les phases d'analyse et de conception. Elle est organisée autour de trois axes principaux :

- l'analyse,
- la conception du système,
- la conception objet.

Comme pour toutes les méthodes, la démarche d'analyse proposée dans OMT s'appuie sur les spécifications fournies par un cahier des charges. Elle définit trois modèles : un modèle objet, un modèle dynamique et un modèle fonctionnel. Le modèle objet définit, comme les diagrammes de classes et d'objets de la méthode Booch, les classes et les instances présentes dans une application. Rumbaugh suggère en plus de constituer un dictionnaire qui contient la description de l'ensemble de classes, attributs et associations de l'application. Le modèle dynamique comprend un modèle états/transitions fondé sur une extension des automate de Harel et de même type que celui de la méthode Booch, auquel est associé un diagramme de flots d'événements. Ce modèle décrit les aspects dynamiques de chaque classe issue du modèle objet. Finalement, le modèle fonctionnel correspond à un diagramme de flots de données et un ensemble de contraintes sur ces données. Les modèles dynamiques et fonctionnels sont définis à partir des éléments présents dans le modèle objet. Leur description permet de mettre en avant un certain nombre de fonctionnalités et d'opérations qui n'avaient pas forcément été identifiées dans le modèle objet. Ces éléments doivent alors être introduits dans ce dernier modèle. Ce processus itératif se poursuit jusqu'à ce que le concepteur juge qu'un niveau de détail satisfaisant a été atteint.

Le second axe présent dans OMT concerne la conception de l'architecture de base du système. Cette phase consiste essentiellement à organiser les objets en sous-systèmes, à allouer les sous-systèmes à des processus ou à des tâches, à identifier à partir du modèle dynamique les tâches parallèles, à gérer la gestion des données en fichiers, mémoire et/ou bases de données, et à organiser l'utilisation des ressources physiques partagées. Cela correspond donc en partie au diagrammes de processus et de modules de la méthode Booch.

Finalement, lors de la conception objet, le concepteur est amené à détailler les modèles objet, dynamiques et fonctionnels définis lors de la phase d'analyse. Parmi les tâches à accomplir au cours de cette phase, il faut associer chaque événement du modèle dynamique et chaque processus du modèle fonctionnel à une opération du modèle objet, concevoir les algorithmes qui implantent ces opérations, optimiser les chemins d'accès aux données en éliminant par exemple les associations redondantes ou en sauvegardant des résultats intermédiaires pour éviter des recalculs inutiles, implanter le contrôle, ajuster la structure des classes afin d'optimiser la réutilisation, implanter les associations, déterminer la représentation des attributs et regrouper les classes en modules. Cet ensemble d'opérations permet de vérifier que rien n'a été omis dans la phase d'analyse.

OMT est une méthode très riche. Elle a certainement été la première méthode de conception orientée objet à connaître une large diffusion. Nous n'avons pas présenté sa syntaxe graphique qui comporte de nombreux icônes. Cette richesse entraîne parfois une certaine confusion qui la dessert. En partant de cette constatation, Booch a pris soin, dans sa méthode, de limiter le nombre des variantes graphiques, et surtout, de définir un sous-ensemble de la notation, appelée notation Booch Lite, qui rassemble les quelques icônes qui apparaissent le plus souvent.

1.3.3.2 UML

UML (*Unified Modeling Language*) [BRJ97] est un langage de modélisation qui unifie les concepts des méthodes Booch [Boo94], OMT [RBP+91] et OOSE [JCJO92]. Elle a été définie par Booch, Rumbaugh et Jacobson, auteurs respectifs de ces trois méthodes. Trois raisons principales ont incité les auteurs à faire converger leurs travaux. Tout d'abord, plutôt que de faire évoluer leurs méthodes indépendemment les unes de autres, ils ont préféré mettre leurs réflexions en commun afin d'introduire de concert, les mêmes extensions. Deuxièmement, ils espèrent appor-

ter une certaine stabilité dans le domaine de la modélisation objet qui ne compte pas moins d'une trentaine de propositions différentes. Finalement, ils souhaitent que leur collaboration apporte des solutions à des problèmes nouveaux que leurs méthodes précédentes ne permettaient pas d'envisager. Cette démarche d'unification a abouti à une notation qui, dans sa version actuelle (1.0), est en cours de standardisation par l'OMG. UML est donc très certainement destinée à être utilisée pour la conception d'applications CORBA. On peut également penser qu'elle serve de support au point de vue de l'information du modèle ODP (CORBA servant de support, quant à lui, au point de vue des traitements).

Le langage de modélisation UML est avant tout une notation plutôt qu'une méthode. L'unification mise en place par les auteurs concerne les éléments et la syntaxe textuelle et graphique. Ces éléments décrivent le détail d'une architecture logicielle plutôt que les étapes méthodologiques qui permettent d'aboutir à cette architecture. Les auteurs se contentent de suggérer de suivre une démarche de développement incrémentale et itérative, et de valider les différentes étapes à l'aide de scénarios. Ils justifient ce choix en faisant remarquer que, par exemple, le processus de développement d'une application bureautique et celui d'une application temps réel dur n'ont pas à répondre aux mêmes impératifs. Chaque processus de développement est donc spécifique au contexte, à la culture d'entreprise et au domaine d'application auquel il est appliqué. Il n'est donc pas possible d'en extraire des éléments méthodologiques communs. Néanmoins, les auteurs pensent que les définitions d'un méta-modèle et d'une notation uniques qui servent de support commun à tous les processus de développement, sont envisageable. Dans la suite de cette partie, nous donnons un aperçu des différents éléments présents dans cette notation.

UML définit huit diagrammes principaux que l'on peut répartir en quatre catégories qui concernent respectivement : les cas d'utilisation, les classes, les comportements et l'implantation. Les deux premières catégories comportent chacune un seul diagramme : les diagrammes de cas d'utilisation et les diagrammes de classes. La catégorie des comportements inclut, quant à elle, les diagrammes d'états, d'activités, de séquences et de collaboration. Finalement, la catégorie des diagrammes d'implantation définit les diagrammes de composants et de mise en place.

Catégories des diagrammes de cas d'utilisation

Les diagrammes de cas d'utilisation sont des descriptions d'interactions type entre un utilisateur et le système. La syntaxe et la sémantique de ces diagrammes sont identiques à ceux de la méthode OOSE. Jacobson définit un cas d'utilisation (*use case* en anglais) comme "un scénario dans lequel un utilisateur initie une série d'événements dans un système". Une analyse en terme de cas d'utilisation consiste donc à énumérer les principaux scénarios prévus dans une application. Un diagramme de cas d'utilisation comporte des acteurs et des scénarios d'utilisation. Par exemple, dans un système bancaire, les clients et les guichetiers sont des acteurs, tandis que l'action "accorder un prêt" est un scénario qui met en relation ces deux acteurs. Les diagrammes de cas d'utilisation peuvent être instanciés et implantés. L'implantation se fait en général à l'aide d'objets collaborant entre eux. Les échanges de messages entre objets sont modélisés par des diagrammes de collaboration.

Catégories des diagrammes de classes

Les diagrammes de classe définissent les fonctionnalités de l'application pour l'utilisateur final. Leur syntaxe est proche de celle d'OMT. Ils traduisent la structure statique des applications en

définissant les classes existantes, leur structure interne et leurs relations. Ils peuvent également contenir des instances. Le diagramme de classe présente un graphe statique de classes et/ou d'instances. S'il n'y a que des instances, on parle de diagramme d'objets. Néanmoins, contrairement aux diagrammes d'objets de la méthode Booch, les diagrammes d'objets d'UML ne traduisent pas les échanges dynamiques de messages entre instances. Ce point de vue est fourni par le diagramme de collaborations qui est donc le diagramme d'objets au sens de la méthode Booch. Celui-ci fait partie de la catégorie des diagrammes de comportement.

Les classes existent rarement de façon isolées. Dans la majorité des cas, elles sont reliées entre elles par différents liens sémantiques. UML distingue quatre types de relations entre classes : l'association, l'héritage, la dépendance et le raffinement.

- la relation d'association lie deux ou plusieurs classes dont les comportements se référencent entre eux.
- l'héritage désigne la relation de classification entre une classe générale et une classe plus spécifique. Les deux classes doivent présenter une cohérence sémantique entre elles. La sous-classe peut ajouter des informations à celles fournies par la super-classe. L'héritage peut être multiple. Dans ce cas une sous-classe possède plusieurs sur-classes. L'héritage peut être qualifié avec différents attributs (chevauchant, disjoint, complet, incomplet).
- la relation de dépendance traduit un lien logique entre classes. Contrairement à l'association, ce lien n'est pas instancié au niveau physique. Il est utilisé dans toutes les situations où un changement dans la classe cible entraîne une modification de la classe source.
- la relation de raffinement traduit un lien entre deux éléments identiques spécifiés avec deux niveaux de détails différents. Ce lien peut caractériser une relation entre un type et une classe qui l'implante (relation de réalisation), une classe issue du processus d'analyse et une classe issue du processus de conception (relation de conception), une construction de haut niveau et une construction de bas niveau (relation de précision), une construction et son implantation (relation d'implantation), une implantation de base et une implantation optimisée (relation d'optimisation).

Catégories des diagrammes de comportement

La catégorie des diagrammes de comportement traduit les aspects dynamiques d'une application. Elle comprend les diagrammes d'états, d'activités, de séquences et de collaboration.

Les diagrammes d'états sont des extensions des automates de Harel. Leur sémantique et leur syntaxe sont identiques à celles des diagrammes états/transitions de la méthode Booch et de la méthode OMT.

Les diagrammes d'activités sont, quant à eux, des diagrammes de flots de données. Ils sont identiques dans leur principe aux diagrammes de flots de données du modèle fonctionnel d'OMT. Chaque diagramme d'activité est attaché à une classe. Ils peuvent être vus comme des cas spéciaux de diagrammes d'états dans lesquels aucun événement extérieur à la classe n'intervient. Ils se focalisent sur les flots induits par l'exécution d'actions internes à la classe sans prendre en compte les événements extérieurs asynchrones. Leur syntaxe est identique à celle des diagrammes d'états.

Les diagrammes de séquences sont identiques dans leurs principes aux diagrammes d'interactions de la méthode Booch. Ils traduisent les interactions entre les objets participant à une

application. Leur but principal est de mettre en avant l'évolution et l'enchaînement temporel des messages échangés.

Les diagrammes de collaboration sont une adaptation des diagrammes d'objets de la méthode Booch. Ils décrivent les interactions entre les objets participant à la réalisation d'une application. Contrairement aux diagrammes de séquences qui prennent un point de vue temporel, les diagrammes de collaboration s'attachent avant tout, à mettre en avant les liens entre objets et les messages échangés au travers de ces liens.

Catégories des diagrammes d'implantation

La catégorie des diagrammes d'implantation regroupe les diagrammes de composants et les diagrammes de mise en place. Les premiers décrivent l'organisation du code des applications. Ils sont essentiellement destinés aux programmeurs. Les diagrammes de mise en place décrivent quant à eux, le déploiement des applications sur un réseau. Ils prennent en compte les aspects liés à la topologie, à l'intégration de systèmes et aux communications. La syntaxe et la sémantique de ces deux diagrammes sont proches de celles des diagrammes de modules et de processus de la méthode Booch.

1.3.4 Conclusion sur les méthodologies orientées objet

UML est une notation à la syntaxe très riche. Le tableau 1.1 résume les différents éléments qu'elle emprunte aux méthodologies dont elle est issue. Ainsi, les diagrammes de cas d'utilisation sont issues de la méthode OOSE. Les diagrammes de classes sont identiques au modèle objet d'OMT et constituent donc une synthèse des diagrammes de classes et d'objets de la méthode Booch. Les diagrammes d'états sont identiques à ceux du modèle dynamique d'OMT et proches des diagrammes états/transitions de la méthode Booch. Les diagrammes d'activité sont des diagrammes de flots de données issus du modèle fonctionnel d'OMT. Les diagrammes de séquences sont les diagrammes d'interaction de Booch tandis que les diagrammes de collaboration sont identiques aux diagrammes d'objets de Booch et donc, proches du modèle objet d'OMT. Finalement, les diagrammes de composants et de mise en place sont identiques aux diagrammes de modules et de processus de la méthode Booch.

UML		Booch	OMT	OOSE
cas d'utilisation				cas d'utilisation
classes		diagrammes classes + objets	modèle objet	
comportements	états	diagrammes états/transitions	mod. dynamique états/transitions	
	activités		mod. fonctionnel flots de données	
	séquences	diag. d'interactions		
	collaboration	diag. d'objets	modèle objet	
implantation	composants	diag. de modules		
	mise en place	diag. de processus		

TAB. 1.1 – *Apports des différentes méthodologies à la notation UML*

La volonté unificatrice d'UML devrait lui assurer un succès non négligeable parmi la communauté des développeurs objet. Néanmoins, elle n'apporte pas plus d'aspects novateurs dans le domaine de la conception des applications concurrentes et distribuées que les méthodes dont elle est issue. Plusieurs remarques peuvent être émises sur ce domaine. Tout d'abord, même si les processus de développement accompagnant ces notations (Cf. par exemple celui de Booch paragraphe 1.3.2.2) suggère une démarche de conception incrémentale, il ne propose pas d'approche formelle du raffinement comme c'est par exemple le cas dans la méthode B [Abr96]. Dans les méthodologies de type OMT ou UML, la preuve qu'un comportement raffiné implante sa spécification est laissée au soin des développeurs. Or, dès que les comportements à mettre en place sont complexes (ce qui est pratiquement toujours le cas pour une application concurrente et/ou distribuée non triviale), cet aspect est une source potentielle d'erreurs. La deuxième limitation des méthodologies orientées objet existantes concernent la conception de comportements distribués. Dans la plupart des cas, elles n'adoptent qu'une vue statique de la répartition (Cf. par exemple le diagrammes de processus de la méthode Booch paragraphe 1.3.2.1). Elles se contentent de fournir des notations permettant d'attribuer un processus à un nœud physique du système. Elles ne s'intéressent pas, par exemple, aux schémas de collaboration mis en place entre les différents objets ou aux informations et aux connaissances échangées au cours de telles interactions.

Les méthodologies de conception orientées objet telles que Booch, OMT ou UML, offrent donc une notation très intéressante permettant de décrire la structure d'une application. Leur syntaxe est très intuitive, elle s'intègre bien dans les ateliers de génie logiciel et elle permet de produire rapidement des squelettes de codes. Néanmoins, la complexité introduite par la concurrence et la distribution amène un besoin de traitement formel de ces aspects. Dans le chapitre suivant, nous présentons différentes notions logiques permettant de prendre en compte ces aspects. Ce sont les logiques temporelles et épistémiques.

1.4 Conclusion sur l'approche objet

Dans ce chapitre, nous avons présenté un certain nombre de concepts de base de la programmation et de la conception orientées objet. Sans vouloir être exhaustif, nous avons choisi de développer les aspects qui, à notre avis, sont employés le plus fréquemment. Nous avons couvert, au paragraphe 1.1, les langages de programmation. Nous nous sommes intéressé, au paragraphe 1.2, aux environnements de programmation en univers distribué en prenant l'exemple du standard d'interopérabilité CORBA. Finalement, nous avons abordé, au paragraphe 1.3, la conception des applications orientées objet en introduisant plus particulièrement les méthodologies Booch, OMT et UML.

L'approche objet part du principe que la modélisation d'un système est plus facile à appréhender lorsque celui-ci est scindé en plusieurs composants et lorsque la taille de chaque composant est relativement modeste. La programmation procédurale a initié cette démarche en introduisant une segmentation des traitements en procédures autonomes et réutilisables. L'approche objet a étendu la démarche aux données. Ainsi, un objet encapsule des variables gérées par des méthodes. Il exporte vers le reste du système une interface et interagit avec les autres objets du système en invoquant les méthodes proposées par de telles interfaces. En partant de ce concept de base, nous avons examiné trois caractéristiques des langages de programmation orientés objets modernes : la concurrence, la synchronisation et les relations de réutilisation. Les deux modèles principaux de la

concurrence en univers objet sont celui à objets passifs et celui à objets actifs. En ce qui concerne la synchronisation si historiquement, des solutions classiques à base de sémaphores ou de moniteurs ont été reprises et adaptées à l'univers objet, la tendance dans les langages modernes semble se porter vers des solutions à base de compteurs d'événements, d'ensembles d'états acceptables pour les méthodes ou encore de solutions réflexives. Cette dernière approche semble prometteuse puisqu'elle permet de séparer clairement les différentes fonctionnalités. Nous avons poursuivi cet état de l'art en examinant différentes relations de réutilisation. De façon courante, deux relations majeures sont employées : l'héritage et la délégation. Chacune présentent des avantages et des inconvénients. Brièvement, l'héritage est plus simple à utiliser, tandis que la délégation est celle qui, du point de vue de l'encapsulation, respecte le plus la démarche objet. Enfin, nous avons examiné les problèmes liés à l'utilisation conjointe de l'héritage et de la synchronisation, et désignés dans la littérature sous le terme d'anomalies d'héritage.

Après avoir traité les aspects langages, nous nous sommes intéressés, au paragraphe 1.2, aux concepts introduits par les environnements objets répartis en prenant l'exemple de CORBA. L'apport majeur de ce type d'environnements ou des systèmes répartis orientés objet comme Amoeba, Arjuna ou GUIDE, est de masquer la distribution. L'environnement distribué fournit un mécanisme global de désignation d'objets. Les objets communiquent par invocations de méthodes. Lorsqu'émetteur et récepteur sont localisés sur un même site, l'invocation est réalisée par un appel procédural classique. Dans les autres cas, l'invocation emprunte un appel de procédure distante. En plus de ce mécanisme de base, CORBA permet à des objets implantés dans des langages différents et sur des systèmes différents d'interopérer. Par exemple, un objet écrit en C++ peut invoquer une méthode d'un objet Smalltalk. Pour cela, CORBA définit un langage de définition d'interfaces (IDL pour *Interface Definition Language* en anglais) et un protocole d'interopérabilité (IIOP pour *Internet Inter-ORB Protocol*). Le premier permet de spécifier, indépendamment du langage choisi pour l'implantation, les interfaces de tout objet serveur mis à disposition dans l'environnement. Le second est un protocole de communication servant de support à l'invocation de méthode et commun à toutes les plateformes CORBA.

Finalement, nous avons examiné, au paragraphe 1.3, la conception des applications orientées objet. Pour cela, nous avons pris l'exemple de trois méthodologies couramment employées : la méthode Booch, OMT et UML. De façon générale, ces méthodes offrent des notations pour décrire en détail la structure des applications. Elles sont en général très complètes et proposent de nombreuses catégories de diagrammes. Elles permettent, par exemple, de décrire facilement les hiérarchies de classes et d'objets, les séquences d'échanges de messages ou les diagrammes d'états d'une classe. Néanmoins, elles sont beaucoup plus orientées vers la description des données que vers la description des comportements. De plus, elles n'abordent que de façon succincte la distribution. Dans la majorité des cas, elles se contentent de décrire, de façon statique, la répartition des différents composants d'une application sur les nœuds physiques d'un système. Elles ne s'intéressent pas aux comportements distribués, à la coordination et à la synchronisation inter-objets. Or, ces aspects nous semblent jouer un rôle important dans la conception des applications coopératives et distribuées. Dans la suite de ce document, nous proposons donc un ensemble de concepts et de notations permettant d'aborder ces problèmes.

Chapitre 2

Logiques pour l'approche objets répartis

La conception d'applications distribuées, qu'elle se fasse en approche objet ou en approche procédurale, présente un ensemble de caractéristiques originales par rapport à la conception des logiciels en univers centralisé. Les deux aspects majeurs que l'on peut citer sont la concurrence et la distribution. Les concepteurs d'applications distribuées doivent donc avoir à leur disposition des formalismes permettant d'une part d'exprimer les contraintes et les spécificités liées à la concurrence et à la distribution et d'autre part de raisonner sur ces spécifications. La plupart des démarches existantes de spécification et de preuve de programmes s'appuient sur une formalisation mathématique des applications informatiques. Chaque élément de base du programme est vu comme un composant logique et le programme complet se traduit par une formule de logique. La preuve d'une propriété se résume alors à l'étude de la formule associée au programme. Par exemple, prouver une propriété de correction revient à exhiber un invariant conservé par cette formule. Les systèmes formels existants, comme par exemple les méthodes B ou Z, pour les applications centralisées s'appuient dans la plupart des cas sur la théorie des ensembles et sur la logique booléenne classique. L'introduction du parallélisme et de la répartition dans un programme complexifie de façon importante les comportements à décrire. Les formules manipulées deviennent plus lourdes et les raisonnements plus délicats. Dans certains cas, on choisit de simplifier cette approche en adoptant des logiques plus expressives que la logique classique. La logique modale, dont nous rappelons les principes dans ce paragraphe, fait partie de celles-ci. Elle se présente comme une extension de la logique booléenne avec deux opérateurs appelés modalités, exprimant respectivement les notions de nécessité et de possibilité (cette première approche est due à [Lew12]). Ensuite, différents auteurs ont introduit d'autres interprétations pour ces modalités. Selon les interprétations choisies, plusieurs catégories de logiques modales ont été définies. Ainsi, nous présentons au paragraphe 2.2 les logiques temporelles. Celles-ci s'intéressent aux évolutions nécessaires ou inévitables d'un système et sont couramment employées pour spécifier des algorithmes concurrents. Nous utilisons au chapitre 7, une de ses variantes, la logique temporelle d'actions, pour décrire la sémantique du modèle de synchronisation pour objets concurrents que nous proposons. Puis, nous présentons au paragraphe 2.3 les logiques épistémiques qui introduisent des notions de connaissance. Nous l'utilisons au chapitre 4 pour exprimer la répartition des connaissances dans une application distribuée. Finalement, le paragraphe 2.4 conclut ce chapitre sur les logiques modales.

2.1 Logique modale

La logique modale [Lew12][vW51][Kri63][Che80][HC68, HC84] étend la logique classique avec les notions de nécessité et de possibilité. Les propositions de la logique modale peuvent être vraies ou fausses mais également nécessaires ou possibles. Les formules modales s'écrivent avec les opérateurs booléens traditionnels $\neg$ $\wedge$ $\vee$ $\Rightarrow \Leftrightarrow$ et les symboles (appelés modalités) $\Box\,\Diamond$. La modalité $\Box$ est l'opérateur de nécessité et l'expression $\Box p$ se lit comme la formule *p est nécessaire*. La modalité $\Diamond$ est l'opérateur de possibilité. L'expression $\Diamond p$ se lit comme la formule *p est possible*. Ces deux opérateurs sont duaux. Ainsi, les opérateurs $\Box$ et $\Diamond$ sont équivalents, respectivement, à $\neg\Diamond\neg$ et à $\neg\Box\neg$.

Dans les paragraphes qui suivent, nous présentons deux logiques modales : la logique temporelle présentée au paragraphe 2.2, et la logique épistémique présentée au paragraphe 2.3. Chacune d'elles introduit des interprétations différentes pour les modalités $\Box$ et $\Diamond$. La logique temporelle s'intéresse aux évolutions futures nécessaires ou inévitables d'un système tandis que la logique épistémique est une logique de la connaissance. D'autres instances de logique modale peuvent être définies et utilisées dans le domaine de l'informatique. En particulier, la logique déontique [vW51, vW80], employée par certains auteurs en informatique distribuée, introduit les notions d'obligation et de permission pour les modalités $\Box$ et $\Diamond$. Elle permet, par exemple, de décrire le fonctionnement d'un moteur d'exécution ou d'une machine virtuelle. Néanmoins, nous nous limitons dans cette partie, à une présentation des logiques temporelle et épistémique.

2.2 Logique temporelle

Les logiques temporelles sont des variantes de la logique modale. Elle utilise les interprétations *toujours dans le futur* et *inévitablement dans le futur* pour les modalités $\Box$ et $\Diamond$. Elles sont utilisées en informatique pour raisonner sur les algorithmes concurrents. De nombreuses classes de logiques temporelles ont été définies [AEdC90][KM94][Eme90]. Leur point commun consiste à considérer que la valeur de vérité d'une formule n'est pas absolue et immuable mais qu'elle varie avec le temps. Elles définissent pour cela une relation d'ordre total sur la variable temps. Dans de telles logiques, une formule n'est donc pas vraie ou fausse de façon définitive mais vraie ou fausse à un instant donné ou sur une période donnée. De très nombreux travaux en informatique parallèle et distribuée utilisent les logiques temporelles. Leur avantage réside dans le fait qu'elles permettent de spécifier des comportements déterministes et indéterministes évoluant dans le temps, de définir des propriétés et de les prouver.

Dans le paragraphe suivant, nous donnons un aperçu des différentes catégories de logiques temporelles. Le paragraphe 2.2.2 présente plus en détail la logique temporelle d'actions. C'est la logique que nous utilisons au chapitre 7 pour définir la sémantique du modèle de synchronisation pour objets concurrents que nous proposons. Finalement, le paragraphe 2.2.3 conclut cette partie sur la logique temporelle.

2.2.1 Différentes catégories de logiques temporelles

Parmi les différentes catégories de logiques temporelles définies, on peut citer, sans être exhaustif, les logiques temporelles propositionnelle ou prédicative, du passé et/ou du futur, du temps discret ou continu, avec ou sans origine et fin des temps, ponctuelle ou d'intervalle, du temps linéaire

ou arborescent dans le passé et/ou le futur. Dans ce paragraphe, nous présentons la sémantique de la logique temporelle propositionnelle qui est commune à presque toutes les autres logiques temporelles, et nous présentons brièvement les autres variantes.

Logique temporelle propositionnelle

Les systèmes propositionnels sont construits à partir d'un ensemble fini de symboles appelés propositions, de constantes logiques (*vrai* et *faux*) et d'opérateurs logiques ($\neg \wedge \vee \Rightarrow \Leftrightarrow$). Les systèmes temporels introduisent en plus les modalités suivantes : $\Box \Diamond \bigcirc \mathcal{U}$.

- $\Box$ se lit *toujours dans le futur* et signifie que la formule est vérifiée dans l'état courant et dans tous les états futurs,
- $\Diamond$ se lit *inévitablement dans le futur* et signifie que la formule est vérifiée dans l'état courant ou dans un état futur,
- $\bigcirc$ se lit *dans l'état suivant* et signifie que la formule est vérifiée dans l'état suivant,
- $\mathcal{U}$ se lit *jusqu'à* et la formule $f_1 \mathcal{U} f_2$ signifie, soit que f_1 est vérifiée dans l'état courant et dans tous les états futurs, soit que f_1 est vérifiée dans l'état courant et dans tous les états futurs jusqu'à ce que f_2 soit vérifiée.

Les formules de la logique temporelle propositionnelle sont obtenues à partir des règles suivantes :

1. les propositions atomiques p sont des formules,
2. si f_1 et f_2 sont des formules
   alors $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$, $f_1 \Rightarrow f_2$, $f_1 \Leftrightarrow f_2$, $\Box f_1$, $\Diamond f_1$, $\bigcirc f_1$, $f_1 \mathcal{U} f_2$ sont des formules,
3. toute expression obtenue en appliquant un nombre fini de fois les deux règles précédentes est une formule.

La sémantique de la logique temporelle propositionnelle est définie en terme de modèle et de relation de satisfaisabilité. Un modèle permet d'interpréter sémantiquement des formules d'une logique temporelle. Il comprend un ensemble non vide S d'états (qui correspondent à des dates ou instants précis), une relation binaire entre ces états définissant la relation d'accessibilité entre dates, et une fonction $\pi : P \rightarrow 2^S$ de l'ensemble des variables propositionnelles P dans l'ensemble des parties de l'ensemble des états (ou, symétriquement, de l'ensemble des états dans l'ensemble des parties de l'ensemble des variables propositionnelles). Nous présentons ici la définition d'une logique temporelle pour des modèles dont la relation d'accessibilité est un ordre total selon une séquence infinie $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$ d'états visités. Une autre possibilité est de définir pour chaque état e, l'ensemble $C(e)$ des sous-séquences issues de e, et de définir la sémantique des opérateurs sur l'ensemble de ces sous-séquences.

Soit, donc, S un ensemble non vide d'états, $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$ une séquence infinie d'états et $\pi : P \rightarrow 2^S$ une fonction associant à chaque proposition atomique un sous-ensemble de l'ensemble S. Pour toute proposition p la valeur de la fonction π représente l'ensemble des états pour lequel p est vraie. La satisfaisabilité d'une formule f dans un état $s \in S$ d'une séquence σ est notée $(\sigma, s) \models f$ et est déduite des règles suivantes :

$(\sigma, s_i) \models p$	si et seulement si	$s_i \in \pi(p)$
$(\sigma, s_i) \not\models p$	si et seulement si	$s_i \notin \pi(p)$
$(\sigma, s_i) \models \neg f_1$	si et seulement si	non $(\sigma, s_i) \models f_1$
$(\sigma, s_i) \models f_1 \wedge f_2$	si et seulement si	$(\sigma, s_i) \models f_1$ et $(\sigma, s_i) \models f_2$
$(\sigma, s_i) \models f_1 \vee f_2$	si et seulement si	$(\sigma, s_i) \models f_1$ ou $(\sigma, s_i) \models f_2$
$(\sigma, s_i) \models f_1 \Rightarrow f_2$	si et seulement si	$(\sigma, s_i) \models (\neg f_1) \vee f_2$
$(\sigma, s_i) \models f_1 \Leftrightarrow f_2$	si et seulement si	$(\sigma, s_i) \models (f_1 \Rightarrow f_2) \wedge (f_2 \Rightarrow f_1)$
$(\sigma, s_i) \models \Box f_1$	si et seulement si	$\forall j \geq i, (\sigma, s_j) \models f_1$
$(\sigma, s_i) \models \Diamond f_1$	si et seulement si	$\exists j \geq i, (\sigma, s_j) \models f_1$
$(\sigma, s_i) \models \bigcirc f_1$	si et seulement si	$(\sigma, s_{i+1}) \models f_1$
$(\sigma, s_i) \models f_1 \mathcal{U} f_2$	si et seulement si	$\forall j \geq i, (\sigma, s_j) \models f_1$
		ou $\exists j \geq i, (\sigma, s_j) \models f_2$ et $\forall k, i \leq k < j, (\sigma, s_k) \models f_1$

Logique temporelle prédicative

Comme pour la logique classique, on peut distinguer une logique temporelle des propositions et une logique temporelle des prédicats. Le calcul des prédicats étend le calcul des propositions en y ajoutant des variables, des prédicats et des quantifications existentielle et universelle ($\exists$ et $\forall$) sur les variables. Malgré son pouvoir d'expression plus élevé que celui du calcul des propositions, la logique prédicative présente comme difficulté majeure de ne pas constituer un système décidable, c'est à dire un système pour lequel une axiomatisation complète peut être produite [AEdC90]. En d'autres termes, il n'existe pas d'algorithme universel permettant de déterminer la valeur de vérité d'une formule de logique temporelle des prédicats.

Logique temporelle du passé et/ou du futur

Les opérateurs temporels introduits jusqu'à présent ($\Box \Diamond \bigcirc \mathcal{U}$) traitent des évolutions futures d'un comportement. Dans certaines situations, il est nécessaire de raisonner sur les évolutions passées d'un comportement. Pour cela, les opérateurs temporels du passé ■♦● et $\mathcal{S}$ ont été introduits. Ils définissent, respectivement, les notions de *toujours dans le passé*, *inévitablement dans le passé*, *dans l'état précédent* et *depuis*.

Logique temporelle du temps discret ou continu

On peut choisir de représenter la variable temps à l'aide d'un ensemble discret (par exemple avec des entiers) ou à l'aide d'un ensemble continu (par exemple avec des réels). La plupart des formalismes temporels employés en informatique (comme TLA [Lam94] ou TSOM [Ara92, Ara95]) sont à base de temps discret. Une représentation continue du temps permet d'aborder des questions plus théoriques telles que : peut-on toujours supposer "qu'il existe un instant entre deux instants distincts" ou "qu'un intervalle peut être décomposé en deux intervalles distincts" ? Néanmoins, cette approche est peu adaptée au domaine informatique dans la mesure où les instructions d'un programme s'exécutent au cours de cycles du processeur. Il existe donc une identification et une discrétisation naturelle de la variable temps fournie par les tics d'une horloge. L'hypothèse d'un temps discret consiste donc à supposer, d'après la définition donnée par [Gal90], que "chaque instant (sauf le premier et le dernier s'il y a une origine et une fin des temps) a un unique instant passé et un unique instant futur".

Logique temporelle ponctuelle ou d'intervalle

Parmi les formalismes temporels, on peut distinguer ceux dans lesquels il est possible de raisonner sur les instants et les durées, de ceux qui ne prennent en compte que les instants. Etant donné deux instants distincts, on peut, dans le premier cas dit des logiques d'intervalle, attribuer une valeur (c'est à dire une durée) à l'intervalle de temps compris entre ces deux points. Pour cela, le formalisme doit prendre en compte une fonction associant à tout couple de points une valeur positive entière ou réelle. De façon intuitive, la durée d'un instant vaut zéro. On est alors en mesure d'effectuer des comparaisons sur des intervalles de temps. Allen [AH89] définit six opérateurs (*before*, *overlaps*, *starts*, *finishes*, *during*, *meets*) et leurs inverses (*after*, *overlapped-by*, *started-by*, *finished-by*, *contains*, *met-by*). Cette notion devient difficilement exploitable dans les systèmes répartis ne disposant pas d'horloge globale. En effet, la durée de l'intervalle entre deux points situés sur des machines différentes et sans référentiel commun n'a pas vraiment de sens. De ce fait, la plupart des formalismes temporels utilisés en informatique distribuée sont ponctuels.

Logique temporelle du temps linéaire ou arborescent

La relation d'accessibilité entre états peut être considérée comme une relation d'ordre total ou comme une relation d'ordre partiel. On parle alors de LTL logique du temps linéaire (en anglais *Linear Time Logic*) ou de LTA logique du temps arborescent (en anglais CTL pour *Computational Tree Logic*). Cette dernière approche est surtout utilisée pour modéliser le comportement de programmes non déterministes. Lamport a montré dans [Lam90] que dans cette dernière situation, deux interprétations temporelles de la modalité $\Diamond$ pouvaient être dégagées. Dans le cas d'un temps discret pour des programmes déterministes, l'interprétation de la modalité $\Diamond$ est *inévitablement dans le futur*. Autrement dit, si la formule $\Diamond p$ est vraie à l'instant i alors il existe $k \geq 0$ tel que la formule p est vraie à l'instant $i{+}k$. Dans le cas d'un temps ramifié pour des programmes non déterministes, les deux interprétations possibles sont :

- soit il existe au moins un futur dans lequel p sera réalisée à un instant,
- soit pour tous les futurs il existe un instant où p sera réalisée.

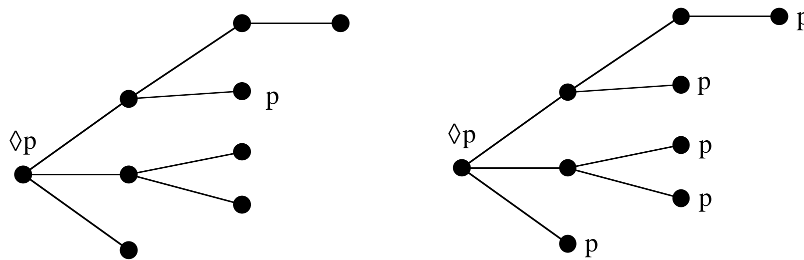


FIG. 2.1 – *Deux interprétations de la modalité $\Diamond$ dans une logique du temps ramifié*

Afin de distinguer clairement ces deux interprétations, la modalité $\Diamond$ est remplacée dans les logiques du temps ramifié par $\exists\Diamond$ et $\forall\Diamond$. De ce fait, l'opérateur dual $\Box$ est également remplacé par $\forall\Box$ et $\exists\Box$. Ces modalités sont parfois notées à l'aide de symboles alphabétiques. Le tableau 2.1 donne les correspondances entre ces différentes écritures.

$\Box$	G	$\forall\Box$	AG
$\Diamond$	F	$\forall\Diamond$	AF
$\bigcirc$	X	$\exists\Box$	EG
$\mathcal{U}$	U	$\exists\Diamond$	EF

TAB. 2.1 – *Différentes écritures des modalités temporelles*

Conclusion

Nous avons vu, dans ce paragraphe, que de nombreuses classes de logiques temporelles ont été étudiées. La diversité des comportements qui peuvent être décrits est potentiellement très élevée. Cependant, la plupart des systèmes logiques adoptent un point de vue spécifique, adapté à une classe de problèmes donnée. Beaucoup de systèmes de logique temporelle s'intéressent à la logique du temps linéaire. Certains ne s'intéressent qu'au temps ramifié, d'autres envisagent des logiques d'intervalles pour un temps continu tandis que d'autres encore raisonnent sur les évolutions passées et futures. Dans le paragraphe suivant, nous introduisons la logique temporelle d'actions qui est une logique prédicative du futur. La variable temps est discrète, ponctuelle et linéaire. Ce formalisme est employé pour la description de comportements concurrents.

2.2.2 TLA

La logique temporelle d'actions [Lam91, Lam94] (TLA pour *Temporal Logic of Actions*) est un formalisme développé par Lamport pour spécifier des algorithmes concurrents. L'apport majeur de cette approche est d'unifier au sein d'un même système formel un mécanisme de description d'instructions algorithmiques appelé logique d'actions par Lamport, et une logique temporelle pour décrire les enchaînements valides de ces actions. La logique temporelle d'actions présente l'intérêt d'utiliser le même formalisme pour décrire une application et ses propriétés. De ce fait, la preuve qu'une application réalise sa spécification, qu'elle respecte des propriétés de sûreté ou de vivacité ou qu'elle en raffine une autre, se ramène à la preuve d'une implication entre deux formules de logique temporelle d'actions. Nous utilisons cette logique, au chapitre 7, pour décrire la sémantique du modèle de synchronisation pour objets concurrents que nous proposons.

Dans les paragraphes qui suivent, nous présentons la logique d'actions et la logique temporelle utilisées par TLA. Puis nous montrons comment l'auteur a rassemblé ces deux aspects au sein d'un même formalisme. Nous donnons alors un aperçu des mécanismes de preuve des propriétés de sûreté, de vivacité et de raffinement de TLA.

Logique d'actions

La partie logique d'actions de TLA manipule des valeurs, des variables et des états. Une action est une expression à valeur booléenne composée de variables, de variables primées et de valeurs. Par exemple, $x' + 1 = y$ et $x \Leftrightarrow 1 \notin z'$ sont des actions. Une action représente une relation entre un état passé et un état futur, dans lesquels les variables non primées se réfèrent à l'état passé, tandis que celles primées se réfèrent à l'état futur.

Les valeurs sont regroupées au sein d'un ensemble noté **Val** qui comprend entre autres l'ensemble **Bool** des booléens *true* et *false*, l'ensemble **Nat** des entiers naturels, les chaînes de caractères comme "*abc*". L'ensemble **Val** n'est pas défini plus précisement mais on suppose qu'il contient toutes les valeurs nécessaires à une application donnée. Les variables sont regroupées au

sein d'un ensemble noté **Var**. Elles sont notées classiquement à l'aide d'identificateurs alphanumériques comme x, i ou sem. La sémantique de la logique d'actions est définie en termes d'états. Un état est une affectation de valeurs à des variables c'est à dire une relation entre l'ensemble **Var** des identificateurs de variables et l'ensemble **Val** des valeurs. Une fonction d'état est une expression construite à partir de variables et de valeurs (par exemple $x^2 + y \Leftrightarrow 3$). Un état s associe donc une valeur $s(x)$ à une variable x. Lamport note $[\![F]\!]$ l'interprétation sémantique d'un objet syntaxique F. L'interprétation sémantique $[\![f]\!]$ d'une fonction d'état f est une relation entre l'ensemble **St** des états et l'ensemble **Val** des valeurs. Par exemple, $s[\![x^2 + y \Leftrightarrow 3]\!]$ est la relation qui associe à l'état s la valeur $(s(x))^2 + s(y) \Leftrightarrow 3$. La notation $s[\![f]\!]$ désigne la valeur que l'interprétation sémantique $[\![f]\!]$ associe à l'état s. D'une manière générale, la définition de $s[\![f]\!]$ est :

$$s[\![f]\!] \quad \widehat{=} \quad f(\forall v : s(v)/v)$$

$f(\forall v : s(v)/v)$ désigne la valeur obtenue à partir de f en substituant $s(v)$ à v pour toutes les variables v. Un prédicat est une fonction d'état à valeur booléenne (par exemple $x^2 = y \Leftrightarrow 3$). Un prédicat P est donc une fonction d'état telle que $s[\![p]\!]$ vaut *vrai* ou *faux* pour tout état s. Un état s satisfait un prédicat P si et seulement si $s[\![p]\!]$ vaut *vrai*. L'interprétation sémantique $[\![\mathcal{A}]\!]$ d'une action $\mathcal{A}$ est la fonction qui associe le booléen $s[\![\mathcal{A}]\!]t$ à la paire d'états s et t. Par définition :

$$s[\![\mathcal{A}]\!]t \quad \widehat{=} \quad (\forall v : s(v)/v, t(v)/v')$$

Par exemple, $s[\![y = x' + 1]\!]t$ est égal à la valeur booléenne $s(y) = t(x) + 1$.

Logique temporelle

La partie logique temporelle de TLA est construite à partir des opérateurs de la logique booléenne et de l'opérateur unaire $\Box$ (se lit *toujours*). Par exemple, si E_1 et E_2 sont des formules élémentaires alors $\neg E_1 \wedge \Box(\neg E_2)$ ou $\Box(E_1 \Rightarrow \Box(E_1 \vee E_2))$ sont des formules temporelles. La sémantique de cette logique est fondée sur des séquences infinies d'états. Soit $\sigma = \langle s_0, s_1, s_{2,\ldots} \rangle$ une séquence infinie d'états. $\sigma[\![F]\!]$ désigne la valeur booléenne que la formule F assigne à la séquence infinie σ et on dit que σ satisfait F si et seulement si $\sigma[\![F]\!]$ vaut *vrai*. Les définitions de $[\![F \wedge G]\!]$, $[\![\neg F]\!]$ et $[\![\Box F]\!]$ pour une séquence σ sont les suivantes :

$$\begin{aligned}
[\![F \wedge G]\!] \quad &\widehat{=} \quad \sigma[\![F]\!] \wedge \sigma[\![G]\!] \\
[\![\neg F]\!] \quad &\widehat{=} \quad \neg\sigma[\![F]\!] \\
[\![\Box F]\!] \quad &\widehat{=} \quad \forall n \in \mathbb{N} : \langle s_n, s_{n+1}, s_{n+2}, \ldots \rangle\, [\![F]\!]
\end{aligned}$$

La séquence infinie σ représente l'évolution de l'algorithme tel que s_n est l'état de l'algorithme à l'instant n. La formule $\langle s_{n,\ldots} \rangle\, [\![F]\!]$ spécifie que F est vraie à l'instant n. Donc, la formule $\langle s_0, \ldots \rangle\, [\![\Box F]\!]$ spécifie que F est vraie à tout instant durant la séquence σ. En d'autres termes, $\Box F$ spécifie que F est un invariant de la séquence σ.

L'opérateur dual de *toujours*, appelé *inévitablement* et noté $\Diamond$, est défini de la façon suivante :

$$\Diamond F \quad \widehat{=} \quad \neg\Box\neg F$$

La formule $\Diamond F$ spécifie que F est inévitablement vraie, c'est à dire qu'il existe un instant n dans la séquence σ où F est vraie. Etant donnée une séquence infinie d'états $\sigma = \langle s_0, s_1, s_{2,\ldots} \rangle$, on obtient :

$$\sigma[\![\Diamond F]\!] \quad \equiv \quad \exists n \in \mathbb{N} : \langle s_n, s_{n+1}, s_{n+2}, \ldots \rangle\, [\![F]\!]$$

A partir des opérateurs *toujours* et *inévitablement*, on peut définir les expressions *infiniment souvent* ($\Box\Diamond$) et *inévitablement toujours* ($\Diamond\Box$). Ainsi, la formule $\Box\Diamond F$ est vraie pour une séquence infinie si et seulement si $\Diamond F$ est vraie à tout instant. $\Diamond F$ est vraie à l'instant n si et seulement si F est vraie à un instant m supérieur ou égal à n. On obtient donc :

$$\langle s_0, s_1, \ldots \rangle \llbracket \Box\Diamond F \rrbracket \quad \equiv \quad \forall n \in \mathbb{N} : \exists m \in \mathbb{N} : \langle s_{n+m}, s_{n+m+1}, \ldots \rangle \llbracket F \rrbracket$$

L'expression $\forall n : \exists m$ établit qu'il existe un nombre infini d'instants m. Donc, la formule $\Box\Diamond F$ établit que F est infiniment souvent vraie. De la même façon, $\Diamond\Box F$ établit que F est inévitablement toujours vraie.

Logique temporelle d'actions

La combinaison des notions d'actions et de logique temporelle permet d'exprimer des algorithmes concurrents comme des séquences d'actions devant toujours ou inévitablement avoir lieu et permet de décrire un ordonnancement d'actions. Un programme TLA représenté par la formule Φ s'écrit de manière générale sous la forme suivante : $\Phi \mathrel{\hat{=}} Init_\Phi \wedge \Box[\mathcal{M}]_f \wedge F$. La formule $Init_\Phi$ décrit l'initialisation des variables du programme. La formule $\mathcal{M}$ représente les actions de l'algorithme à appliquer sur l'ensemble f des variables du problème. Finalement, la formule F donne les conditions à respecter afin d'obtenir une exécution équitable des différentes actions du programme. Nous illustrons ci-dessous ces notions en incorporant au fur et à mesure ces différents éléments dans la formule décrivant un programme TLA.

Prenons l'exemple (issu de [Lam91]) d'un programme qui manipule deux variables entières x et y. Les variables sont initialisées à 0 puis le programme exécute une boucle infinie dans laquelle, à chaque étape, il incrémente x ou y. Le choix de la variable à incrémenter se fait de manière non déterministe. Si x est incrémentée, alors y reste inchangée. Réciproquement, si y est incrémentée, alors x reste inchangée. La figure 2.2 présente ce programme, à la fois, sous une forme algorithmique et sous la forme d'une formule TLA (formule Φ).

```
var natural x,y = 0;
do   ⟨ true → x := x + 1 ⟩
     □
     ⟨ true → y := y + 1 ⟩
od
```

$$\begin{aligned} Init_\Phi &\mathrel{\hat{=}} (x = 0) \wedge (y = 0) \\ \mathcal{M}_1 &\mathrel{\hat{=}} (x' = x + 1) \wedge (y' = y) \\ \mathcal{M}_2 &\mathrel{\hat{=}} (y' = y + 1) \wedge (x' = x) \\ \mathcal{M} &\mathrel{\hat{=}} \mathcal{M}_1 \vee \mathcal{M}_2 \\ \Phi &\mathrel{\hat{=}} Init_\Phi \wedge \Box\mathcal{M} \end{aligned}$$

FIG. 2.2 – *Forme algorithmique et formule TLA du programme* Φ

La formule $Init_\Phi$ modélise l'état initial de l'algorithme. La formule $\mathcal{M}$ modélise les actions entreprises par la suite : $\mathcal{M}_1$ (x est incrémentée et y reste inchangée) ou $\mathcal{M}_2$ (y est incrémentée et x reste inchangée). Finalement, le programme Φ est tel que, à l'instant 0, $Init_\Phi$ est vraie puis, aux instants suivants, $\mathcal{M}$ est toujours vraie.

Cette première version du programme peut être complétée en prenant en compte des étapes dites de "bégaiement" (*stuttering steps* en anglais) et des conditions de vivacité et d'équité. En effet, le programme Φ est légèrement inexact car il impose qu'au cours d'une étape, au moins une des deux variables x et y soit incrémentée. Or, il se peut qu'aucune des deux variables ne soient

modifiées (la spécification textuelle indique que x ou y est incrémentée). Pour cela, on définit l'expression $[\mathcal{M}]_{\langle x,y\rangle}$, qui spécifie que, soit $\mathcal{M}$ est exécutée, soit rien ne se passe :

$$[\![\mathcal{M}]\!]_{\langle x,y\rangle} \quad \widehat{=} \quad \mathcal{M} \vee ((x' = x) \wedge (y' = y))$$

Le programme Φ s'écrit alors :

$$\Phi \quad \widehat{=} \quad Init_\Phi \wedge \Box[\![\mathcal{M}]\!]_{\langle x,y\rangle}$$

Cependant, cette formule autorise des comportements qui débutent par $Init_\Phi$ puis dans lesquels plus rien ne se passe. Pour éliminer de telles situations, on peut spécifier que les expressions $\mathcal{M}_1$ et $\mathcal{M}_2$ doivent s'exécuter infiniment souvent. Une telle propriété est une propriété de vivacité car elle garantit qu'un comportement a lieu (par opposition aux propriétés de sûreté qui garantissent qu'un comportement n'a pas lieu). Le programme Φ s'écrit alors :

$$\Phi \quad \widehat{=} \quad Init_\Phi \wedge \Box[\![\mathcal{M}]\!]_{\langle x,y\rangle} \wedge \Box\Diamond\mathcal{M}_1 \wedge \Box\Diamond\mathcal{M}_2$$

Cette formule garantit que l'une des deux actions $\mathcal{M}_1$ ou $\mathcal{M}_2$ est entreprise, mais elle ne garantit pas que, par exemple, seule l'action $\mathcal{M}_1$ soit systématiquement exécutée en laissant de côté $\mathcal{M}_2$. Pour éliminer de telles situations, des conditions d'équité doivent être introduites entre les actions $\mathcal{M}_1$ et $\mathcal{M}_2$. Deux catégories de conditions d'équité sont définies : l'équité faible (*weak fairness* en anglais) et l'équité forte (*strong fairness* en anglais). Par exemple, le programme Φ peut s'écrire :

$$\Phi \quad \widehat{=} \quad Init_\Phi \wedge \Box[\![\mathcal{M}]\!]_{\langle x,y\rangle} \wedge WF_{\langle x,y\rangle}\mathcal{M}_1 \wedge WF_{\langle x,y\rangle}\mathcal{M}_2$$

Une condition d'équité faible $WF_{\langle x,y\rangle}\mathcal{A}$ garantit que, soit l'action $\mathcal{A}$ est exécutée, soit elle devient inévitablement impossible à exécuter (en d'autres termes, soit $\mathcal{A}$ est infiniment souvent exécutée, soit elle est infiniment souvent impossible à exécuter). Une condition d'équité forte $SF_{\langle x,y\rangle}\mathcal{A}$ garantit que, soit l'action $\mathcal{A}$ est exécutée, soit elle devient toujours impossible à exécuter (en d'autres termes, soit $\mathcal{A}$ est infiniment souvent exécutée, soit elle est inévitablement toujours impossible à exécuter). Par définition, on obtient donc :

$$\begin{aligned} WF_{\langle x,y\rangle}\mathcal{A} \quad &\widehat{=} \quad (\Box\Diamond\mathcal{A}) \vee (\Box\Diamond\neg Enabled\langle\mathcal{A}\rangle) \\ SF_{\langle x,y\rangle}\mathcal{A} \quad &\widehat{=} \quad (\Box\Diamond\mathcal{A}) \vee (\Diamond\Box\neg Enabled\langle\mathcal{A}\rangle) \end{aligned}$$

Formellement, le prédicat $Enabled\langle\mathcal{A}\rangle$ est vrai pour l'action $\mathcal{A}$ dans l'état s s'il est possible d'exécuter $\mathcal{A}$ dans cet état :

$$s[\![Enabled\langle\mathcal{A}\rangle]\!] \quad \widehat{=} \quad \exists t \in St : s[\![\mathcal{A}]\!]t$$

Le prédicat *Enabled* teste donc si la garde associée à l'action $\mathcal{A}$ s'évalue à vrai. Jusqu'à présent les actions manipulées ($\mathcal{M}_1$ et $\mathcal{M}_2$) sont des actions exécutables dans toutes les situations. D'une façon plus générale, une action peut être considérée comme une commande gardée [Dij75, Dij76] comprenant deux parties : une garde (une expression booléenne) et une commande (une série d'instructions). Lorsque la garde s'évalue à vraie, la commande peut être exécutée, sinon la commande reste bloquée.

Propriétés de sûreté

Une propriété de sûreté consiste à établir qu'une certaine configuration n'est jamais atteinte au cours de l'exécution d'un programme. Par exemple, on va chercher à vérifier que les instructions

d'une section critique dans un algorithme concurrent ne sont jamais exécutées simultanément par plusieurs procédures ou que la valeur d'un compteur ne dépasse jamais une certaine limite. Dans le cadre d'une logique temporelle cette démarche consiste à établir que la propriété de sûreté représentée par la formule I est vérifiée à toutes les étapes de l'exécution du programme (c'est à dire que I est toujours vraie ou en d'autres termes que I est un invariant). Etant donné un programme TLA Φ, cela revient à prouver l'implication logique : $\Phi \Rightarrow \Box I$.

Propriétés de vivacité

Contrairement aux propriétés de sûreté qui cherchent à établir qu'une configuration erronée n'est jamais atteinte, les propriétés de vivacité établissent qu'un certain état doit inévitablement être atteint lors de l'exécution d'un programme. Par exemple, on va chercher à déterminer que le programme termine ou que tout message envoyé est reçu. Dans le cadre d'une logique temporelle, cette démarche consiste à montrer qu'il existe une étape de l'exécution du programme qui vérifie la propriété de vivacité représentée par la formule V. De ce fait, les propriétés de vivacité doivent être déduites des conditions d'équité du programme. Etant donné un programme TLA Φ, cela revient à prouver l'implication logique : $\Phi \Rightarrow \Diamond V$.

Raffinement

De façon informelle, un programme Ψ est un raffinement d'un programme Φ si $\Psi \Rightarrow \Phi$. On parle alors de programme de niveau supérieur pour Φ et de programme de niveau inférieur ou raffiné pour Ψ. Dans le cas d'actions s'exprimant sous la forme de commandes gardées, le raffinement consiste, par exemple, à renforcer les gardes ou à ajouter des instructions qui ne sont pas en contradiction avec les actions de niveau supérieur.

2.2.3 Conclusion sur la logique temporelle

Dans cette partie, nous avons présenté différentes classes de logiques temporelles. Elles sont employées en informatique, pour les raisonnements sur les programmes concurrents. Nous avons vu que ces logiques sont des variantes de la logique modale. Les opérateurs de nécessité et de possibilité s'interprètent alors à partir d'une variable temps et traduisent respectivement qu'une formule est toujours vraie dans le futur ou qu'une formule est inévitablement vraie dans le futur. Nous avons vu au paragraphe 2.1 que de nombreuses catégories de logiques temporelles pouvaient être définies. Ainsi, on peut étendre à des raisonnements temporels soit le calcul des propositions soit le calcul des prédicats.

A partir de ces définitions, nous nous sommes attachés à décrire plus en détail la logique temporelle d'actions de Lamport. Ce formalisme, employé en informatique pour la modélisation d'algorithmes concurrents, intègre une logique d'actions pour la description des instructions d'un programme et une logique temporelle pour la description des enchaînements valides de ces actions. Une action représente, à l'aide de variables et de valeurs, une relation entre un état passé et un état futur. Les raisonnements temporels en TLA se font à partir d'une logique prédicative du futur. La variable temps est discrète, ponctuelle et linéaire. Un programme TLA est une formule comprenant un ensemble d'actions exécutées selon des conditions d'équité. Des preuves de propriétés de sûreté, de vivacité et de raffinement peuvent alors être établies. En unifiant les aspects de spécification

et de preuve de programmes, la logique temporelle d'actions fournit donc un cadre théorique intéressant pour les applications concurrentes.

2.3 Logique épistémique

La logique épistémique peut être vue comme une instance particulière de logique modale. C'est une logique de la connaissance [MvdH95][FHMV95] que nous utilisons au chapitre 4 pour exprimer la répartition des connaissances dans une application distribuée. Elle utilise les notions de connaissance et de possibilité pour les modalités $\Box$ et $\Diamond$. La logique épistémique permet de représenter l'état de connaissance d'un ensemble d'agents qui peut être, à un instant donné, extrêmement variable si l'application fonctionne dans un environnement lui-même très variable. C'est le cas des applications distribuées qui sont soumises à des aléas de fonctionnement très nombreux : indéterminisme des communications, indéterminisme des exécutions dû, en particulier, aux problèmes de pannes ou de charge des systèmes support. Par l'analyse des différentes situations possibles (voir plus loin la notion de mondes possibles) et de leurs relations, le modélisateur peut représenter précisément l'état de connaissance d'un ensemble d'agents dans une situation donnée, sans avoir à décrire la variété des évolutions ayant conduit à cette situation. On peut donc considérer, en quelque sorte, que la logique épistémique est un moyen souple de modéliser une évolution passée alors que la logique temporelle permet de modéliser de façon adéquate l'évolution future.

Le paragraphe suivant introduit la notion de connaissance. Le paragraphe 2.3.2 étend la logique épistémique avec différents degrés de connaissance : connaissance distribuée, instanciée, de tous et commune. Le paragraphe 2.3.3 introduit la notion de croyance. Finalement, le paragraphe 2.3.4 résume les principales notions introduites par la logique épistémique et conclut ce paragraphe.

2.3.1 Connaissance

La logique épistémique note les modalités $\Box$ et $\Diamond$ à l'aide des symboles K et M. Les formules épistémiques sont construites à partir d'un ensemble P de propositions atomiques et s'interprètent à partir d'un ensemble A d'agents. Ainsi, étant donné un agent $i \in A$, étant donnée la formule φ (on emploie également le terme *fait*), l'expression $K_i\varphi$ s'interprète comme *l'agent i connaît le fait φ*. L'opérateur M est le dual de K (c'est à dire $M \equiv \neg K \neg$) et l'expression $M_i\varphi$ s'interprète comme *l'agent i considère le fait φ comme possible.*

Une sémantique de la logique épistémique peut être définie à partir des notions de mondes possibles et de structures de Kripke. Une structure de Kripke $\mathbb{M}$ est un tuple $\langle S, \pi, R_1, \ldots, R_m \rangle$ où :

- S est un ensemble non vide d'états,
- $\pi : S \rightarrow (P \rightarrow \{true, false\})$ est une fonction associant une valeur de vérité à chaque proposition dans chaque état,
- $R_i \subseteq S \times S$ pour $i = 1, \ldots, m$ sont les relations d'accessibilité pour tous les agents de l'ensemble A.

Un monde est alors un couple $(\mathbb{M}, s)$ composé par une structure de Kripke et un état $s \in S$. L'interprétation d'un élément (s, t) de l'ensemble R_i est que, dans le monde $(\mathbb{M}, s)$, l'agent i

considère le monde $(\mathbb{M}, t)$ comme un monde possible. Le fait que la formule φ soit satisfaite dans un monde $(\mathbb{M}, s)$ est notée $(\mathbb{M}, s) \models \varphi$ et est déduit des règles suivantes :

$(\mathbb{M}, s) \models p$	si et seulement si	$\pi(s)(p) = true$
$(\mathbb{M}, s) \models \varphi \wedge \psi$	si et seulement si	$(\mathbb{M}, s) \models \varphi$ et $(\mathbb{M}, s) \models \psi$
$(\mathbb{M}, s) \models \neg\varphi$	si et seulement si	$(\mathbb{M}, s) \not\models \varphi$
$(\mathbb{M}, s) \models K_i\varphi$	si et seulement si	$(\mathbb{M}, t) \models \varphi$ pour tous les t tels que $(s, t) \in R_i$

La dernière règle signifie que l'agent i connaît le fait φ dans le monde $(\mathbb{M}, s)$ si et seulement si la formule φ est vraie dans tous les états t que i considère comme possible à partir de l'état s (Cf. figure 2.3). Les relations d'accessibilité d'un modèle de Kripke sont réflexives, transitives et symétriques.

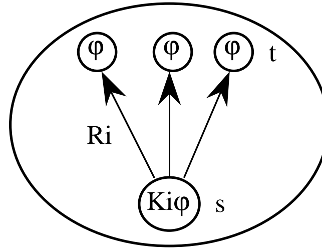


FIG. 2.3 – *Interprétation en terme de mondes possibles de la notion de connaissance*

2.3.2 Différents degrés de connaissance

La notion de connaissance présentée au paragraphe précédent peut être complétée par les notions de connaissance distribuée, instanciée, de tous et commune. Par rapport aux modalités K et M issues de la logique modale, on introduit ainsi quatre modalités auxiliaires ou opérateurs, notés respectivement I, S, E et C. Alors que les modalités de connaissance et de possibilité (K et M) concernent un agent particulier, ces quatre nouveaux opérateurs concernent l'ensemble A de tous les agents du système. On obtient ainsi différents degrés de connaissance dite de groupe qui permettent par exemple de modéliser l'élévation du niveau de connaissance dans une application distribuée.

Connaissance distribuée et instanciée

Contrairement à la modalité K qui désigne un fait connu par un agent désigné de façon explicite, l'opérateur I de connaissance distribuée (appelé également connaissance implicite) représente un fait présent de façon latente au sein de l'ensemble d'agents. Cet opérateur est parfois noté dans la littérature avec la lettre D. Néanmoins, certains auteurs réservent cette lettre pour la notion de croyance commune (Cf. paragraphe 2.3.3). Ainsi, pour ne pas créer d'ambiguïté, l'opérateur de connaissance distribuée ou implicite est noté I. Informellement, l'ensemble d'agents A a une connaissance implicite d'un fait φ si cette connaissance est distribuée parmi les éléments de A. Il existe un programme qui, exécuté par un ou plusieurs agents de l'ensemble A, permet à partir de données représentées par un fait ψ, d'inférer le fait φ. Il est certain que la notion de connaissance distribuée n'est à utiliser que dans le cas où aucun agent ne connaît, avant l'exécution du programme d'inférence, le fait φ. L'idée sous-jacente à la notion de connaissance distribuée est qu'un

groupe d'agents peut mettre des connaissances en commun pour inférer un fait φ.

On peut alors remarquer que si l'on considère qu'une application centralisée ou distribuée est un système fermé, les connaissances "nouvelles" qu'elle peut instancier ne peuvent être que des connaissances distribuées. La sémantique de cet opérateur utilise, comme précédemment, un ensemble P de propositions, un ensemble A d'agents et une structure de Kripke $\mathbb{M} = \langle S, \pi, R_1, \ldots, R_m \rangle$ dans laquelle S désigne un ensemble non vide d'états, π une fonction associant une valeur de vérité à chaque proposition et R_m des fonctions d'accessibilité. La sémantique de l'opérateur I dans un monde $(\mathbb{M}, s)$ est définie alors de la façon suivante :

$(\mathbb{M}, s) \models I\varphi$ si et seulement si $(\mathbb{M}, t) \models \varphi$ pour tous les t tels que $(s, t) \in R_1 \cap \ldots \cap R_m$

De façon intuitive, cette règle stipule que la formule φ est une connaissance distribuée si et seulement si, elle est vérifiée dans les mondes considérés comme possibles par tous les agents simultanément. La notion de connaissance pour un agent i dans un état s est définie à partir de l'ensemble R des états accessibles par i à partir de s. Un fait n'est connu que s'il est vérifié dans tous les états de cet ensemble. Néanmoins, il se peut que ce fait ne soit vérifié que dans un sous-ensemble de l'ensemble R. La connaissance distribuée pour un ensemble d'agents A dans un état s est définie à partir du sous-ensemble d'états accessibles par tous les agents de A. Si le fait est vrai dans tous les états de ce sous-ensemble, alors il est qualifié de connaissance distribuée.

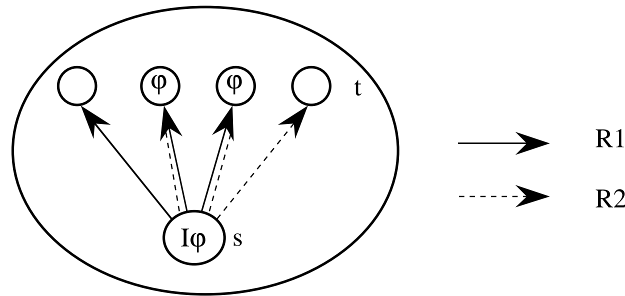


FIG. 2.4 – *Interprétation en terme de mondes possibles de la notion de connaissance distribuée*

La notion suivante introduite par les systèmes épistémiques dans les systèmes à base de connaissance est appelée connaissance instanciée (opérateur S). Une connaissance φ est dite instanciée au sein de l'ensemble A de tous les agents du système si il existe un agent i qui connaît le fait φ. La sémantique de l'opérateur S dans un monde $(\mathbb{M}, s)$ est définie de la façon suivante :

$(\mathbb{M}, s) \models S\varphi$ si et seulement si $\exists i \in A, K_i\varphi$

Connaissance de tous et connaissance commune

On a vu que la notion de connaissance a été définie à partir d'un ensemble A contenant m agents. La modalité K concerne la connaissance d'un agent c'est à dire d'un membre particulier de l'ensemble A. Lorsqu'un même fait est connu par plusieurs agents, il est intéressant de définir une notation permettant de factoriser la connaissance possédée par tous les agents concernés. Pour cela, on définit les opérateurs E (connaissance de tous) et C (connaissance commune). De façon informelle, l'expression $E\varphi$ signifie que tous les agents de l'ensemble A connaissent le fait φ. $C\varphi$ signifie que tous les agents de l'ensemble A savent que tous savent ... que tous connaissent le fait φ. Ainsi on obtient les définitions suivantes :

$$E\varphi \mathrel{\widehat{=}} K_1\varphi \wedge \ldots \wedge K_m\varphi$$
$$C\varphi \mathrel{\widehat{=}} E\varphi \wedge EE\varphi \wedge \ldots \wedge E^k\varphi \wedge \ldots$$

La sémantique des opérateurs E et C utilise en plus des relations d'accessibilité R_m les relations $\rightarrow$, $\rightarrow^k$ et R^* suivantes :

- $s \rightarrow t$ si et seulement si un agent de l'ensemble A considère t comme un état accessible à partir de s, c'est à dire si et seulement si $(s, t) \in R_1 \cup \ldots \cup R_m$,
- $s \rightarrow^k t$ si et seulement si il existe pour la relation $\rightarrow$ une séquence d'états $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$ telle que $s_0 = s$ et $s_k = t$,
- $(s, t) \in R^*$ si et seulement si $\forall k \geq 0, s \rightarrow^k t$ (R^* est la fermeture transitive de la relation d'accessibilité $\rightarrow$).

La sémantique des opérateurs E et C dans un monde $(\mathbb{M}, s)$ est définie de la façon suivante :

$(\mathbb{M}, s) \models E\varphi$ si et seulement si $(\mathbb{M}, t) \models \varphi$ pour tous les t tel que $(s, t) \in R_1 \cup \ldots \cup R_m$
$(\mathbb{M}, s) \models C\varphi$ si et seulement si $(\mathbb{M}, t) \models \varphi$ pour tous les t tel que $(s, t) \in R^*$

Les sémantiques des opérateurs E, C ou I ont toutes une forme proche. Seuls changent les états accessibles considérés. Dans le cas de l'opérateur E, tous les états que tous les agents de l'ensemble A considèrent comme accessibles sont pris en compte (Cf. figure 2.5). Tous les agents doivent connaître le fait φ pour que la formule $E\varphi$ soit vérifiée (d'où le terme connaissance de tous).

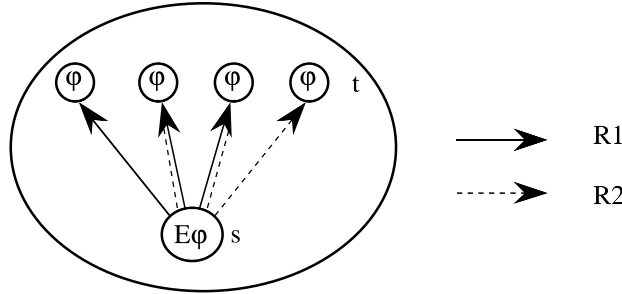


FIG. 2.5 – *Interprétation en terme de mondes possibles de la notion de connaissance de tous*

Pour l'opérateur C, on prend en compte la fermeture transitive de la relation d'accessibilité sur tous les mondes que tous les agents de l'ensemble A considèrent comme possibles (Cf. figure 2.6). Comme on peut le constater, cette notion est très forte. Ce niveau de connaissance n'est atteint qu'en de rares occasions. Dans le cas de programmes informatiques distribués communiquant avec des délais non bornés, Fagin et al prouvent dans [FHMV96, FHMV95] que ce niveau de connaissance ne peut être atteint. Tout au plus, la seule connaissance commune dans de tels programmes est celle qui est apportée par une hypothèse de fonctionnement concernant la phase d'initialisation qui permettrait de postuler l'existence d'un état initial où une propriété de connaissance est satisfaite. Par exemple, on peut considérer que l'unicité des identificateurs de sites dans une application distribuée est une connaissance partagée par tous les agents du système et garantie par l'environnement d'exécution. De même, dans certains cas, les codes s'exécutant sur les différents sites de l'environnement ou les interfaces des objets sont des connaissances communes. En fait, dans un

monde réel soumis, entre autres, à des erreurs humaines, à des pannes, à un indéterminisme des communications, l'attribution d'un identificateur à chaque site sans homonymie, le chargement à distance d'un ensemble de codes sur des sites, sont des traitements probabilistes qui ont toujours une probabilité d'échouer (mais suffisamment faible pour qu'en pratique on s'en accommode).

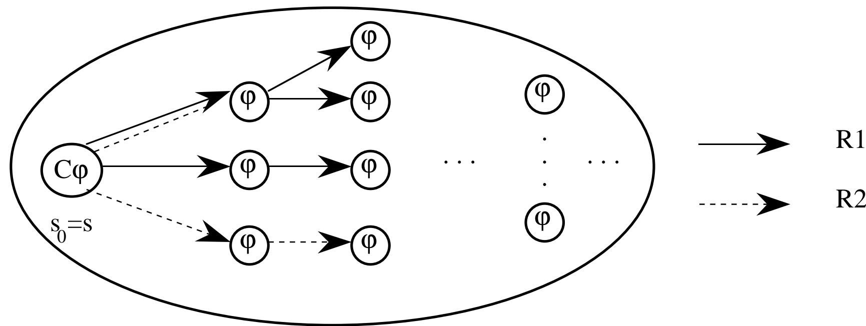


FIG. 2.6 – *Interprétation en terme de mondes possibles de la notion de connaissance commune*

Gradation des niveaux de connaissance

Les quatre opérateurs de connaissance de groupe introduits précédemment définissent des degrés progressifs de connaissance. En termes d'échanges, on peut considérer que, peu ou prou, tout programme distribué qui termine, est un système qui fait évoluer un groupe d'agents (ou d'objets ou de sites) d'un niveau de connaissance distribuée à un niveau de connaissance instanciée puis éventuellement à un niveau de connaissance de tous et plus rarement à un niveau de connaissance commune.

2.3.3 Croyance

Dans ce paragraphe, nous introduisons la notion de croyance (modalité B). Cette modalité offre un éclairage intéressant pour les applications distribuées (en particulier dans le cas où la variabilité de l'environnement a conduit à ne pas faire une analyse exhaustive, mais seulement partielle, de l'ensemble des mondes possibles et des conditions de vérité des faits dans tous ces mondes). En effet, un univers réparti introduit de nombreuses sources d'indéterminisme dans la manipulation des données. Par exemple, lorsqu'un agent a lit une variable manipulée par un agent b situé sur un site distant, rien ne garantit qu'entre le moment où l'agent b envoie la valeur et le moment où l'agent a reçoit le résultat, la variable n'a pas été modifiée. Sachant cela, l'agent a ne peut pas attribuer à la valeur reçue le statut d'une connaissance sûre. La modalité B est donc un moyen d'exprimer ce statut et de différencier les connaissances sur les états locaux qui sont sûres, des connaissances sur les états distants qui sont non sûres. Dans l'exemple précédent, on peut utiliser la notion de croyance par l'agent a que l'agent b a toujours conservé à la variable la même valeur que celle reçue alors qu'il ne peut en être certain. Dans ce paragraphe, nous présentons deux logiques. Une logique de la connaissance et une logique de la connaissance et de la croyance.

Logique épistémique de la croyance

Levesque dans [Lev84] introduit la notion de croyance en tant que "connaissance qui n'est pas nécessairement vraie". La sémantique de la logique épistémique de croyance peut s'inter-

préter en terme de mondes possibles. Comme pour la logique épistémique de la connaissance, on définit un ensemble P de propositions, un ensemble A d'agents et une structure de Kripke $\mathbb{M} = \langle S, \pi, T_1, \ldots, T_m \rangle$. Les relations d'accessibilité $T_1, \ldots, T_m$ sont fondées sur les croyances des agents et non plus sur leurs connaissances. Ainsi l'expression $(s,t) \in T_i$ signifie que, dans l'état s, l'agent i croit que l'état t est un état possible. Le fait qu'une formule $B_i\varphi$ soit satisfaite dans un monde $(\mathbb{M}, s)$ est déduit de la règle suivante :

$$(\mathbb{M}, s) \models B_i\varphi \quad \text{si et seulement si} \quad (\mathbb{M}, t) \models \varphi \text{ pour tous les t tels que } (s,t) \in T_i$$

Logique épistémique de la connaissance et de la croyance

Kraus et Lehmann dans [KL86] définissent une logique épistémique notée KB qui manipule à la fois des connaissances et des croyances. Ils prennent en compte les opérateurs de connaissance et de croyance K et B. Ils étendent leur logique avec les opérateurs E et C de connaissance de tous et de connaissance commune. De plus, ils introduisent les opérateurs F et D pour traduire les notions de croyance de tous et de croyance commune. Les différentes expressions de cette logique épistémique notée KB, s'interprètent de la façon suivante à partir d'une ensemble P de propositions et d'un ensemble A d'agents :

$K_i\varphi$	l'agent i connaît le fait φ	
$B_i\varphi$	l'agent i croit au fait φ	
$E\varphi$	tous les agents connaissent le fait φ	i.e. $E\varphi \mathrel{\widehat{=}} K_1\varphi \wedge \ldots \wedge K_m\varphi$
$F\varphi$	tous les agents croient au fait φ	i.e. $F\varphi \mathrel{\widehat{=}} B_1\varphi \wedge \ldots \wedge B_m\varphi$
$C\varphi$	le fait φ est une connaissance commune	i.e. $C\varphi \mathrel{\widehat{=}} E\varphi \wedge \ldots \wedge E^k\varphi \wedge \ldots$
$D\varphi$	le fait φ est une croyance commune	i.e. $D\varphi \mathrel{\widehat{=}} F\varphi \wedge \ldots \wedge F^k\varphi \wedge \ldots$

La structure de Kripke $\mathbb{M}$ associée à cette logique épistémique comprend deux catégories de relations d'accessibilité R et T: $\mathbb{M} = \langle S, \pi, R_1, \ldots, R_m, T_1, \ldots, T_m \rangle$. Les relations R_i sont les relations d'accessibilité relatives aux connaissances. Les relations T_i sont les relations d'accessibilité relatives aux croyances. Elles présentent les propriétés suivantes :

$$T_i \subseteq R_i$$
$$\forall s, t, u \in S, R_i(s,t) \wedge T_i(t,u) \Rightarrow T_i(s,u)$$

La première propriété établit que si un état est accessible sur la base de croyances, alors il est également accessible sur la base de connaissances (en effet, tout couple en relation par T_i l'est également par R_i). La seconde propriété stipule que si s et t sont des états accessibles sur la base des connaissances de l'agent i et si celui-ci croit que l'état u est accessible à partir de l'état t, alors il croit que u est accessible à partir de s. La sémantique de la logique épistémique KB se définit aussi à partir de la notion de mondes possibles.

2.3.4 Conclusion sur la logique épistémique

Dans ce paragraphe, nous avons décrit la sémantique de différents opérateurs de connaissance. Les logiques intégrant ces opérateurs sont dites épistémiques et sont des instances de logique modale. Elles sont utilisées en informatique pour raisonner sur les systèmes multi-agents et les applications distribuées. Le paragraphe 2.3.1 présente la modalité de connaissance K traduit la situation où un agent i d'un groupe G connaît un fait φ. A partir de cet opérateur principal,

le paragraphe 2.3.2 introduit différents degrés dans la connaissance d'un fait pour un groupe d'agents. Ainsi, la connaissance peut être implicite ou instanciée. De même, le fait peut être connu de tous ou la connaissance peut être commune. Comme pour la modalité K, la sémantique de ces opérateurs est définie en terme de mondes possibles et de structure de Kripke. Finalement, le paragraphe 2.3.3 introduit la notion de croyance pour raisonner sur les faits qui ne sont pas connus de façon certaine.

2.4 Conclusion sur la logique modale

Les différents formalismes de spécification existants s'appuient sur une description mathématique des comportements à mettre en œuvre. Les bénéfices attendus sont multiples. On espère ainsi imposer un cadre mathématique strict qui contraigne les développeurs à une démarche de conception rigoureuse. De même, on souhaite que la construction de modèles fassent apparaître clairement, dès les premières phases de conception, les fonctionnalités majeures de l'application en reléguant l'introduction des détails techniques à des phases ultérieures. Finalement, en s'appuyant sur des systèmes axiomatiques, la vérification de propriétés formelles permet de détecter en amont des phases de codage, et donc à un moindre coût, certaines erreurs de conception.

La plupart des méthodes formelles pour les systèmes séquentiels et centralisés utilisent une logique du premier ordre. La concurrence et de la répartition des applications distribuées complexifient de manière importante les comportements à mettre en œuvre. Afin de simplifier les modèles, on choisit en général d'adopter des logiques plus expressives. La logique modale fait partie de celles-ci. Elle introduit, par rapport à la logique classique, deux opérateurs supplémentaires, appelés modalités : c'est l'opérateur de nécessité (noté $\Box$) et l'opérateur de possibilité (noté $\Diamond$). Ainsi, les faits dans une telle logique peuvent être vrais ou faux, mais également possibles ou nécessaires. Deux interprétations de la logique modale nous intéressent plus particulièrement pour les programmes informatiques : la logique temporelle et la logique épistémique. Elles sont employées respectivement pour décrire la concurrence et la distribution. La logique temporelle utilise des séquences d'états et interprète les modalités $\Box$ et $\Diamond$ avec les termes *toujours* et *inévitablement*. Nous avons présenté la logique temporelle d'actions développée par Lamport. La principale originalité de cette logique est d'inclure dans un même formalisme une logique d'actions pour décrire des constructions algorithmiques comme des affectations ou des lectures de variables, et une logique temporelle pour décrire des séquences d'actions. Un algorithme concurrent se ramène alors à la description de toutes les séquences valides d'actions qui peuvent être entreprises simultanément. Des conditions d'équité peuvent être attribuées à chacune de ces séquences afin de garantir qu'elles se déroulent correctement. Au chapitre 6, nous proposons un modèle de synchronisation pour des objets concurrents. Nous utilisons alors au chapitre 7, la logique temporelle d'actions de Lamport pour définir sa sémantiques.

La seconde interprétation de la logique modale que nous présentons dans ce chapitre, concerne la logique épistémique. Celle-ci attribue la notion de connaissance à la modalité de nécessité. Elle permet de décrire l'état d'un système réparti selon les connaissances manipulées par chacun de ces agents. Différents degrés de connaissance, comme la connaissance distribuée, la connaissance de tous ou la connaissance commune, permettent alors de qualifier la répartition de la connaissance dans un système multi-agents. Ces opérateurs sont employés dans les programmes à base de connaissance que nous présentons au chapitre suivant. Ceux-ci proposent un paradigme de

programmation permettant de décrire le comportement d'un agent. Nous utilisons également ces opérateurs au chapitre 4, pour étendre la notion de programme à base de connaissances d'un niveau agent à un niveau groupe d'agents.

Chapitre 3

Programmation à base de connaissances

De nombreux paradigmes de programmation d'applications distribuées ont été proposés. Parmi les principaux, on peut citer ceux issus respectivement du domaine des systèmes répartis et des protocoles réseaux, de l'approche objet et des systèmes multi-agents. Dans le premier cas, l'application est envisagée comme un ensemble de processus communiquant par envoi de message asynchrone. Dans le second, c'est un ensemble d'objets interagissant par invocation de méthode. Finalement, un système multi-agents est composé d'un ensemble d'agents autonomes, poursuivant des buts locaux et échangeant des connaissances. Dans ce chapitre, nous nous intéressons à ce dernier paradigme. Sans être exhaustif, nous présentons les grandes lignes de l'approche proposée par Fagin, Halpern, Moses et Vardi dans [HF89, FHMV95, FHMV96] pour la mise en place de systèmes multi-agents.

Le concept clé défini par ces auteurs est celui de connaissance. Selon eux, un système multi-agents est un système qui, bien sûr, échange des données, mais également dans de nombreux cas, manipule, échange et transforme non pas des faits bruts mais une connaissance de ces faits. Ceci est par exemple tout à fait évident dans le cas des horloges logiques qui ne sont pas uniquement des compteurs entiers, mais beaucoup plus des connaissances sur l'exécution passée d'actions. Ils proposent donc une logique modale de la connaissance qui permet de décrire les comportements des différents agents d'un système multi-agents. Chaque agent manipule localement de la connaissance et interagit avec ses pairs. Le comportement distribué résulte alors de l'ensemble des comportements locaux coordonnés par les opérations de communication. Pour pouvoir l'interpréter, nous présentons au paragraphe 3.1 la notion d'état global d'un système. Le paragraphe 3.2 présente alors la logique épistémique et temporelle proposée par Fagin et al pour effectuer des raisonnements en terme de connaissance dans les systèmes multi-agents. Puis, à partir de cette logique, le paragraphe 3.3 présente la notion de programme à base de connaissance qui permet de décrire le traitement de la connaissance effectué par un agent d'un système multi-agents. Finalement, le paragraphe 3.4 conclut cette partie et résume les apports de la programmation à base de connaissance.

3.1 Notion d'état global d'un système réparti

De façon mathématique, l'état global d'un système peut être défini comme un n-uplet composé par les états de chacun de ces membres. L'état d'un membre (par exemple un processus, un objet ou un agent) est alors une fonction entre l'ensemble de ses variables d'instance et un ensemble de valeurs. Selon les hypothèses émises sur l'environnement distribué, certaines entités comme les canaux de communication entre objets ou entre sites, peuvent être considérées comme ayant un état (dans ce cas, c'est l'ensemble des messages en transit entre deux objets ou deux sites). Ces types d'éléments sont alors ajoutés au n-uplet définissant l'état du groupe. Parmi tous les n-uplets possibles, seul un certain nombre correspond à des configurations qui ont pu exister lors de l'exécution : ce sont les états dits cohérents. Ceux-ci sont définis à partir de la notion de relation d'ordre. De façon intuitive, une relation d'ordre définit des précédences entre les états du groupe. Ces précédences sont qualifiées de causales car elles traduisent, à un niveau sémantique, des liens de cause à effet entre les actions des différents membres de l'application.

3.1.1 Relation de causalité (arrive avant ou happened before)

Les nombreuses études sur la notion de causalité dans les systèmes répartis s'appuient pour la plupart sur une modélisation en terme de processus communicant par envoi de message asynchrone et sur la relation de précédence causale entre événements, dite *Happened before*, définie par Lamport [Lam78]. Dans ce modèle, aucune hypothèse n'est réalisée sur les partages de mémoire commune, les canaux de communication qui ne sont pas FIFO, les délais de communication qui ne sont pas bornés; il n'y a pas d'horloge globale et les horloges locales ne sont pas synchronisées. Il n'est donc pas possible, comme dans les univers centralisés, de dater de façon univoques les événements d'un programme. Le but de la relation proposée est donc de permettre une datation logique de ces événements.

Deux types d'événements sont considérés au sein d'un processus : des événements locaux modélisant des opérations internes à un processus et des événements de communication qui sont soit des envois de message soit des réceptions de message. Tous les événements sont atomiques, c'est à dire que leur exécution est non interruptible. De plus ils représentent des transitions entre deux états d'un processus. La relation de précédence causale définit un ordre partiel entre ces événements. On peut alors définir les notions d'indépendance causale et d'histoire causale associées à cette relation.

Soient n processus $P_1, \ldots, P_n$
Soient n ensembles d'événements $E_1, \ldots, E_n$
Chaque ensemble E_i contient les événements locaux e_{ij} au processus P_i
Soit $E = E_1 \cup \ldots \cup E_n$ l'ensemble de tous les événements de l'application

Définition 3.1 *La relation de précédence causale $\rightarrow$ entre les éléments d'un ensemble E ($\rightarrow \subseteq E \times E$) est la plus petite relation transitive satisfaisant les deux axiomes suivants :*

1. *si $e_{ij} \in E_i$, si $e_{ik} \in E_i$ et si $j < k$ alors $e_{ij} \rightarrow e_{ik}$*
2. *si $s \in E_i$ est un événement d'envoi de message et si $r \in E_j$ est l'événement de réception correspondant alors $s \rightarrow r$*

Définition 3.2 *La relation de concurrence* $\|$ *entre les éléments d'un ensemble* E ($\| \subseteq E \times E$) *traduit l'indépendance causale entre deux événements e et e'. Elle est définie de la façon suivante :*

$$e \parallel e' \quad \text{si et seulement si} \quad \neg(e \rightarrow e') \wedge \neg(e' \rightarrow e)$$

Définition 3.3 *L'histoire causale d'un événement e notée* $C(e)$ *est définie par :*

$$C(e) = \{e' \in E | e' \rightarrow e\} \cup \{e\}$$

La projection de $C(e)$ *sur* E_i *notée* $C(e)[i]$ *est définie par* $C(e)[i] = \{e' \in C(e) | e' \in E_i\}$

De façon intuitive, la relation de causalité traduit que tout événement local précède tout autre événement local futur du même processus et qu'un événement d'envoi de message précède l'événement de réception correspondant. L'histoire causale d'un événement e peut se définir comme l'union des histoires causales locales de chaque processus P_i (c'est à dire $C(e) = C(e)[1] \cup \ldots \cup C(e)[n]$). Donc, si $e_{ij} \in C(e)[i]$ alors tous les événements locaux précédents e_{ij} appartiennent aussi à l'histoire causale locale (c'est à dire $e_{i1}, \ldots, e_{i,j-1} \in C(e)[i]$). De ce fait, chaque histoire causale d'un événement est caractérisée de façon suffisante par l'indice de l'événement le plus récent dans chaque histoire locale (si les événements sont numérotés par des entiers de façon non discontinue à partir d'un instant initial). De plus, cet indice est égal à la cardinalité de l'ensemble $C(e)[i]$ notée $|C(e)[i]|$.

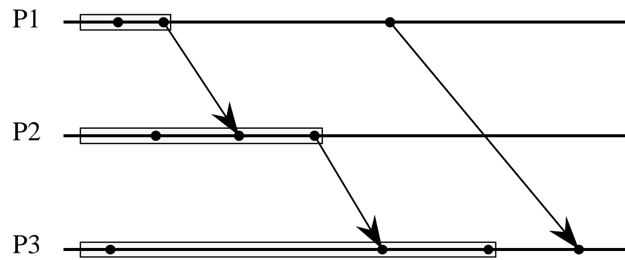


FIG. 3.1 – *Relations de dépendance et histoire causale*

L'absence d'horloge commune et les délais de communication ne permettent pas dans une exécution répartie de distinguer les notions de concurrence et de simultanéité. La première se définit comme l'absence de relation de dépendance causale entre deux événements. Cette relation n'est pas transitive. Par exemple, on peut avoir $e_1 \parallel e_2$ et $e_2 \parallel e_3$ alors que $\neg(e_1 \parallel e_3)$. La relation de simultanéité (définie par l'attribution de dates communes à tous les processus), parfois appelée vrai-parallélisme, est quant à elle transitive. A moins d'émettre l'hypothèse de la présence d'un observateur global, parfait et omniscient, ou ce qui est équivalent dans ce cas d'une synchronisation d'horloge parfaite, elle n'est pas calculable pour une exécution répartie. La notion de concurrence, qui est elle calculable dans les cas généraux, traduit donc tout au plus, une possibilité de simultanéité entre deux événements.

3.1.2 Estampillage des événements

A partir des dépendances définies par la relation de causalité, Lamport propose une datation des événements à l'aide d'horloges logiques. Chaque estampille est représentée par un entier naturel et chaque événement a une estampille supérieure à celles des événements dont il dépend causalement. Néanmoins, si l'estampillage par horloge logique est compatible avec la relation de précédence causale, le contraire n'est pas vrai.

Afin de lever cette limitation, Fidge dans [Fid88] et Mattern et Schwarz dans [Mat88, SM92] ont défini une représentation des histoires causales à l'aide de vecteurs d'estampilles. Cette technique se base sur le fait que chaque événement e_{ij} d'un processus P_i dépend de l'événement local l'ayant immédiatement précédé et des $n \Leftrightarrow 1$ derniers événements d'envoi de message appartenant aux processus autres que P_i. L'histoire causale d'un événement e peut alors être représentée de façon unique par un vecteur de dimension n. Cette méthode de datation par vecteurs d'estampille est souple et pratique. Elle requiert néanmoins la connaissance du nombre de processus utilisés par l'application. En effet, cette valeur est nécessaire pour dimensionner les vecteurs. Il faut donc, soit fixer de manière statique le nombre maximal de processus pour faire de l'estampillage à la volée, soit faire de l'estampillage *post mortem* (dans ce cas on est capable de calculer le nombre exact de processus qui a été créé lors de l'exécution). D'autres représentations, comme par exemple des listes ou des arbres d'estampilles, doivent donc être utilisées si on désire dater des événements à la volée avec un nombre de processus dynamique. Néanmoins, dans ce cas, la gestion des estampilles et de leur comparaison est plus lourde à mettre en œuvre qu'avec de simples vecteurs de taille fixe.

3.1.3 Notion de coupe cohérente

La notion de relation d'ordre permet de déterminer la précédence ou l'indépendance causale de deux événements ou de deux actions d'une exécution répartie. L'état global d'une exécution répartie peut alors être défini comme un n-uplet composé par les états locaux des n processus participant à l'algorithme. Cet état change sous l'effet de l'exécution d'une action ou de la prise en compte d'un événement. Parmi tous les n-uplet d'états locaux qu'il est possible de déterminer, Chandy et Lamport ont montré que seul un certain nombre correspond à des états globaux qui ont pu être réellement visités : ce sont les états globaux dits cohérents [CL85, FZ90, SM92, Fro96]. Il est clair que la notion d'état global cohérent ne définit pas seulement des états globaux composés d'états locaux observables à la même date, mais tous les états globaux qui auraient pu être observés à la même date si les conditions d'exécution sur les différents sites avaient été légèrement différentes (sans, toute fois, remettre en cause le séquencement imposé par les échanges de messages). Dans ce paragraphe, nous définissons plus précisément cette notion.

Définition 3.4 *Etant donné un ensemble E d'événements ou d'actions et une relation d'ordre $\rightarrow$, un sous-ensemble fini $C \subseteq E$ est appelé une coupe cohérente de l'exécution répartie si et seulement si, si $e \in C$ alors $\{e' \in E | e' \rightarrow e\} \subseteq C$*

De manière informelle, cette définition stipule que si un élément (événement ou action) e appartient à une coupe cohérente alors tous les éléments e' dépendant causalement de e appartiennent également à la coupe. En d'autres termes, le passé causal de tout élément d'une coupe cohérente appartient également à la coupe. Par exemple, la coupe C_1 de la figure 3.2 n'est pas cohérente. En effet, l'élément e_{22} appartient à la coupe alors que e_{12} qui dépend causalement de e_{22} n'en fait pas partie. Par contre, la coupe C_2 est cohérente. Une coupe cohérente sépare donc une exécution en un passé et un futur.

On peut alors définir l'ensemble des coupes cohérentes qui peuvent être observées lors d'une exécution. Cet ensemble a la structure mathématique d'un treillis. La figure 3.3 présente le treillis des états globaux cohérents correspondant à l'exécution de la figure 3.2. Chaque axe du diagramme en trois dimensions est associé à un processus. Les points représentent les états globaux cohérents.

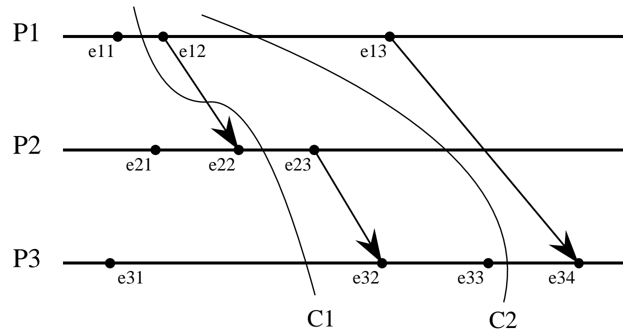


FIG. 3.2 – *Coupes non cohérente et cohérente*

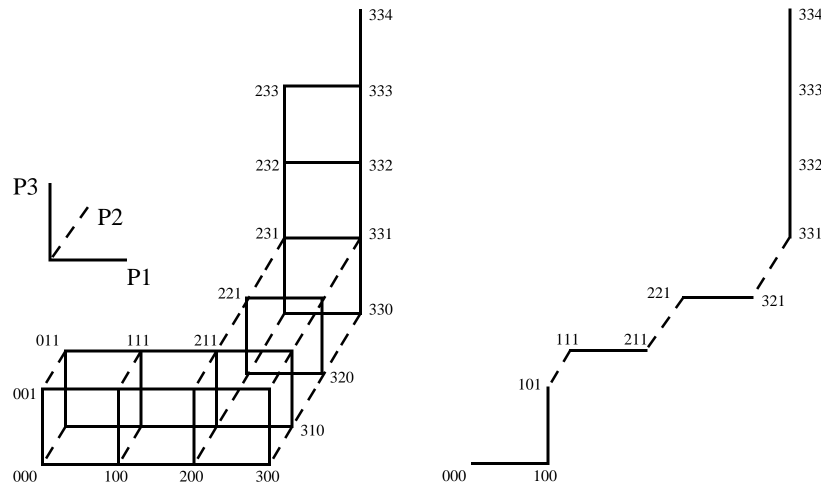


FIG. 3.3 – *Treillis des états globaux cohérents et observation possible de l'exécution*

Par exemple, le point $(2, 3, 1)$ correspond à la coupe qui passe par les éléments e_{12}, e_{23} et e_{31}. Chaque segment de droite sur un axe correspond à un changement d'état dans l'un des trois processus. Finalement, un chemin de l'état initial vers l'état final sans retour vers le passé correspond à une observation de l'exécution répartie. Le treillis regroupe donc toutes les observations possibles de l'exécution.

3.1.4 Conclusion sur la notion d'état global

La notion de causalité permet d'ordonner les événements d'une exécution répartie. Elle fournit ainsi un temps logique pour les systèmes répartis ne possédant ni horloge globale, ni horloges locales synchronisées. La notion de coupe cohérente permet alors de reconstituer les différents états globaux pouvant être observés au cours d'une exécution répartie. Cette notion est à la base des techniques de vérification de propriétés, de déverminage, de reprise arrière d'exécution et de rejeu dans les environnements répartis. Dans cette partie, nous avons présenté la relation *Happened before* proposée par Lamport. C'est la relation de causalité la plus couramment employée dans le domaine des systèmes répartis. Elle comprend une source d'ordre local qui définit des précédences entre événements d'un même processus et une source d'ordre de communication qui, à partir d'un mode par envoi de message asynchrone, définit des précédences entre événements de processus

distants.

Néanmoins, deux limitations peuvent être mentionnées à propos de la relation *Happened before* : les événements datés sont de trop bas niveau ce qui entraîne un nombre de dépendances élevé et c'est une relation purement observationnelle. Nous avons donc proposé (Cf. [PDFS95b, PDFS95a]), dans le cadre d'une approche de programmation orientée objet, un ensemble de relations de causalité basées non pas sur la notion d'événement mais sur celle d'action. Une action est par exemple, une méthode, un bloc d'instructions ou une construction algorithmique de base. Elle factorise ainsi des événements présentant une cohérence logique entre eux et peut être raffinée afin de fournir différents degrés de précision dans la description d'un comportement. De plus, les opérations de communication n'étant pas les seules sources d'ordre dans les environnements répartis, nous avons proposé des relations permettant de traduire les dépendances issues, par exemple, de la synchronisation d'actions concurrentes ou des mises à jour transactionnelles de données. Enfin, la prise en compte de différentes sémantiques de communication comme l'invocation synchrone ou semi-synchrone de méthode et l'invocation sur groupe avec un ordre de soumission ou d'exécution, permet de traduire plus fidèlement un comportement réparti. La seconde limitation concernant la relation *Happened before* provient du fait qu'elle s'intéresse uniquement à l'observation d'une application répartie. Or, la notion d'ordre partiel intervient également lors des phases de spécification et d'exécution. En effet, la spécification fournit un ordre partiel entre actions qui doit être respecté lors de l'exécution. Puis, le moteur d'exécution interprète cette spécification et fournit un ordre d'exécution qui représente toutes les exécutions possibles conformes à la spécification. Finalement, l'ordre d'observation consiste à traduire une exécution particulière parmi toutes celles possibles. Nous avons donc proposé dans [CDFS96] d'intégrer ces trois points de vue (spécification, exécution et observation) dans une même démarche. Ils fournissent une vision plus précise d'une application répartie que celle fournie par la relation *Happened before*.

Dans la suite de ce chapitre, nous présentons une utilisation de la relation d'ordre *Happened before* dans le cadre de la spécification du comportement d'un système multi-agents. Le formalisme proposé par Fagin, Halpern, Moses et Vardi, étend également le point de vue observationnel de la notion d'état global cohérent au sens de Chandy et Lamport avec la notion de configurations possibles appelées mondes pour une exécution répartie.

3.2 Logique épistémique et temporelle

Fagin, Halpern, Moses et Vardi proposent dans [HF89, FHMV95, FHMV96] une logique pour la modélisation de systèmes multi-agents. L'originalité de cette approche est d'incorporer au sein d'un même formalisme les modalités temporelles $\Box$, $\Diamond$, $\bigcirc$ et $\mathcal{U}$ et les modalités épistémiques K, E, et C présentées respectivement aux paragraphes 2.2 et 2.3. Les auteurs envisagent une application distribuée comme un ensemble d'agents autonomes poursuivant des buts locaux. Ces agents interagissent par le biais d'un réseau de communication. La poursuite de buts locaux associée à cette forme de coopération permet d'atteindre le but global de l'application distribuée. Cette approche place la notion de connaissance distribuée au centre de tout comportement multi-agents. Dans le paragraphe suivant, nous définissons plus précisément les notions, au sens de Fagin et al, de système multi-agents et d'exécution d'un système. En particulier, nous comparons leur notion d'état global avec la notion d'état global cohérent issue du domaine des systèmes répartis. Le paragraphe 3.2.2 propose en terme de structure de Kripke [Kri63] et de mondes possibles une sé-

mantique pour les modalités épistémiques et temporelles retenues par les auteurs. Ces opérateurs permettent d'une part, de spécifier des comportements pour les systèmes multi-agents et d'autre part, de raisonner sur les différentes exécutions possibles de tels systèmes.

3.2.1 Définition

Les auteurs définissent l'état global d'un système par un tuple $(s_e, s_1, \ldots, s_n)$ comprenant l'état de l'environnement s_e (c'est à dire les interactions en transit dans les canaux de communication) et les états locaux s_i de chacun des agents du système. L'état de l'environnement et les états locaux peuvent à leur tour être définis comme une fonction entre un ensemble de variables et un ensemble de valeurs. Une exécution du système est alors une fonction de la variable temps vers l'ensemble des états globaux. Les auteurs retiennent l'hypothèse d'un temps linéaire et discret. Dans ce cas, la variable temps peut être représentée par des entiers naturels. La prise en compte d'un temps arborescent (Cf. [EH85, EH86]) et d'opérateurs temporels du passé (Cf. [LPZ85]) semble apporter une plus grande richesse dans la description des comportements. Néanmoins, elle introduit de nouveaux opérateurs et complique l'interprétation des programmes temporels à base de connaissances. Ces extensions n'ont donc pas été retenues par les auteurs.

Une exécution r du système est une séquence infinie $\langle r(0), r(1), \ldots \rangle$ d'états globaux. Un système $\mathcal{R}$ est modélisé par l'ensemble de toutes les exécutions possibles de l'application distribuée. La structure mathématique de cet ensemble est celle d'un treillis. Panangaden et Taylor ont suggéré dans [PT92] de limiter cette définition aux seuls états globaux associés à des coupes cohérentes au sens de Chandy et Lamport (Cf. paragraphe 3.1.3). Bien que cette démarche semble pertinente puisque seules les coupes de ce type sont des observations cohérentes d'une exécution répartie, elle n'a pas été retenue par Fagin et al. La notion de système étend donc le treillis des états globaux cohérents selon trois directions :

- l'état de l'environnement est pris en compte (un axe spécialisé est donc ajouté au treillis pour modéliser l'état des canaux de communication),
- le treillis est étendu à tous les états globaux possibles,
- alors que la démarche de Chandy et Lamport s'intéresse à toutes les observations possibles d'une exécution, la notion de système au sens de Fagin et al comprend toutes les exécutions possibles d'une application.

Etant donné un système $\mathcal{R}$, le couple (r, m) comprenant une exécution $r \in \mathcal{R}$ et un instant m est désigné sous le terme de point. L'expression $r_i(m)$ désigne alors l'état local s_i de l'agent i à l'instant m. Deux points (r, m) et (r', m') sont dits indistinguables pour l'agent i et sont notés $(r, m) \sim_i (r', m')$ si $r_i(m) = r'_i(m')$ c'est à dire si l'agent i a le même état local aux deux points.

3.2.2 Sémantique

La logique mise en place par Fagin et al utilise les modalités épistémiques K (connaissance), E (connaissance de tous) et C (connaissance commune) ainsi que les modalités temporelles $\Box$ (toujours), $\Diamond$ (inévitablement), $\bigcirc$ (dans l'état suivant) et $\mathcal{U}$ (jusqu'à). Ils considèrent un ensemble G de n agents et un ensemble P de propositions. La sémantique de cette logique s'exprime en terme de mondes possibles. Elle est définie à l'aide de la structure de Kripke $\mathcal{I} = \langle \mathcal{G}, \pi, \sim_1, \ldots, \sim_n \rangle$ où $\mathcal{G}$

est l'ensemble de tous les états globaux, $\pi : \mathcal{G} \rightarrow (P \rightarrow \{true, false\})$ une fonction associant une valeur de vérité à chaque proposition dans chaque état et $\sim_i$ les relations définies précédemment. Les points (r, m) sont les états, les triplets $(\mathcal{I}, r, m)$ sont les mondes possibles et les relations $\sim_i$ sont les relations d'accessibilité entre états. La satisfaisabilité d'une formule φ dans un monde $(\mathcal{I}, r, m)$ est notée $(\mathcal{I}, r, m) \models \varphi$. Elle est déduite des règles présentées figure 3.4.

$(\mathcal{I}, r, m) \models p$	si et seulement si	$\pi(s)(p) = true$
$(\mathcal{I}, r, m) \models \varphi \wedge \psi$	si et seulement si	$(\mathcal{I}, r, m) \models \varphi$ et $(\mathcal{I}, r, m) \models \psi$
$(\mathcal{I}, r, m) \models \neg\varphi$	si et seulement si	$(\mathcal{I}, r, m) \not\models \varphi$
$(\mathcal{I}, r, m) \models K_i\varphi$	si et seulement si	$(\mathcal{I}, r', m') \models \varphi$ pour tous les points (r', m') tels que $(r', m') \sim_i (r, m)$
$(\mathcal{I}, r, m) \models E_G\varphi$	si et seulement si	$\forall i \in G,\ (\mathcal{I}, r, m) \models K_i\varphi$
$(\mathcal{I}, r, m) \models C_G\varphi$	si et seulement si	$\forall i \in G,\ \forall k,\ (\mathcal{I}, r, m) \models (E_G\varphi)^k$
$(\mathcal{I}, r, m) \models \Box\varphi$	si et seulement si	$\forall m' \geq m,\ (\mathcal{I}, r, m') \models \varphi$
$(\mathcal{I}, r, m) \models \Diamond\varphi$	si et seulement si	$\exists m' \geq m,\ (\mathcal{I}, r, m') \models \varphi$
$(\mathcal{I}, r, m) \models \bigcirc\varphi$	si et seulement si	$(\mathcal{I}, r, m+1) \models \varphi$
$(\mathcal{I}, r, m) \models \varphi\mathcal{U}\psi$	si et seulement si	$\exists m' \geq m,\ (\mathcal{I}, r, m) \models \psi$ et $\forall m''/m \leq m'' \leq m',\ (\mathcal{I}, r, m'') \models \varphi$

FIG. 3.4 – *Sémantique de la logique épistémique et temporelle*

Cette interprétation des modalités épistémiques et temporelles est très proche des interprétations habituelles présentées aux paragraphes 2.2 et 2.3. La sémantique des opérateurs temporels est celle que l'on retrouve couramment pour un temps linéaire et discret. La définition des opérateurs de connaissance de tous et de connaissance commune est fondée sur celle de la modalité K de connaissance. Cette dernière stipule qu'un fait est connu s'il est vrai dans tous les mondes considérés comme possibles à partir de l'état courant. Cet ensemble comprend tous les états globaux de toutes les exécutions possibles dans lesquels l'état local de l'agent est identique à l'état local courant. Par exemple dans le système $\mathcal{R}$ représenté par le treillis de la figure 3.5, l'ensemble des mondes possibles au point (r, m) pour l'agent P_3 comprend tous les points du plan P. D'une façon générale, l'ensemble des mondes possibles pour un agent à un point (r, m) comprend l'ensemble des points du plan passant par (r, m) et perpendiculaire à l'axe associé à l'agent.

3.3 Programmes à base de connaissance de niveau agent

La logique pour les systèmes multi-agents que nous venons de présenter a pour but de faciliter la description de comportements répartis. Elle est le support formel de la notion de programme à base de connaissance. Ceux-ci permettent de spécifier des comportements mettant en jeu des connaissances distribuées et de raisonner sur ces spécifications c'est à dire de vérifier un ensemble de propriétés temporelles sur les séquences d'exécutions possibles.

Dans cette approche, chaque agent est considéré comme une entité autonome dont l'évolution dépend de ses connaissances sur l'environnement et sur l'état de ses pairs. Le paragraphe 3.3.1 définit la notion de programme à base de connaissance pour un agent. Le paragraphe 3.3.2 illustre cette notion avec la spécification d'un protocole de transmission de données pour un réseau non fiable. Le paragraphe 3.3.3 introduit alors le concept de raffinement et montre comment une spécification à base de connaissances peut être dérivée afin de fournir une implantation opérationnelle.

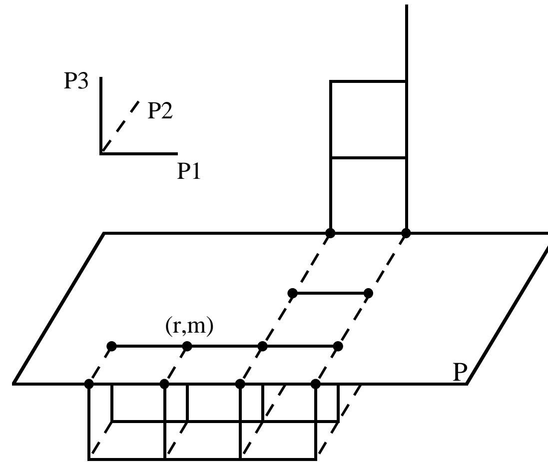


FIG. 3.5 – *Ensemble des mondes possibles pour l'agent P_3 au point (r, m)*

Finalement, le paragraphe 3.3.4 conclut ce paragraphe et résume les principales caractéristiques introduites par Fagin et al dans les programmes à base de connaissance de niveau agent.

3.3.1 Définition

Plutôt que de définir l'état local d'un agent par une fonction entre un ensemble de variables et un ensemble de valeurs, une approche épistémique permet d'abstraire en terme de connaissance une partie des informations contenues dans cette fonction. Dans le point de vue de Fagin et al, on peut exprimer le fait qu'un agent i a bien reçu une donnée *bit* transmise par un émetteur en spécifiant qu'il connaît sa valeur et en écrivant la formule $K_i(bit)$. Dans le cas d'une variable binaire, cette expression résume en fait la formule $K_i(bit = 1) \vee K_i(bit = 0)$. Au delà d'une simple réécriture à l'aide d'opérateurs épistémiques de la fonction $\{bit\} \rightarrow \{true, false\}$, cette formulation distingue de façon précise deux situations : un état antérieur dans lequel la valeur n'est pas connue (c'est à dire un état dans lequel la variable n'a encore jamais été assignée) et un état postérieur à l'affectation de la variable. En étendant cette démarche à l'ensemble des agents du système on est en mesure d'exprimer les connaissances d'un agent sur l'état d'un de ses pairs. Par exemple, la formule $K_j K_i(bit)$ exprime le fait que l'agent j sait que l'agent i connaît la valeur de la variable *bit*. En poursuivant l'analogie réseau, on peut dire que cela correspond, pour l'agent j, à la réception de l'acquittement indiquant que la donnée bit a été reçue par l'agent i. Fagin et al dans [FHMV95, FHMV96] définissent alors la notion de programme à base de connaissance comme un ensemble de règles qui en fonction de tests sur la connaissance d'un agent, détermine les actions à entreprendre. Chaque agent i exécute un programme ayant la forme suivante [1] :

1. Fagin et al. notent les programmes à l'aide de la structure **case of** plutôt qu'avec **while** true **do**. La sémantique du programme étant de tester en permanence l'état des connaissances, l'emploi d'une boucle infinie nous semble plus explicite.

while true **do**
  if $t_1 \wedge k_1$ **do** a_1
  if $t_2 \wedge k_2$ **do** a_2
  ...
end do

Les tests t_j sont des tests standards sur les valeurs des variables locales de l'agent tandis que les expressions k_j sont des tests de connaissance. Les termes de type a_j sont les actions à entreprendre par l'agent. Les tests de connaissance pour un agent i sont des combinaisons booléennes de formules de type $K_i\varphi$ où φ peut être n'importe quelle formule comprenant des opérateurs modaux temporels ou de connaissance. De façon intuitive, un agent sélectionne une action en fonction du résultat des tests standards sur son état local et des tests de connaissance sur son état de connaissance. Si l'évaluation des tests standards ne pose *a priori* pas de difficultés, celle des tests de connaissance se heurte au problème de l'interprétation opérationnelle des modalités épistémiques et temporelles. Deux voies principales peuvent être dégagées :

– les formules épistémiques peuvent être considérées comme des prédicats calculables de type Prolog. Chaque agent possède alors une base de fait dans laquelle sont stockées les expressions qu'il considère comme étant vraies. La mise à jour de la connaissance de l'agent se fait par ajouts et par retraits d'éléments dans cette base. L'évaluation d'un test de connaissance consiste alors à inférer la valeur de vérité d'un prédicat. Ce processus peut être réalisé par un simple test de la présence ou de l'absence du prédicat dans la base de faits ou peut nécessiter dans les cas plus complexes l'utilisation de règles de déduction.

– les formules épistémiques peuvent être considérées comme des spécifications. Ce sont alors des expressions abstraites qui doivent être raffinées en vue d'obtenir une implantation opérationnelle. Dans ce cas, un programme à base de connaissances ne devient implantable que lorsque tous les tests de connaissance k_j ont été traduits en terme de tests standards t_j.

La première solution est très opérationnelle car elle assigne une interprétation informatique aux modalités épistémiques et temporelles. Elle amène néanmoins plusieurs problèmes. Tout d'abord, certaines notions épistémiques telles que la connaissance distribuée ne sont pas représentables dans cette approche. En effet, la définition même d'une telle modalité stipule qu'aucun agent ne connaît le fait distribué mais qu'il existe un algorithme qui, à partir de données fournies en entrée, permet d'inférer ce fait. Donc, si aucun agent ne connaît directement le fait distribué, il ne peut être ajouté dans aucune des bases de connaissance. En effet, à partir du moment où on l'insère dans une base de connaissance, il perd son caractère distribué pour acquérir un statut instancié. La seconde difficulté introduite par cette solution concerne la cohérence du contenu des différentes bases de connaissance. Par exemple, la notion de connaissance de tous est associée aux faits connus par l'ensemble des agents. Ces faits doivent donc être enregistrés dans chacune des bases du système. L'absence de contrôle centralisé et les délais de communication non bornés font qu'il n'est pas possible de garantir à un instant donné, que toutes les bases de connaissance possèdent ce prédicat et qu'elles le retirent simultanément dès qu'il est révoqué.

En considérant les modalités épistémiques et temporelles comme des éléments liés uniquement au processus de spécification, la seconde solution s'affranchit de ces problèmes. Ainsi, les tests de connaissance conservent leur caractéristique principale qui est de fournir un formalisme de haut niveau pour abstraire les détails de l'implantation. Les programmes à base de connaissances doivent

donc être raffinés étape après étape, en des programmes de plus en plus précis afin de transformer tous les tests de connaissances en tests standards. Le nombre d'itérations de ce processus de développement n'est en aucune manière fixe. Il comprend autant d'étapes de raffinement que le concepteur le juge nécessaire.

3.3.2 Exemple

Afin d'illustrer la notion de programme à base de connaissance et le processus de raffinement associé, nous présentons l'exemple tiré de [FHMV95] d'un programme de transfert de données. Ce programme comprend deux processus, un émetteur E et un récepteur R, qui communiquent au travers d'un réseau. L'émetteur débute son protocole en envoyant un bit (soit 0, soit 1) au récepteur. Le médium de communication est considéré comme non fiable et des messages peuvent être perdus. Il n'y a donc pas de garantie que le message envoyé par E sera reçu par R. Les délais de transmission sont non bornés et les canaux de communication sont FIFO. Afin de simplifier la présentation, on suppose que la perte de message est le seul type de comportement fautif pouvant avoir lieu. Le protocole est le suivant : l'émetteur envoie la donnée au récepteur jusqu'à ce que celui-ci l'acquitte avec un message de type *ack*. Le récepteur commence l'envoi des acquittements tout de suite après la réception du premier message. Pour être sûr que l'émetteur arrête l'envoi des données, le récepteur transmet continuellement des messages d'acquittement afin qu'il y en ait au moins un qui parvienne à l'émetteur. Bien que le protocole se limite à ces deux phases, on peut néanmoins supposer que l'envoi des acquittements ne dure pas indéfiniment. On peut décider par exemple qu'il cesse lors de la phase suivante de transmission de données. Ce protocole comprend deux actions *sendbit* pour l'envoi des données et *sendack* pour l'envoi de l'acquittement. Le comportement de l'émetteur peut être décrit par le programme TB_E (pour transmission de bit par l'émetteur) suivant [2] :

TB_E : **if** $\neg K_E K_R(bit)$ **do** sendbit

En d'autres termes, l'émetteur envoie la donnée tant qu'il ne sait pas si le récepteur la connaît. Le comportement du récepteur consiste quant à lui, à envoyer le message d'acquittement dès qu'il connaît la valeur de la donnée (c'est à dire dès que $K_R(bit)$ est vrai) et tant qu'il ne sait pas si l'émetteur sait qu'il connaît cette donnée (c'est à dire tant que $K_R K_E K_R(bit)$ est faux). Le programme TB_R du récepteur comporte donc l'action suivante :

TB_R : **if** $K_R(bit) \wedge \neg K_R K_E K_R(bit)$ **do** sendack

3.3.3 Raffinement

Le programme présenté ci-dessus comprend un certain nombre de tests de connaissance. Afin d'implanter cette spécification, nous allons raffiner ce programme.

Pour cela, nous introduisons les propositions booléennes *recbit* et *recack*. La première est associée à l'agent récepteur et vaut vraie si ce dernier a reçu la donnée et faux dans les autres cas. La proposition *recack* est quant à elle, associée à l'émetteur et vaut vraie si l'émetteur a reçu le message d'acquittement. L'expression $K_R(bit)$ qui indique que le récepteur connaît la valeur de la donnée, est alors équivalente à la proposition *recbit* qui est un test standard et local au récepteur.

2. Afin de simplifier les notations, nous omettons systématiquement la boucle infinie **while** true **do** dans l'écriture des programmes à base de connaissances.

De même, lorsque l'émetteur a reçu un acquittement, c'est à dire lorsqu'il sait que le récepteur connaît la valeur de la donnée (en d'autres termes lorsque $K_E K_R(bit)$ est vrai) alors le test standard local à l'émetteur *recack* est vrai. Le programme précédent TB peut alors être raffiné par le programme $TB^{'}$ suivant :

$TB^{'}_E$: **if** $\neg recack$ **do** sendbit
$TB^{'}_R$: **if** $recbit \wedge \neg K_R(recack)$ **do** sendack

Le principe de raffinement consiste donc à introduire de plus en plus de détails dans les programmes des agents. D'un point de vue pratique, cela revient à remplacer les tests de connaissance par des tests standards. Le programme $TB^{'}$ est un raffinement du programme TB de transmission de bit. Il peut quasiment être implanté tel quel. En effet, seul le test de connaissance $\neg K_R(recack)$ est encore présent dans le programme du récepteur. Cette condition a pour but de stopper l'envoi des acquittements lorsque l'émetteur en a reçu au moins un. Les raffinements précédents, $K_E K_R(bit)$ en *recack* d'une part, et $K_R(bit)$ en *recbit* d'autre part, sont des tests sur l'état de connaissance du possesseur des variables booléennes *recack* et *recbit*. L'expression $\neg K_R(recack)$ est pour sa part, un test de l'état de connaissance du récepteur sur une variable de l'émetteur. Cette condition vaut vrai lorsque le récepteur sait que l'émetteur a reçu l'acquittement. Or, les deux seuls messages échangés par ce protocole sont *sendbit* et *sendack*. Donc, en l'absence de spécification plus détaillée, aucune interaction n'apporte à l'émetteur la connaissance de la réception ou de la non réception de son acquittement. Comme nous l'avons déjà suggéré, ce niveau de connaissance pourrait être atteint si le protocole se poursuivait par un envoi de message par l'émetteur. Cela permettrait d'acquitter l'acquittement. Néanmoins, cette solution n'est pas satisfaisante car le problème de l'acquittement du dernier message subsiste. On peut alors constater que ce problème de terminaison revient à tenter d'atteindre un niveau de connaissance commune entre l'émetteur et le récepteur. Fagin et al ont montré dans [FHMV95, FHMV96] qu'un tel niveau n'est pas atteignable dynamiquement dans un réseau asynchrone possédant des délais de transmission non bornés. Les seules connaissances communes possibles dans un tel système sont celles incorporées à l'initialisation aux programmes de chacun des agents. On constate donc que l'expression $\neg K_R(recack)$ ne peut être traduite de façon satisfaisante en un test standard. On peut alors imaginer que le récepteur envoie un nombre fini de messages d'acquittement afin que la probabilité qu'il y en ait au moins un qui arrive à l'émetteur soit suffisamment élevée. On transforme ainsi une connaissance en croyance. L'expression $\neg K_R(recack)$ est remplacée par $\neg B_R(recack)$ où B est la modalité de croyance. On peut alors décider par exemple, que cette expression se raffine par $\#sendack < Max$ où $\#sendack$ représente le nombre de tentatives d'envoi de l'acquittement et *Max* une constante prédéfinie. Le programme précédent $TB^{'}$ peut alors être raffiné par le programme $TB^{''}$ qui ne comporte que des tests standards :

$TB^{''}_E$: **if** $\neg reacack$ **do** sendbit
$TB^{''}_R$: **if** $recbit \wedge \#sendack < Max$ **do** sendack

Le raffinement d'un programme à base de connaissance en terme d'exécutions possibles permet d'introduire des raisonnements temporels sur les différentes séquences d'états globaux. Par exemple on exprime que l'émetteur finit inévitablement par savoir que le récepteur connaît la valeur du bit par la formule $\Diamond K_E K_R(bit)$. De même l'invariant qui stipule que lorsque l'émetteur sait que le récepteur connaît la donnée alors plus aucun message de donnée n'est envoyé se traduit par la formule $\Box(K_E K_R(bit) \Rightarrow \neg sendbit)$.

3.3.4 Conclusion sur les programmes de niveau agent

Dans ce paragraphe, nous avons présenté la notion de programmes à base de connaissances pour les systèmes multi-agents telle qu'elle a été introduite par Fagin et al dans [HF89, FHMV95, FHMV96]. Dans de tels systèmes, les agents sont vus comme des entités autonomes qui possèdent un état local et un état de connaissance. Le premier est une fonction entre un ensemble de variables et un ensemble de valeurs. Le second résume à l'aide de prédicats modaux la perception qu'a chaque agent de l'état de ses pairs. Un programme à base de connaissances pour un agent est alors un ensemble de règles qui en fonction de l'évaluation de tests dits standards sur l'état local et de tests de connaissance détermine les actions à entreprendre. La notion de raffinement pour de tels programmes consiste essentiellement à transformer les tests de connaissance en tests standards. Dans l'approche de Fagin et al, le comportement global d'une application distribuée se déduit des exécutions locales des programmes à base de connaissances et des opérations d'interactions entre les agents.

3.4 Conclusion sur la programmation à base de connaissances

La programmation à base de connaissances est une démarche issue du domaine des logiques épistémiques et des systèmes multi-agents. Elle propose un paradigme de programmation fondée sur la notion de connaissance. Plutôt que de manipuler des faits bruts, un programme à base de connaissances manipule une connaissance de ces faits. Cette approche est particulièrement intéressante dans le domaine des systèmes répartis. En effet, dans de tels environnements, les nombreuses sources d'indéterminisme créées par les systèmes et les réseaux, rendent délicate la manipulation de certaines données. Par exemple, on ne peut pas garantir qu'entre le moment où on acquiert une variable distante et celui où on manipule sa valeur, celle-ci n'a pas été modifiée. Une solution serait d'évaluer quantitativement la pertinence de l'information manipulée en lui attribuant, par exemple, un pourcentage de confiance. Cette approche est néanmoins, très difficile et ne peut vraiment être mise en place que dans certains cas particuliers. Les modalités épistémiques retenues par Fagin et al permettent quant à elles, d'introduire un critère qualitatif dans l'évaluation et le traitement des connaissances.

Fagin, Halpern, Moses et Vardi proposent une formalisation à base de connaissances pour la modélisation des applications distribuées dans les environnements multi-agents. Chaque agent est autonome, poursuit des buts locaux et interagit au travers d'un réseau de communication. Le but global de l'application distribuée est alors atteint par la mise en commun de ces buts locaux et de ces interactions. Celles-ci se définissent par des opérations d'échange et de manipulation de la connaissance. Le formalisme proposé par les auteurs contient des modalités épistémiques permettant de décrire les différents niveaux de connaissance et des modalités temporelles permettant de décrire l'évolution de cette connaissance. Un programme à base de connaissance pour un agent est alors un ensemble de règles de type système expert qui testent en permanence l'état local et l'état des connaissances d'un agent. Les modalités retenues par les auteurs permettent de s'affranchir des détails des implantations et d'abstraire en terme de connaissances les comportements d'un système multi-agents. La notion de raffinement permet alors de transformer, étape après étape, les modalités de connaissance en comportements concrets.

L'interprétation des modalités dans un tel contexte fait appel à la notion d'état global cohérent

que nous avons donc présenté au début de ce chapitre. Puis, nous avons présenté les modalités retenues par les auteurs et leurs sémantiques. Finalement, nous avons illustré, à l'aide d'un exemple d'école, la formalisation d'un problème distribué. Le principal avantage de cette approche est d'offrir un point de vue de haut niveau qui traduit de façon synthétique les comportements répartis. Elle impose néanmoins une démarche de conception ascendante : les buts globaux poursuivis émergent des différents buts locaux. Dans le chapitre 4, nous adaptons cette approche à une démarche de conception descendante qui privilégie la déduction de buts locaux à partir de buts globaux clairement identifiés.

Deuxième partie

Coordination inter-objets

Chapitre 4

Conception de comportements de groupe

Dans les chapitres précédents, nous avons présenté un certain nombre de propositions existantes pour la description de comportements répartis avec une approche objet. Ce chapitre présente notre approche pour la spécification de comportements de niveau groupe. Notre but est de mettre en place une démarche algorithmique de groupe.

Les méthodologies de conception des logiciels définissent en général deux types de modèles de spécification : un modèle pour les données et un modèle pour les comportements. Les premiers s'intéressent aux aspects statiques d'une application. Ils définissent les données manipulées, leur type, les contraintes d'intégrité qu'elles doivent respecter, la façon dont elles sont organisées hiérarchiquement et leurs relations de dépendances mutuelles. La plupart des méthodologies existantes utilisent pour cela un modèle entité-relation [Che76], des diagrammes de classe et d'objets ou des extensions et variantes de ces deux approches. Les modèles de comportement s'intéressent, quant à eux, aux aspects dynamiques d'une application. Ils décrivent l'évolution d'une application en réponse à des événements externes ou internes. La plupart de ces modèles sont fondés sur des extensions ou des modifications des automates états/transitions. Par exemple, les automates de Harel introduisent une approche hiérarchique dans la conception des modèles états/transitions et sont utilisés, entre autres, dans les méthodes Booch, OMT et UML présentées au paragraphe 1.3.

L'approche que nous avons développée s'intéresse essentiellement aux aspects comportementaux d'une application distribuée. L'originalité de notre démarche est d'étendre le point de vue local des formalismes précédents à un niveau d'abstraction de groupe. Nous introduisons un processus de développement et des gabarits de conception qui permettent de décrire de façon globale des schémas standards de comportement pour un ensemble d'objets distribués. Le processus de développement comprend trois niveaux méthodologiques et est fondé sur la notion de raffinement. Les trois niveaux sont le niveau groupe, le niveau objet et le niveau méthode. Par rapport au travail antérieur, l'apport de cette thèse concerne, d'une part les outils et les formalismes disponibles à chaque niveau de conception, et d'autre part, une précision de la sémantique du modèle proposé. Ainsi, en ce qui concerne le niveau groupe, nous proposons, dans ce chapitre, la notion de programme à base de connaissance de niveau groupe et une notation associée permettant l'expression des comportements répartis. Cette approche est complétée au chapitre 5 par la définition de structures de contrôle réparties. En ce qui concerne le niveau objet, nous proposons au cha-

pitre 6 un modèle de synchronisation ainsi qu'un protocole de niveau méta-objet permettant la synchronisation d'objets concurrents.

Le paragraphe 4.1 présente le processus de développement proposé. Il définit le niveau groupe pour le comportement d'un ensemble d'objets, le niveau objet pour le comportement d'un objet au sein du groupe, et le niveau méthode pour le comportement d'une méthode au sein d'un objet. C'est un processus descendant qui procède par raffinements successifs. Le paragraphe 4.2 définit plus précisement la notion de comportement de groupe qui est l'apport majeur de ce processus. Le paragraphe 4.3 présente alors les structures de données réparties manipulées par ces comportements. Le paragraphe 4.4 propose l'utilisation d'un certain nombre d'opérateurs épistémiques pour la description des degrés de connaissance d'une structure répartie. Ces opérateurs sont essentiellement ceux retenus par Fagin, Halpern, Moses et Vardi dans la notion de programme à base de connaissances pour un agent (Cf. chapitre 3). On retrouve donc les opérateurs de connaissance, de connaissance de tous et de connaissance commune. Nous justifions l'utilisation de trois opérateurs supplémentaires pour la conception d'applications distribuées. Ces opérateurs, qui existent par ailleurs dans d'autres logiques épistémiques (Cf. chapitre 2), permettent d'exprimer les notions de connaissance distribuée, de connaissance instanciée et de croyance. Finalement, nous présentons, au paragraphe 4.5, la notion de programme à base de connaissances de niveau groupe. Ces programmes décrivent les connaissances échangées au sein d'un groupe d'objets distribués et peuvent être vus comme des extensions de la notion de programme à base de connaissances pour un agent.

4.1 Processus de développement

Le processus de développement que nous présentons dans ce chapitre est issu des travaux réalisés au sein de notre équipe par Bonnet [Bon94, BDFS97], au cours de son mémoire d'ingénieur, et enrichis par cette thèse. Il est associé à un système semi-formel de preuve de propriétés de correction. La modélisation complète d'un algorithme est rarement instantanée. Ce travail se fait étape par étape, par raffinements successifs. Trois niveaux méthodologiques sont proposés : groupe, objet et méthode. Chacun d'eux permet d'adopter un point de vue particulier dans la spécification d'un comportement réparti. Ils constituent une hiérarchie de niveaux d'abstraction qui va du plus général au plus détaillé. La conception complète d'une application comporte donc un ensemble de modèles de spécification qui appartiennent respectivement à l'un de ces trois niveaux proposés. On part donc d'un modèle de niveau groupe très général, comportant peu de détails. On raffine ce modèle jusqu'à atteindre une spécification assez détaillée permettant de faire apparaître le comportement des objets. On projette alors le comportement sur chaque objet et on poursuit le processus de raffinement jusqu'à faire apparaître les différentes méthodes composant l'objet. Enfin, on achève ce processus en raffinant la spécification des méthodes et en faisant apparaître le pseudo-code composant la structure des méthodes. Ce processus de développement linéaire permet de garantir que les comportements définis au niveau méthode découlent de ceux définis au niveau objet, qui eux-mêmes, découlent de ceux définis au niveau groupe.

Le formalisme originel utilise une syntaxe graphique à base d'automates états/transitions pour dénoter les comportements dans les trois niveaux. Chaque état est associé à un prédicat et les transitions représentent des opérations de type condition/action. Au niveau groupe, les prédicats sont écrits à l'aide des opérateurs de connaissance distribuée présentés au paragraphe 2.3. Chaque transition peut donc être vue comme un transformateur de connaissance de groupe. Au niveau ob-

jet, les prédicats sont des fonctions de l'ensemble des variables globales de l'objet dans un ensemble de valeurs. Ils traduisent localement les connaissances définies au niveau groupe. Finalement, au niveau méthode, les prédicats utilisent en plus les variables locales aux méthodes. La sémantique informelle de ces modèles est la suivante : lorsque l'état est atteint par un flot de contrôle, il reste actif tant que son prédicat est vérifié. Nous avons étendu cette approche en lui associant une syntaxe textuelle. Au niveau groupe, les comportements sont décrits à l'aide de la notion de programme à base de connaissances présentée au paragraphe 4.5 et utilisent des structures de contrôle réparties définies au chapitre 5. Pour la spécification des objets, nous présentons au chapitre 6 un modèle de synchronisation d'objets concurrents dont la sémantique est définie, au chapitre 7, en terme de logique temporelle d'actions.

A chaque étape du processus de développement, le formalisme proposé permet de prouver des propriétés de correction. Au niveau groupe, les preuves utilisent les systèmes axiomatiques de la logique épistémique. Par exemple, on va chercher à établir que les niveaux de connaissance de groupe peuvent être atteints. On utilise pour cela des schémas de preuve par induction et par récurrence. Au niveau objet, les preuves conduites concernent essentiellement le non-interblocage des politiques de synchronisation ainsi que des conditions de sûreté ou d'équité. Finalement, au niveau méthode, on s'intéresse à des propriétés algorithmiques telles que des invariants de boucle.

Le processus de développement est donc essentiellement descendant. On part du plus général pour aboutir au plus spécifique. Cette démarche est différente de celle adoptée dans les systèmes multi-agents. En effet, ceux-ci envisagent une démarche ascendante dans laquelle le comportement de groupe émerge des buts locaux poursuivis par les agents. A contrario, notre processus nécessite des buts globaux clairement identifiés. Les buts locaux des objets ne peuvent être que déduits des buts globaux. Néanmoins, bien que nous n'ayons pas retenu la démarche de conception issue du domaine multi-agents, nous avons conservé une approche de spécification à base de connaissances. En effet, nous pensons que celles-ci offrent un pouvoir d'expression élevé qui permet d'abstraire de façon simple les comportements complexes mis en œuvre dans les applications distribuées.

4.1.1 Niveau groupe

C'est le niveau de spécification le plus élevé. Un groupe est composé par un ensemble d'objets distribués sur les différentes machines d'un réseau. Ce niveau donne une vision globale du comportement réparti. Il n'y a pas de notion de contrôle centralisé. Celui-ci est complètement décentralisé dans chacun des objets du groupe. Il n'y a pas non plus nécessairement de notion de localité. Ainsi, la composition du groupe n'est pas instanciée sur un des sites de l'environnement. Elle se définit par les voisinages. Chaque objet connaît un ensemble de voisins avec qui il interagit. Ceux-ci connaissent, à leur tour, un ensemble de voisins et le groupe peut se définir ainsi de proche en proche.

Il est important de noter que le niveau groupe est un point de vue abstrait. Bien qu'il n'y ait pas d'observateur global dédié, on suppose être capable d'appréhender, de façon instantanée et parfaite, l'évolution de tous les membres du groupe. La mise en œuvre de ce comportement global est assuré par un ensemble d'objets, concurrents, autonomes et éventuellement hétérogènes.

A ce niveau, les spécifications manipulent des connaissances distribuées. En fonction du problème posé et de la stratégie de résolution choisie, elles définissent des actions globales. Celles-ci impliquent tous les membres du groupe. Chaque membre exécute alors une version locale de l'action globale. Dans le chapitre 5, nous identifions un certain nombre d'actions globales types. Elles

couvrent des situations fréquement rencontrées en algorithmique répartie. Ce sont, par exemple, des schémas de phasage au sein du groupe. Nous définissons de manière plus précise au paragraphe 4.2 la notion de comportement de groupe. Ces comportements, ainsi que ceux que nous définissons aux niveaux objet et méthode, peuvent être dénotés par des automates états/transitions (Cf. figure 4.2). Au niveau groupe, chaque état est associé à un prédicat épistémique représentant un niveau de connaissance distribuée, et l'action globale peut être vue comme un transformateur de prédicat épistémique.

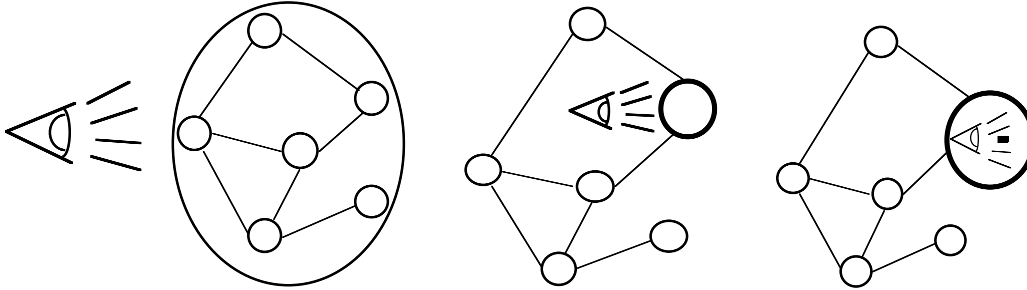


FIG. 4.1 – *Niveaux groupe, objet et méthode*

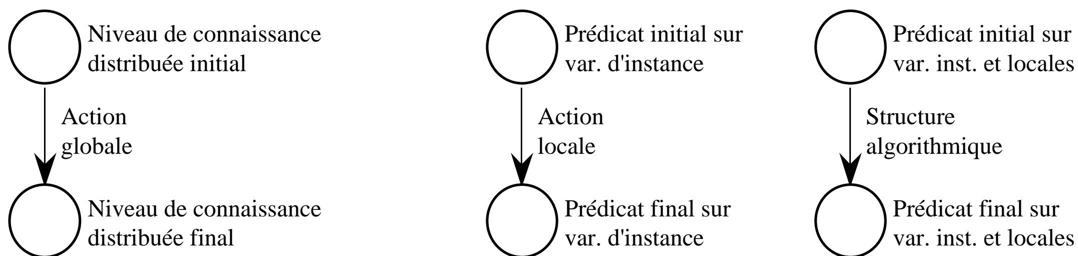


FIG. 4.2 – *Formalisme états/transitions pour les trois niveaux groupe, objet et méthode*

4.1.2 Niveau objet

Le niveau objet traduit le comportement d'un objet au sein du groupe. Ce niveau correspond au niveau de modélisation des agents dans un système multi-agents. Pour nous le comportement d'un agent est implanté à l'aide d'un objet. Nous employons le terme agent lorsque nous faisons référence aux concepts du domaine multi-agents comme par exemple la notion de connaissance. Nous employons le terme objet lorsque nous faisons référence plus spécifiquement à la façon dont ces comportements sont implantés. Dans le cas de groupes complètement hétérogènes, chaque objet peut être unique au sein du groupe. Néanmoins, dans de nombreux algorithmes répartis plusieurs objets exécutent le même comportement. Certains présentent même un comportement complètement symétrique : tous les objets du groupe sont à la fois clients et serveurs et appartiennent à une seule et même classe.

A ce niveau, on s'intéresse à un objet particulier au sein du groupe. Cet objet est implanté sur un site et met en œuvre l'algorithme réparti. La spécification manipule les données et les méthodes de l'objet. Elle définit la politique de synchronisation à mettre en place pour les méthodes de l'objet. La spécification de niveau objet fournit les mécanismes permettant d'implanter la spécification

définie au niveau groupe. Comme au niveau groupe, ces comportements sont dénotés par des diagrammes états/transitions (Cf. figure 4.2). Chaque état est associé à un prédicat sur les variables d'instance de l'objet. Chaque transition est une action locale qui correspond par exemple, à la prise en compte d'une méthode ou au retour de cette méthode.

4.1.3 Niveau méthode

Le comportement de niveau méthode est la dernière étape. Il définit les mécanismes internes à l'objet qui permettent d'obtenir son comportement. Ceux-ci sont regroupés au sein des méthodes de l'objet étudié. On adopte donc une approche locale aux méthodes.

A ce niveau, on manipule les variables locales aux méthodes. Les actions spécifiées permettent d'implanter le comportement défini au niveau objet, qui lui-même, implante le comportement de niveau groupe. Comme précédemment, les comportements de niveau méthode peuvent être dénotés par des diagrammes états/transitions (Cf. figure 4.2). Chaque état est associé à un prédicat sur les variables d'instance de l'objet et les variables locales de la méthode. Chaque transition est une action locale qui correspond à des structures algorithmiques de base telles que des boucles, des conditionnelles, des affectations ou des lectures de variables. Le résultat final est une spécification suffisamment précise de l'algorithme pouvant être codé dans un langage de programmation.

4.2 Comportement de groupe

La notion de comportement de groupe constitue l'un des apports majeurs du processus de développement présenté au paragraphe précédent. Il vise à décrire l'évolution globale d'un ensemble d'objets distribués. Nous en donnons une définition qui généralise la notion de comportement d'objet. Afin de positionner le discours, il semble utile de rappeler quelques caractéristiques importantes de l'approche objet :

- un objet est implanté sur un seul site.
- il encapsule une structure de données gérée dans la mémoire locale du site de résidence.
- il permet l'accès à ces données au moyen d'un ensemble de primitives.
- la définition de la sémantique des primitives d'accès forme la spécification externe de l'objet (c'est à dire les services fournis). Elle peut, selon Meyer, être vue en termes de conditions sur les résultats d'une exécution correcte (en somme, sur les pré et les post-conditions). Cette spécification peut être logique, algébrique ou à modalités temporelles selon les besoins ou le style de la description.
- l'implantation est définie par une classe et représente l'ensemble des mécanismes permettant de réaliser la sémantique externe. C'est en quelque sorte la partie protocole au sens des réseaux.

En résume, d'un point de vue algorithmique répartie, un objet encapsule une structure de données purement locale et offre une vue centralisée d'un service. Notre démarche consiste à donner d'un comportement de groupe une définition qui généralise la notion de comportement d'objet.

Celui-ci est alors un cas particulier qui présente un point de vue centralisé alors que le comportement de groupe représente un point de vue distribué. Nous reformulons donc les définitions précédentes de la façon suivante :

- un comportement de groupe est implanté sur plusieurs sites.
- il encapsule une structure de données gérée dans les mémoires des différents sites. Celle-ci est donc, dans le cas général, composée de différents objets ayant une relation sémantique entre eux et localisés sur différents sites.
- cette structure de données de groupe est accédée via des primitives d'accès qui constituent le comportement de groupe.
- la définition de la sémantique des primitives d'accès forme la spécification externe du comportement de groupe. Cette spécification du service peut être, selon les besoins, logique, algébrique, à modalités temporelles ou épistémiques.
- l'implantation d'un comportement de groupe est définie par l'analogue d'une classe qui est un protocole multipoints ou protocole coopératif. Il généralise la notion de protocole point à point

Un comportement de groupe présente donc une définition abstraite de ce qui va être implanté sous la forme d'un ensemble d'objets. Le service de groupe est réalisé par des services d'objets, tandis que le protocole est réalisé par des interactions entre objets. Par exemple, un comportement de groupe est :

- un protocole de communication point à point : c'est un comportement de groupe qui maintient une structure file d'attente répartie accédée en mode producteur consommateur distribué à l'aide de deux primitives *envoyer* et *recevoir*.
- un algorithme de routage : il maintient, idéalement, pour chaque site d'un réseau, un arbre couvrant enraciné en ce site et permettant d'atteindre tous les destinataires. Le service comprend essentiellement une primitive qui, à partir d'un destinataire et d'un site source, détermine le site voisin auquel doit être envoyé le message afin d'atteindre le destinataire.
- une mémoire virtuelle partagée répartie : elle maintient un ensemble de données partiellement ou totalement répliquées selon une certaine politique de consistance. Le service comprend deux primitives *lire* et *écrire*.

De même que les objets gèrent les données qu'ils encapsulent, les comportements de groupe manipulent des structures de données réparties au sein du groupe. Dans le paragraphe suivant nous présentons plus en détail cette notion.

4.3 Structure de données de groupe

Dans les langages où tout est objet comme par exemple Smalltalk, il n'est pas fait de différence entre les différents types d'objets. En particulier on ne distingue pas un objet primitif comme un entier, d'un objet construit dans le cadre d'une application. Nous adoptons un point de vue analogue en nous plaçant au niveau des comportements de groupe. Nous considérons que les

programmes de niveau groupe manipulent des données qui sont elles-mêmes des programmes de niveau groupe

Les structures de données de groupe peuvent être vues comme des ensembles de variables disséminées dans le système. Les composants de ces structures (ensembles, files, piles, enregistrements, arbres, anneaux, etc ...) sont distribués au sein du groupe. Elles peuvent être vues comme des variables d'instances de l'entité groupe. On peut distinguer deux grandes catégories de structures de groupe :

- les structures topologiques qui traduisent une organisation des relations entre objets au sein du groupe,

- les structures de stockage qui enregistrent des informations de type applicatif.

Ces deux catégories de structures sont manipulées, interrogées et mises à jour par les programmes à base de connaissance. Bien qu'il n'y ait fondamentalement pas de différence dans la façon dont elles sont gérées, leurs finalités, présentées dans la suite de ce paragraphe, sont légèrement différentes.

Les structures topologiques de groupe enregistrent des informations sur le statut des objets au sein du groupe et sur leurs liens. Le statut d'un objet définit son rôle dans l'organisation du groupe. Ce statut peut éventuellement évoluer au cours de l'exécution de l'algorithme. Selon les situations, on peut par exemple distinguer des objets coordinateurs des objets participants. On peut également les classer selon qu'ils sont racine, nœuds ou feuilles. De même, on peut leur attribuer le rôle de maître ou d'esclave. Les liens entre objets définissent les types de relations et les possibilités de communication qui existent entre les membres d'un groupe. Un objet peut, par exemple, être associé à un ensemble d'objets voisins ou à un ensemble d'objets fils. Ces relations définissent entre autre choses, les compétences qu'un objet peut attendre de ses pairs. Ainsi, un objet participant n'interagit pas de la même façon ou ne demande pas les mêmes services à un objet coordinateur ou à un autre objet participant. Les attributs d'état et de liens permettent de définir différentes organisations de groupe. Par exemple, on peut maintenir des topologies en anneau afin de faire circuler des jetons garantissant l'accès à une ressource partagée. On peut également envisager des topologies en arbre pour router des messages, effectuer des transactions imbriquées ou garantir un accès hiérarchique à une ressource. Ainsi, le paragraphe 5.4 introduit une structure arborescente pour le parcours d'un ensemble d'objets distribués. Chaque nœud possède un identificateur unique et un ensemble de nœuds fils. Un élément particulier, la racine, n'est le fils d'aucun autre nœud. Il existe une bijection entre l'ensemble des nœuds et celui des objets : chaque nœud est associé à un objet et un seul, et chaque objet ne représente qu'un nœud. Un objet ne connaît donc que l'identificateur et l'ensemble des fils du nœud auquel il est associé et aucun objet ne connaît la totalité de l'arbre. Ainsi, on obtient bien une distribution de la structure de données parmi l'ensemble des membres du groupe.

Les structures de stockage de groupe enregistrent les données gérées par les applications. Ce sont par exemple des données répliquées et/ou distribuées. Dans le cas de données répliquées, chaque objet du système possède une copie des données qu'il gère selon une politique de cohérence. On peut définir des politiques fortes (pour garantir qu'à tout instant toutes les copies répliquées sont identiques) ou des politiques faibles (avec par exemple des règles de mise à jour selon des dépendances causales [Cor97]). Les composants des structures de données distribuées sont répartis parmi différents objets. Chaque objet gère de façon autonome une partie des éléments de

l'ensemble mais c'est l'algorithme réparti qui est garant de la cohérence de l'ensemble. L'objet est responsable de la mise à jour des éléments sous sa responsabilité et sert les requêtes les concernant. Les structures de données répliquées s'instancient de la même façon sur chacun des objets du système. Ainsi, une donnée de groupe de type fichier ou de type enregistrement s'instancie en une variable de type fichier ou de type enregistrement sur chaque objet concerné par la réplication. L'instanciation des données applicatives distribuées est moins mécanique et requiert la plupart du temps, un certain nombre de choix de la part du concepteur. Par exemple, l'instanciation d'une liste d'enregistrements peut se faire de trois façons : horizontalement, verticalement ou de façon mixte. Dans le premier cas, chaque objet stocke un sous-ensemble d'enregistrements et tous les champs d'un enregistrement sont sauvegardés sur le même objet. Dans le cas d'une distribution verticale, les champs d'un même enregistrement sont répartis sur plusieurs objets. Chaque objet gère ainsi une fraction de tous les éléments de la liste. Finalement dans le cas d'une distribution mixte, d'une part chaque enregistrement est fractionné sur un ensemble d'objets et d'autre part, les éléments de la liste sont distribués sur un ensemble d'objets.

Ces structures de données réparties véhiculent des informations qui sont lues et mises à jour par les comportements de groupe. Contrairement aux univers centralisés, les objets du système n'ont qu'une vue partielle et pas nécessairement à jour, de la structure complète. Il est alors important de définir de façon précise le degré de visibilité des informations gérées par le groupe. Nous proposons, au paragraphe suivant, l'emploi d'un certain nombre d'opérateurs épistémiques qui permettent de traduire différents degrés dans la connaissance d'une information répartie.

4.4 Opérateurs de connaissance de groupe

Les opérateurs épistémiques que nous proposons d'employer pour la description d'une information distribuée, sont essentiellement ceux retenus par Fagin, Halpern, Moses et Vardi dans la notion de programme à base de connaissances de niveau agent (Cf. paragraphe 3.2). On retrouve donc les opérateurs de connaissance, de connaissance de tous et de connaissance commune. Nous justifions l'utilisation de trois opérateurs supplémentaires : ce sont les opérateurs de connaissance distribuée D (parfois désigné dans la littérature sous le terme d'opérateur I de connaissance implicite), de connaissance instanciée S et de croyance B. Ces opérateurs ne sont pas originaux puisqu'ils existent dans d'autres systèmes épistémiques (Cf. paragraphe 2.3). Nous pensons que leur utilisation offre un pouvoir d'expression élevé et qu'ils méritent d'être retenus dans une démarche de spécification. De plus, ils facilitent l'expression de l'élévation de la connaissance au sein d'un groupe. Nous utilisons ces opérateurs au paragraphe 4.5 dans l'écriture des programmes à base de connaissances de niveau groupe.

4.4.1 Opérateurs retenus

Nous rappelons brièvement dans ce paragraphe la définition des différents opérateurs épistémiques retenus. Leur sémantique a été donnée au paragraphe 2.3. Nous précisons leur utilisation dans le cadre de la spécification de comportements distribués.

Connaissance distribuée : l'opérateur de connaissance distribuée permet d'exprimer simplement les faits connus de manière implicite par les agents du système. Comme nous l'avons expliqué au paragraphe 2.3.2, toute connaissance dans un système distribué fermé ne peut provenir que de

faits disséminés parmi ses agents. Le but d'un algorithme réparti est alors le plus souvent d'instancier ces faits, c'est à dire de les calculer à partir d'autres faits pris comme données de départ. L'opérateur D permet de définir clairement les faits présents de manière latente à un instant donné, c'est à dire les faits distribués parmi les agents mais non encore inférés. Ainsi, la connaissance d'un fait φ est dite distribuée si aucun agent ne le manipule explicitement en tant que connaissance, mais si il existe un programme qui, exécuté par un ou plusieurs agents du groupe, permet à partir de données en entrée représentées par un fait ψ, d'inférer le fait φ.

Connaissance : c'est l'opérateur de base de la logique épistémique. Il fournit une spécification des faits qui peuvent être manipulés de façon sûre par un agent.

Connaissance instanciée : l'opérateur de connaissance instanciée désigne quant à lui, les faits connus par au moins un agent du groupe. Cet opérateur peut être vu comme une simple réécriture au niveau groupe de la connaissance d'un agent. Il permet d'exprimer clairement le fait qu'au moins un agent du groupe connaît le fait.

Connaissance de tous et **connaissance commune** : ce sont les niveaux suivants dans la hiérarchie des opérateurs épistémiques. Ils permettent de désigner respectivement les faits qui sont connus par tous les agents du système et les faits dont tous les agents savent que tous savent ... (un nombre infini de fois) que tous le connaissent.

Croyance : l'opérateur de croyance B est utilisé dans les situations où l'on n'a pas pu analyser l'ensemble des mondes possibles. Dans la plupart des cas, on souhaite néanmoins, dans le cadre d'une action à entreprendre, restreindre cet ensemble. Ceci est réalisé en définissant une croyance, c'est à dire un prédicat que l'on ne peut trouver dans l'ensemble des mondes possibles, mais qui peut être vrai dans un sous-ensemble. La figure 4.3 présente une situation qui vérifie les prédicats $B_i\varphi$ et $\neg K_i\varphi$. En effet, le fait φ n'est vrai que dans un sous-ensemble de mondes possibles (ceux situés dans la zone hachurée). Il n'est donc pas connu de façon certaine mais juste cru.

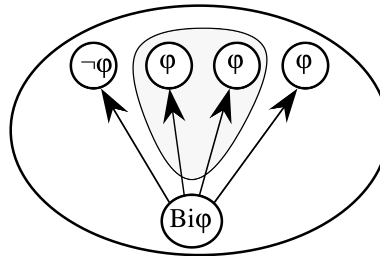


FIG. 4.3 – *Le fait φ est seulement vrai dans un sous-ensemble des mondes possibles*

L'exemple le plus classique est celui de l'émission d'un message. Deux situations sont possibles : sa transmission est correcte ou sa transmission est incorrecte. Dans le cadre d'un protocole donné, il est possible de croire la transmission correcte si cette hypothèse est intéressante dans le cadre considéré (par exemple si le taux d'erreur est considéré comme suffisamment faible sans être réellement nul). On élimine alors dans le domaine de la croyance la situation où le message n'est pas arrivé.

Un autre cas simple est celui d'une hypothèse de connexité. Développer un algorithme réparti dans un réseau quelconque suppose que l'ensemble des mondes possibles est l'ensemble des réseaux possibles, c'est à dire l'ensemble des graphes orientés finis. Certaines solutions ne marchent réelle-

ment que pour les réseaux connexes, c'est à dire les réseaux dans lesquels on peut atteindre tous les sites à partir de n'importe quel site. La vérification de cette propriété est longue et dans certains cas irréaliste. Lorsque la réponse est obtenue, la propriété peut être redevenue fausse même si elle a été détectée vraie. L'hypothèse la plus pertinente est de restreindre les mondes possibles dans le domaine des croyances en ne considérant que les réseaux connexes.

4.4.2 Notation

Dans la suite de ce mémoire, ces modalités épistémiques sont utilisées, la plupart du temps, non pas avec des prédicats, mais avec des variables. Ainsi, nous utilisons souvent des écritures comme $D_G(statut)$, plutôt que $K_i(fils(i,j))$. Si la seconde écriture ne pose pas de problèmes ($fils(i,j)$ est clairement un prédicat), la première peut prêter à confusion. Or, nous cherchons à exprimer que, quelle que soit sa valeur, la variable *statut* est une connaissance distribuée. L'abus de notation consiste donc à considérer le terme *statut* dans l'expression $D_G(statut)$, comme un paramètre et non comme un prédicat. Nous avons signalé, au paragraphe 3.3.1, que Fagin, Halpern, Moses et Vardi résolvent ce problème en proposant de considérer l'expression $D_G(statut)$ comme un raccourci pour $\bigvee_i D_G(statut = val_i)$. Les termes val_i représentent toutes les valeurs possibles pour la variable *statut*. Par exemple, dans le cas d'une variable booléenne *Bit*, la notation $K_i(Bit)$ représente l'expression $K_i(Bit = true) \vee K_i(Bit = false)$. Ainsi, l'ambiguïté est levée, et quelle que soit la valeur de la variable *Bit*, le prédicat $K_i(Bit)$ est vérifié.

On peut, néanmoins, poursuivre cette réflexion en proposant des interprétations plus informatiques aux modalités épistémiques. Par exemple, on peut considérer que, dire qu'une variable est connue et écrire $K_i(var)$, signifie que la variable existe, qu'elle a été instanciée (c'est à dire qu'un emplacement mémoire lui a été alloué) et qu'elle a été initialisée (c'est à dire qu'une valeur lui a été affectée). Dans une optique programmation objet, on peut alors définir une classe *var* implantant la variable et une méta-classe *MetaVar* possédant, en plus des constructeurs et destructeurs habituels, des méthodes *existe*, *est_instancié*, *est_initialisé* et *K*. La notion de connaissance acquiert ainsi un sens informatique et peut être utilisée dans un programme.

```
class MetaVar is
  method New;
  method Delete;
  method existe : boolean;
  method est_instancié : boolean;
  method est_initialisé : boolean;
  method K : boolean;
end MetaVar.
```

FIG. 4.4 – *Méta-classe de gestion de la variable var*

Néanmoins, cet aspect reste à l'état de proposition. Un travail important reste à effectuer pour définir l'interprétation informatique de l'ensemble des modalités retenues. Nous en restons donc à l'interprétation des expressions de type $D_G(statut)$ par une disjonction sur l'ensemble des valeurs possibles : $\bigvee_i D_G(statut = val_i)$.

4.4.3 Gradation des niveaux de connaissance

Les opérateurs de connaissances distribuée et instanciée introduits précédemment, associés aux opérateurs de connaissance commune et de connaissance de tous présentés au paragraphe 3.2, définissent des degrés progressifs de connaissance. On peut considérer que, peu ou prou, tout programme distribué qui termine fait évoluer un groupe d'objets d'un niveau de connaissance distribuée à un niveau de instancié puis éventuellement de tous et plus rarement commune.

Considérons, par exemple, le cas d'un protocole de validation à deux phases ayant pour but d'engager une transaction entre un coordinateur (appelé maître de la transaction) et un ou plusieurs participants (appelés esclaves). Dans un premier temps, le coordinateur effectue une phase de vote en envoyant les données de la transaction aux participants, en leur demandant s'ils peuvent ou non effectuer leur part de la transaction et en collectant les réponses. Dans une deuxième phase, si tous les participants peuvent effectuer leur part de la transaction, alors le coordinateur leur envoie un message d'engagement afin qu'ils enregistrent en mémoire stable les résultats de cette transaction. Sinon, il leur envoie un message d'annulation pour qu'ils reviennent à l'état initial et qu'ils annulent les effets de la transaction. Dans les deux cas (phase d'engagement ou phase d'annulation), les participants retournent au coordinateur un acquittement positif ou négatif décrivant le succès ou l'échec de l'opération demandée. Au niveau du groupe constitué par le coordinateur et les participants, ce protocole peut se représenter à l'aide des quatre états et des trois transitions du modèle de la figure 4.5.

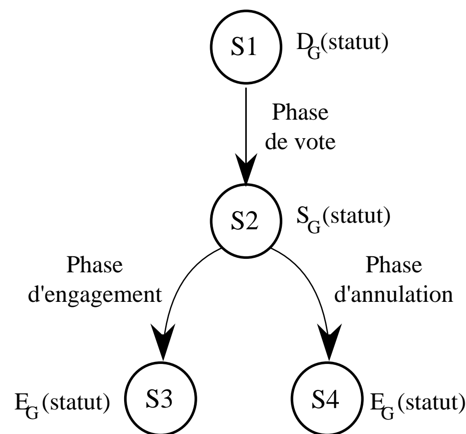


FIG. 4.5 – *Modèle d'un protocole de validation à deux phases*

Selon que la transaction est réalisable ou non, le groupe exécute, soit la séquence *Phase De vote* puis *Phase d'engagement*, soit la séquence *Phase de vote* puis *Phase d'annulation*. Dans l'état initial S_1, toutes les données nécessaires au déroulement de l'algorithme sont présentes de manière latente au sein du groupe. La connaissance du statut de la transaction est donc distribuée parmi le groupe. Le niveau de connaissance associé à cet état est donc noté $D_G(statut)$. A l'état intermédiaire S_2, le coordinateur a collecté les réponses de tous les participants. Il sait donc si la transaction est validable ou annulable. De plus, il est le seul à le savoir puisque les participants n'ont pas encore été informés des réponses de leurs pairs. Cet état est donc un état de connaissance instanciée (il existe un membre du groupe, le coordinateur, qui connaît le résultat de la transaction). Son niveau de connaissance est noté $S_G(statut)$. Les états finals S_3 et S_4 sont des états de connaissance de tous. En effet, tous les membres du groupe (coordinateur et participants) savent si la transaction a

été engagée ou annulée. Son niveau de connaissance est noté $E_G(statut)$. On passe donc bien d'un état de connaissance distribuée à un état de connaissance instanciée puis à un état de connaissance de tous. De façon informelle, cette élévation peut se traduire par une implication logique entre deux formules épistémiques. Ainsi, l'exemple proposé vérifie bien les propriétés : $E_G(statut) \Rightarrow S_G(statut)$ et $S_G(statut) \Rightarrow D_G(statut)$.

L'emploi d'un opérateur de connaissance distribuée suppose que l'on modélise un système fermé. En effet, rappelons que les groupes pour lesquels nous effectuons des modélisations de comportement sont vus comme des systèmes fermés. Cela signifie par exemple que chaque fois que l'un des membres du groupe a besoin de prendre une décision, toutes les données nécessaires sont présentes à l'intérieur du système. De ce fait, la prise de décision ne requiert l'intervention d'aucun agent extérieur. Cette hypothèse est, à première vue, plutôt réductrice puisqu'elle exclut l'environnement physique et l'intervention humaine du processus de modélisation. Néanmoins, il semble important de distinguer clairement dans la vie d'un système, les périodes pendant lesquelles le système est ouvert (et où une intervention humaine est nécessaire par exemple) de celles pendant lesquelles il est fermé. En effet, les raisonnements à partir de la notion de connaissance distribuée sont plus faciles lorsqu'un système est fermé. Les agents extérieurs au système ne sont pas, par définition, connus et on ne peut donc pas modéliser la connaissance qu'ils manipulent. Néanmoins, si le système est ouvert et si cette notion doit tout de même être employée, on incorpore l'environnement physique et/ou humain dans la modélisation afin de clore artificiellement le système.

4.4.4 Niveaux de connaissance et structures de données réparties

Cette approche permet de caractériser une donnée par son niveau dans la hiérarchie D, S, E, E^2, ... , E^k, ... , C. On peut alors construire un langage dans lequel toute structure de données est associée à un niveau de connaissance dans la hiérarchie précédente. On peut ainsi imaginer de déclarer des objets distribués caractérisés par un niveau de connaissance D, des objets locaux caractérisés par le niveau S, des objets partagés caractérisés par le niveau E, atomiques par E^2, ... , communs par C.

Malheureusement, le fait de déclarer un objet local ou global renseigne très insuffisamment sur sa sémantique, sur ce qui peut être réalisé sur les données encapsulées, et encore moins sur la consistance d'un ensemble d'objets manipulés simultanément. Par ailleurs, le projet de nombreux algorithmes distribués étant de traiter de l'élévation de connaissance dans la hiérarchie, l'objectif relatif à un objet est souvent de modifier son niveau de connaissance. Ainsi, une variable initialement déclarée comme distribuée peut acquérir, après l'exécution de l'algorithme, un statut instancié. De ce fait, le niveau de connaissance n'est pas tant un attribut statique des structures de données, qu'un formalisme permettant de dénoter un comportement de groupe.

Nous avons vu jusqu'à présent qu'une application distribuée peut être vue comme un ensemble d'objets distribués coopérant afin d'atteindre un but global. Ce groupe exécute un comportement et manipule une structure de données réparties. L'exécution du comportement peut être traduit en terme d'élévation du niveau de connaissances distribuées. Dans le paragraphe suivant, nous proposons une notation permettant l'expression des comportements de niveau groupe.

4.5 Programmes à base de connaissances de niveau groupe

Nous avons exposé au paragraphe 3.3 la notion de programme à base de connaissances pour chaque agent d'un système distribué. Cette notion, proposée par Fagin, Halpern, Moses et Vardi, permet de décrire le comportement d'un agent au sein d'un système multi-agents. Plutôt que d'envisager le comportement de ces agents indépendamment les uns des autres, il peut être intéressant de définir l'évolution du système, non plus localement, mais dans son ensemble. On aboutit alors à la notion de comportement de groupe et de programme à base de connaissances pour le système distribué. Dans ce paragraphe, nous définissons donc la notion de programme à base de connaissances de niveau groupe.

4.5.1 Définition

Un programme à base de connaissances de niveau groupe fournit une description générale de l'évolution du système telle qu'elle pourrait être appréhendée par un observateur global ayant une connaissance parfaite et instantanée de l'état de chacun des objets du système et non soumis aux contraintes du réseau telles que les délais de communication. Elle propose une vue d'ensemble du système et unifie dans un même formalisme la définition du protocole de communication et des comportements locaux. Alors que l'interaction est au centre de toute application distribuée, la notion de comportement de groupe masque paradoxalement cet aspect. Elle ne retrouve son statut privilégié que lorsque le comportement de groupe est raffiné au niveau des comportements locaux. Les comportements de groupe introduisent donc un niveau d'abstraction supplémentaire dans la définition des applications distribuées. Les programmes à base de connaissances pour les agents abstraient en terme de connaissances les états locaux des agents et leur perception des états de leurs pairs. Les programmes à base de connaissances de niveau groupe abstraient, quant à eux, l'évolution globale des groupes d'objets distribués. Ils permettent dans un premier niveau de spécification de faire abstraction de la répartition et des problèmes qui y sont liés.

Un programme à base de connaissance de niveau groupe décrit donc de façon globale l'évolution d'un ensemble G de n objets. Il manipule des structures de données réparties. Il effectue des tests de connaissance sur ces structures et spécifie des actions globales à entreprendre. Ces actions sont construites à l'aide de structures de contrôle présentées au chapitre 5. Les programmes à base de connaissance de niveau groupe doivent pouvoir être raffinés. Le processus de raffinement peut suivre deux axes. Tout d'abord, un programme de niveau groupe peut être raffiné en un autre programme de niveau groupe comportant plus de détails. Dans un second temps, après un nombre d'étapes de ce type jugé suffisant par le concepteur, le programme de niveau groupe doit être instancié au niveau de chaque objet du système. Ainsi, on raffine un programme de niveau groupe en n programmes de niveau objet. Selon les types d'objets envisagés et selon leurs rôles respectifs dans la résolution du problème distribué, cette traduction n'est pas forcément identique pour les n objets. En effet, la plupart des protocoles réseau introduisent une dissymétrie dans les comportements des objets. Par exemple, les protocoles transactionnels différencient généralement le comportement des entités coordinatrices du comportement des participants. Le programme global qui consiste à transmettre les éléments de la transaction, puis à l'engager ou à l'abandonner, doit donc être raffiné en deux types de programmes locaux différents.

4.5.2 Notation

On peut envisager, comme Fagin, Halpern, Moses et Vardi, un programme à base de connaissances comme un ensemble de tests standards et de tests de connaissances évalués continuellement. Cette notation présentée au paragraphe 3.3.1 est de type système expert et utilise deux types de constructions algorithmiques : des conditionnelles **if then** imbriquées dans une boucle infinie **while** true **do**. Sans perte de généralités, cette notation peut être transformée à l'aide de séquences, de tests et de boucles en une forme plus impérative. La figure 4.6 présente la syntaxe textuelle d'un tel programme. Celui-ci comprend des variables de groupe, des hypothèses sur l'environnement et une séquence d'actions. Lorsque le programme comprend plusieurs parties, chaque comportement est défini au sein d'une méthode de groupe. On utilise alors le mot clé **method** suivi d'un identificateur. Cet en-tête est omis lorsque le programme comporte un seul comportement et qu'il n'y a pas d'ambiguïté dans la notation.

- le délimiteur **group vars** permet de débuter la déclarations des structures de données réparties représentées par les termes gv_j. Ces structures sont typées. Ce sont des types topologiques tels que les arbres ou les anneaux et des types applicatifs tels que les scalaires répliqués ou les types construits répliqués. Un type particulier **object** est réservé pour désigner les membres du groupe.

- la section **environment** définit à l'aide des prédicats h_j les hypothèses de fonctionnement nécessaires au bon déroulement de l'algorithme. Elle permet d'exprimer en terme de connaissances un ensemble de contraintes sur les variables de groupe. Ce sont, à la fois des invariants qui doivent être conservés par les opérations du programme, et des hypothèses de bon fonctionnement qui ne peuvent être modifiées par des éléments extérieurs au groupe. Par exemple, un grand nombre d'algorithmes répartis ne fonctionne que si la topologie du réseau reste fixe du début à la fin. En instanciant un objet par site, cela revient à spécifier que tous les objets connaissent les mêmes objets pendant la durée de l'algorithme.

- les instructions du programme de niveau groupe sont comprises entre les délimiteurs **begin** et **end**. Elles sont composées à l'aide des structures de contrôle présentées au chapitre 5. Par exemple, nous proposons comme structure : la phase notée par un point-virgule, la conditionnelle notée par les mots clés **if then else** et l'itération notée par les mots clés **while do enddo**. Ces structures définissent des comportements globaux pour le groupe d'objets. Ce sont des structures génériques qui utilisent des actions globales notées ag_j, des tests standards t_j, des tests de connaissance k_j et des tests de connaissance de groupe kg_j. De façon plus précise, les tests de connaissance de groupe kg_j sont des combinaisons booléennes de formules de type $\mathcal{X}_G\varphi$ où $\mathcal{X}$ est un des opérateurs modaux de groupe D, S, E ou C et φ est une formule quelconque. Les pas d'exécution des actions globales peuvent être dénotés à l'aide de prédicats épistémiques. Ceux-ci définissent les états globaux observés lors de l'exécution. Ainsi, on associe la forme impérative d'un programme de niveau groupe à un automate états/transitions. Chaque transition correspond à une action globale et les prédicats épistémiques des états traduisent l'élévation du niveau de connaissance lors de l'exécution.

4.5.3 Exemple

La figure 4.7 reprend le programme de transmission de bit du paragraphe 3.3.2 et en donne une spécification qui illustre la notion de programme à base de connaissances de niveau groupe. Le

group vars
  $gv_1, gv_2, \ldots$
vars
  $v_1, v_2, \ldots$
environment
  $h_1 \wedge h_2 \wedge \ldots$
begin
  a_1 ; ag_1 ; . . .
  if $t_2 \wedge k_2 \wedge kg_2$
  then . . . **else** . . . **endif**
  while . . . **do** . . . **enddo**
  . . .
end

FIG. 4.6 – *Version impérative d'un programme à base de connaissances de niveau groupe*

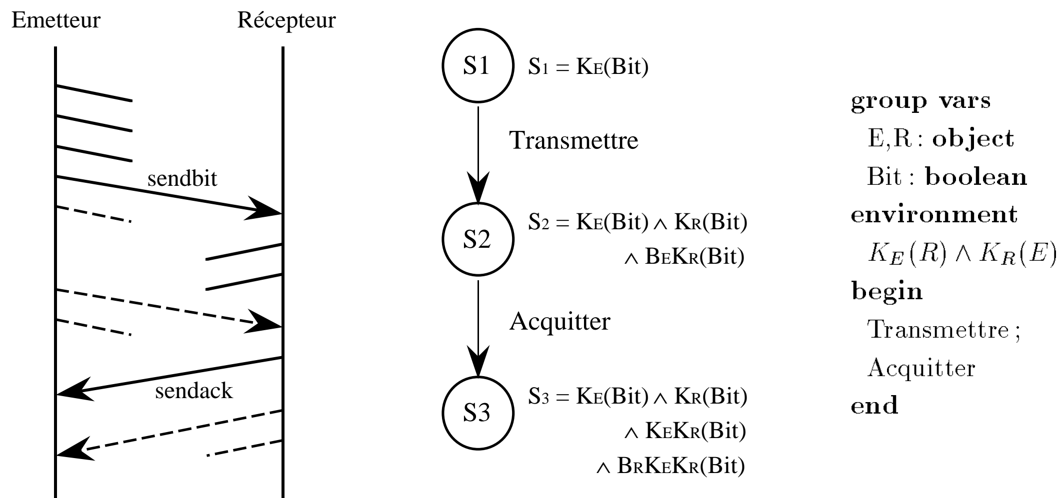


FIG. 4.7 – *Programme de niveau groupe TB_G de transmission de bit*

système comprend deux éléments : l'objet émetteur E et l'objet récepteur R. La variable *Bit* est une structure de données réparties de type booléen. Les hypothèses d'environnement spécifient que pour que ce programme fonctionne correctement, il faut que pendant toute la durée de son exécution l'émetteur connaisse le récepteur et réciproquement. Deux phases peuvent être distinguées dans ce comportement global : la phase de transmission des données et la phase d'acquittement. La partie gauche de la figure 4.7 présente le diagramme d'échange de messages de ce programme. Il correspond à une exécution possible du protocole parmi toutes celles envisageables. Deux messages peuvent être échangés : *sendbit* entre l'émetteur et le récepteur et *sendack* entre le récepteur et l'émetteur. Les flèches représentent les messages correctement délivrés. Les traits interrompus correspondent aux pertes de messages. Les traits pointillés symbolisent les émissions redondantes de messages.

La partie droite de la figure 4.7 présente sous la forme d'un automate états/transitions et sous une forme textuelle le comportement de niveau groupe de ce programme. Chaque transition correspond à l'exécution d'une action globale et chaque état est associé à un prédicat épistémique. Il se résume à l'enchaînement séquentiel des phases de transmission et d'acquittement. La phase de transmission est exécutée tant que la donnée n'a pas été acquittée. Ainsi, on constate dans le diagramme d'échange de messages que l'émetteur envoie des messages *sendbit* tant qu'il n'a pas reçu un acquittement et cela, même si un message *sendbit* a correctement été délivré au récepteur. En effet, le seul moyen qu'a l'émetteur de prendre connaissance de ce fait lui est fourni par la réception d'un acquittement. De même, le récepteur exécute la phase d'acquittement en envoyant des messages *sendack* de façon continue même si un de ces messages est parvenu correctement à l'émetteur.

Les phases de transmission et d'acquittement peuvent être dénotées par les états S_1, S_2 et S_3. Ceux-ci correspondent respectivement aux situations où aucun message n'a été transmis, où la donnée a été transmise et où l'acquittement a été transmis. Chacun d'eux peut être associé à un niveau de connaissance de groupe. Les prédicats épistémiques définissant ces niveaux utilisent la donnée répartie *Bit*. Dans le premier état, seul l'émetteur connaît cette donnée et le prédicat correspondant est $K_E(Bit)$. Dans l'état S_2, après l'exécution de la phase *Transmettre*, le récepteur acquiert la valeur de cette donnée. Il vérifie donc $K_R(Bit)$. L'émetteur est toujours en possession de cette donnée, mais il ne sait pas de façon sûre que le récepteur l'a acquise. Il vérifie donc le prédicat $K_E(Bit) \wedge B_E K_R(Bit)$. Finalement, l'état S_2 est associé au prédicat $K_E(Bit) \wedge K_R(Bit) \wedge B_E K_R(Bit)$. Dans l'état S_3, l'acquittement permet à l'émetteur de savoir que le récepteur connaît la valeur de la donnée. Le récepteur croit, quant à lui, que l'acquittement a bien été reçu par l'émetteur. Comme l'émetteur et le récepteur connaissent toujours la valeur de la donnée, on définit l'état S_3 par le prédicat $K_E(Bit) \wedge K_R(Bit) \wedge K_E K_R(Bit) \wedge B_R K_E K_R(Bit)$. Ces formules nous permettent d'une part d'axiomatiser le programme TB_G en annotant ses actions par des pré et des post-conditions et d'autre part, de définir l'élévation du niveau de connaissance en vue d'une preuve de bon fonctionnement.

Axiomatisation

Une technique d'axiomatisation de type Hoare permet de définir des mécanismes de preuve de programmes. Chaque action est associée à deux prédicats. Le premier, appelé pré-condition, définit un ensemble d'hypothèses tandis que le second, appelé post-condition, définit des conclusions. Une preuve de correction consiste alors à établir que, sous l'effet de l'exécution de l'action, les hypo-

thèses permettent de déduire les conclusions. Cette technique, issue de l'algorithmique centralisée, peut être appliquée aux programmes de niveau groupe. Les prédicats de connaissance associés aux actions globales remplissent le rôle des pré et des post-conditions. Dans le cas du programme TB_G, le schéma de preuve comprend la démonstration des théorèmes suivants. D'une part, il faut prouver que si le prédicat S_1 est vérifié et si l'action globale *Transmettre* est exécutée alors on peut déduire le prédicat S_2. D'autre part, il faut établir que sous les hypothèses du prédicat S_2 et de l'exécution de l'action globale *Acquitter*, alors le prédicat S_3 est vérifié. Ces prédicats sont présentés en général entre accolades dans le corps du programme de la façon suivante :

$TB_G : \{S_1\}$ Transmettre $\{S_2\}$ Acquitter $\{S_3\}$

Elévation du niveau de connaissance

Comme nous l'avons vu au paragraphe 4.4.3, l'élévation de la connaissance dans un groupe d'objets se traduit par une implication logique entre deux prédicats épistémiques. Ainsi, dans cet exemple, on vérifie aisément que $S_3 \Rightarrow S_2$ et que $S_2 \Rightarrow S_1$. En effet, S_1 vaut $K_E(Bit)$ qui est un des termes de la conjonction du prédicat S_2. De même, ce prédicat est égal à $K_E(Bit) \wedge K_R(Bit) \wedge B_E K_R(Bit)$. Les deux premiers termes font partie de l'écriture de S_3. Pour le dernier terme, on vérifie aisément que $K_E K_R(Bit) \Rightarrow B_E K_R(Bit)$. En constatant que $K_E(Bit) = S_G(bit)$ et que $K_E(Bit) \wedge K_R(Bit) = E_G(bit)$, on vérifie que l'on passe d'un niveau de connaissance instanciée sur l'émetteur à un niveau de connaissance de tous puis à un niveau de connaissance de tous dans lequel l'émetteur sait en plus que le récepteur connaît la donnée.

4.5.4 Raffinement

Nous désignons sous le terme de raffinement le processus qui consiste à dériver un programme père en un ou plusieurs programmes fils. De cette façon les programmes raffinés introduisent des détails supplémentaires dans le modèle du programme père. Ils peuvent donc être perçus comme des implantations de celui-ci. Deux hypothèses fondamentales sont à la base d'un tel mécanisme :

- le processus de raffinement doit réduire l'espace des solutions du problème,
- les modèles raffinés ne doivent pas contredire le modèle du programme père.

En d'autres termes, ces conditions signifient que l'ensemble des implantations correctes du programme raffiné doit être un sous-ensemble de l'ensemble des implantations correctes du programme père. Deux types de raffinements sont à mettre en place dans notre approche. Tout d'abord un programme de niveau groupe peut être raffiné en un autre programme de niveau groupe. La raison d'un tel choix par le concepteur de l'application est le plus souvent d'introduire plus de détails dans la spécification. Dans un deuxième temps un programme de niveau groupe peut être raffiné en un ou plusieurs programmes de niveau objet afin de fournir une spécification du comportement des objets participants à la réalisation de l'application.

Nous adoptons pour les programmes de niveau objet le même style de notation impérative que pour les programmes de niveau groupe. Nous utilisons à la fois une forme graphique (Cf. par exemple figure 4.9) avec des automates états/transitions et une forme textuelle (Cf. par exemple figure 4.10). La première traduit de façon visuelle l'avancement de l'exécution, l'élévation de la

connaissance et par la même occasion les pré et les post-conditions des structures, tandis que la seconde fournit le détail des structures algorithmiques utilisées.

La figure 4.8 présente la structure type d'un programme à base de connaissance de niveau objet dans un style d'écriture impératif. Les termes v_j sont des variables locales gérées par les objets. Les instructions du programme de niveau groupe sont comprises entre les délimiteurs **begin** et **end**. Elles sont composées à l'aide des structures algorithmiques habituelles : la séquence symbolisée par un point-virgule, les conditionnelles et les itérations. Ces structures utilisent des actions locales notées a_j, des tests standards t_j et des tests de connaissance k_j. Comme pour les programmes de niveau groupe, les pas d'exécution de ces structures peuvent être dénotés à l'aide de prédicats. Ceux-ci définissent les états locaux observés au sein du groupe lors de l'exécution. On associe ainsi la forme impérative d'un programme de niveau groupe à un automate états/transitions.

vars
  $v_1, v_2, \ldots$
begin
  a_1 ; ...
  if $t_2 \wedge k_2$ **then** ... **else** ... **endif**
  while ... **do** ... **enddo**
  ...
end

FIG. 4.8 – *Version impérative d'un programme à base de connaissance de niveau objet*

Les programmes de niveau groupe doivent être raffinés à un niveau local en programmes à base de connaissances pour chacun des objets. Trois types de transpositions sont à mettre en œuvre :

- les actions de niveau groupe doivent être traduites en actions locales et en interactions (afin de prendre en compte effectivement la répartition),
- les structures de données réparties doivent être associées à des variables locales,
- les états globaux définis dans le programme de niveau groupe doivent être mis en correspondance avec les états locaux des objets.

En suivant ces trois points, le raffinement du programme de niveau groupe TB_G en programmes de niveau objet doit donc comprendre la traduction des actions globales *Transmettre* et *Acquitter*, l'instanciation de la structure de donnée *Bit* et l'identification des états de groupe S_1, S_2 et S_3 au niveau de l'émetteur et du récepteur. L'action globale *Transmettre* consiste pour l'émetteur, à exécuter l'action locale *sendbit* et pour le récepteur, à attendre la réception de ce message. De même, l'action globale *Acquitter* correspond à l'exécution simultanée d'un envoi d'acquittement *sendack* par le récepteur et d'une attente de ce message par l'émetteur. Le programme TB_G utilise la donnée répartie *Bit*. Cette variable de groupe de type booléen modélise l'information échangée lors de la phase de transmission. C'est une donnée répliquée instanciée à la fois sur l'objet émetteur et sur l'objet récepteur. Ces deux instanciations se font sous la forme des variables binaires bit_e et bit_r.

La partie gauche de la figure 4.9 présente, sous la forme d'un automate états/transitions, le programme TB_E, raffinement pour l'émetteur du programme de groupe TB_G. Chaque transition est associée à une action locale. Ainsi, l'émetteur envoie la donnée et attend l'acquittement. La

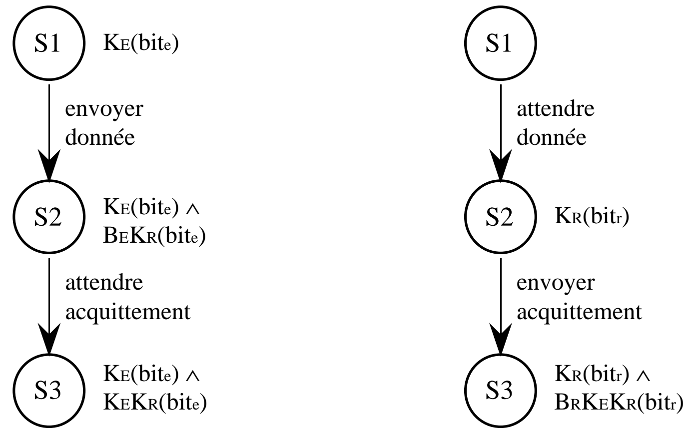


FIG. 4.9 – *Automates des programmes TB_E et TB_R pour les objets émetteur et récepteur*

partie droite de la figure présente le programme TB_R, raffinement pour le récepteur du programme de groupe TB_G. Ce programme, exécuté concurremment à TB_E, attend le message de données puis envoie l'acquittement. Les états S_1, S_2 et S_3 du programme de niveau groupe ont ainsi été instanciés au niveau objet pour l'émetteur et pour le récepteur. Ils correspondent chacun à des niveaux de connaissances locales différents. Ils s'obtiennent en transformant dans les prédicats de niveau groupe les données globales en données locales et en ne conservant, pour chaque objet, que les informations qui le concernent. Ainsi, dans l'état S_1, l'émetteur connaît la donnée, donc $K_E(bit_e)$ est vérifiée, alors que le récepteur n'a encore acquis aucune connaissance. Dans l'état S_2, l'émetteur a envoyé la donnée mais n'a pas encore reçu l'acquittement. Il croit donc que le récepteur l'a bien reçue. Son niveau de connaissance par rapport à la variable bit_e correspond donc à $K_E(bit_e) \wedge B_E K_R(bit_e)$. Le récepteur a, quant à lui, acquis la donnée et son niveau de connaissance passe à $K_R(bit_r)$. Finalement, dans l'état S_3 l'émetteur connaît toujours la donnée et il sait en plus que le récepteur la connaît. Son niveau de connaissance est donc $K_E(bit_e) \wedge K_E K_R(bit_e)$. Le récepteur connaît toujours la valeur de la variable bit_r. De plus, il croit que l'émetteur a bien reçu l'acquittement. Son prédicat vaut donc $K_R(bit_r) \wedge B_R K_E K_R(bit_r)$.

Cet exemple montre que les états globaux d'un programme de niveau groupe sont perçus à des degrés divers par les objets du système. Chaque objet développe sa propre perception des états globaux et cette perception est éventuellement différente de celle de ses pairs. Ce décalage est inévitable dans le modèle de système réparti retenu. En effet, l'asynchronisme des communications et les délais de transmission non bornés font que seul un observateur omniscient et parfait pourrait capturer un état global exact à un instant donné. Les objets du système ne peuvent qu'appréhender cet état global avec plus ou moins de retard et de précision.

La figure 4.10 présente la syntaxe textuelle des programmes TB_E et TB_R. Dans le premier, la phase d'envoi de la donnée entre les états S_1 et S_2 est réalisée par une boucle **while**. Celle-ci envoie la donnée au récepteur à l'aide de la primitive *sendbit* tant que le récepteur ne l'a pas acquittée c'est à dire tant que $\neg K_E K_R(bit_e)$. En ce qui concerne le programme du récepteur TB_R, la phase d'attente de la donnée correspond à une boucle vide exécutée tant que le message *sendbit* n'a pas été reçu, c'est à dire tant que $\neg K_R(bit_r)$. La phase d'envoi de l'acquittement est associée à la primitive *sendack* insérée au sein d'une boucle exécuté tant que le récepteur ne croit pas que l'émetteur a reçu l'acquittement c'est à dire tant que $\neg B_R K_E K_R(bit_r)$.

vars	**vars**
bit_e : **boolean**	bit_r : **boolean**
begin	**begin**
$\{K_E(bit_e)\}$	$\{\}$
while $\neg K_E K_R(bit_e)$ **do**	**while** $\neg K_R(bit_r)$ **do**
sendbit	**enddo**
$\{K_E(bit_e) \wedge B_E K_R(bit_e)\}$	$\{K_R(bit_r)\}$
enddo	**while** $\neg B_R K_E K_R(bit_r)$ **do**
	sendack
	enddo
$\{K_E(bit_e) \wedge K_E K_R(bit_e)\}$	$\{K_R(bit_r) \wedge B_R K_E K_R(bit_r)\}$
end	**end**

FIG. 4.10 – *Programmes TB_E et TB_R pour les objets émetteur et récepteur*

4.5.5 Conclusion sur les programmes de niveau groupe

Dans ce paragraphe, nous avons présenté la notion de programme à base de connaissances de niveau groupe. Ce concept étend la notion de programme de niveau agent introduite par Fagin, Halpern, Moses et Vardi. Plutôt que de spécifier les comportements locaux des agents indépendemment les uns des autres, nous adoptons un point de vue global et nous décrivons en terme de connaissances l'évolution du groupe d'objets. Un programme à base de connaissances de niveau groupe est donc vu comme une entité qui manipule, interroge et met à jour des structures de données réparties. Ces structures présentées au paragraphe 4.3, sont des types de données de groupe véhiculant des informations qui sont fragmentées et/ou répliquées parmi l'ensemble des objets du système. Le paragraphe 4.5.2 présente alors une notation textuelle pour ces programmes. La notation textuelle met en jeu des actions locales et globales, des tests de connaissance locaux et globaux, et utilise différentes structures de contrôle de niveau groupe. Ces structures sont présentées plus en détail dans le chapitre suivant. Elles étendent à un niveau distribué les constructions de base de l'algorithmique centralisée. Ce sont des briques de base qui peuvent être composées, assemblées et raffinées afin de mettre en place une application distribuée. Ces programmes peuvent être dénotés par des automates états/transitions. Chaque état est associé à un prédicat épistémique. Cela permet de définir l'élévation du niveau de connaissance et l'axiomatisation du programme de niveau groupe. Le paragraphe 4.5.3 reprend le protocole de transmission de données sur un réseau non fiable du paragraphe 3.3.2 et le spécifie à un niveau groupe. Le paragraphe 4.5.4 propose un raffinement de ce programme pour les objets émetteur et récepteur.

4.6 Conclusion sur la conception de niveau groupe

La conception de comportement distribués est une tâche difficile pour laquelle peu d'outils et de méthodologies existent. Les méthodes de conception orientées objet existantes telles que

Booch, OMT ou UML, ne placent pas vraiment la distribution au centre de leurs préoccupations. Elles se contentent d'en donner une vision statique en proposant par exemple, une répartition des composants sur les différents nœuds physiques du système. Les interactions sont modélisées à l'aide de diagrammes d'échange de messages, mais leur enchaînement ou leur conséquence restent peu étudiées. Les notions algorithmiques réparties telles que l'état global cohérent d'un système ou le but global poursuivi par un ensemble d'objets ne sont pas abordées par ces approches. Dans ce chapitre, nous avons voulu apporter une réponse à ce problème. Nous avons donc proposé l'utilisation d'un processus de développement pour les applications distribuées, d'un ensemble d'opérateurs épistémiques de connaissance et de programmes à base de connaissances de niveau groupe.

Le processus de développement présenté dans ce chapitre est issu des travaux réalisés au sein de notre équipe par de Bonnet [Bon94, BDFS97], au cours de son mémoire d'ingénieur, et enrichis par cette thèse. Trois niveaux méthodologiques pour la conception d'applications distribuées sont suggérés : le niveau groupe, le niveau objet et le niveau méthode. Le niveau groupe prend le point de vue d'un observateur global, parfait et non soumis aux contraintes du réseau telles que les délais de communication. Un groupe est un ensemble d'objets autonomes distribués sur les sites d'un réseau. Au niveau groupe, le concepteur spécifie les buts globaux de l'application distribuée et la politique d'échanges de connaissances permettant d'atteindre ces buts. Le niveau objet prend le point de vue d'un objet au sein du groupe et fournit la politique d'enchaînement des actions locales permettant d'implanter le comportement défini au niveau groupe. Finalement, le niveau méthode prend le point de vue d'une méthode au sein d'un objet et spécifie les structures algorithmiques permettant d'implanter le comportement de niveau objet. Le processus suggéré est donc essentiellement descendant et procède par raffinements successifs. La principale originalité de cette approche est la notion de comportement de groupe. Dans notre approche, ce type de comportement peut être vu comme l'extension, à un niveau distribué, du comportement d'un objet. Ainsi, un comportement de groupe est implanté sur plusieurs sites. Il encapsule une structure de données répartie dans les mémoires des différents sites. Cette structure est accédée par des primitives dont la sémantique fournit la spécification externe du comportement de groupe. L'implantation de cette spécification est décrite en terme de connaissances manipulées et échangées. Elle manipule des structures de données réparties. Celles-ci sont des ensembles de variables disséminées entre les membres du groupe présentant des liens sémantiques entre elles. Ce sont, par exemple, des arbres couvrants de réseau ou des anneaux représentant la topologie physique du réseau. Les comportements de groupe entreprennent alors des actions globales qui permettent d'atteindre différents degrés dans la connaissance de ces structures. Nous avons suggéré d'exprimer ces degrés à l'aide de six opérateurs issus de la logique épistémique. Finalement, nous avons proposé de noter ces comportements à l'aide de la notion de programme à base de connaissances de niveau groupe. Celle-ci étend la notion de programme à base de connaissances de niveau agent introduite par Fagin, Halpern, Moses et Vardi.

Nous allons maintenant nous intéresser aux actions qui sont, au niveau groupe, entreprises par ces comportements. En particulier, nous proposons, dans le chapitre suivant, quatre structures de contrôle qui sont, à notre avis, couramment employées en algorithmique répartie. Ces structures peuvent être vues comme l'extension, à un niveau réparti, des structures de base que sont la séquence, les conditionnelles de type *case* ou *if*, les boucles de type *while* et le parcours récursif.

Chapitre 5

Gabarits de conception de niveau groupe

Quel que soit le domaine considéré, la mise en place des aspects comportementaux d'un programme passe par la définition de structures de base. Ces structures sont en général choisies pour leur caractère universel et générique. Par exemple, en algorithmique séquentielle, des structures telles que la séquence, l'affectation, la conditionnelle ou l'itération ont été définies. L'algorithmique parallèle et concurrente a introduit des constructions spécifiques telles que les instructions *parbegin/parend* (pour exécuter en parallèle plusieurs blocs d'opérations), *fork* (pour créer et démarrer l'exécution d'une nouvelle activité) ou *join* (pour effectuer un rendez-vous d'activités). Dans le domaine de la conception des applications orientées objet, la définition de gabarits de conception (*design patterns* en anglais) suit une optique identique. Basée sur les travaux architecturaux de Alexander [AIS+77], cette démarche consiste à identifier certains schémas de programmation qui apparaissent de façon récurrente dans de nombreuses applications et qui sont de ce fait fréquemment employés par les développeurs. Gamma, Helm, Johnson et Vlissides dans [GHJV95] ont ainsi identifié une vingtaine de gabarits, répartis en trois catégories : les gabarits de création, les gabarits structurels et les gabarits comportementaux. Les premiers concernent les processus de création des objets. Les seconds s'intéressent à la façon dont les classes et les objets sont organisés et composés. Finalement, les gabarits comportementaux définissent la façon dont les classes et les objets interagissent et se partagent les responsabilités. Par exemple, le gabarit *itérateur* définit le comportement consistant à parcourir séquentiellement les éléments d'un objet de type agrégat. De même, le gabarit *proxy* définit une structure permettant d'accéder à un objet distant. Le but de cette démarche est de spécifier de façon claire et précise ces composants et d'en proposer des implantations standards afin de réduire autant que faire se peut les temps de développement et d'augmenter le niveau de réutilisation du code. Plusieurs auteurs ont appliqué cette démarche à la conception de programmes concurrents et distribués. Par exemple, Jézéquel et Pacherie dans [JP96] définissent un gabarit *parallel operator*. Ce gabarit est instancié sur tous les processeurs mis en jeu dans un calcul distribué et se comporte comme une entité unique gérant en parallèle la manipulation de données partagées et distribuées. De même, Maffeis dans [Maf96] définit un gabarit *object group* pour l'instanciation des objets répliqués, le partage de charge et la diffusion sur groupe selon différentes sémantiques. Les gabarits de conception peuvent donc être vus comme des composants de base qui peuvent être adaptés, réutilisés et assemblés pour la mise en place d'une application

distribuée.

Nous avons suivi une démarche identique et nous proposons quatre gabarits de conception pour l'algorithmique répartie. Ils définissent des structures de contrôle exécutées par des ensembles d'objets distribués. Les quatre structures proposées dans ce chapitre et dans [SD96], sont le phasage, la conditionnelle, l'itération et la récursion. Ces structures sont les extensions à un niveau réparti des structures de base de l'algorithmique classique. Elles sont spécifiées par des programmes à base de connaissance de niveau groupe. Elles peuvent être composées afin de construire des applications complètes et raffinées afin de fournir des implantations au niveau objet.

5.1 Le schéma de phasage

La notion de phase, ou action de groupe, est la généralisation au niveau réparti de la notion d'action en univers centralisé. Elle a pour but de rassembler sous une même dénomination des actions locales qui font partie d'un même schéma global.

Chaque phase fournit une solution à un problème distribué. D'après Amyay [Amy91], c'est “une tâche distribuée qui amène les différents agents [ou objets] d'une connaissance initiale à une connaissance finale qui caractérise l'objectif de la phase”. Elle factorise donc un ensemble d'actions locales exécutées par les objets de l'application. C'est le gabarit le plus général. Les autres (conditionnelle, itération et récursion) peuvent être vus comme des raffinements de celui-ci. La phase est une transition entre deux états de groupe. Chaque état étant associé à un niveau de connaissance de groupe, c'est donc une action modifiant l'état de connaissance du groupe. Par exemple, dans les applications d'échange de données comme le protocole de transmission de bit (Cf. paragraphe 4.5.3), les actions de groupe *Transmettre* et *Acquitter* sont des phases. De même, dans un protocole transactionnel les actions d'engagement ou d'annulation sont des phases.

La notion de phase est la brique de base qui permet de spécifier puis de raffiner un modèle de comportement dans ses dimensions temporelle, c'est à dire dire d'évolution ordonnée, et épistémique, c'est à dire de connaissance.

5.1.1 Dimension temporelle

La phase est associée à la réalisation d'une étape entre deux états distinguables et immédiatement consécutifs. Chaque état est défini par un prédicat (précisé plus loin dans le paragraphe sur la dimension épistémique). La spécification d'une phase implique qu'il existe une relation d'ordre dans le temps absolu entre les instants de vérification du prédicat associé à l'état initial et du prédicat associé à l'état final. Un état est considéré comme atteint lorsque son prédicat est vérifié pour la première fois. En définissant une phase, on définit donc un élément d'une relation d'accessibilité entre des situations indexées par le temps. Cette relation peut alors servir de modèle pour l'expression des prédicats d'une logique temporelle.

On peut représenter une phase de façon graphique dans une forme habituelle des systèmes de transitions comme une transition entre un état initial et un état final. De manière identique, on peut représenter la même chose dans une approche langage avec une opération associée à une pré et une post-condition. En exprimant la pré-condition et la post-condition sous la forme d'une pré-condition et d'un invariant, on se rapproche alors de la notion de machine abstraite introduite par la méthode B [Abr96].

5.1.2 Dimension épistémique

La construction d'un raisonnement épistémique impose la définition d'un modèle pour une logique modale épistémique, c'est à dire l'identification d'un ensemble de mondes possibles et l'identification d'une relation d'accessibilité dans cet ensemble. La mise en œuvre de cette définition extensive est, dans la plupart des cas, excessivement longue. Aussi, dans l'optique de spécification avec raffinement que nous poursuivons, nous cherchons à donner une description minimale utilisable dans le cadre d'un modèle et qui permettrait, en cas de besoin, d'atteindre l'ensemble des mondes possibles. Nous illustrons notre démarche par l'exemple qui suit.

5.1.2.1 Variables et groupes

La dimension épistémique d'un modèle impose, peu ou prou, de définir le ou les groupes concernés par la phase. Une définition minimale de ce référentiel des états de connaissance est indispensable pour pouvoir parler des données manipulées par la phase et de ses états caractéristiques. Ce n'est qu'à cette condition que l'on peut envisager la représentation de l'ensemble des mondes possibles et de la relation d'accessibilité dans ces mondes.

Si l'on prend l'exemple d'un protocole de communication atomique pour échanger un seul bit d'information, un ensemble minimum de données manipulées par ce protocole est un groupe G d'objets communicants et une variable Bit de type booléen :

group vars
G : **set of ref object**
Bit : **boolean**

L'un des problèmes difficiles de la spécification est la complétude des notions introduites pour préciser, documenter et prouver un comportement. En particulier, pour des comportements de groupe qui partagent des objets, il est important de bien préciser le statut de connaissance des objets partagés ainsi que la sémantique d'accès aux variables partagées. En fait, la première version de la spécification comprenant G et Bit s'adapte à une variété très large de comportements visant à atteindre un consensus dans le groupe G. Si l'on suppose une communication point à point, on peut tout de suite effectuer la spécification suivante qui est, en fait, un raffinement de la précédente :

G : **set of ref object** $:= \{\text{Entité}_i | i := 1, 2\}$

Mais cette spécification, qui ne distingue pas les entités, suppose plutôt une communication bidirectionnelle entre $Entité_1$ et $Entité_2$. On peut vouloir ne modéliser qu'une communication unidirectionnelle. Dans ce cas, on spécifie plutôt :

G : **set of ref object** $:= \{\text{Emetteur, Récepteur}\}$

Dans ce dernier cas on précise, dès le niveau groupe, les rôles respectifs différents de l'émetteur et du récepteur. La pré-condition associée à l'état initial est alors : $S_G(Bit)$ (c'est à dire au moins un site connaît le bit au début de la phase). La post-condition pourrait être $E_G(bit)$ (les deux sites connaissent le bit) ou $E_G^2(Bit)$ (les deux sites connaissent le bit et savent que l'autre site le connaît aussi). Nous retenons ce second objectif. On a donc une spécification plus précise de la phase :

group vars
G : **set of ref object** := {Emetteur, Récepteur}
Bit : **boolean**
begin
$\{S_G(Bit)\}$
Communication bit atomique
$\{E_G^2(Bit)\}$
end

On peut raffiner les niveaux de connaissance en précisant que :

- d'une part, la pré-condition vaut $K_{Emetteur}(Bit)$. C'est bien un raffinement puisque $K_{Emetteur}(Bit) \Rightarrow S_G(bit)$
- et d'autre part, la post-condition vaut $K_{Emetteur}(Bit) \wedge K_{Récepteur}(Bit) \wedge K_{Emetteur}(K_{Récepteur}(Bit)) \wedge K_{Répteur}(K_{Emetteur}(Bit))$ C'est bien un raffinement puisque ce prédicat est une simple réécriture du prédicat $E_G^2(Bit)$

5.1.2.2 Statut des variables

Dans une telle spécification, les deux variables G et Bit ont un statut assez différent. Le groupe G des objets communicants apparaît plutôt comme une constante. On dirait dans le langage de l'algorithmique répartie qu'il est statique et invariant pendant toute la durée de l'algorithme. Or, il est excessivement difficile de préciser, dans un contexte réseau donné, le comportement de la variable G. Ce problème d'appartenance à un groupe peut être traduit de plusieurs façons : la connaissance du groupe peut être répartie (dans ce cas il faut déterminer à quelles informations accède chaque membre) ou elle peut être instanciée (dans ce cas il faut déterminer sur quel site). Ici nous pourrions, compte tenu du caractère simple du problème, postuler que chaque site connaît l'identifiant de son interlocuteur. Cependant, nous n'avons pas un besoin immédiat de cette précision et nous pouvons laisser sous-spécifiée cette définition.

La variable Bit est, quant à elle, beaucoup plus importante. C'est en effet sur elle que porte le schéma de phasage. Par sa spécification même, cette phase spécifie la sémantique d'accès et de connaissance pour cette variable. Le problème à résoudre a pour vocation de communiquer la connaissance de la valeur du bit entre deux sites. Les deux sites doivent donc gérer une copie de cette variable et peuvent avoir une perception différente de sa valeur. On peut utiliser, pour préciser le fait que la variable Bit a pour vocation d'être en copies multiples dans tous les membres du groupe, le mot clé *replicated* et l'indication du groupe de réplication :

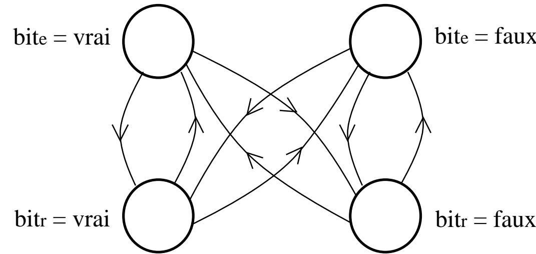
Bit : **boolean replicated on** G

Ceci ne définit pas la sémantique d'accès à la variable partagée Bit, mais indique que l'implantation vise une instance bit_e sur l'émetteur et une instance bit_r sur le récepteur.

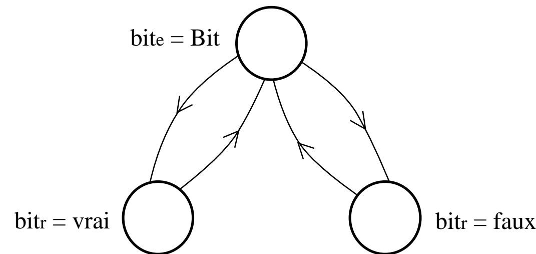
5.1.2.3 Mondes possibles

Dans ce cas simple, on peut maintenant identifier la relation d'accessibilité et l'ensemble des mondes possibles. Il y a deux états de contrôle : avant et après la phase. La variable G initialisée

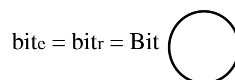
avec l'ensemble $\{Emetteur, Récepteur\}$ ne définit qu'un seul monde possible. La déclaration Bit: **boolean** définit deux mondes possibles sans relation entre eux puisque si le bit est vrai, il est impossible qu'il soit faux. Finalement, le raffinement de la déclaration Bit: **boolean replicated on** G permet d'envisager quatre modes possibles et les relations suivantes :



Finalement, la notation synthétique de la pré-condition $K_{Emetteur}(Bit)$ implique que, pour l'émetteur, au début de l'algorithme, il n'existe qu'une seule situation possible où la valeur du bit est fixée. La relation d'accessibilité sur l'ensemble des mondes possibles se réduit donc pour l'état initial à :



C'est sur ce modèle de départ que l'on raisonne et l'on applique une phase qui vise à atteindre pour le groupe de l'émetteur et du récepteur le niveau E^2, soit en fait un seul monde possible avec :



Remarquons qu'aucune variable concernant l'identification des états de connaissance visées par l'algorithme n'apparaît dans l'identification des mondes possibles. En particulier, on ne manipule pas un booléen qui contient l'information $E^2_G(Bit)$. Celle-ci va donc rester implicite relativement à l'algorithme. Elle est établie dans une preuve mais n'est pas accessible dans une dimension réflexive par programme.

5.1.3 Approche observationnelle

Une phase représente l'exécution coordonnée d'un ensemble d'objets au sein d'un groupe. Nous avons vu qu'elle correspond à une action globale dénotée par deux prédicats épistémiques. Cette description est avant tout une démarche de spécification. On peut s'intéresser à la façon dont l'exécution de cette spécification va être observée.

Nous avons vu au paragraphe 3.1 que l'état global d'un système au sens de Chandy et Lamport, est associé à la notion de coupe cohérente. Celle-ci a pour but d'identifier parmi tous les n-uplets d'états locaux des objets du système, ceux qui correspondent à des situations qui ont été effectivement observées. Les coupes cohérentes, appelées également états globaux cohérents, peuvent être organisées en treillis. Elles permettent de reconstituer toutes les observations possibles d'une exécution donnée. Les états globaux cohérents appartenant à toutes les observations possibles d'une exécution sont dits inévitables. Ces états sont plus importants que les autres puisqu'on est sûr qu'ils seront observés dans tous les cas.

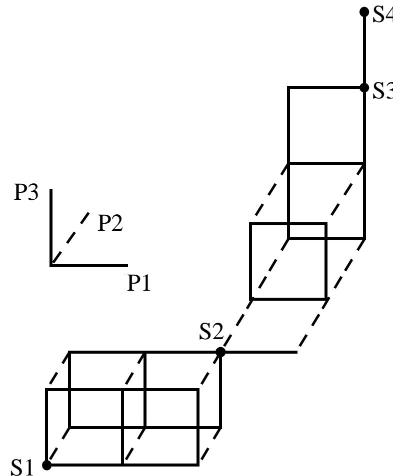


FIG. 5.1 – *Observation d'une exécution et états globaux inévitables*

Les états inévitables permettent donc de découper un treillis d'états globaux cohérents en blocs. Chaque bloc situé en aval d'un état inévitable s'exécute avant le bloc situé en amont. Par exemple, le treillis de la figure 5.1 comporte quatre états inévitables : S_1, S_2, S_3 et S_4. On est sûr que tous les chemins possibles entre S_1 et S_2 sont avant tous ceux entre S_2 et S_3 (de même pour S_2, S_3 et S_3, S_4). Dans notre approche, les niveaux de connaissance dénotant une action globale correspondent à des états globaux inévitables. En effet, si ce n'était pas le cas, certaines observations pourraient ne pas capturer l'état vérifiant ce niveau de connaissance. On ne serait pas alors en mesure de déterminer si l'action globale a été exécutée ou non.

Néanmoins, la propriété d'état inévitable ne caractérise que partiellement les niveaux de connaissance d'une phase. En effet, le treillis des états globaux cohérents de Chandy et Lamport ne représente pas le modèle d'accessibilité complet d'une spécification. Il ne traduit que l'ensemble des observations possibles pour une seule exécution. Or, plusieurs exécutions sont possibles à partir d'une même spécification. En effet, certaines actions, comme par exemple des conditionnelles *if then else* à un niveau global, entraînent des développements alternatifs du treillis. En construisant le graphe d'accessibilité associé à la spécification, on obtient alors avec plusieurs treillis possibles. On caractérise ainsi un niveau de connaissance par une classe d'équivalence d'états inévitables. Chaque état de cette classe appartient à un treillis particulier. Afin d'être exhaustif, il faudrait donc ajouter aux treillis (comme par exemple celui de la figure 5.1) des axes supplémentaires représentant toutes les exécutions possibles. Néanmoins, comme il n'est pas évident d'ajouter des axes à un graphique en trois dimensions représenté en perspective, nous nous contentons, en général, de ne présenter qu'une treillis parmi tous ceux possibles.

5.1.4 Spécialisation

La figure 5.2 reprend le programme de niveau groupe du protocole de transmission de bit comprenant les phases *Transmettre* et *Acquitter* (Cf. paragraphe 4.5.3). Deux types de phases peuvent être définis : soit la phase entraîne une élévation de la connaissance de groupe, soit elle révoque un certain nombre de connaissances.

Le premier cas est le plus courant. Les actions globales associées à la phase permettent aux objets, au travers d'opérations locales et d'opérations d'interactions, d'acquérir un certain nombre d'informations. Par exemple, la phase *Transmettre* de la figure 5.2 fait passer le groupe d'un état de connaissance instanciée (S_G) de la variable *Bit* à un niveau de connaissance de tous de niveau 2 (E_G^2). D'un point de vue quantitatif il y a donc bien une élévation de la connaissance de groupe.

Le second type de phase révoque une partie de la connaissance acquise par le groupe. Cette situation se rencontre entre autres, dès que l'on recommence un algorithme terminé. Ainsi le programme de la figure 5.2 peut être inséré dans une boucle infinie afin de recommencer le processus de transmission dès que l'acquittement a été effectué (Cf. figure 5.3). Cela revient à ajouter une transition entre les états S_3 et S_1 de l'automate. On constate alors que l'on passe d'un état où la variable *Bit* est connue de tous à un état où sa valeur n'est plus connue que de l'émetteur. Cette phase révoque donc une partie de la connaissance liée à la variable *Bit*. On pourrait arguer que, bien que pour des facilités d'écriture, un seul et même identificateur *Bit* est utilisé, il n'y a aucun rapport entre les valeurs transmises d'un cycle à l'autre. Les cycles peuvent donc être numérotés de façon à créer des numéros d'époque, et on peut transformer la variable binaire *Bit* en un tableau (Cf. figure 5.4). De cette façon, à chaque début de cycle, la connaissance du bit d'indice n reste instanciée tandis que les $n \Leftrightarrow 1$ précédents bits sont toujours connus de tous. Néanmoins, cette tentative de transformation des phases révoquant la connaissance en phases standards se heurte à un nouveau problème : la taille du tableau de bits ne peut être infinie. Au bout d'un nombre fini de cycles, son contenu doit être purgé, ce qui entraîne la destruction de données antérieures. Il n'est donc, en pratique, pas possible de se passer des phases révoquant la connaissance.

5.1.5 Composition

Les phases d'une application distribuées peuvent être composées de nombreuses façons. Elles peuvent être exécutées en séquence les unes après les autres, elles peuvent être exécutées concurremment les unes par rapport aux autres, ou elles peuvent être organisées en commandes gardées.

5.1.5.1 Séquence

La composition séquentielle impose un enchaînement strict des phases les unes après les autres. Elle est notée par un point-virgule. Chaque phase se traduit par des actions locales sur chaque objet du groupe. Toutes ces actions locales ne commencent, ni ne finissent forcément, à un même instant global. La coordination de groupe impose que, sur chaque objet, l'action locale correspondant à la phase k est exécutée après celle correspondant à la phase $k \Leftrightarrow 1$ et avant celle correspondant à la phase $k + 1$.

5.1.5.2 Constructeur de parallélisme

La composition parallèle permet d'exécuter concurremment plusieurs phases au sein d'un groupe. Elle est notée à l'aide des instructions *co_begin*/*co_end*. Les différentes actions de groupe

```
group vars
  Bit : boolean replicated on G
begin
  Transmettre ;
  Acquitter
end
```

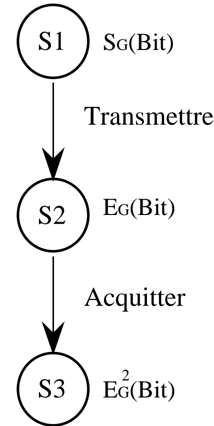


FIG. 5.2 – *Phase d'élévation de la connaissance*

```
group vars
  Bit : boolean replicated on G
begin
  while true do
    Transmettre ;
    Acquitter
  end while
end
```

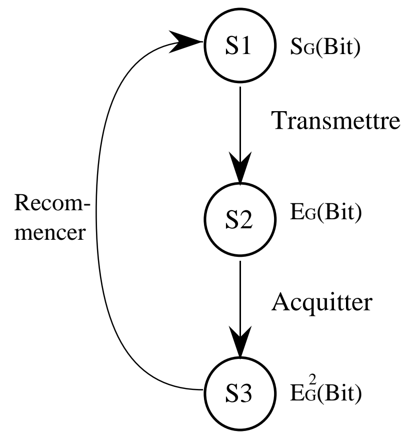


FIG. 5.3 – *Boucle avec phase de révocation de connaissance*

```
group vars
  TBit :   array [1..n] of
           boolean replicated on G
  n : integer replicated
begin
  n := 1;
  while true do
    Transmettre( TBit[n] );
    Acquitter( TBit[n] );
    n := n+1
  end while
end
```

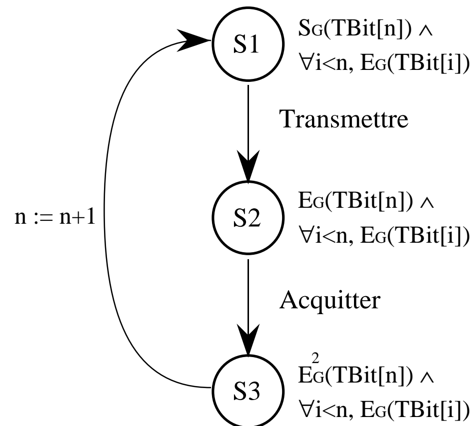


FIG. 5.4 – *Boucle sans phase de révocation de connaissance*

peuvent être exécutées par des sous-groupes distincts ou non. La composition parallèle prend fin lorsque toutes les actions concurrentes sont terminées. On pourrait autoriser des conditions plus riches comme par exemple, la fin de la construction parallèle dès que la première action de groupe est terminée ou dès qu'un quorum est atteint. Néanmoins, la gestion des activités restantes pose un certain nombre de problèmes. Par exemple, la terminaison d'actions distribuées est une tâche difficile imposant la mise en place de mécanismes coûteux. Inversement, si on laisse ces activités se dérouler jusqu'à leur terminaison, on risque d'introduire des effets de bord avec les actions suivantes. En l'état actuel, il nous semble donc plus simple de retenir une sémantique fondée sur la terminaison de toutes les activités parallèles.

FIG. 5.5 – *Composition parallèle de phases*

5.1.5.3 Phases gardées

L'organisation des phases en commandes gardées a pour but de reproduire, à un niveau réparti, un schéma de contrôle équivalent, par exemple, à celui de l'instruction *select* du langage Ada. Dans les programmes de niveau groupe, nous le notons à l'aide des instruction *choice/endchoice*. Dans les automates graphiques, nous adoptons une notation de style CSP dans laquelle les conditions sont précédées par un point d'interrogation et les actions par un point d'exclamation (Cf. figure 5.6). Lorsqu'il n'y a pas de condition, l'action est notée sans point d'exclamation. La notion de commande gardée [Dij75, Dij76] met en place des couples condition, action. La sémantique de cette construction est que, si la condition est vérifiée, alors l'action est entreprise. Lorsque plusieurs commandes gardées sont issues d'un même état, une seule commande est éventuellement déclenchée. Si plusieurs gardes sont vraies simultanément, alors l'action à entreprendre est choisie de manière non déterministe.

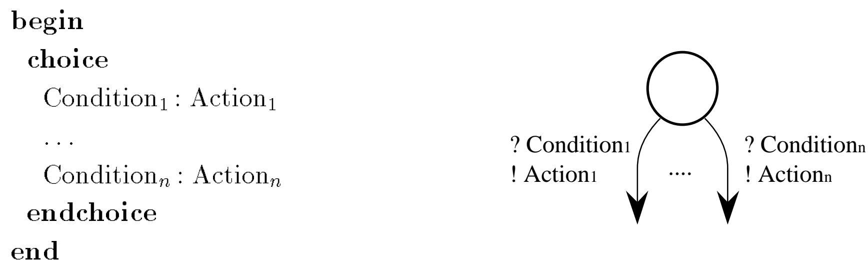


FIG. 5.6 – *Phases gardées*

5.2 La conditionnelle distribuée

Le gabarit *conditionnelle distribuée* est la généralisation à un niveau réparti de la structure *case* des langages de programmation habituels. Ce gabarit peut être utilisé par exemple pour :

- changer dynamiquement la stratégie de résolution du problème distribué,
- choisir des comportements alternants en cas de panne,
- effectuer une transaction,
- détecter une condition globale.

Ce gabarit modélise un schéma de contrôle dans lequel le groupe modifie son comportement en fonction d'une condition sur son état global. Deux phases peuvent être dégagées : dans un premier temps, l'état global est collecté et la condition est évaluée, puis dans un second temps, les actions alternatives sont choisies.

5.2.1 Définition

La figure 5.7 présente le programme de niveau groupe associé à ce comportement. Une variable globale *Var* est déclarée. Avant que sa valeur ne soit calculée, c'est à dire dans l'état S_1, son statut est celui d'une connaissance distribuée. En effet, c'est une fonction sur l'état du groupe, donc tous les éléments permettant de l'évaluer sont présents dans le groupe, mais aucun objet ne les a tous en sa possession. Le prédicat de l'état S_1 est noté $D_G(Var)$. Un algorithme de collecte d'état global et d'évaluation permet alors de faire passer cette connaissance distribuée à un statut instancié. Le prédicat de l'état S_2 vaut donc $S_G(Var)$. Selon la valeur de la variable *Var*, une des actions $Action_1, \ldots, Action_n, Action_e$ est entreprise. Dans la perspective de l'implantation de ce schéma à un niveau objet, nous préfixons les appels de ces actions avec la référence de l'objet courant *self* (en fait, il faudrait développer cette notion et parler plutôt ici de la référence du groupe courant). Notons que ce schéma peut être simplifié en supprimant la phase de collecte lorsque la valeur de la variable est directement accessible (c'est le cas d'une variable simple ou locale à un objet).

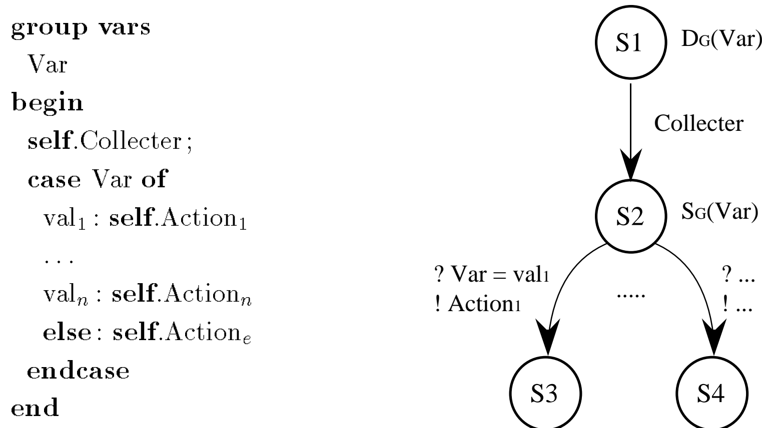


FIG. 5.7 – *Gabarit conditionnelle distribuée*

5.2.2 Raffinement

Le schéma de comportement présenté ci-dessus peut être raffiné de nombreuses façons. On peut, par exemple, choisir une variable de type booléen et ne retenir que deux branches alternatives. On obtient alors une structure de type *if then else*. La phase de collecte peut alors être complétée par une phase de diffusion du résultat à tous les objets du groupe et les branches *then* et *else* peuvent être associées, respectivement, à des actions d'engagement et d'annulation. On obtient alors un schéma transactionnel avec un protocole de validation à deux phases. Nous développons plus particulièrement ce dernier exemple en explicitant les différentes phases et en proposant une implantation type.

5.2.2.1 Structure if then else

La structure précédente peut être simplifiée lorsque la variable de choix est de type booléen. Les deux seules valeurs possibles étant vrai et faux, on obtient seulement deux branches alternatives. Il est alors plus simple d'adopter une notation de style *if then else*.

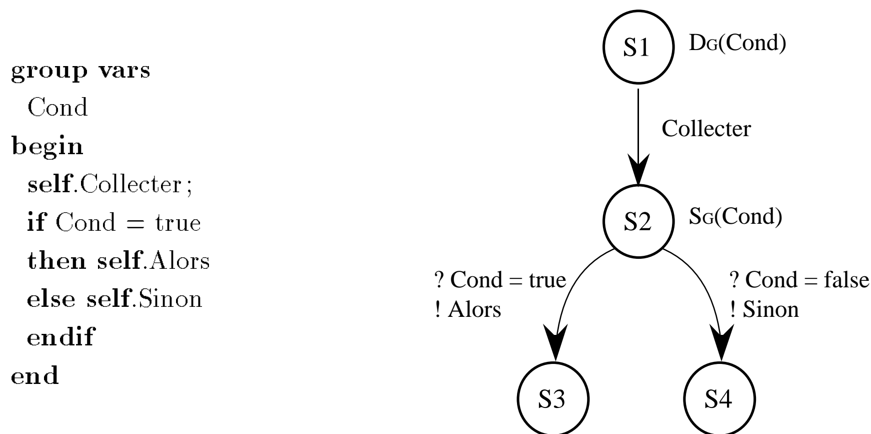


FIG. 5.8 – *Structure if then else pour le gabarit conditionnelle distribuée*

5.2.2.2 Schéma transactionnel

Le schéma de comportement précédent peut être spécialisé en un schéma transactionnel pour un protocole de validation à deux phases. La phase de collecte est une phase de vote notée *Voter* et les phases *Alors* et *Sinon* deviennent des phases d'engagement et d'annulation notées respectivement *Engager* et *Annuler*.

On peut remarquer tout d'abord que ce gabarit particularise un objet au sein du groupe : c'est celui pour lequel la connaissance de la condition est instanciée après la phase d'évaluation. Dans les protocoles transactionnels, cet objet est désigné sous le terme de coordinateur ou de maître de la transaction. Les autres sont dits participants. La phase de vote est initiée par le coordinateur qui contacte tous les participants. Les interactions entre le coordinateur et les participants peuvent se faire de deux manières : soit par un parcours séquentiel, soit par un parcours récursif. Dans le premier cas, le coordinateur contacte tous les participants individuellement, tandis que dans le second cas, il initie une vague de parcours qui se propage récursivement dans tout le groupe. Nous ne détaillons dans ce paragraphe que le mode par parcours séquentiel. Le mode par parcours récursif est présenté au paragraphe 5.4 consacré au gabarit *récursion distribuée*.

```
method Main
group vars
  Cond : boolean
begin
  self.Voter ;
  if Cond = true
  then self.Engager
  else self.Annuler
  endif
end
```

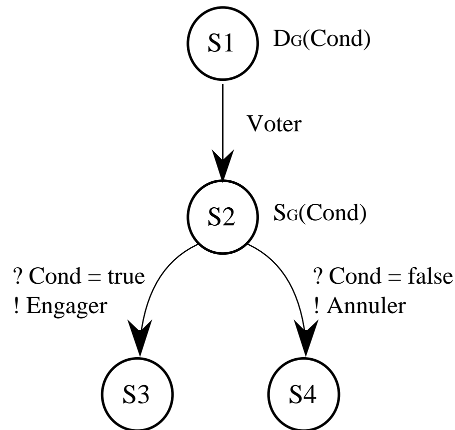


FIG. 5.9 – *Schéma transactionnel*

Le mode par parcours séquentiel nécessite que le coordinateur connaisse tous les participants. En d'autres termes, cela signifie que l'objet coordinateur doit connaître les références de tous les objets participants. Il demande à chacun de voter, c'est à dire d'évaluer une condition correspondant à la transaction à effectuer. La condition est différente pour chaque participant (par exemple, il va demander à l'un s'il peut créditer un compte, tandis qu'il va demander à l'autre s'il peut en débiter un autre). Elle peut être identique dans d'autres schémas de contrôle, comme par exemple pour un traitement effectué en redondance massive. Dans le cas d'une transaction simple, la condition met en jeu uniquement des données locales au participant. Dans le cas d'une transaction imbriquée, elle nécessite l'évaluation d'une condition globale puisque la réponse des participants dépend d'un mécanisme d'évaluation de sous-transactions. Une fois les expressions transmises, le coordinateur collecte toutes les réponses. Il les consolide et fournit une évaluation de la condition globale. Il notifie ce résultat aux participants qui choisissent d'exécuter, soit la phase *Engager*, soit la phase *Annuler*.

5.3 L'itération distribuée

Le gabarit *itération distribuée* est la généralisation à un niveau réparti de la boucle des langages de programmation habituels. Il est utilisé chaque fois qu'un comportement global doit être itéré. On l'emploie par exemple pour :

- réexécuter plusieurs fois un algorithme,
- dans les algorithmes à train de vagues pour détecter une terminaison,
- dans les algorithmes de recalcul périodique des tables de routage d'un réseau,
- dans les algorithmes répartis dits auto-stabilisants [Dij74][Tel94] qui offrent des comportements tolérants les fautes avec une approche dite optimiste. Deux états peuvent être distingués dans un algorithme auto-stabilisant : l'état stable et l'état instable. L'apparition d'une faute place le groupe dans un état instable, ce qui a pour effet de déclencher l'exécution d'un mécanisme de compensation jusqu'au retour à l'état stable. Plutôt que de suspecter toute information reçue ou de scruter périodiquement l'activité des objets du groupe comme dans les approches dites pessimistes de l'algorithmique répartie avec détecteurs de fautes [CT91],

les algorithmes auto-stabilisants font l'hypothèse que tous les objets se comportent normalement mais, que si certains d'entre eux deviennent fautifs, alors ils retournent à un mode de fonctionnement correct dans un laps de temps fini. Ces algorithmes sont utilisés par exemple pour maintenir en permanence une structure de données répartie (un arbre, un anneau, etc ...) ou pour garantir un accès uniforme à une ressource partagée (par exemple un accès en exclusion mutuelle). L'hypothèse de base est que chaque objet détermine son comportement à partir de son état local et à partir de l'état distant de ces voisins immédiats dans le réseau.

5.3.1 Définition

La figure 5.10 présente le programme de niveau groupe associé à ce gabarit. Une variable globale *Cond* est déclarée. Elle est de type booléen et contient le résultat de l'évaluation de la condition. Dans l'état S_1, avant que l'état global ne soit collecté et sa valeur ne soit calculée, son statut est celui d'une connaissance distribuée. Puis après une phase d'évaluation cette variable acquiert un statut instancié. Le contrôle s'oriente alors, soit vers la phase *Fin de boucle* entre les états S_2 et S_3 si cette variable est fausse, soit vers la phase *Traiter* entre les états S_2 et S_1 si elle est vraie. Notons, que ce schéma peut être simplifié en supprimant la phase de collecte dans le cas où la valeur de la variable *Cond* est directement accessible. La figure 5.11 présente un treillis des états globaux correspondant aux observations possibles d'un tel gabarit. Le schéma de comportement associé à la phase *Traiter* est répété un nombre éventuellement nul, fini ou infini de fois.

```
method Main
group vars
  Cond : boolean
begin
  self.Collecter ;
  while Cond do
    self.Traiter ;
    self.Collecter
  enddo
end
```

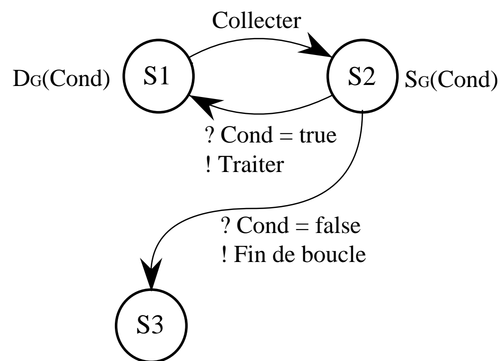


FIG. 5.10 – *Gabarit itération distribuée*

5.3.2 Raffinement

De nombreuses variations peuvent être mises en place pour ce gabarit. On peut, par exemple, faire varier le degré de synchronisation des phases *Traiter*. On obtient ainsi un gabarit itération distribuée synchrone dans lequel les phases *Traiter* s'exécutent de façon synchronisée sur tous les objets du groupe, et un gabarit itération distribuée asynchrone dans lequel les phases *Traiter* s'exécutent indépendamment les unes des autres. On peut également mettre en place des versions dans lesquelles la désynchronisation est bornée par une valeur maximale. Néanmoins, nous n'abordons que les deux premières versions de ce gabarit.

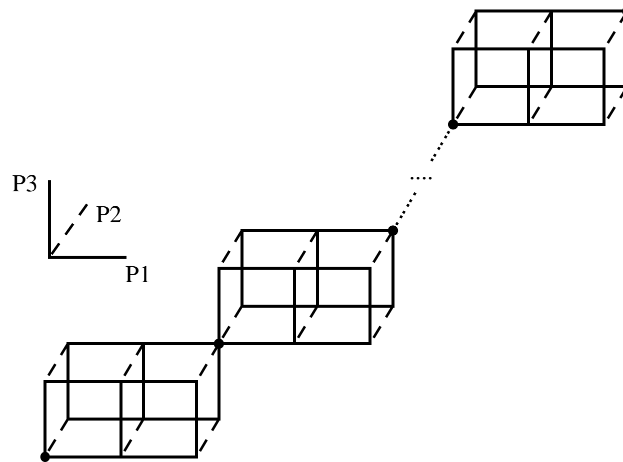


FIG. 5.11 – *Treillis des états globaux pour une itération distribuée*

5.3.2.1 Itération synchrone

Dans la version synchrone, la phase de collecte est identique à celle définie au paragraphe 5.2 pour la *conditionnelle distribuée*. Un objet particulier, le coordinateur, est chargé de transmettre à tous les autres objets, appelés participants, les expressions à évaluer. Il collecte les réponses, consolide le résultat pour obtenir la valeur de la condition globale et la diffuse aux participants. Ces deux étapes d'évaluation et de diffusion peuvent s'effectuer avec un mode de communication par parcours séquentiel ou par parcours récursif. Une fois que le coordinateur et les participants connaissent la valeur de la condition globale, ils peuvent choisir d'exécuter soit la phase *Traiter*, soit de sortir de la boucle par la phase *Fin de boucle*. Cette version du gabarit *itération distribuée* est dite synchrone car tous les objets du groupe exécutent simultanément et de façon coordonnée la phase de collecte puis la phase *Traiter*.

5.3.2.2 Itération asynchrone

Dans la version asynchrone, du gabarit *itération distribuée*, chaque objet exécute en séquence mais indépendamment des autres objets, les phases de *Collecter* et *Traiter*. Il n'y a pas de coordinateur. Chaque objet collecte lui-même les données nécessaires à l'évaluation de la condition globale. Il n'y a pas d'étape de diffusion du résultat. Ce type d'itération est rencontré dans les algorithmes dits auto-stabilisants. Au cours d'une itération, chaque objet collecte l'état de ces voisins, évalue une condition et détermine si une règle d'évolution est applicable. Si c'est le cas, il l'exécute (cela correspond à la phase *Traiter*). Sinon, il considère qu'il a atteint l'état stable et exécute la phase *Fin de boucle*. Cette version de l'itération distribuée est dite asynchrone car les objets ne sont pas synchronisés dans leur exécution des phases d'évaluation et *Traiter*.

5.4 La récursion distribuée

Le gabarit *récursion distribuée* est la généralisation à un niveau réparti du parcours arborescent de l'algorithmique centralisée. Ce gabarit est utilisé pour des groupes de taille importante dans lesquels il n'est pas envisageable que les objets connaissent en permanence l'identité de tous les

membres du groupe. Chaque objet possède donc un ensemble d'accointances et la composition du groupe est définie de proche en proche. Le gabarit *récursion distribuée* est alors utilisé pour :

- parcourir récursivement l'ensemble des objets du groupe,
- diffuser une information à tous les objets,
- collecter un ensemble d'états locaux en vue du calcul d'un état global,
- construire une structure de données répartie.

5.4.1 Définition

Ce gabarit est parfois désigné dans le domaine des algorithmes réseaux sous le terme de vague (Cf. [FGL93][Tel94][Gom95]). Au cours d'une récursion distribuée, tous les objets du groupe sont visités et une action locale est exécutée à l'occasion de cette visite.

5.4.1.1 Description informelle

Ce gabarit est déclenché par un ou plusieurs objets appelés initiateurs, qui le transmettent à l'ensemble de leurs accointances, c'est à dire à l'ensemble de leurs voisins dans le groupe. La vague se propage de proche en proche dans le groupe et développe récursivement un arbre de parcours. Lorsqu'une branche a été complètement développée, la vague reflue jusqu'à un nœud intermédiaire et reprend son parcours de descente dans une autre branche. Plusieurs versions de ce gabarit peuvent être définies. Sans évoquer de façon exhaustive toutes les déclinaisons possibles, on peut, par exemple, mettre en place des versions qui autorisent la construction simultanée d'un seul ou de plusieurs parcours, avec une visite séquentielle ou parallèle des branches. De plus, chaque objet peut devoir être visité une seule fois (parcours simple) ou plusieurs fois (c'est la cas, par exemple, dans l'algorithme des généraux byzantins de Lamport, Shostak et Pease [LSP82]). Finalement, ce gabarit peut être associé à une itération répartie asynchrone [DFGS96] afin d'entretenir de manière auto-stabilisante une structure de données répartie à l'aide de parcours récursifs successifs.

5.4.1.2 Programme de niveau groupe

Nous nous limitons à une présentation du gabarit pour un parcours simple, c'est à dire avec un seul passage sur chaque objet. Néanmoins, le programme de la figure 5.12 recouvre les versions avec un ou plusieurs initiateurs et avec une visite séquentielle ou parallèle des branches. Il doit être complété, selon la réutilisation que l'on en fait, par le code à exécuter sur chaque objet lors de la première visite de la vague (c'est la phase de *Pré-traitement*) et par les résultats à collecter lors du reflux (c'est la phase de *Post-traitement*).

Ce gabarit peut être utilisé, par exemple, pour élire un objet au sein d'un groupe afin de lui accorder un accès privilégié à une ressource partagée, ou pour construire un arbre couvrant afin de router des messages. Dans le premier cas, il faut retourner à chaque reflux de vague l'identité du meilleur objet rencontré jusqu'à ce point, tandis que dans le second cas, il faut enregistrer lors de la première visite l'identité de l'objet père ayant transmis la vague.

```
group vars
  const Sites : set of object
  Initiateurs : set of object

method Rec( in Visités : set of object )
  Ajout : set of object
begin
  if self.Poursuivre( Visités ) then
    Ajout := self.SitesAjoutés( Visités ) ;
    self.Pré-traitement ;
    self.Rec( Visités ∪ Ajout ) ;
    self.Post-traitement
  endif
end

method Poursuivre( in Visités : set of object ) : boolean
begin
  return Visités ≠ Sites
end

method SitesAjoutés( in Visités : set of object ) : set of object
  Ajout : set of object
begin
  Ajout ⊆ Sites ⇔ Visités ;
  return Ajout
end

method Main
begin
  self.Rec( Initiateurs )
end
```

FIG. 5.12 – *Programme de niveau groupe RD_G pour le gabarit récursion distribuée*

5.4.1.3 Description du programme

Le programme de la figure 5.12 déclare deux variables globales *Sites* et *Initiateurs*. Ce sont toutes les deux des ensembles de références d'objets. La première est constante et désigne l'ensemble de tous les objets du groupe. Cette variable est utilisée uniquement dans la modélisation de niveau groupe et n'intervient pas au niveau objet. La seconde *Initiateurs* contient, quant à elle, la liste du ou des sites initiateurs de la vague. C'est également une variable de modélisation qui n'est pas instanciée au niveau objet.

Le corps du programme proprement dit comprend l'appel de la méthode *Rec* avec la liste des initiateurs. Cette méthode accepte comme paramètre d'entrée un ensemble *Visités* d'objets déjà visités par la récursion. La méthode *Poursuivre* teste si la récursion doit être poursuivie ou non. Dans le cas d'un parcours simple, elle retourne la valeur vrai tant qu'il reste des objets qui n'ont pas encore été visités, c'est à dire tant que *Visités* est différent de *Sites*. Cette méthode est plus complexe lorsque la récursion nécessite plusieurs passages sur chaque objet comme dans le cas de l'algorithme des généraux byzantins de Lamport, Shostak et Pease [LSP82]. A chaque pas de la récursion, la méthode *SiteAjoutés* sélectionne un certain nombre de sites à ajouter. Dans une première version de ce programme, ils sont choisis parmi ceux de $Sites \Leftrightarrow Visités$. Ils sont affectés à la variable *Ajout*. Comme pour les variables *Visités* et *Sites*, *Ajout* est utilisée uniquement dans la modélisation de niveau groupe. La phase de *Pré-traitement* est exécutée pour ces nouveaux éléments, puis la méthode *Rec* est appelée récursivement avec en paramètre l'ensemble des objets précédemment visités augmenté des objets nouvellement incorporés. Au retour de cette méthode, la phase de *Post-traitement* est exécutée.

La figure 5.13 présente deux configurations intermédiaires possibles : la première concerne une récursion avec un seul initiateur, tandis que la seconde concerne une récursion avec plusieurs initiateurs. Ceux-ci sont représentés en gras. Le contenu de l'ensemble des sites visités apparaît en grisé.

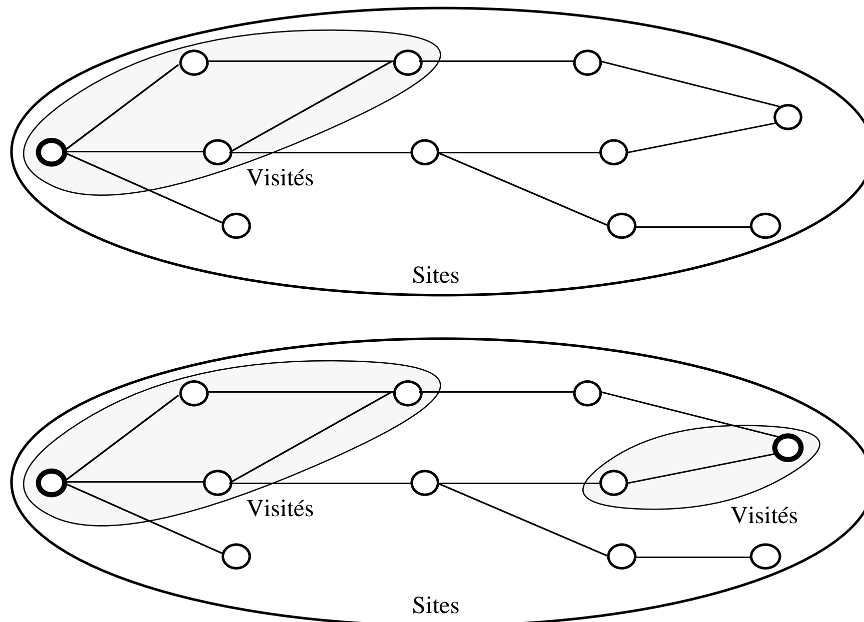


FIG. 5.13 – *Etapes intermédiaires d'une récursion distribuée*

5.4.2 Raffinement

Afin de préciser la définition du gabarit *récursion distribuée*, nous proposons deux raffinements de sa spécification. La première précise le programme de niveau groupe, tandis que la seconde fournit les comportements de niveau objet.

5.4.2.1 Raffinement de niveau groupe

Comme l'exemple de la figure 5.13 le suggère, le programme de niveau groupe précédent est légèrement sous-spécifié. En effet, la sélection des éléments de l'ensemble *Ajout* dans l'ensemble *Sites* ⇔ *Visités* des objets non encore visités est trop imprécise. De façon plus rigoureuse et afin d'obtenir un parcours récursif cohérent, seuls les éléments situés immédiatement après la frontière de l'ensemble *Visités* (c'est à dire les éléments s qui n'ont pas encore été visités mais qui sont voisins de sites i déjà visités) sont sélectionnables. Ainsi, la définition de la méthode *SitesAjoutés* doit être remplacée par :

method SitesAjoutés(**in** Visités : **set of object**) : **set of object**
  Ajout : **set of object**
begin
  Ajout $\subseteq \{s | s \notin Visités \wedge \exists i \in Visités, s \in voisins(i)\}$;
  return Ajout
end

Cette spécification amène plusieurs remarques :

- comme précédemment, seuls des sites non visités peuvent être ajoutés. Ainsi, chaque site n'est parcouru qu'une seule fois par la vague.
- tous les objets potentiellement ajoutables (c'est à dire tous les voisins des sites déjà visités) ne le sont pas forcément au cours de la même étape.
- que le parcours soit séquentiel ou parallèle, il se déroule toujours de façon indépendante dans chacune des branches. Pour un parcours séquentiel, un seul élément est ajouté à chaque étape, tandis que plusieurs le sont pour un parcours parallèle. Dans ce dernier cas, on atteint un degré de parallélisme maximum si toutes les branches sont développées simultanément, c'est à dire si à chaque étape de la récursion, tous les voisins des sites déjà visités sont incorporés.
- comme pour les schémas de parcours arborescents de l'algorithmique centralisée, ce gabarit peut être transformé en une itération lorsque la récursion est terminale, c'est à dire lorsqu'aucun post-traitement n'est effectué après l'appel récursif. La transformation fournit alors le programme de la figure 5.14.

5.4.2.2 Raffinement de niveau objet

Le gabarit *récursion distribuée* particularise deux types de comportements : celui des objets initiateurs et celui des objets non initiateurs. Nous désignons ces derniers sous le terme de participants. Ces deux types d'objets exécutent une version locale de la méthode de niveau groupe *Rec* présentée figure 5.12. Les initiateurs exécutent en plus le code de lancement du parcours, c'est à dire la méthode *Main*.

```
group vars
  const Sites : set of object
  Initiateurs, Ajout : set of object
begin
  Visités := Initiateurs ;
  while self.Poursuivre( Visités ) do
    Ajout := self.SitesAjoutés( Visités ) ;
    self.Pré-traitement ;
    Visités := Visités ∪ Ajout
  enddo
end
```

FIG. 5.14 – *Récursion distribuée terminale transformée en itération*

Chaque objet initiateur commence le parcours en propageant la récursion parmi ses voisins. Si le parcours est séquentiel alors un seul voisin est sélectionné, tandis que s'il est parallèle tous les voisins sont retenus. La propagation est réalisée par appels distants de la méthode *Rec* des objets sélectionnés. Tous les objets contactés itèrent alors le processus de sélection propagation jusqu'à ce que tout le groupe ait été visité.

Ce test s'exprime dans le programme de niveau groupe par une conditionnelle distribuée associée à l'expression $Visités \subset Sites$. On constate que ce gabarit, défini au paragraphe 5.2, requiert, soit la présence d'un objet coordinateur capable d'interroger tous les membres du groupe, soit un parcours récursif des objets. Or, la définition du gabarit *récursion distribuée* se base sur une connaissance de la composition du groupe à partir des voisinages de chaque objet. Cela exclut donc l'utilisation d'une solution à base de coordinateur. Par ailleurs, l'hypothèse d'un parcours récursif pour calculer la condition globale en vue de la réalisation du gabarit *récursion distribuée* est bien évidemment irréalisable. La condition $Visités \subset Sites$ du programme de la figure 5.12 ne pouvant pas être calculée à partir d'un algorithme de capture d'état global, il est nécessaire de la déduire des comportements locaux des objets. Par exemple, chaque objet peut enregistrer le passage de la vague et la condition $Visités \subset Sites$ est alors vérifiée localement pour un objet si lui-même et tous ses voisins ont été visités par la vague.

Chaque objet transmet donc une proposition de parcours à un ou à plusieurs voisins. Chacun d'eux teste s'il a déjà été visité par cette vague. Si c'est le cas, il retourne simplement la proposition à l'objet demandeur. Sinon, il poursuit le processus normal de pré-traitement, propagation, post-traitement, puis retourne la proposition. Lorsque l'objet demandeur a reçu les retours de tous ses voisins, il considère que la condition $Visités \subset Sites$ est vérifiée localement. Il retourne à son tour la proposition de vague. Lorsque toutes les propositions ont été retournées aux objets initiateurs, la récursion prend fin.

Bien que la spécification de niveau groupe soit la même pour les quatre versions du gabarit (un ou plusieurs initiateurs et parcours séquentiel ou parallèle) les versions dans lesquelles un seul initiateur est autorisé sont les plus délicates à réaliser. En effet, dans un environnement où le contrôle est complètement décentralisé, chaque objet décide de façon indépendante d'initier ou non une vague. A partir du moment où un objet prend une telle décision, il faut interdire aux autres de faire de même. Cette exclusion peut être réalisée de deux façons :

– soit en posant un verrou d'interdiction sur tous les autres objets,

– soit en désignant un coordinateur du parcours chargé d'accorder les droits d'initiation.

La première solution introduit un certain nombre de difficultés. En effet puisque la composition complète du groupe ne peut se déduire que des voisinages, la pose d'un verrou nécessite un parcours de tous les objets avant le parcours proprement dit. Comme plusieurs initiateurs potentiels peuvent exécuter simultanément ce parcours de pose de verrou, cela revient à gérer une version parallèle du gabarit *récursion distribuée*. Quant à la seconde solution, elle n'est pas envisageable dans un réseau défini uniquement par les voisinages. On constate donc, qu'avec un réseau sans contrôle centralisé et défini par les voisinages, les parcours à plusieurs initiateurs simultanés sont, paradoxalement, moins complexes à concevoir.

5.5 Conclusion sur les gabarits de niveau groupe

Les gabarits de conception ou *design patterns*, visent à définir des schémas d'organisation ou de comportement, qui apparaissent de façon récurrente. Dans le domaine de la conception informatique, ils définissent, par exemple, des hiérarchies d'organisation ou des schémas de coordination entre entités logicielles. Leurs apports peuvent être comparés à ceux de l'approche objet dans le domaine de la programmation. En effet, le principal atout des langages orientés objets est de favoriser la réutilisabilité des composants logiciels. Les gabarits visent, dans le domaine de la conception, un objectif similaire. Ils ont pour but de définir des schémas de conception types pouvant être réutilisés dans différentes applications. Les gabarits et l'approche objet sont d'ailleurs étroitement liés. En effet, sauf exception, les gabarits proposés actuellement sont implantés dans des langages objets. Si la communauté des développeurs d'informatique de gestion emploie fréquemment les démarches à base de gabarits, celles-ci sont encore rares dans les développements d'applications système et réseau.

Dans ce chapitre, nous avons donc voulu contribuer au développement de ces démarches. Nous proposons quatre structures de contrôle qui peuvent être employées comme gabarits pour la conception d'applications distribuées. Ce sont la phase, la conditionnelle distribuée, l'itération distribuée et la récursion distribuée. L'objectif est de définir des structures réutilisables, raffinables et composables. Bien que le compromis ne soit pas évident à mettre en place, ces structures ne doivent être, ni trop, ni pas assez génériques. Ainsi, elles sont assez génériques pour pouvoir être réutilisées dans de multiples applications. Néanmoins, leur définition comporte suffisamment d'éléments afin qu'elles puissent être appliquées à des cas concrets. Par ailleurs, ces structures peuvent être raffinées afin d'introduire des éléments nouveaux permettant de spécialiser leur emploi. Finalement, elles peuvent être assemblées et composées les unes avec les autres afin de faciliter la mise en place d'applications complètes.

Les quatre structures que nous proposons peuvent être vues comme des extensions à un niveau distribué des structures de base de l'algorithmique. Ainsi la phase, la conditionnelle distribuée, l'itération distribuée et la récursion distribuée sont des extensions de la séquence, des structures alternantes de type *case*, des boucles *while* et des parcours arborescents. Elles définissent des schémas de coordination type pour des groupes d'objets. Ce sont donc des schémas de coordination inter-objets. Dans les deux chapitres de la partie suivante, nous nous intéressons à un autre aspect la coordination des applications distribuées. Ainsi, nous abordons la coordination intra-objet, c'est à dire la coordination des activités internes à un objet.

Troisième partie

Coordination intra-objets

Chapitre 6

Langage CAOLAC

Nous présentons dans ce chapitre un langage permettant de spécifier et d'implanter des politiques synchronisation pour des objets concurrents. En particulier nous souhaitons faciliter l'implantation des comportements de groupe et des structures de contrôle distribuées présentées aux chapitres 4 et 5. En particulier, nous avons vu que ces dernières définissent des schémas d'évolution standards pour des groupes d'objets distribués et qu'elles engendrent des ensembles complexes de comportements locaux et d'interactions. La gestion du parallélisme intra-objet et de la synchronisation pose un certain nombre de problèmes spécifiques. Le but de ce chapitre est donc de présenter un langage permettant de mener à bien une telle tâche. Il s'insère dans notre processus de développement en trois niveaux (groupe, objet, méthode) et concerne donc les comportements de niveau objet.

Le langage CAOLAC (acronyme de Conception d'Algorithmes orientés Objet pour Les Applications Coopératives) [Sei96, Sei97b, SDF97] est un formalisme de coordination. Il gère les aspects liés à la concurrence et à la synchronisation. Il doit de ce fait être associé à un langage de base afin de fournir un environnement de programmation complet et opérationnel. Dans le prototype que nous avons réalisé, cette fonction est remplie par le langage objet du système réparti GUIDE [BBD+91]. Néanmoins, les concepts introduits par le langage CAOLAC sont assez généraux pour être adaptés à d'autres langages et/ou à d'autres environnements distribués (par exemple C++ au-dessus d'un bus à objets CORBA). Le langage CAOLAC a pour but de définir, de structurer et de réutiliser des politiques de synchronisation et de coopération complexes. Pour cela, il utilise des modèles états/transitions dont nous proposons une sémantique spécifique. Nous définissons alors deux relations de réutilisation qui permettent d'enrichir ces modèles. La technique suggérée consiste, à partir d'une spécification très générale avec peu de détails, à aboutir étape après étape à des modèles de synchronisation de plus en plus précis comportant de plus en plus de détails.

Le paragraphe 6.1 positionne notre approche de la synchronisation intra-objet par rapport aux formalismes existants. Le paragraphe 6.2 introduit les concepts généraux manipulés par le langage CAOLAC. Le paragraphe 6.3 présente le modèle états/transitions retenu. En particulier nous exposons comment les traitements de synchronisation d'une classe d'objets concurrents peuvent être isolés du reste du code et définis au sein de classes comportementales. Nous avons choisi d'exprimer ces classes de synchronisation à l'aide de modèles états/transitions étendus prenant en compte le parallélisme intra-objet. Nous présentons donc les différentes extensions que nous

introduisons puis nous montrons dans quelle mesure ces classes comportementales peuvent être héritées. Le paragraphe 6.4 présente une évaluation du langage CAOLAC par rapport aux langages ACT++, DRAGOON et GUIDE. Puis, le paragraphe 6.5 propose un bilan de l'implantation du compilateur du langage CAOLAC. Finalement, le paragraphe 6.6 conclut cette présentation.

6.1 Introduction

Le langage CAOLAC permet de définir des politiques de synchronisation pour les différentes méthodes des objets concurrents. Celles-ci permettent de définir des règles de précédence ou de concurrence entre les exécutions des méthodes d'un objet. Cependant, le langage CAOLAC a, au-delà des aspects de synchronisation, un objectif de hiérarchisation et de spécification par raffinements successifs des fonctions réalisées par un objet isolé. Nous présentons cette approche dans le paragraphe suivant, puis nous la comparons, au paragraphe 6.1.2, avec des approches existantes.

6.1.1 Présentation de l'approche

Le concept de politique de synchronisation peut être illustré avec l'exemple simple de la gestion d'un tampon de taille fixe. On considère une structure de données pouvant stocker au plus n éléments d'un type prédéfini. Deux méthodes *Put* et *Get* gèrent cette structure. La première ajoute un élément au tampon tandis que la seconde en retire un. De façon évidente, il apparaît que la méthode *Put* ne peut être exécutée lorsque le tampon est plein et que, réciproquement, la méthode *Get* ne peut être exécutée lorsque le tampon est vide. On est ainsi amené à caractériser le taux de remplissage du tampon à l'aide de trois états : *Vide*, *Partiel* et *Plein*. Dans l'état *Vide*, seule la méthode *Put* peut être exécutée. Dans l'état *Partiel*, les deux méthodes peuvent être exécutées. Finalement, dans l'état *Plein*, seule la méthode *Get* peut être exécutée. Les trois règles de synchronisation énoncées ci-dessus définissent le comportement (ou politique de contrôle) de la structure de données *BufferDeTailleFixe*. Ce comportement est alors associé à une classe non synchronisée qui fournit le code des méthodes *Put* et *Get*, c'est à dire l'ajout et le retrait d'un élément de la structure de données. La figure 6.1 présente sous une forme graphique, les états et les changements d'états opérés par ce comportement. L'objectif d'un langage pour l'expression de la synchronisation est donc, ici, de décrire par un modèle états/transitions le comportement de l'objet.

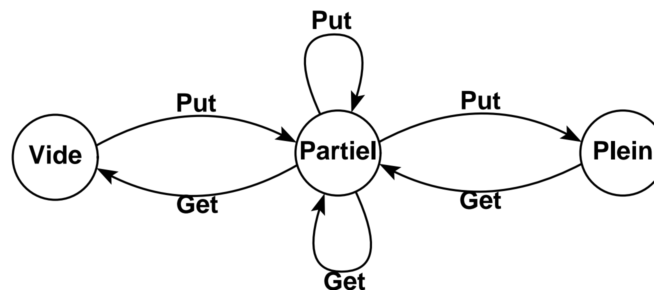


FIG. 6.1 – *Modèle états/transitions du comportement BufferDeTailleFixe*

Au delà de cet exemple simple, le concept de politique de synchronisation recouvre également des comportements très variés (des plus simples aux plus complexes). Prenons pour second exemple

la gestion de variables partagées. Ainsi une classe encapsulant une donnée gérée selon une cohérence forte accepte d'exécuter simultanément soit une opération d'écriture, soit plusieurs opérations de lecture. Le modèle décrivant cette synchronisation comporte trois états : *En attente*, *En lecture* et *En écriture*. Les transitions sont associées aux méthodes acceptables dans chaque état. On constate alors qu'un modèle dans lequel une seule transition est franchissable simultanément à partir d'un état est insuffisant pour décrire les exécutions concurrentes de l'opération de lecture. Le but des extensions au modèle états/transitions que nous proposons est donc d'introduire de nouvelles sémantiques d'états et de transitions afin de décrire des comportements mettant en jeu le parallélisme intra-objet.

6.1.2 Comparaison avec d'autres approches

Notre objectif est de pouvoir décrire des politiques de synchronisation pour des objets concurrents. Pour cela nous avons besoin de contrôler le nombre de méthodes en cours d'exécution au sein d'une instance pour garantir par exemple, un accès exclusif à une section critique. Comme nous l'avons vu au paragraphe 1.1.3 de l'état de l'art, plusieurs mécanismes de synchronisation sont envisageables. Nous écartons les sémaphores et les moniteurs qui présentent l'inconvénient de ne pas séparer les règles de synchronisation des actions à synchroniser. De plus, l'approche à base de clauses comportementales et de compteurs d'événements de GUIDE est trop limitée lorsque la politique de synchronisation est complexe et comporte de nombreux états. De manière générale, les implantations existantes à base d'états comme ACT++ [KL89b] ou Rosette [TS89], présentent l'inconvénient de ne pas séparer clairement les aspects traitements des aspects synchronisation. Finalement, les approches réflexives comme celle d'ABCL/R [WY88], ont un pouvoir d'expression important mais sont souvent lourdes à mettre en place. De plus, nous n'avons pas besoin, au niveau réflexif, de manipuler une abstraction complète de la structure de base de l'objet (*a Causally-Connected Self-Representation* selon les termes de Matsuoka, Watanabe et Yonezawa [MWY91]). Le degré de réflexivité qui nous intéresse concerne essentiellement les aspects dynamiques du mécanisme de prise en compte des invocations. Nous choisissons donc de suivre une voie à mi-chemin entre ces deux dernières solutions : le langage CAOLAC offre un degré de réflexivité identique à celui des protocoles méta-objets comme Open C++ [Chi95], MetaJava [GK97a, GK97b], PC++ [WSMB95] ou CodA [McA95] et adopte une expression de la synchronisation à base d'états.

Par ailleurs, notre approche de la synchronisation diffère de celle des langages à objets concurrents comme Java [GM95], Ada 95 [Bar95, BW95] ou DRAGOON [AGMB91, Atk91]. Les objets synchronisés de Java utilisent des primitives *wait* et *notify* à la manières des moniteurs de Hoare [Hoa74] tandis que les objets protégés d'Ada 95 garantissent un accès exclusif à chaque méthode. Le langage DRAGOON est une extension d'Ada 83. Il introduit une approche objet et prend en compte la concurrence et la distribution. Si, en ce qui concerne la distribution, une partie des propositions de DRAGOON, comme par exemple le concept de nœud virtuel, a été reprise dans Ada 95, les extensions concernant l'approche objet et la gestion de la synchronisation restent différentes. Ainsi, DRAGOON introduit la notion de classe comportementale. Un composant standard comprend alors trois entités : une interface de classe, une implantation (c'est à dire le corps des fonctions et des procédures) et un comportement (c'est à dire le code de synchronisation). Le comportement définit pour chaque procédure de la classe une garde construite à l'aide de fonctions historiques (*history functions*). Ces fonctions, *req*, *act* et *fin*, sont identiques aux compteurs

d'événements du langage GUIDE. Néanmoins, l'écriture d'une politique de synchronisation dans DRAGOON souffre, comme dans tous les langages à base de commandes gardées et de compteurs d'événements, d'une certaine lourdeur lorsque le nombre de situations à prendre en compte est important.

Tout en conservant cette séparation des différents codes, le langage CAOLAC étend cette approche en fondant la synchronisation, non pas sur les gardes associées aux méthodes, mais sur des états représentant différentes étapes du processus de synchronisation. L'objectif d'une telle démarche est double. On souhaite, d'une part, augmenter le niveau de réutilisation du code et, d'autre part, proposer des solutions aux problèmes d'anomalie d'héritage liés à l'introduction de la concurrence dans les langages objet.

6.2 Concepts de base

Le langage CAOLAC fait la distinction entre deux types d'entités : les entités de base définissant les fonctionnalités séquentielles de l'application et les entités dites de niveau méta définissant les fonctionnalités de synchronisation. Les premières sont des classes au sens usuel du terme avec une structure de données et des méthodes. Les secondes sont appelées comportements et possèdent des variables et un code assurant une synchronisation. Ce code est décrit à l'aide d'un modèle états/transitions. Un comportement a donc le statut d'une classe. Il n'est pas employé de façon isolée mais est associé à une classe de base dont il synchronise les méthodes. Les instances de comportement sont désignées sous le terme de méta-objets. Un comportement est dit de niveau méta car il modifie le mécanisme habituel de prise en compte des méthodes d'un objet (par exemple, dans le modèle objet passif, ce mécanisme spécifie que toute méthode peut être invoquée et immédiatement exécutée à tout instant, sans aucune limitation).

6.2.1 Définition d'un comportement

De façon plus détaillée, chaque comportement du langage CAOLAC possède un identifiant unique, hérite, éventuellement, la structure d'un sur-comportement (Cf. paragraphe 6.3.7) et comporte les éléments suivants :

- un ensemble fini de constantes et de variables (cette section commence avec le mot clé `variables`),
- un ensemble fini d'invocations typées (section `invocations`) qui peuvent être prises en compte par le comportement,
- un ensemble fini de méthodes (section `methods`) de l'objet de base qui peuvent être appelées par le comportement,
- un ensemble fini d'états dont l'un d'entre eux est l'état initial (section `initial state`),
- des ensembles finis de transitions associées à chaque état.

L'identificateur permet de désigner de façon univoque le comportement ainsi que tous les éléments qui le composent. La relation d'héritage permet d'enrichir et d'étendre la définition des comportements. Les variables sont utilisées dans les modèles états/transitions pour définir la synchronisation. Les invocations constituent l'interface de communication du comportement. Finalement, les états et leurs transitions associées définissent le modèle de synchronisation.

6.2.2 Exemple de comportement

La figure 6.2 donne la définition du comportement *BufferDeTailleFixe* qui synchronise l'ajout et le retrait d'éléments dans un tampon de taille fixe. Dans cet exemple, aucune variable n'est définie dans la section `variables`. Deux invocations *Put* et *Get*, pour ajouter et retirer un élément, peuvent être prises en compte. La section `methods` déclare deux méthodes *IsFull* et *IsEmpty*, pour tester le taux de remplissage du tampon. Finalement, trois états *Vide*, *Partiel*, *Plein* sont définis. Chaque état définit un ensemble de transitions. La définition des transitions est présentée plus en détails au paragraphe 6.3.4.

6.2.3 Relation méta-comportementale

Comme nous l'avons déjà précisé, les comportements du langage CAOLAC permettent de définir, à un niveau méta, les politiques de synchronisation de classes concurrentes. Il existe donc, entre ces deux entités, un lien particulier que nous appelons relation méta-comportementale et que nous présentons dans le paragraphe suivant. Par ailleurs, ce lien peut également être généralisé et servir d'association entre deux comportements afin de créer des tours méta de synchronisation.

6.2.3.1 Association avec une classe

L'association entre un comportement CAOLAC et une classe du langage de base (dans notre cas GUIDE) se fait statiquement au moment de la compilation. La classe fournit le code séquentiel des invocations prises en compte et des méthodes appelées par le comportement. Elle déclare avec les mots clés `with behaviour` l'identificateur du comportement dont elle utilise la synchronisation.

La séparation entre la synchronisation des méthodes et les traitements effectifs associés à ces méthodes introduit un degré de modularité supplémentaire dans la programmation objet. La partie synchronisation peut ainsi être réutilisée dans différentes classes. Par exemple, le comportement *BufferDeTailleFixe* peut être utilisé aussi bien pour la définition d'une file d'attente que d'une pile. Dans les deux cas, la synchronisation des méthodes *Put* et *Get* est la même : on ne peut enlever un élément si la pile ou la file est vide et on ne peut ajouter un élément si la pile ou la file est pleine. Seule la gestion des ajouts et des retraits change. Dans le cas d'une file, le premier élément ajouté est le premier retiré, tandis que dans le cas d'une pile, le dernier ajouté est le premier retiré. La figure 6.2 donne en exemple le code de la classe *File*.

D'un point de vue statique, l'association entre un comportement et une classe de base fournit une classe finale qui hérite à la fois de la politique de synchronisation et du code effectif des méthodes (Cf. figure 6.3). D'un point de vue dynamique, chaque objet dans l'environnement réparti, c'est à dire chaque instance de classe, est associé à un méta-objet, c'est à dire une instance de comportement (Cf. figure 6.4). Quand un message est envoyé à un objet x, il est d'abord pris en compte par son méta-objet $\uparrow x$ ou par le méta-objet de plus haut niveau dans la tour méta (nous adoptons la même notation que Matsuoka et al dans [MY90] et [MWY91] : $\uparrow$ désigne la relation méta). Le méta-objet applique sa politique de synchronisation au message puis le délivre à l'objet ou au méta-objet inférieur dans la tour méta.

Plusieurs méta-objets, et donc plusieurs politiques de synchronisation, peuvent être définies pour un même objet. Néanmoins, dans la version actuelle du langage CAOLAC, le lien entre un objet et son méta-objet est statique. Il est déterminé au moment de la compilation et ne peut être changé dynamiquement lors de l'exécution. Une extension envisageable consiste à autoriser une

```
behaviour BufferDeTailleFixe {
 variables:

 invocations:
  Put( IN e : REF TElement );
  Get : REF TElement;

 methods:
  IsFull : Boolean;
  IsEmpty : Boolean;

 initial state : Vide;

 state Vide sequential {
  TPut {
   invocation( Put(e) );
   BASE.Put(e); return;
   become( Partiel );
 }}

 state Partiel sequential {
  TPut {
   invocation( Put(e) );
   BASE.Put(e); return;
   if BASE.IsFull = true
   then become( Plein );
   else become( Partiel );
   end;
  }
  TGet {
   invocation( Get );
   return BASE.Get;
   if BASE.IsEmpty = true
   then become( Vide );
   else become( Partiel );
   end;
 }}

 state Plein sequential {
  TGet {
   invocation( Get );
   return BASE.Get;
   become( Partiel );
}}}
```

```
class File
 with behaviour BufferDeTailleFixe
is

 const Max : Integer = 50;
 buf : Array [Max] OF REF TElement;
 ptrEntree, ptrSortie : Integer = 0;

 method Put( IN e : REF TElement );
 begin
  buf[ptrEntree] := e;
  ptrEntree := (ptrEntree+1) mod Max;
 end Put;

 method Get : REF TElement;
  e: REF TElement;
 begin
  e := buf[ptrSortie];
  ptrSortie := (ptrSortie+1) mod Max;
 end Get;

 method IsFull : Boolean;
 begin
  return (ptrEntree-ptrSortie) mod Max
           = Max-1;
 end IsFull;

 method IsEmpty : Boolean;
 begin
  return ptrEntree=ptrSortie;
 end IsEmpty;

end File.
```

FIG. 6.2 – *Définition d'un comportement CAOLAC et d'une classe GUIDE associée*

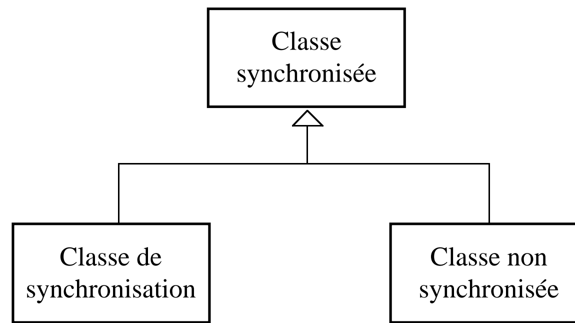


FIG. 6.3 – *Double héritage du code de synchronisation et du code effectif*

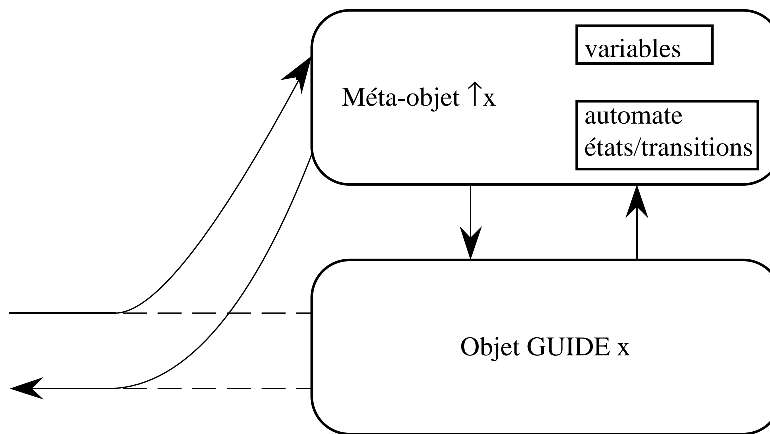


FIG. 6.4 – *Association entre un objet et son méta-objet*

liaison dynamique entre un objet et son méta-objet. De plus, on peut prévoir qu'au moment du changement, et à condition que la transition entre les modes de fonctionnement soit bien contrôlée, un choix soit possible parmi un ensemble de plusieurs méta-objets.

6.2.3.2 Tour méta

La relation méta-comportementale que nous avons présenté jusqu'à présent, définit un lien entre un comportement et une classe de base. Ce lien permet au comportement de synchroniser l'exécution de la classe. On peut généraliser et décider de synchroniser un comportement. On associe, ainsi, par la relation méta-comportementale, un comportement à un autre comportement. La structure créée est désignée sous le terme de tour méta.

La sémantique de la relation méta est la même entre deux comportements que entre un comportement et une classe. Ainsi, quand une invocation est envoyée à un objet, elle est d'abord prise en compte par le méta-objet de plus haut niveau dans la tour. Celui-ci lui applique sa politique de synchronisation, puis la délivre au méta-objet inférieur dans la tour. L'invocation "descend" ainsi, la tour jusqu'à l'objet de base. Les paramètres de retour suivent un chemin inverse avant d'être renvoyés à l'appelant.

Notons que l'association entre deux comportements dans une tour peut se faire dans deux sens : soit le comportement déclare qu'il est associé à un méta (mots clés `with behaviour`), soit il déclare qu'il est un méta d'un autre comportement (mots clés `meta of`). De cette façon, la démarche de

conception de la politique de synchronisation peut se faire, soit de manière descendante, soit de manière ascendante.

6.3 Comportements

Dans cette partie, nous détaillons les différents éléments composant un comportement CAOLAC. En particulier, les paragraphes 6.3.1 et 6.3.2 présentent la définition des données et des interfaces de messages au sein d'un comportement tandis que les paragraphes 6.3.3 et 6.3.4 présentent le modèle états/transitions retenu ainsi que les différentes extensions que nous introduisons. Le paragraphe 6.3.5 fait le point sur les conventions de nommage utilisées dans les comportements CAOLAC. Le paragraphe 6.3.6 présente un ensemble de fonctions prédéfinies désignées sous le terme de compteurs et d'historiques d'événements, qui enregistrent des informations de synchronisation. Finalement, le paragraphe 6.3.7 définit la notion d'héritage de comportements.

6.3.1 Variables

Les variables déclarées au sein d'un comportement CAOLAC représentent des données de synchronisation. Elles sont utilisées dans la définition des modèles états/transitions. Elles peuvent être employées, par exemple, dans les actions associées aux transitions ou dans les clauses d'activation des états. Les variables de synchronisation sont à distinguer des compteurs d'événements présentés au paragraphe 6.3.6. Bien que ces deux types de données enregistrent des informations ayant trait à la synchronisation, les premières sont définies par le concepteur tandis que les secondes sont des fonctions prédéfinies, gérées par l'environnement CAOLAC.

Chaque variable est typée et possède, éventuellement, une expression d'initialisation. Cette expression est évaluée lors du processus d'instanciation. Des constantes peuvent également être définies. La syntaxe de ces déclarations, l'ensemble des types valides ainsi que l'expression d'initialisation dépendent du langage de base associé à CAOLAC. Dans le cas de GUIDE, on retrouve donc les types simples habituels (caractère, entier, réel, booléen), les types construits (chaîne, tableau, enregistrement), les types références, ainsi que le mot clé `const` pour désigner les constantes.

Les variables d'un comportement sont de statut privé. Elles sont visibles à l'intérieur du comportement et dans les comportements hérités (Cf. paragraphe 6.3.7). Elles ne sont pas accessibles par les classes ou comportements extérieurs. Du fait de la relation particulière existant entre un comportement et une classe l'implantant, il peut être nécessaire, dans certaines situations, de partager des variables de synchronisation entre ces deux entités. On obtient alors les deux solutions suivantes :

- la variable de synchronisation n'est pas visible dans la classe implantant le comportement : on respecte ainsi le principe d'encapsulation des variables dans leurs instances.
- la variable de synchronisation est visible dans la classe implantant le comportement : la classe de base peut accéder à cette information de synchronisation. La classe et le comportement fonctionnant de manière concurrente, il est nécessaire de définir la consistance de partage. Différentes politiques peuvent être envisagées : accès en exclusion mutuelle, sérialisation transactionnelle ou séquentielle, linéarisation, etc ...

Ces deux solutions ont chacune des avantages et des inconvénients. La première respecte la philosophie de la programmation objet et contribue à une meilleure lisibilité du code. La seconde

est plus expressive, mais nécessite une maîtrise des politiques de partage parfois complexes. Il nous a semblé plus intéressant, dans la version actuelle du langage CAOLAC, d'opter pour la seconde solution. Néanmoins, par manque de temps et afin de faciliter l'implantation, nous n'avons retenu qu'une seule politique de partage : celle en exclusion mutuelle. Ainsi, toutes les variables de synchronisation sont accessibles par les classes implantant le comportement. L'objet et l'instance de comportement s'exécutent de façon concurrente, mais seule une de ces deux entités accède simultanément à la variable.

6.3.2 Interfaces

Une interface définit la signature des appels qui peuvent être pris en compte par une entité. Elle comprend un ensemble de messages possédant un identificateur et un profil de paramètres. Chaque paramètre est défini par un identificateur, un type et un mode de passage (entrée, sortie, entrée/sortie). Dans le cas du langage CAOLAC, nous avons choisi de définir deux types d'interfaces : une dite externe qui définit les requêtes qui sont traitées par le comportement puis transmises à l'objet de base, et une dite interne qui définit les requêtes de l'objet de base qui peuvent uniquement être invoquées par le comportement. On fournit ainsi le type de la classe de base en séparant les requêtes publiques (interface externe) des requêtes privées (interface interne).

L'interface externe définit les appels qui peuvent être invoqués par une entité externe. Deux types d'éléments sont envisagés pour cette interface : les invocations, annoncées par la section **`invocations`**, et les invocations génériques, annoncées par la section **`generic invocations`**. Les premiers sont des types précis de messages (par exemple, les messages *Put* et *Get* pour le comportement *BufferDeTailleFixe*), tandis que les seconds sont des gabarits de messages permettant de définir des comportements génériques (l'exemple d'une invocation générique désignant indifféremment un message *Put* ou un message *Get* est fourni au paragraphe 6.3.7.4).

L'interface interne, annoncée par la section **`methods`**, définit quant à elle, les méthodes de la classe de base qui peuvent être appelées par le comportement. Les fonctions *IsFull* et *IsEmpty* font, par exemple, partie de l'interface interne.

6.3.3 Etats

Dans ce paragraphe, nous nous intéressons à la notion d'état pour un comportement. En effet, dans notre approche, la spécification d'une politique de synchronisation consiste à indiquer, pour chaque configuration de cette politique, l'ensemble des méthodes exécutables dans cet état. Nous étudions donc, tout d'abord, la représentation d'un état par un prédicat portant sur l'ensemble des variables de synchronisation. Puis, nous présentons, au paragraphe 6.3.3.2, les différentes sémantiques d'état offertes par le langage CAOLAC.

6.3.3.1 Représentation des états

Un état définit une configuration valide rencontrée par un comportement au cours de sa période de bon fonctionnement, c'est à dire en dehors de toute panne ou faute non prévue dans les spécifications. Chaque état modélise une configuration des variables de synchronisation. Il est dit actif s'il représente un point d'avancement atteint par le comportement, et inactif sinon.

Tout d'abord, l'état d'un objet peut être défini par une fonction entre un ensemble de variables d'instances et un ensemble de valeurs. Dans la terminologie objet, les variables d'instances dési-

gnent aussi bien les variables privées accessibles uniquement par les méthodes de l'objet, que les variables publiques accessibles par n'importe quelle méthode. Toutes les variables sont typées et appartiennent à un ensemble fini de valeurs. Les types considérés peuvent être simples, construits ou être des références à tout autre type d'objets. Dans les deux premiers cas, les ensembles de valeurs possibles sont ceux qui sont définis naturellement par les types considérés tandis que, dans ce dernier cas, c'est l'ensemble des références mémoire accessibles dans l'environnement réparti. Par exemple, dans le comportement *BufferDeTailleFixe* les états sont définis par une fonction de l'ensemble $\{ptrEntree, ptrSortie\}$ vers $\mathbb{N} \times \mathbb{N}$. Le couple $(ptrEntree = 1, ptrSortie = 3)$ est alors un état possible. La variable *Max* est une constante. Comme sa valeur ne varie pas, son omission ne modifie pas la fonction d'état.

Le nombre élevé de configurations définies par ce type de fonction s'avère relativement difficile à manipuler dans le cadre d'une spécification de la synchronisation. Plutôt que de manipuler cette fonction, on peut choisir de partitionner l'ensemble des configurations valides pour faire apparaître un nombre raisonnable de macro-états. Chaque macro-état est une classe d'équivalence et représente un sous-ensemble particulier de l'ensemble des configurations. Il se caractérise alors par un prédicat à valeur booléenne. Par exemple, *Vide* regroupe l'ensemble des configurations tel que $ptrEntree = ptrSortie$. De même *Partiel* et *Plein*[1] regroupent les configurations telles que respectivement $(ptrEntree \Leftrightarrow ptrSortie)$ mod $Max < Max \Leftrightarrow 1$ et $(ptrEntree \Leftrightarrow ptrSortie)$ mod $Max = Max \Leftrightarrow 1$. De ce fait, alors que $(ptrEntree = 1, ptrSortie = 1)$ et $(ptrEntree = 3, ptrSortie = 3)$ sont deux états distincts au sens d'une fonction, ils représentent tous les deux l'état *Vide* au sens d'un prédicat puisqu'ils vérifient $ptrEntree = ptrSortie$. Avec une telle approche, l'activité d'un état est définie par la valeur de vérité de son prédicat. Il est alors possible d'envisager des situations dans lesquelles plusieurs états sont actifs simultanément. Par exemple, l'état *UnElement* défini par le prédicat $(ptrEntree \Leftrightarrow ptrSortie)$ mod $Max = 1$ regroupe les situations où le tampon contient un seul élément. On constate alors que, lorsque le prédicat de l'état *UnElement* est vrai, celui de l'état *Partiel* l'est aussi. Dans ce cas, les deux états sont actifs simultanément.

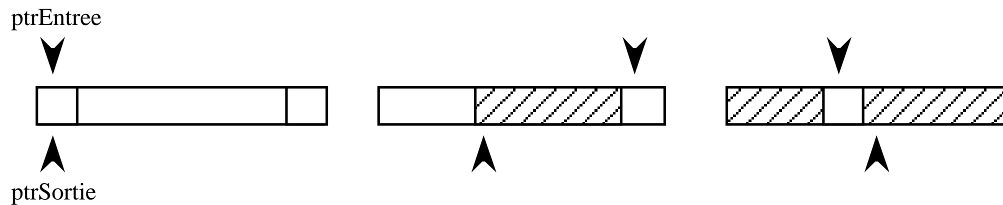


FIG. 6.5 – *Trois configurations, Vide, Partiel et Plein, d'une instance de BufferDeTailleFixe*

Nous retenons la définition à base de prédicats pour les états d'un comportement. En effet, cette approche s'avère plus intéressante lorsque les politiques de synchronisation doivent prendre en compte des exécutions simultanées de méthodes. Par exemple, un objet concurrent remplissant le rôle de serveur de données peut à la fois être en attente de requêtes et en servir d'autres. Il est alors plus aisé de synchroniser de tels comportements à l'aide d'un ensemble d'états définis par des prédicats et éventuellement actifs simultanément plutôt qu'avec une fonction décrivant un ensemble élevé de configurations atteignables.

1. Le mode de gestion du buffer que nous avons choisi permet de stocker *n-1* élements dans un tableau de dimension *n*. En effet lorsque le buffer contient *n-1* éléments, le dernier emplacement libre ne peut être utilisé car alors la valeur de *ptrEntree* serait égale à celle de *ptrSortie*. On ne pourrait plus alors distinguer les cas *Vide* et *Plein*.

6.3.3.2 Sémantiques d'état

Dans les paragraphes précédents nous avons présenté comment la définition d'une politique de synchronisation peut se faire à partir d'un ensemble d'états. Cela revient à spécifier les configurations valides du comportement et à indiquer les méthodes exécutables à partir de chacune d'elles. Nous allons maintenant nous intéresser à la façon dont un comportement évolue entre deux états.

A l'instanciation d'un objet, son méta-objet (et éventuellement son méta-méta-objet, etc ...) est instancié et l'état initial est actif. Pour des comportements simples tels que le tampon de taille fixe, un seul état est actif simultanément. Néanmoins, la prise en compte du parallélisme intra-objet entraîne des situations dans lesquelles plusieurs états sont actifs simultanément. Afin de réaliser de tels scénarios et de gérer les flots d'exécution concurrents au sein d'un comportement, il est nécessaire d'étendre la sémantique habituelle des états issue de la théorie des automates à états finis. Dans cette approche un seul état est actif simultanément. Dès que l'une des gardes des transitions issues de cet état est vraie, l'état est désactivé, la transition est exécutée et l'état conséquent est activé. Si plusieurs transitions sont déclenchables simultanément (c'est à dire si plusieurs gardes sont vraies en même temps), une seule est choisie de manière non déterministe. Nous étendons cette approche en proposant cinq sémantiques d'état qui définissent les contraintes et les possibilités d'évolution que doit respecter le moteur d'exécution interprétant les états. La sémantique d'un état se décompose en un ensemble de règles pour les transitions entrantes et un ensemble de règles pour les transitions sortantes. Deux sémantiques entrantes, *nulle* (pas de mot clé) et *rendez-vous* (mot clé `join`) sont proposées ainsi que trois sémantiques sortantes : *séquentielle* (mot clé `sequential`), *parallèle* (mot clé `parallel`) et *tant que* (mot clé `server`). Dans la suite de ce paragraphe, nous définissons ces sémantiques. Nous en proposons une interprétation en terme de logique temporelle d'actions au chapitre 7.

Sémantiques entrantes

La première sémantique entrante, appelée *nulle*, recouvre l'interprétation habituelle issue de la théorie des automates à états finis pour les transitions entrantes. Un état de ce type est activé dès que l'une des transitions entrantes a été exécutée une fois.

La seconde sémantique entrante est appelée *rendez-vous*. Un état de ce type est activé lorsque toutes ses transitions entrantes ont été exécutées au moins une fois chacune. Ce mécanisme est identique à celui de la primitive *wait* et à ceux des instructions *coend* et *parend*.

Sémantiques sortantes

La première sémantique sortante, appelée *séquentielle*, désigne la sémantique habituelle issue de la théorie des automates à états finis pour les transitions sortantes. Dès que l'une des gardes des transitions issues de cet état est vraie, l'état est désactivé, la transition est exécutée et l'état conséquent est activé. Si plusieurs gardes sont vérifiées simultanément alors la transition à franchir est choisie de manière non déterministe. Ce mécanisme est identique à celui des commandes gardées de CSP ou à celui de l'instruction *select* du langage Ada.

La deuxième sémantique sortante, appelée *parallèle*, permet dès qu'un état est actif, d'examiner toutes ces transitions sortantes. Dès que l'une de ces transitions est déclenchable, elle est exécutée. L'état reste actif tant que chaque transition sortante n'a pas été exécutée exactement une fois. Ce mécanisme est utilisé pour décrire des états dans lesquels sont exécutées des primitives *fork* ou

des instructions *cobegin* ou *parbegin*.

La troisième sémantique sortante est appelée *tant que* ou *serveur*. Dès qu'un état de ce type est devenu actif selon l'une des règles de sémantique entrante précédemment définies, il le reste tant qu'une condition fournie par le programmeur s'évalue à vrai. Cette condition, dite clause d'activation, est une expression booléenne écrite à partir des variables d'instance du comportement, des informations d'historique comme les compteurs d'événements fournis par le langage CAOLAC et des opérateurs booléens habituels. Pendant la période d'activation de l'état, toutes les transitions sortantes, dont les gardes comportent des types de message non nul, sont examinées. Dès que l'une d'entre elles est déclenchable, elle est exécutée. Ce mécanisme est identique à celui d'un objet passif comme ceux du système GUIDE qui déclenche un fil d'activité sur réception d'une requête. Notons que seules les transitions prenant en compte des invocations sont scrutées pendant la période d'activation d'un tel état. En particulier les transitions libres de toute garde ou déclenchables après évaluation d'une condition booléenne sont exclues de ce processus. Ce choix est motivé par le modèle à objets passifs retenu par le système GUIDE. Dans cette approche, le mécanisme d'invocation est la cause première de toute exécution. Contrairement au modèle à objets actifs, on n'envisage pas qu'une activité soit attachée à un objet ou à un comportement dès son instanciation. Ainsi, au sein d'un objet, aucun traitement ne s'exécute de manière indépendante et tout traitement fait suite à une invocation. De ce fait, les transitions libres de toute invocation ne font pas partie du processus de scrutation associé aux états à sémantique *tant que*.

Discussion

Le tableau 6.1 résume les deux sémantiques entrantes et les trois sémantiques sortantes. La sémantique *tant que* définit l'évolution d'un modèle état/transition à partir de conditions sur les données du comportement : elle est dite orientée par les données. Les quatre autres définissent une évolution à partir d'exécutions passées ou à venir de transitions : elles sont dites orientées par le contrôle. Nous n'avons pas retenu de sémantique entrante orientée par les données. Cela signifie que, mis à part l'état initial qui est actif à l'instanciation du comportement, tous les autres états doivent être atteints par un flot d'exécution pour être activés. On pourrait envisager une sémantique entrante orientée par les données qui activerait un état, dès qu'une condition serait vérifiée ou dès qu'un événement arriverait. Cependant, il nous semble qu'une telle caractéristique présente deux inconvénients majeurs :

- elle introduit un caractère asynchrone dans les modèles états/transitions au sens où, une action pourrait être déclenchée en dehors d'une invocation,
- elle impose une scrutation permanente des états à activer.

L'introduction d'une sémantique entrante orientée par les données irait donc d'une part, à l'encontre d'une structuration aussi synchrone que possible de la gestion du contrôle et d'autre part, introduirait de manière implicite un modèle d'objets actifs. Cela irait donc à l'encontre du choix initial d'un modèle synchrone à objets passifs. De plus la scrutation permanente introduirait un coût non négligeable en temps processeur.

La figure 6.6 présente la définition d'états dans le langage CAOLAC avec les six couples possibles, sémantiques entrantes sémantiques sortantes. Chaque état est défini par un identificateur, une sémantique, un prédicat et un ensemble de transitions sortantes. Lorsque la sémantique entrante est omise, le type *nulle* est choisi par défaut. Lorsque la sémantique sortante est omise, le

Sémantiques		Transitions entrantes	Transitions sortantes
contrôle	nulle	une	
	rendez-vous	toutes une fois	
	séquentielle		une
	parallèle		toutes une fois
données	tant que		toutes plusieurs fois

Tab. 6.1 – *Sémantiques d'état du langage CAOLAC*

type *séquentielle* est choisi par défaut. Le prédicat caractérisant l'état est précédé par le mot clé `predicate`. C'est une expression booléenne écrite à partir des variables déclarées dans la section `variables` du comportement. Il est facultatif.

```
behaviour Foo {
  // ...
  state s1 /*sequential est choisi par défaut*/ predicate (aPredicate) {}
  state s2 parallel predicate (aPredicate) {}
  state s3 server while (aCondition) predicate (aPredicate) {}
  state s4 join, sequential predicate (aPredicate) {}
  state s5 join, parallel predicate (aPredicate) {}
  state s6 join, server while (aCondition)
           predicate (aPredicate) {}
  // ...
}
```

Fig. 6.6 – *Sémantiques d'état du langage CAOLAC*

6.3.4 Transitions

Les transitions peuvent être vues comme des commandes gardées de Dijkstra [Dij75, Dij76]. Contrairement aux transitions de la théorie des automates à états finis ou à celles des automates de Harel [Har87], les transitions de notre modèle sont des blocs d'instructions qui ont un début, une fin et une durée d'exécution non nulle. Au sein d'un comportement plusieurs transitions peuvent s'exécuter concurremment. Les transitions définissent des phases entre deux états et sont associées à des méthodes et/ou à des blocs d'instructions. Dans les cas simples, chaque transition est associée à une méthode qu'elle délivre à l'objet de base. C'est le cas, par exemple, dans le comportement *BufferDeTailleFixe*. Néanmoins dans certaines situations plus complexes, le flot d'exécution d'une méthode peut s'étendre sur plusieurs transitions. Par exemple, un bloc particulier d'une méthode peut nécessiter un accès en exclusion mutuelle qui n'est pas nécessaire pour les autres instructions de la méthode. On répartit alors le flot d'exécution sur deux ou plusieurs transitions. Seule la transition associée au bloc critique, est alors synchronisée en exclusion mutuelle.

6.3.4.1 Définition

De façon plus détaillée, chaque transition possède un identifiant unique au sein de l'état et comprend trois types d'informations: une garde, des commandes et un état conséquent.

1. la garde : c'est un couple invocation condition. L'invocation (mot clé `invocation`) spécifie un type d'appel, c'est à dire un identificateur et un profil de paramètres. Le type peut éventuellement être nul. Dans ce cas, la présence d'une invocation n'est pas requise pour le franchissement de la transition. Les types d'invocations valides sont spécifiés dans la section `invocations` de la définition d'un comportement. La condition (mot clé `require`) est une expression booléenne. Elle peut éventuellement être toujours vraie. Les conditions peuvent utiliser les variables définies dans la section `variables` et des compteurs d'événements. Ceux-ci enregistrent des informations de synchronisation comme, par exemple, le nombre de messages en attente, le nombre de méthodes en cours d'exécution ou le nombre d'instances de transitions bloquées sur leur garde.

2. les commandes : c'est un ensemble de structures algorithmiques de base qui peuvent être des opérations élémentaires (lecture, écriture d'une variable), des structures algorithmiques itératives (test, boucle), des changements de contexte (branchement, appel fonctionnel, appel procédural) ou des opérations de communication (envoi de message, appel de procédure distante, communication de groupe). Cet ensemble peut éventuellement être vide.

3. l'état conséquent : c'est l'état qui est activé lorsque les commandes de la transition ont été exécutées (mot clé `become`).

6.3.4.2 Sémantique

Une transition est franchissable si sa garde s'évalue à vraie (dans ce cas elle est dite ouverte). Une garde est ouverte, si la file d'attente contient une invocation de même type que celle spécifiée par l'instruction `invocation`, et si la condition de l'instruction `require` est vraie. Si le type d'invocation et la condition sont omis, alors la garde est toujours ouverte. Si plusieurs invocations de même type sont présentes, alors une des invocations est choisie de manière non déterministe. Dès qu'une transition est franchie, l'invocation est retirée de la file d'attente et les instructions sont exécutées. Toutes les instructions du langage GUIDE sont des commandes valides dans une transition. Le langage CAOLAC fournit les instructions supplémentaires `BASE`, `invocation`, `require` et `become` :

- `BASE` définit une référence à l'objet de base associé au comportement. Elle a le même statut que les références `SELF` ou `SUPER` qui désignent respectivement l'objet courant et l'objet hérité. Elle permet d'appeler une méthode de l'objet de base ou de délivrer un message après synchronisation.

- les instructions `invocation` et `require` définissent les éléments d'une garde. Elles ne peuvent se trouver qu'en tête de transition. `invocation` prend en paramètre un type d'invocation, tandis que `require` accepte une expression booléenne.

- l'instruction `become` définit l'état conséquent d'une transition. Elle peut être associée aux structures algorithmiques du langage de base. Toute instruction placée après l'instruction `become` dans un bloc est un code mort qui ne sera jamais exécuté.

6.3.5 Désignation des éléments d'un comportement

Dans ce paragraphe, nous présentons les conventions de nommage que nous utilisons pour désigner les invocations, les états et les transitions d'un comportement CAOLAC.

Les invocations sont désignées de façon unique par l'identificateur du comportement auquel elles appartiennent, suivi par l'opérateur : :, suivi par leur identificateur. Ainsi, *BufferDeTailleFixe::Get* désigne l'invocation *Get* dans le comportement *BufferDeTailleFixe*. Lorsqu'il n'y a pas de risque de confusion, le préfixage par l'identificateur du comportement peut être omis.

De la même façon que les invocations, les états sont désignés de façon unique par l'identificateur du comportement auquel ils appartiennent, suivi par l'opérateur : :, suivi par leur identificateur. Ainsi *BufferDeTailleFixe::Vide* désigne l'état *Vide* dans le comportement *BufferDeTailleFixe*. Lorsqu'il n'y a pas de risque de confusion, le préfixage par l'identificateur du comportement peut aussi être omis.

Les transitions sont désignées de façon unique par l'identificateur du comportement auquel elles appartiennent, suivi par l'opérateur : :, suivi par l'identificateur de l'état dont elles sont issues, suivi par un point et finalement suivi par leur identificateur. Ainsi *BufferDeTailleFixe::Vide.TPut* désigne la transition *TPut* issue de l'état *Vide* dans le comportement *BufferDeTailleFixe*. Lorsqu'il n'y a pas de risque de confusion, le préfixage par l'identificateur du comportement et/ou de l'état peut être omis.

6.3.6 Compteurs et historiques d'événements

Les compteurs et les historiques d'événements enregistrent des informations sur l'état d'activité d'un comportement et sur ses événements passés. Ce sont des objets prédéfinis gérés par le langage CAOLAC. Ces notions ne sont pas entièrement nouvelles. Par exemple, les compteurs d'événements sont déjà présents dans le langage GUIDE. Ils sont, néanmoins, limités aux seules invocations. Nous les étendons afin de prendre en compte les transitions et les états. En ce qui concerne les historiques d'événements, et bien que la version actuelle du langage CAOLAC soit quelque peu limitée dans ce domaine (notamment en ce qui concerne les fonctions de recherche), aucun autre langage ne les propose, à notre connaissance, de façon standard.

Les compteurs et des historiques peuvent être interrogés à partir de n'importe quelle partie de code définie au sein d'un comportement. Les compteurs enregistrent des informations concernant les invocations, les transitions et les états. Les compteurs sur les invocations et les transitions retournent un nombre d'occurrences tandis que les compteurs sur les états retournent un statut de type booléen. Les historiques d'événements sont, quant à eux, des listes chaînées qui enregistrent les occurrences d'événements au sein des comportements et de leurs objets associés.

6.3.6.1 Compteurs d'événements

Les compteurs d'événements sont au nombre de neuf (Cf. tableau 6.2). Cinq compteurs concernent les invocations, trois les transitions et un compteur est lié aux états.

Les compteurs `INVOKEDINVOCATION`(i), `STARTEDINVOCATION`(i) et `COMPLETEDINVOCATION`(i) enregistrent respectivement le nombre d'occurrences arrivées, commencées et terminées d'une invocation *i*. *i* est l'identificateur d'une invocation dans le comportement courant. Il peut également désigner à l'aide de l'opérateur : : une invocation d'un comportement de niveau supérieur dans une tour méta (Cf. paragraphe 6.3.5). Les compteurs `PENDINGINVOCATION`(i) et `CURRENTINVOCATION`(i)

retournent respectivement le nombre d'occurrences en attente et en cours d'exécution d'une invocation *i*. Ils sont définis à partir des trois compteurs précédents de la façon suivante : *pending*:= *invoked* - *current* et *current*:= *started* - *completed*. Ces cinq compteurs sont identiques, au niveau d'un comportement, aux compteurs *invoked*, *started*, *completed*, *pending* et *current* du langage GUIDE qui s'appliquent à des objets de base.

Comme le traitement d'une invocation n'est pas simplement associé à la réalisation d'une transition, il est apparu souhaitable de permettre la manipulation de compteurs de transitions. Les compteurs `STARTEDTRANSITION`(t) et `COMPLETEDTRANSITION`(t) enregistrent donc, respectivement, le nombre d'occurrences commencées et terminées d'une transition *t*. *t* est l'identificateur d'une transition dans le comportement courant. Il peut également désigner à l'aide des opérateurs `::` et `.` une transition d'un sur-comportement (Cf. paragraphe 6.3.5). Le compteur `CURRENTTRANSITION`(t) retourne le nombre d'occurrences en cours d'exécution d'une transition *t*. Il est défini à partir des deux compteurs précédents de la façon suivante : *current*:= *started* - *completed*.

Le compteur `CURRENTSTATE`(s) retourne un booléen représentant le statut d'activation d'un état *s*.

	Invocations	Transitions	Etats
principaux	`INVOKEDINVOCATION`(i)		
	`STARTEDINVOCATION`(i)	`STARTEDTRANSITION`(t)	
	`COMPLETEDINVOCATION`(i)	`COMPLETEDTRANSITION`(t)	`CURRENTSTATE`(s)
secondaires	`PENDINGINVOCATION`(i)		
	`CURRENTINVOCATION`(i)	`CURRENTTRANSITION`(t)	

TAB. 6.2 – *Compteurs d'événements*

6.3.6.2 Historiques d'événements

De très nombreux algorithmes distribués s'appuient sur l'histoire causale des actions successives conduisant à l'action courante. Cette histoire causale est souvent résumée sous la forme d'horloges logiques ou vectorielles. Elle est aussi traitée, dans d'autres solutions, sous la forme de graphes de liens de causalité. L'analyse *a posteriori* de la trace d'exécution tire, bien sûr, partie de l'enregistrement de l'historique.

Nous avons donc, dans une première version du langage CAOLAC, inclus une gestion automatisée des historiques. Ceux-ci se présentent sous la forme de listes chaînées d'enregistrements gérées dynamiquement et dont les champs contiennent des informations sur chaque événement enregistré. Ces listes sont gérées par un objet prédéfini désigné par le mot clé `HISTORY`. Celui-ci est associé à chaque objet contrôlé par un comportement CAOLAC. Il peut être interrogé pendant toute la durée de vie de l'objet afin, par exemple, d'orienter le contrôle en fonction des occurrences passées d'événements. Les types d'événements pris en compte dans les historiques concernent :

- les événements d'arrivée, de début et de fin des invocations prises en compte par le comportement et des méthodes de l'objet de base appelées par le comportement,
- les appels séquentiels et parallèles de méthodes,
- les opérations de lecture et d'écriture de variables.

Afin de limiter la taille des historiques, le langage CAOLAC offre la possibilité de restreindre le nombre d'événements enregistrés. Ainsi, seuls les événements des invocations, méthodes et variables marquées pour l'enregistrement à l'aide du mot clé `TRACED`, sont enregistrés. Dans la version actuelle du langage CAOLAC, l'objet prédéfini `HISTORY` fournit, essentiellement, une seule opération d'extraction de données dans l'historique. Cette opération, nommée `SEARCHFORLASTEV`, permet de rechercher le dernier événement exécuté dans une invocation ou une méthode.

6.3.6.3 Attendus sur les compteurs et les historiques d'événements

Les primitives de manipulation de compteurs d'événements concernant des méthodes ou des actions sont, maintenant, anciennes. Un consensus est assez largement établi quant au jeu des objets compteurs et de leurs primitives associées.

Le cas des historiques est relativement différent. Si de nombreuses solutions d'algorithmes distribués utilisent des historiques, leur mise en œuvre de façon systématique et automatisée est, à notre connaissance, encore rare. En fait, le seul cas connu concerne le déverminage en approche répartie où l'usage d'un observateur permet le rejeu d'une exécution.

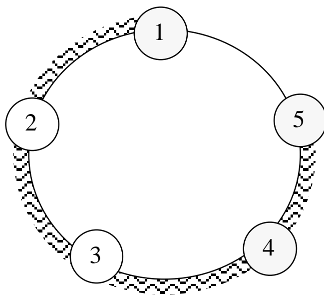
Le choix des primitives de manipulation de manipulation des compteurs et historiques est donc, dans une première version, assez contingent des exemples traités. D'autres travaux pourront préciser les besoins et affiner les ensembles d'objets d'historiques et leurs primitives de manipulation.

6.3.6.4 Exemple d'utilisation

Dans ce paragraphe, nous illustrons l'utilisation des compteurs et des historiques, à l'aide d'un algorithme d'élection sur un anneau. Les modèles de synchronisation sont fournis, sous forme graphique et dans la syntaxe CAOLAC, aux figures 6.8 et 6.9. Cet algorithme a pour but d'élire un leader sur un réseau dont la topologie est celle d'un anneau. Chaque site est identifié de façon unique et héberge un objet qui participe à l'algorithme. La version proposée utilise deux vagues qui se propagent en sens inverse (Cf. figure 6.7). L'initiateur déclenche simultanément deux vagues, une vers son voisin droit et une vers son voisin gauche. Les vagues se propagent récursivement de site en site, et refluent lorsqu'elles se rencontrent. Nous omettons volontairement certains détails de l'algorithme.

Les compteurs d'invocations servent, en général, à contrôler le nombre d'invocations exécutables par un comportement. Par exemple, dans l'algorithme précédent, il faut, afin de garantir la cohérence du résultat, qu'il n'y ait qu'une élection simultanément sur l'anneau. En appelant *Election* l'invocation qui réalise ce traitement, cette condition s'exprime en spécifiant que, pour qu'une instance d'*Election* soit acceptée, il faut qu'aucune autre instance d'*Election* ne soit en cours d'exécution. Nous utilisons pour cela le compteur *CurrentInvocation*. Ainsi, la transition *Repos.t1* figure 6.9, prend en compte les invocations de type *Election* et comprend dans sa garde (clause `require`) la condition `CurrentInvocation(Election) = 0`.

Les compteurs de transitions vont servir, par exemple, à assurer une exclusion mutuelle entre deux blocs de codes. Chaque bloc est associé à une transition et les gardes de ces dernières spécifient qu'aucune instance de transition (compteur *CurrentTransition*) ne doit être en cours d'exécution pour déclencher la transition. Une autre utilisation intéressante de ces compteurs concerne le comptage des numéros d'époque dans les algorithmes phasés. Par exemple, dans l'algorithme précédent, on peut considérer que tout lancement d'élection définit une époque nouvelle dans l'évolution du réseau. Si le traitement global correspondant à l'élection est représenté par une



Le site 1 est initiateur

Une exécution possible donne par exemple :

- la vague dextrogyre (qui tourne vers la droite) atteint les sites 1, 5 et 4
- la vague levogyre (qui tourne vers la gauche) atteint les sites 1, 2, 3, 4 et 5

FIG. 6.7 – *Algorithme d'élection sur un anneau par vagues contra-rotatives*

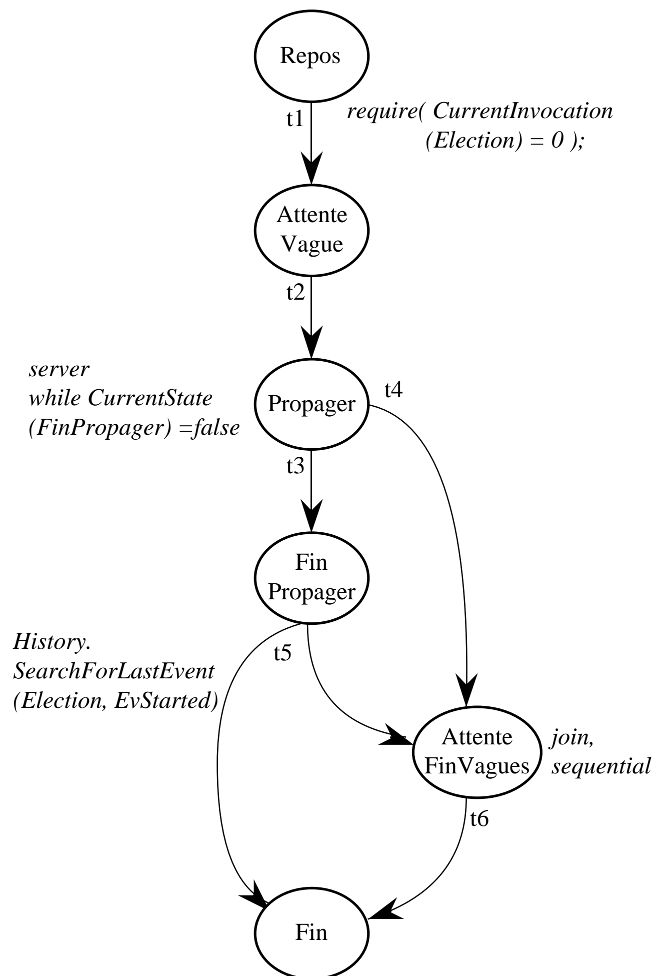


FIG. 6.8 – *Modèle états/transitions de l'algorithme d'élection*

```
// Modèle états/transitions de l'algorithme d'élection
// sur un anneau par vagues contra-rotatives

behaviour AlgoElec {

 invocations:
   Election;
   Vague;

 initial state: Repos;
 state Repos {
  t1 {
   invocation( Election );
   require( CurrentInvocation(Election) = 0 );
   // pas plus de 2 élections simultanées
   become( AttenteVague );
 }}

 state AttenteVague {
  t2 {
   invocation( Vague );
   // une proposition de vague
   become( Propager );
 }}

 state Propager server while ( CurrentState(FinPropager) = false ) {
  t3 {
   // propager la vague courante
   become( FinPropager );
  }
  t4 {
   invocation( Vague );
   // une proposition en sens inverse
   // retourner cette proposition
   become( AttenteFinVagues );
 }}

 state FinPropager {
  t5 {
   if History.SearchForLastEv(Vague,EvStarted).NumOrder # CurrentNumOrder
   then become( AttenteFinVagues );
   else become( Fin );
   end;
 }}

 state AttenteFinVagues
     join, sequential {
  t6 {
   become( Fin );
}}}
```

FIG. 6.9 – *Modèle CAOLAC de l'algorithme d'élection*

phase, et donc par une transition du modèle états/transitions, alors le numéro d'époque correspond au nombre d'instances commencées de cette transition.

Les compteurs d'états sont utilisés, en général, pour tester si le comportement a atteint, ou non, un certain point d'avancement. Par exemple, dans l'algorithme précédent, après une première prise en compte de vague, les propositions ultérieures sont acceptées tant que la vague courante est en cours de propagation. La propagation de la vague initiale est réalisée par la transition *Propager.t3*, la fin de cette propagation est réalisée lorsque l'état *FinPropager* est atteint et la prise en compte d'une proposition ultérieure se fait par la transition *Propager.t4*. L'état *Propager* est donc actif et accepte les propositions ultérieures (transition *Propager.t4*), tant que l'état *FinPropager* n'a pas été atteint. Cela s'exprime, avec le compteur *CurrentState*, par un état à sémantique *tant que* associé à la condition suivante : `server while ( CurrentState(FinPropager) = false )`.

Finalement, les historiques d'exécution sont utilisés, en général, pour tester si un événement quelconque a, ou non, eu lieu. Par exemple, dans l'algorithme précédent, une fois la propagation de la vague initiale terminée, il est nécessaire de savoir si une vague circulant en sens inverse a été reçue. En effet, si c'est le cas, il faut attendre que le traitement de cette seconde vague soit terminé pour retourner à l'état *Repos*, tandis que dans le cas contraire, cela peut se faire immédiatement. Le code de la transition *FinPropager.t5* recherche donc, dans l'historique des événements, la dernière occurrence d'arrivée de l'invocation *Vague* à l'aide de l'appel `History.SearchForLastEv(Vague,EvStarted)`. Pour chaque événement enregistré, le système stocke, entre autres, la méthode et le numéro d'ordre (c'est à dire le numéro de l'instance de cette méthode) auquel il appartient. Si le numéro d'ordre est différent du numéro d'ordre de la méthode courante (`CurrentNumOrder`), alors cela signifie qu'une nouvelle invocation de la méthode *Vague* est arrivée depuis que l'on a commencé l'invocation courante. On obtient donc le test recherché.

En résumé, les compteurs et les historiques d'événements ont un pouvoir d'expression puissant permettant de contrôler finement l'évolution d'un objet. Alors que les modèles états/transitions définissent un contrôle orienté par le flot d'exécution, les compteurs et les historiques introduisent, quant à eux, un contrôle plus orienté par les données. Nous pensons donc qu'ils apportent un complément utile aux modèles de contrôle états/transitions du langage CAOLAC.

6.3.7 Héritage de comportements

La définition d'une politique de synchronisation correcte est l'une des tâches les plus ardues de la programmation concurrente. Une démarche consistant à définir de façon progressive et incrémentale le contrôle permet de limiter les difficultés introduites par ce problème. Par ailleurs un des bénéfices attendus de l'approche objet est d'augmenter le niveau de réutilisation du code. L'objectif de notre langage est donc, grâce à une programmation modulaire des politiques de synchronisation, d'apporter des réponses à ces deux problèmes.

La technique suggérée consiste, à partir d'une spécification très générale avec peu de détails, à aboutir étape après étape à des modèles de synchronisation de plus en plus précis comportant de plus en plus de détails. On espère ainsi faire apparaître, au fur et à mesure, les différents éléments de synchronisation qui participent à la réalisation de la politique globale. Les objectifs de cette démarche sont doubles :

- la hiérarchie de modèles issue du processus de raffinement doit faciliter l'exposé des différents éléments de synchronisation. Cet aspect est important aussi bien pour le concepteur qui doit structurer son raisonnement que pour le tiers qui doit le comprendre.

– le processus de validation des spécifications doit être plus aisé. Ce travail est en général conduit selon deux axes complémentaires : la preuve et le test. Lorsque les comportements sont spécifiés mathématiquement à l'aide de méthodes formelles (comme, par exemple, dans les approches CCS [Mil80], CSP [Hoa85], VDM [Jon86], Lotos [ISO87a], Estelle [ISO87b] ou B [Abr96]), des propriétés formelles sont prouvées. Parallèlement à cette approche de preuve, des scénarios de test sont définis afin de valider l'évolution du modèle face à des situations types. Dans les deux cas, la modularité introduite par la hiérarchie de raffinements facilite la vérification incrémentale des propriétés de chaque composant.

Comme nous l'avons vu précédemment un comportement est défini par un ensemble fini d'états et de transitions et gère des variables de synchronisation. Un état représente un point d'avancement dans l'évolution de l'exécution. Les transitions définissent des phases entre les états et sont associées à des blocs d'instructions qui sont exécutés lorsque les transitions sont tirées. A partir d'un tel ensemble d'éléments, il est possible de dériver un comportement en une ou plusieurs versions. Le comportement obtenu hérite ainsi de toutes les caractéristiques (variables, interfaces, états, transitions) de son ancêtre. Certains éléments peuvent alors être ajoutés ou modifiés.

6.3.7.1 Définition

La relation sous-comportementale permet, à un comportement fils, de réutiliser et de modifier les états et les transitions d'un comportement père. Les ensembles d'états et de transitions définis dans le comportement père sont dupliqués dans le comportement fils. Des états peuvent être ajoutés, d'autres peuvent être redéfinis ou partitionnés. Des transitions peuvent être ajoutées et d'autres peuvent être redéfinies. Cette réutilisation est de type boîte blanche (en effet un comportement fils a accès aux éléments de son comportement père), et est à peu près identique à l'héritage de classe. C'est un mécanisme d'héritage simple. Le support de l'héritage multiple imposerait de définir une règle de composition des modèles hérités, et n'est pas pris en compte par la version actuelle du langage CAOLAC.

La figure 6.10 présente le comportement *SousComp* qui est un sous-comportement de *SurComp*. L'état *s1* et toutes ses transitions sont héritées dans *SousComp*. L'état *s2* est hérité et redéfini. Sa sémantique (`parallel`) reste inchangée. La transition *t2* est héritée et la transition *t6* redéfinit la transition *t3*. L'état *s4* et toutes ses transitions sont ajoutés. Nous détaillons dans la suite de ce paragraphe les mécanismes d'héritage d'états et de redéfinition de transitions.

6.3.7.2 Héritage d'états

Tous les états d'un sur-comportement sont hérités par ses sous-comportements. La sémantique, le prédicat et toutes les transitions des états sont hérités. Ces éléments peuvent éventuellement être redéfinis dans le comportement fils. En plus de ce mécanisme d'héritage et de redéfinition, des états hérités peuvent être partitionnés, et des états supplémentaires peuvent être ajoutés. Un état partitionné redéfinit et remplace un état d'un sur-comportement. Il est défini par l'identifiant de son état père et par un ensemble fini de partitions. Chaque partition est caractérisée par une expression arithmétique et par un état simple appelé sous-état. Au sein d'un comportement, les identifiants de sous-états doivent être uniques et différents de ceux des états. Le principe général du partitionnement est le suivant : toute activation du sur-état est remplacée par l'activation d'exactement un sous-état. Le choix du sous-état à activer est déterminé par la valeur des expressions arithmétiques associées à chacun d'entre eux. De façon théorique, le partitionnement doit

```
behaviour SurComp {
 initial state : s1;

 state s1 sequential {
  t1 { /*...*/ become(s2); }
 }

 state s2 parallel {
  t2 { /*...*/ become(s3); }
  t3 { /*...*/ become(s3); }
 }

 state s3 join,
         sequential {
  t4 { /*...*/ become(s2); }
}}
```

```
behaviour SousComp
  subbehaviour of SurComp {

 state s2 parallel {
  t5 redefines t3
   { /*...*/ become(s4); }
 }

 state s4 sequential {
  t6 { /*...*/ become(s3); }
}}
```

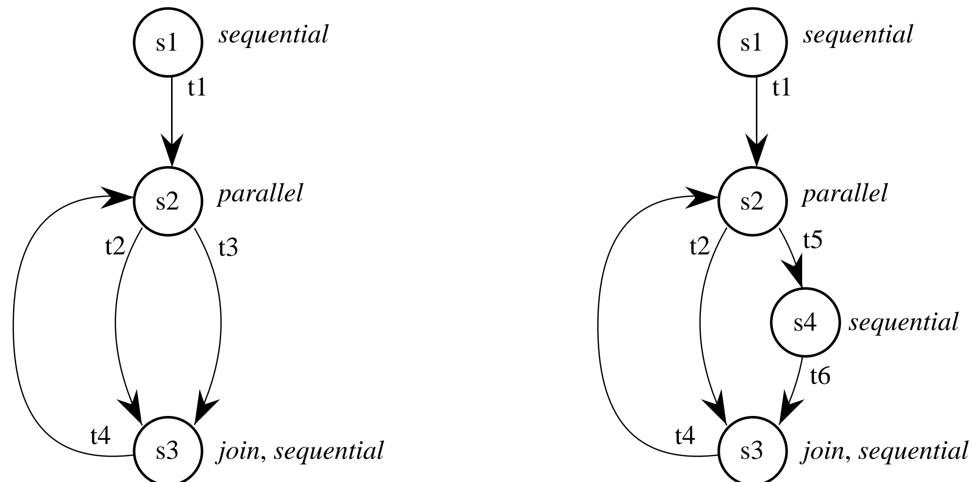
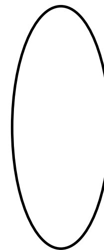


Fig. 6.10 – *Relation d'héritage sous-comportementale*

```
state un_état_du_parent {
 expr1 : state s1 { /*...*/ }
 expr2 : state s2 { /*...*/ }
 // ...
 exprn : state sn { /*...*/ }
 else  : state se { /*...*/ }
}
```

Etat père dans un
sur-comportement



Etat fils
partitionné

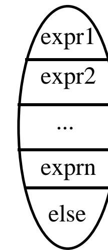


Fig. 6.11 – *Partitionnement d'état*

être complet et les partitions ne doivent pas se chevaucher. En d'autres termes, il ne faut pas que, lors de l'activation du sur-état, aucune expression de partitionnement ne s'évalue à vrai, ni que plusieurs expressions de partitionnement s'évaluent à vrai simultanément.

Nous proposons ici un style de programmation permettant, de façon systématique, de générer de façon correcte un schéma de partitionnement en traitant les différents cas de façon exhaustive. Ainsi, la complétude du partitionnement peut être assurée, en particulier, par la présence obligatoire d'un sous-état précédé par la clause `ELSE`. Ce sous-état n'a pas d'expression arithmétique et est activé lorsqu'aucune autre expression de partitionnement ne s'évalue à vrai. Le chevauchement des partitions peut être résolu par une évaluation séquentielle des expressions de partitionnement. Elles sont évaluées selon l'ordre de définition des sous-états. Si plusieurs expressions s'évaluent à vrai simultanément, le sous-état atteignable évalué en premier est activé. La figure 6.11 présente l'exemple d'un état *parent* partitionné en *n+1* sous-états. Toute instruction `become(un_état_du_parent)` du nouveau comportement est remplacée par un bloc tel que :

```
if expr1 then become(s1);
else if expr2 then become(s2);
     else ...
          if exprn then become(sn);
          else become(se);
          end;
     end;
end;
```

6.3.7.3 Héritage de transitions

Au sein d'un état redéfini, les transitions peuvent être héritées ou redéfinies. Une transition fille redéfinit une transition mère, si elles possèdent le même identifiant, ou si la transition enfant désigne l'identifiant de la transition parent à l'aide du mot clé `redefines`. Les instructions de la transition fille remplacent celles de la transition mère. On peut redéfinir tout ou une partie d'un modèle états/transitions par un ou plusieurs blocs de transitions. Ainsi, dans l'exemple de la figure 6.10, la transition *t3* est remplacée par le bloc composé par les transitions *t5* et *t6*. Les compteurs d'événements associés aux transitions du modèle père continuent à être accessibles dans les modèles fils. Ainsi, chaque fois que la transition *SousComp::s2.t5* est déclenchée, CAOLAC incrémente les compteurs *StartedTransition* associés à cette transition et à la transition redéfinie *SurComp::s2.t3*. De même, lorsque la transition *SousComp::s4.t6* s'achève, les compteurs *CompletedTransition* associés à cette transition et à la transition mère *SurComp::s2.t3* sont mis à jour. Ce mécanisme permet de conserver, dans un comportement de bas niveau comportant de nombreux détails, des informations de synchronisation liées à la structure d'un comportement père plus général. Il permet également, dans l'optique d'un processus de déverminage, d'offrir différents niveaux d'interrogation des comportements.

6.3.7.4 Anomalie d'héritage

Le terme anomalie d'héritage (*inheritance anomaly* en anglais) désigne les limitations liées à l'introduction de la concurrence dans un langage objet supportant l'héritage. Des études précédentes ont montré que le code de synchronisation ne peut pas toujours être réutilisé sans redéfinitions. Une présentation de ce problème est proposée au chapitre 1. Une étude complète se trouve

dans les travaux de Matsuoka et Yonezawa [MY93] et de nombreuses solutions sont proposées dans la littérature [KL89a, TS89, AvdL90, GW91, Frø92]. En particulier, il a été montré que les trois causes principales de l'anomalie d'héritage sont l'introduction d'une prise en compte de l'historique des exécutions non prévues, le partitionnement des états acceptables et la modification des états acceptables.

La première anomalie d'héritage se manifeste lorsque l'historique des exécutions de méthodes est un critère déterminant pour accepter ou non une invocation. Nous utilisons, dans ce cas, la notion de méta-comportement du langage CAOLAC afin d'automatiser l'enregistrement de cet historique. La seconde anomalie concerne le partitionnement des ensembles d'états acceptables. Un état acceptable pour une méthode est un état de l'objet dans lequel la méthode peut être invoquée. La redéfinition d'un état en plusieurs sous-états modifie le caractère acceptable ou non d'une méthode dans cet état. Nous résolvons cette anomalie en utilisant la relation d'héritage sous-comportementale et la notion de sous-état du langage CAOLAC. Nous sommes en mesure de déterminer systématiquement, à partir des expressions de partitionnement, si une méthode héritée est acceptable ou non dans les nouveaux sous-états. Finalement, la troisième anomalie concerne la modification des états acceptables. Elle survient lorsqu'une ou plusieurs méthodes modifient les ensembles d'états acceptables d'une ou plusieurs autres méthodes. Nous la résolvons à l'aide d'un méta-comportement qui définit les nouvelles configurations introduites par les méthodes ajoutées.

Dans la suite de ce paragraphe, nous fournissons plus de détails sur ces différentes solutions et nous les illustrons à l'aide du comportement *BufferDeTailleFixe*.

Historique des exécutions

La première anomalie d'héritage provient de l'historique des exécutions. Elle apparaît lorsqu'un comportement fils a besoin, pour accepter ou non une invocation de méthode, d'informations sur l'historique des méthodes d'un comportement père. Par exemple, le comportement *BufferDeTailleFixe* peut être étendu avec l'invocation *Gget* qui est acceptable lorsque la précédente invocation est de type *Get* (et donc, qui ne l'est pas lorsque la précédente est un *Put*). Le nouveau comportement *BufferDeTailleFixe1* (Cf. figure 6.12) ne peut déterminer si l'invocation *Gget* n'est acceptable que s'il connaît les précédentes exécutions de méthodes. Avec un langage de programmation habituel, toutes les méthodes intervenant en tant que critère de sélection d'une autre méthode doivent être redéfinies ou surchargées afin d'ajouter l'enregistrement de leur début et de leur fin. Dans notre approche, cet enregistrement peut s'exprimer, soit en utilisant l'historique des événements présentés au paragraphe 6.3.6, soit en définissant un méta-comportement. Cette dernière solution met en place, sans modifier la synchronisation initiale, un point de vue différent sur le contrôle qui permet de synchroniser la méthode *Gget*.

Ainsi, le comportement *BufferDeTailleFixe1* est un méta-comportement de *BufferDeTailleFixe*. Il définit deux états : *GgetOff* et *GgetOn*. Dans le premier, la transition *GgetOff.TGet* est tirée chaque fois qu'une invocation *Get* est prise en compte. L'invocation est transmise au niveau inférieur suivant dans la tour méta (instruction `BASE.Get`), et l'état *GgetOn* est activé. Dans cet état, les invocations *Gget* et *Get* sont associées à une transition et peuvent donc être exécutées. Dans ce comportement, l'invocation *Put* n'est pas traitée. Elle est transmise, systématiquement et sans modification, au comportement *BufferDeTailleFixe*.

```
behaviour BufferDeTailleFixe1
 metabehav of BufferDeTailleFixe {

invocations:
 Gget : REF TElement;

initial state : GgetOff;
state GgetOff sequential {
 TGet {
  invocation( Get );
  return BASE.Get;
  become( GgetOn );
}}

state GgetOn sequential {
 TGget {
  invocation( Gget );
  return BASE.Get;
  become( GgetOff );
 }
 TGet {
  invocation( Get );
  return BASE.Get;
  become( GgetOn );
}}}
```

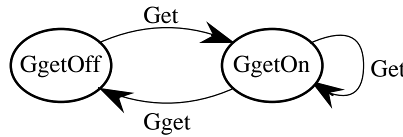


FIG. 6.12 – *Prise en compte des historiques d'exécution pour l'invocation Gget*

```
behaviour BufferDeTailleFixe2
 subbehav of BufferDeTailleFixe {

invocations:
 Get2 : REF TElement;
methods:
 IsOne : Boolean;

state Plein sequential {
 TGet2 {
  invocation( Get2 );
  BASE.Get;
  return BASE.Get;
  become( Partiel );
}}

state Partiel {
 BASE.IsOne = true :
  state Un sequential {}
 else :
  state PlusDUn sequential {
   TGet2 {
    invocation( Get2 );
    BASE.Get;
    return BASE.Get;
    if BASE.IsEmpty = true
    then become( Vide );
    else become( Partiel );
    end;
}} }}
```

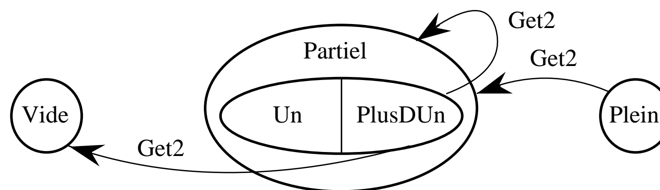


FIG. 6.13 – *Partitionnement des états acceptables par l'invocation Get2*

Partitionnement des états acceptables

La seconde anomalie d'héritage se manifeste lorsqu'un sous-comportement partitionne un ou plusieurs états d'un sur-comportement. Par exemple, on peut vouloir étendre le comportement *BufferDeTailleFixe* avec une invocation *Get2* pour retirer deux éléments simultanément dans le tampon. Le nouveau comportement, noté *BufferDeTailleFixe2*, doit synchroniser les deux autres requêtes de la même façon qu'auparavant, et doit spécifier que l'invocation *Get2* n'est pas acceptable lorsque le tampon est vide ou n'a qu'un seul élément. Cela introduit deux nouveaux états *Un* et *PlusDUn* qui définissent une partition de l'ancien état *Partiel*. Avec une approche habituelle, comme celle du langage C++ par exemple, le partitionnement impose une redéfinition du code de synchronisation de toutes les méthodes acceptables dans l'ancien partitionné (donc, dans cet exemple, une redéfinition des méthodes *Get* et *Put*). Avec notre approche, ce type de partitionnement s'exprime directement au niveau du langage, par des sous-états. Les formules de partitionnement permettent de réexaminer automatiquement les changements d'états du comportement hérité.

La figure 6.13 définit le comportement *BufferDeTailleFixe2* comme une extension de *BufferDeTailleFixe*. Il accepte une nouvelle invocation *Get2* et nécessite une nouvelle méthode de base *IsOne* pour tester si le tampon contient un seul élément. L'état *Vide* et la transition *Vide.TPut* sont hérités ainsi que l'état *Partiel* et les transitions *Partiel.TPut* et *Partiel.TGet*. Deux partitions de l'état *Partiel* sont définies : si la fonction *IsOne* retourne vrai alors le nouvel état *Un* est activé sinon c'est le nouvel état *PlusDUn* qui l'est. L'état *Un* n'introduit pas de nouvelle transition, tandis que *PlusDUn* ajoute la transition *PlusDUn.TGet2* pour servir les invocations *Get2*. Les transitions héritées *Partiel.TPut* et *Partiel.TGet* sont dupliquées pour chacun des deux nouveaux états. Les expressions de partitionnement de l'état *Partiel* permettent de déterminer le sous-état à activer en lieu et place de l'ancien état *Partiel*. Ainsi, toutes les instructions `become(Partiel)` du modèle hérité sont remplacées dans *BufferDeTailleFixe2* par :

```
if BASE.IsOne = true then become(Un); else become(PlusDUn); end;
```

L'état *Plein* et la transition *Plein.TGet* sont hérités. Une transition *Plein.TGet2* est ajoutée pour servir les invocations *Get2* à partir de cet état.

Modification des états acceptables

La troisième anomalie d'héritage concerne la modification des états acceptables d'un comportement. Elle se produit lorsque l'espace d'états d'un comportement père est modifié pour définir un nouvel ensemble d'états acceptables. Par exemple, deux primitives *Lock* et *Unlock* peuvent être définies pour le comportement *BufferDeTailleFixe*. Quand une primitive *Lock* est invoquée, toutes les invocations suivantes sont bloquées. Les ensembles d'états acceptables pour les invocations du comportement fils deviennent vides. Avec un langage de programmation habituel, toutes les méthodes dont les ensembles d'états acceptables sont modifiés doivent être redéfinies ou surchargées. Dans le langage CAOLAC, ces modifications sont traitées comme des configurations spéciales d'un méta-comportement.

Ainsi, le nouveau comportement *BufferDeTailleFixe3* (Cf figure 6.14) est un méta-comportement de *BufferDeTailleFixe*. Deux états *Locked* et *Unlocked* sont définis. Dans le premier, la seule transition franchissable est associée à une invocation *Unlock*. Dans le second état, toutes les invocations sont acceptables et sont délivrées au niveau inférieur dans la tour méta. On utilise une invocation générique *Any* pour désigner indifféremment une invocation *Put* ou une invocation

```
behaviour BufferDeTailleFixe3
 metabehav of BufferDeTailleFixe {

generic invocations:
 Any;
invocations:
 Lock;
 Unlock;

initial state : Unlocked;
state Locked {
 TUnlock {
  invocation( Unlock );
  return;
  become( Unlocked );
}}

state Unlocked server
 while (CurrentState(Locked)=false) {
 TLock {
  invocation( Lock );
  return;
  become( Locked );
 }
 TAny {
  invocation( Any );
  return BASE.Any;
  become( Unlocked );
}}}
```

```
class File with behaviour BufferDeTailleFixe3
          where Any => Put(e), Get is
  // ...
end File.
```

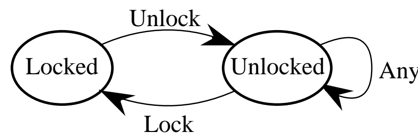


FIG. 6.14 – *Modification des états acceptables par les invocations Lock et Unlock*

Get. *Any* est associée à ces invocations au moment de la déclaration de la classe synchronisée.

6.4 Evaluation du langage CAOLAC

Le langage CAOLAC permet de définir des politiques de synchronisation pour des objets concurrents. Il intègre dans un même formalisme deux styles de programmation: un premier, issu des démarches à base d'états et de transitions dans la lignée du langage ACT++ [KL89a, KL89b] et un second, issu des démarches à base de compteurs d'événements comme celle du langage GUIDE [BBD+91].

6.4.1 Approche états/transitions

Le langage CAOLAC est comparable au langage ACT++ au sens où chaque classe est contrôlée par un ensemble d'états. Dans ACT++, cet ensemble d'états est appelée *abstraction comportementale* et se définit (Cf. figure 6.15), à l'aide du mot clé `behavior`, d'un ensemble d'états (ici *vide*, *partiel* et *plein*), et de l'ensemble des méthodes exécutables dans chaque état (ici, par exemple, seule la méthode *get* est exécutable dans l'état *plein*), à l'intérieur même de la classe. Ceci constitue une première différence avec notre approche: dans CAOLAC, les abstractions comportementales

sont définies à l'extérieur de la classe. Cela nous semble faciliter la réutilisation de comportements puisque les abstractions comportementales n'ont pas besoin d'être réécrites lorsque deux classes différentes utilisent la même politique de synchronisation. De ce point de vue, CAOLAC est plus proche du langage DRAGOON [AGMB91, Atk91] dans lequel les classes synchronisées sont construites par héritage d'une classe séquentielle non synchronisée et d'une classe de synchronisation utilisant des compteurs d'événements.

```
class buf: ACTOR {
 behavior:
  vide = {put};
  partiel = {put, get};
  plein = {get};
public:
  void buf() { /* ... */ }
  void get() { /* ... */ }
  void put() { /* ... */ }
}
```

FIG. 6.15 – *Classe du langage ACT++*

Dans ACT++, l'abstraction comportementale comprend l'ensemble des méthodes exécutables à partir de chaque état. Dans CAOLAC, la notion de transition existe de façon explicite. Ainsi, chaque transition est associée à une garde qui comprend la méthode exécutable et une condition booléenne. Cette dernière permet d'affiner la condition de déclenchement de la transition. De plus, dans CAOLAC, l'exécution d'une méthode n'est pas limitée à une seule transition. Par exemple, une méthode peut être découpée en trois blocs, associés chacun à une transition. Selon les besoins, le bloc correspondant à la seconde transition peut, par exemple, s'exécuter en exclusion mutuelle, tandis que les deux autres s'exécutent en concurrence avec les autres méthodes.

Finalement, l'expression de concurrence intra-objet est facilitée dans les modèles états/transitions de CAOLAC par la présence de différentes sémantiques d'états. Ainsi, il est possible de spécifier, qu'à partir d'un état, plusieurs transitions peuvent être déclenchées et donc, exécutées en parallèle. Ce type de comportement n'est pas, à notre connaissance, possible dans ACT++.

6.4.2 Approche compteurs d'événements

Le modèle de synchronisation du langage GUIDE est fondé sur la notion de clause comportementale et de compteur d'événements. Une clause comportementale est une garde pour l'exécution d'une méthode, tandis que les compteurs d'événements résument l'état d'activité de toutes les méthodes de l'objet. Les modèles de synchronisation de GUIDE et de CAOLAC sont équivalents. Ainsi, la traduction littérale d'une synchronisation "à la GUIDE" peut se faire de la façon suivante en CAOLAC (Cf. figure 6.16) : on déclare un seul état dans le comportement (ici l'état s) actif en permanence (sa sémantique est de type *tant que* associée à la condition vraie) et que l'on associe à autant de transitions qu'il y a de méthodes dans la classe (chaque transition est issue et aboutit à l'état). La garde des transitions reprend alors les clauses comportementales de la classe GUIDE (le compteur *current* correspond au compteur CAOLAC *CurrentInvocation*).

Bien que les modèles de synchronisation de GUIDE et de CAOLAC soient équivalents, plusieurs différences peuvent être constatées :

- CAOLAC est certes plus verbeux que GUIDE. Néanmoins, dans cet exemple d'exclusion mutuelle, les clauses comportementales de GUIDE sont elles-même plus verbeuses qu'un simple sémaphore. Néanmoins, il nous semble indéniable que GUIDE est plus lisible et plus facilement compréhensible que des sémaphores lorsque la synchronisation requiert plusieurs

```
class Pile implements Pile is
 method Put /* ... */
 method Get /* ... */
 control
  Put : current(Put) +
        current(Get) = 0;
  Get : current(Put) +
        current(Get) = 0;
end Pile.
```

```
behaviour BufferDeTailleFixe {
 /* ... */
 initial state : s;
 state s server while (true) {
  TPut {
   invocation( Put(e) );
   require( CurrentInvocation(Put) +
            CurrentInvocation(Get) = 0 );
   BASE.Put(e); return;
   become(s);
  }
  TGet {
   invocation( Get );
   require( CurrentInvocation(Put) +
            CurrentInvocation(Get) = 0 );
   return BASE.Get;
   become(s);
}}}

class Pile with behaviour BufferDeTailleFixe is
 method Put /* ... */
 method Get /* ... */
end Pile.
```

FIG. 6.16 – *Traduction littérale d'une synchronisation GUIDE en CAOLAC*

sémaphores imbriqués les uns dans les autres. De la même façon, nous pensons que le style de programmation de COALAC est plus simple que celui de GUIDE lorsque la synchronisation nécessite de nombreuses clauses comportementales complexes.

- GUIDE fournit un modèle dans lequel toutes les méthodes ont un statut équivalent. En effet, quel que soit l'état de la politique de synchronisation, toutes les clauses comportementales de toutes les méthodes sont systématiquement évaluées. Dans CAOLAC, seul un sous-ensemble de méthodes est concerné par la phase d'évaluation des gardes : ce sont celles associées aux transitions issues des états actifs. De ce fait, on restreint l'ensemble des méthodes potentiellement exécutables et on contribue à clarifier le fonctionnement de la politique de synchronisation.

- dans GUIDE, la politique de synchronisation est incluse dans la classe, alors qu'elle en est séparée dans CAOLAC. Ainsi, l'écriture d'une classe GUIDE *File* impose la réécriture des clauses comportementales, alors que le comportement CAOLAC *BufferDeTailleFixe* peut être réutilisé tel quel.

- à notre avis, les politiques de synchronisation conçues en terme d'expressions de chemin sont plus facilement exprimables en CAOLAC qu'en GUIDE. Même si, encore une fois, le code CAOLAC est plus long, nous pensons que le résultat est plus lisible et plus facilement compréhensible par une personne autre que leur auteur. Par exemple, considérons une classe avec deux méthodes *Lire* et *Ecrire* et respectant la synchronisation $Ecrire; (Lire|Ecrire)*$ (i.e. seule la méthode *Ecrire* est d'abord exécutable, puis soit *Lire* soit *Ecrire* sont exécutables).

L'expression d'une telle synchronisation en CAOLAC s'obtient en traduisant directement l'expression de chemin en automate états/transitions, alors qu'il est nécessaire, en GUIDE, de transformer cette expression pour obtenir les gardes de chacune des méthodes.

6.4.3 Résumé

Le tableau 6.3 résume les approches de la synchronisation mises en place, respectivement, par les langages ACT++, DRAGOON, GUIDE et CAOLAC.

		ACT++	DRAGOON	GUIDE	CAOLAC
modèle de synchronisation	états/ transitions	X			X
	compteurs d'événements		X	X	X
abstraction comportementale	inclus dans les classes	X		X	
	extérieure aux classes		X		X

TAB. 6.3 – *Comparaison de différentes approches de la synchronisation en univers objet*

6.5 Implantation du compilateur CAOLAC

Dans cette partie, nous présentons un bilan de l'implantation du compilateur du langage CAOLAC. Cet outil accepte en entrée du code CAOLAC et produit du code GUIDE. Celui-ci est compilé avec le compilateur `glc` de l'environnement GUIDE (à titre indicatif, `glc` produit du langage C qui est à son tour compilé avec un compilateur ANSI C standard).

Dans le paragraphe suivant, nous établissons un bilan de l'effort de développement investi dans le compilateur du langage CAOLAC. Le paragraphe 6.5.2 présente la chaîne de production d'un fichier exécutable, à partir d'un fichier source CAOLAC. Le paragraphe 6.5.3 présente une évaluation des performances du compilateur du langage CAOLAC. Finalement, le paragraphe 6.5.4 introduit alors les différents traitements effectués par ce compilateur.

6.5.1 Volume de développement

La plate-forme GUIDE que nous utilisons comprend trois stations Sun Sparc exécutant le noyau GUIDE 1.6. Le développement du compilateur CAOLAC représente 5 mois de travail à temps plein (ayant été effectué en parallèle avec d'autres activités, il s'est en fait étalé sur une durée supérieure).

Le développement du compilateur CAOLAC a été réalisé en C++ sur stations Sun sous SunOS 4.1.x. Le compilateur CAOLAC représente 6900 lignes de code qui se répartissent comme suit : 200 lignes de règles d'analyse lexicale, 500 lignes de règles d'analyse syntaxique et 6200 lignes de C++. L'analyseur lexical utilisé est `flex` version 2.5.2 (compatible `lex`). L'analyseur syntaxique est `bison` version 1.24 (compatible `yacc`). Avec les règles d'analyse que nous leur fournissons, ils produisent à eux deux 4900 lignes de code C. Celles-ci, ainsi que les 6200 lignes de C++ qui

réalisent l'analyse sémantique et la génération du code GUIDE, sont compilées avec le compilateur C/C++ `gcc` version 2.7.2. Le fichier exécutable du compilateur a une taille de 136 Ko. Les librairies qui accompagnent le compilateur représentent 600 lignes de code GUIDE et sont utilisées pour la production des exécutables (Cf. paragraphes 6.5.2 et 6.5.4.5).

6.5.2 Chaîne de production

La chaîne de production d'un fichier exécutable à partir d'un programme source CAOLAC comprend deux étapes (Cf. figure 6.17) :

1. à partir d'un ou de plusieurs fichiers CAOLAC comprenant la définition de comportements et/ou de classes implantant ces comportements, le compilateur CAOLAC `cocar` produit un fichier GUIDE (extension `.gui`),

2. le compilateur GUIDE `glc` compile ce fichier et réalise l'édition des liens en y ajoutant un ensemble de librairies (Cf. paragraphe 6.5.4.5).

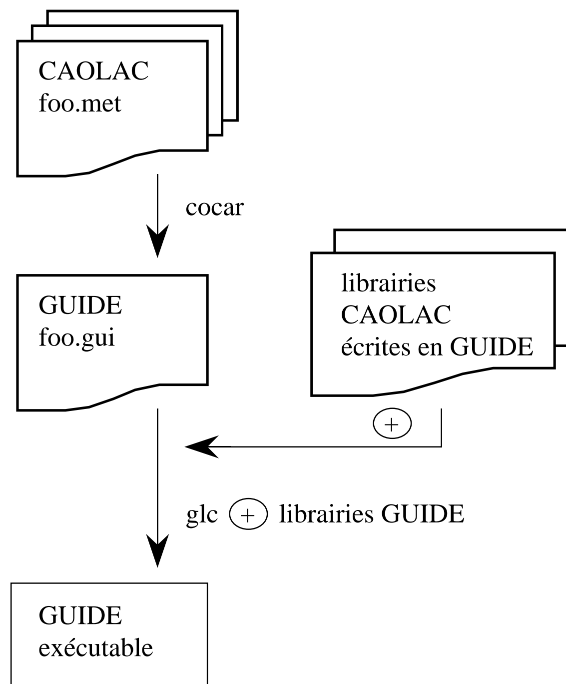


FIG. 6.17 – *Chaîne de production d'un programme CAOLAC*

6.5.3 Performances

Le compilateur `cocar` traite environ 700 lignes de code CAOLAC à la seconde. Cette valeur a été obtenue sur une station Sun Sparc ELC à partir d'une moyenne sur les temps de compilation moyens de six programmes tests (Cf. tableau 6.4). Le code de ces programmes est fourni dans le manuel du langage CAOLAC [Sei97a]. *Arbre couvrant* est un calcul d'arbres couvrants de réseaux dû à Bonnet [Bon94]. *Tampon* est une gestion de tampon synchronisé de taille fixe. *Lecteurs/Ecrivain* est une gestion de données partagées selon un cohérence forte de type plusieurs

lecteurs ou (exclusif) un écrivain. *MacQuillan* est un calcul de tables de routage selon l'algorithme du même nom. *Election sur un anneau* est une élection sur un anneau par deux vagues contra-rotatives due à Florin [Flo95]. *Valide deux phases* est un protocole transactionnel de validation à deux phases (Cf. [LS76] et [BHG87]). Pour chacun de ces programmes, le tableau 6.4 fournit le nombre de lignes CAOLAC, le temps CPU en secondes utilisé par `cocar` et le nombre de lignes GUIDE produites. Le temps de compilation a été obtenu à partir d'une moyenne de 10 compilations pour chaque programme. Chaque mesure est au 50ème de seconde près.

	CAOLAC (lignes)	Compilation (secondes)	GUIDE (lignes)
Arbre couvrant	566	0.55	820
Tampon	128	0.21	334
Lecteurs/Ecrivain	84	0.15	148
MacQuillan	262	0.33	459
Election sur un anneau	215	0.35	421
Validation deux phases	106	0.17	297

TAB. 6.4 – *Temps moyens de compilation CAOLAC*

6.5.4 Présentation du compilateur

Dans ce paragraphe, nous présentons les différents modules composant le compilateur `cocar` du langage CAOLAC : analyse lexicale (en `lex`), analyse syntaxique (en `yacc`), analyse sémantique (en C++) et génération du code GUIDE. Finalement, nous présentons les librairies GUIDE nécessaires à l'édition des liens d'un programme CAOLAC.

6.5.4.1 Analyse lexicale

La partie analyse lexicale reconnaît les 53 mots-clés du langage CAOLAC (Cf. [Sei97a] pour la liste complète). L'analyseur `flex` gère leur reconnaissance quelle que soit leur casse. Chaque mot clé et chaque élément lexical (identificateur, valeur numérique, chaîne de caractères, ...) est associé à un jeton qui est transmis à l'analyseur syntaxique.

Hormis la maîtrise la syntaxe `lex`, les règles lexicales de reconnaissance du langage CAOLAC ne posent pas de problèmes majeurs. Ces règles sont utilisées par l'analyseur `flex` pour produire un automate en langage C. Nous avons développé un programme de modification automatique de cet automate afin que le pointeur de parcours du fichier source enregistre, en plus du numéro de ligne courante, le numéro de colonne. Cela fournit une information supplémentaire au développeur CAOLAC et facilite notamment, l'examen du code source en cas d'erreur.

6.5.4.2 Analyse syntaxique

La grammaire du langage CAOLAC contient 202 règles. Au delà d'une vérification syntaxique des fichiers sources, les actions associées aux règles grammaticales réalisent une première analyse sémantique en vérifiant l'unicité d'un certain nombre d'identificateurs : par exemple, unicité des noms de comportements, des noms de variables, d'invocations, de méthodes et d'états au sein

d'un comportement, des noms de transitions au sein d'un état. De plus, elles construisent une représentation mémoire des programmes examinés. Par exemple, chaque comportement rencontré est associé à une structure qui contient, entre autres, le nom du comportement et des pointeurs sur une liste de variables, d'invocations, de méthodes et d'états (Cf. figure 6.18). Chacun de ces éléments est lui-même, une liste chaînée de structures.

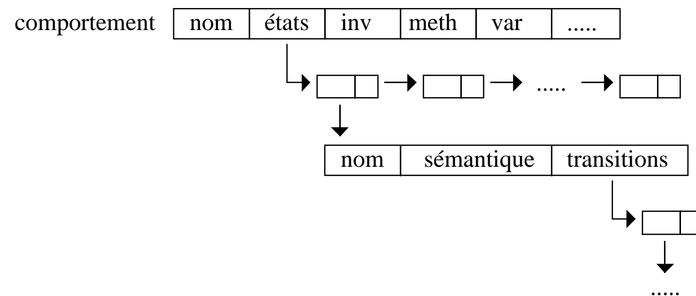


FIG. 6.18 – *Principe de la représentation mémoire des programmes CAOLAC*

La construction de cette représentation mémoire utilise systématiquement les mécanismes d'allocation dynamique. De ce fait, la taille des programmes CAOLAC pouvant être traités est seulement limitée par la taille maximale attribuée, par le système, au processus de compilation. Par ailleurs, les structures de stockage utilisées sont des classes plutôt que des enregistrements. Ainsi, chaque comportement, état, transition, ... est un objet qui possède un ensemble de champs et un ensemble de méthodes permettant de les manipuler. De même, les listes d'états, de transitions, de variables, ... sont des objets membres de classes dérivant d'une classe générique *liste*.

6.5.4.3 Analyse sémantique

Le module d'analyse sémantique a pour but de détecter, dans un programme dont la syntaxe est correcte, les incohérences de sens. C'est, par exemple, le cas d'une transition aboutissant à un état qui n'existe pas. Nous avons signalé, au paragraphe précédent, qu'un certain nombre de vérifications sémantiques sont effectuées "au fil de l'eau" lors de l'analyse syntaxique : ce sont toutes celles qui, comme la vérification de l'unicité d'un nom de comportement, peuvent se contenter du code déjà parcouru. Celles qui nécessitent des recherches ou des traitements plus complexes dans le code, sont traitées séparément dans le module d'analyse sémantique.

Le module d'analyse sémantique effectue principalement les cinq tâches suivantes :

– recherche des états finaux des transitions : pour chaque instruction `become` rencontrée, le module recherche dans la liste des états l'état final spécifié.

– héritage de structures : dans chaque sous-comportement rencontré, le module ajoute les états et les transitions du sur-comportement. Si un état est redéfini, alors seules ses transitions dans le sur-comportement sont ajoutées. Si une transition est redéfinie, alors rien n'est effectué.

– examen des sémantiques d'états : chaque état est associé à un booléen qui vaut vrai lorsque l'état est actif. Sa gestion ne pose pas de difficulté pour la sémantique sortante *séquentielle* et la sémantique entrante *nulle* : la première fait passer sa valeur à faux, tandis que la seconde la fait passer à vrai. Néanmoins, les sémantiques sortantes *parallèle* et *tant que*,

et la sémantique entrante *rendez-vous* nécessitent des mécanismes plus complexes. Le module d'analyse sémantique repère donc les comportements contenant de telles sémantiques d'états pour qu'ils soient instrumentés en conséquence par la phase de génération du code GUIDE. Ainsi, les états à sémantique sortante *parallèle* sont associés à un vecteur d'entiers. Un élément de ce vecteur représente une transition sortante. Chaque élément est incrémenté lorsque l'état est activé, et est décrémenté lorsque sa transition est déclenchée. Lorsque le vecteur est nul, l'état est désactivé. Il en est de même pour les états à sémantique entrante *rendez-vous*. Les éléments du vecteur sont incrémentés lorsque les transitions sont exécutées. L'état est activé lorsque tous les éléments du vecteur sont positifs et les éléments sont décrémentés. Finalement, les états à sémantique sortante *tant que* sont associés à une méthode qui, une fois qu'ils ont été activés, évalue leur condition d'activation à chaque début ou fin de transition dans le comportement, et ce, jusqu'à ce qu'ils soient désactivés.

- instanciation des invocations génériques : les comportements CAOLAC peuvent contenir des invocations dont le type est générique. Celui-ci est alors associé à un ou plusieurs types concrets à la déclaration d'une classe implantant ce comportement. Le module d'analyse sémantique repère donc de telles classes et crée pour chacune une instance de comportement avec les types d'invocations concrets.

- insertion des éléments de trace : les variables, invocations et méthodes d'un comportement CAOLAC peuvent être marquées pour enregistrements (mot clé `TRACED`). Dans ce cas, les événements de lecture, d'écriture, d'arrivée, de début, de fin et d'appels concernant ces éléments sont enregistrés dans un objet historique (désigné par le mot clé `HISTORY`). Le module d'analyse sémantique instrumente donc le code des comportements afin de gérer automatiquement cet historique.

6.5.4.4 Génération du code GUIDE

La dernière phase du compilateur `cocar` consiste en une traduction en classes GUIDE des comportements CAOLAC. Elle comprend deux étapes qui, respectivement, gère les tours méta et produit le code GUIDE proprement dit.

Gestion des tours méta

Le principe retenu dans `cocar` consiste à traduire chaque comportement CAOLAC par une classe GUIDE. La relation méta-comportementale, absente dans GUIDE, est alors simulée par de l'héritage de classe.

Dans le cas d'une tour méta, c'est le composant de plus haut niveau (*méta1* dans l'exemple de la figure 6.19) qui prend en compte, en premier, les invocations de méthodes, puis qui les transmet au niveau inférieur à l'aide de la référence `BASE`. Cependant, dans le cas d'une hiérarchie de classes, c'est l'élément de plus bas niveau qui reçoit en premier l'invocation et qui la propage éventuellement au niveau supérieur à l'aide de la référence `SUPER`. De ce fait, nous sommes obligés de "retourner" chaque branche d'un graphe de tours méta afin de placer son élément de plus haut niveau à la fin de la hiérarchie d'héritage. Nous l'effectuons en parcourant de façon récursive, à partir des feuilles, chaque branche d'une tour méta et en générant la classe GUIDE au retour de l'appel récursif.

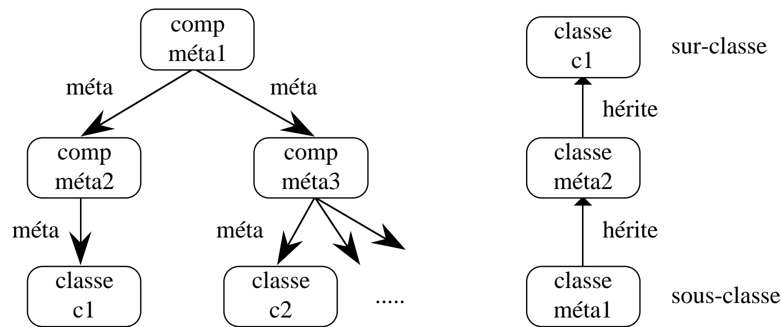


FIG. 6.19 – *Traduction des tours méta CAOLAC en schémas d'héritage de classes GUIDE*

Production du code GUIDE

Comme nous l'avons précisé ci-dessus, chaque comportement CAOLAC est traduit par une classe GUIDE. A l'intérieur de celle-ci, chaque invocation prise en compte par le comportement est traduite par une méthode. La méthode exécute une boucle infinie qui comprend un ensemble de tests correspondant aux états où l'invocation est prise en compte (Cf. figure 6.20). A l'intérieur de chaque test, la garde de l'invocation est évaluée. Si elle est vrai, les actions associées sont exécutées, l'invocation est retournée et l'état conséquent est activé. Notons que, dans le cas où l'exécution d'une invocation se poursuit sur plusieurs transitions, les actions des différentes transitions ainsi que les activations et désactivations d'états intermédiaires sont regroupées au sein de ce bloc.

```
class meta2 is
  method Put;
  begin
    while true do
      if état_où_Put_est_pris_en_compte = true then
        if garde = true then
          // désactiver l'état
          // exécuter les actions
          // retourner l'invocation
          // activer l'état conséquent
        end;
      end;
      if autre_état_où_Put_est_pris_en_compte = true then
      ...
    end;
  end Put;
  ...
end meta2.
```

FIG. 6.20 – *Traduction d'un comportement CAOLAC en classe GUIDE*

6.5.4.5 Librairie de composants

L'édition des liens d'un programme CAOLAC nécessite l'utilisation d'un ensemble de composants prédéfinis. Ce sont des classes GUIDE utilisées par les programmes produits par le compila-

teur `cocar`. Ces classes sont au nombre de cinq :

- vecteur d'entiers : gère un vecteur d'entiers pour les états à sémantiques *parallèle* et *rendez-vous* (Cf. paragraphe 6.5.4.3).
- compteur d'événements : gère les compteurs d'événements associés à une invocation, une méthode ou une transition.
- groupe d'objets : fournit des primitives standards (ajout, retrait, ...) de manipulation de groupes d'objets.
- liste chaînée : fournit des primitives standards de manipulation de listes chaînées gérées dynamiquement.
- historique d'événements : gère les historiques d'événements associés à une classe.

6.6 Conclusion

Dans cette partie, nous avons présenté notre approche pour la gestion de la concurrence et de la synchronisation dans les langages à objets concurrents. Nous avons présenté pour cela le langage CAOLAC qui permet de définir la synchronisation des méthodes d'une classe concurrente. Ce langage s'insère dans le processus de développement d'applications distribuées en trois niveaux (groupe, objet et méthode) proposé au chapitre 4. Il assure la description des comportements de niveau objet.

Le langage CAOLAC se présente sous la forme d'un protocole méta-objets et utilise une approche à base de modèles états/transitions. Nous en avons réalisé un prototype au-dessus du système distribué GUIDE [BBD+91]. Le langage CAOLAC permet de séparer clairement, dans une application, les traitements courants, des traitements de synchronisation. Les premiers sont définis dans des classes au sens habituel du terme, tandis que les seconds le sont dans des classes de synchronisation appelées comportements. Chaque objet, instance d'une classe, est associé à un méta-objet, instance d'un comportement. Les invocations de méthodes sont systématiquement synchronisées par les méta-objets avant d'être délivrées aux objets. De plus, la politique de synchronisation globale peut être découpée en plusieurs modules indépendants reliés entre eux par la relation méta. L'architecture ainsi créée est désignée sous le terme de tour méta. Chaque méta-objet dans cette tour assure une partie de la politique globale. Nous pensons que cette démarche, même si elle entraîne un certain surcoût (chaque niveau méta introduit des appels de méthodes supplémentaires), permet aux différents traitements de synchronisation d'être conçus et testés indépendamment les uns des autres dans des modules différents. Cela facilite ainsi leur compréhension et leur réutilisation.

Une des originalités de CAOLAC est d'utiliser une approche à base de modèles états/transitions pour l'écriture du code des comportements. Chaque état représente un point d'avancement de la politique de synchronisation. Les transitions sont associées à des blocs d'instructions GUIDE et représentent les méthodes qui peuvent être exécutées à partir de ces états. Afin de prendre en compte le parallélisme intra-objets, nous avons proposé, au paragraphe 6.3.3, une sémantique qui étend celle des automates états/transitions habituels. On souhaite autoriser, par exemple, des synchronisations dans lesquelles un état est un point de rendez-vous de transitions ou un point de départ de transitions parallèles. Nous avons donc introduit cinq sémantiques d'états.

Chacune d'elles introduit des règles pour l'activation ou la désactivation d'états. On est ainsi en mesure de contrôler l'évolution d'un objet exécutant plusieurs fils d'activités simultanément. Enfin, afin d'augmenter le taux de réutilisation des comportements CAOLAC, nous avons proposé, au paragraphe 6.3.7, une relation d'héritage. Ce mécanisme, qui s'apparente à de l'héritage de classes, permet de réutiliser la structure des modèles états/transitions et, éventuellement, d'en redéfinir les états et les transitions.

Nous avons veillé à rendre les concepts introduits sont aussi indépendants que possible de la plateforme retenue. Afin de tester cet aspect, nous envisageons d'effectuer une implantation du langage CAOLAC pour d'autres langages objets et d'autres environnements distribués (par exemple C++ au-dessus d'un bus à objets CORBA). Cela nous permettra, d'une part de tester le bien-fondé de notre approche, et d'autre part de comparer CAOLAC à d'autres protocoles méta-objets comme Open C++ [Chi95], MetaJava [GK97a, GK97b].

Dans le chapitre suivant, nous proposons les premiers éléments d'une sémantique pour le modèle de synchronisation du langage CAOLAC. Le but de cette étude est de préciser formellement certaines définitions. En particulier, nous nous intéressons aux sémantiques d'état. D'autres notions, comme par exemple le raffinement de comportements, sont juste évoquées et devront faire l'objet de travaux futurs.

Chapitre 7

Sémantique du modèle de synchronisation

Nous avons présenté dans le chapitre précédent le langage CAOLAC qui permet de décrire des modèles de synchronisation pour des objets concurrents. Dans ce chapitre, nous définissons une sémantique pour le modèle de synchronisation du langage CAOLAC. Nous choisissons pour cela la logique temporelle d'actions de Lamport présentée au chapitre 2. Trois raisons principales ont motivé ce choix : ce sont, respectivement, la façon dont TLA traite les notions d'actions, de concurrence et de raffinement. Tout d'abord TLA est une logique d'actions. Elle est, donc, bien adaptée à des raisonnements sur des algorithmes définis comme des séquences d'actions. Son deuxième avantage réside dans l'association entre une logique d'actions et une logique temporelle. En considérant une sémantique d'entrelacements pour le parallélisme intra-objet, la spécification du comportement d'un objet concurrent se ramène à la description de toutes les séquences valides d'actions. Celles-ci s'expriment, en TLA, à l'aide des opérateurs temporels de nécessité et de possibilité. Finalement, le raffinement s'exprime en TLA comme une implication logique entre deux formules représentant respectivement un programme général et un programme raffiné. En traduisant chaque politique de synchronisation par une formule TLA, on est alors en mesure de donner un support formel à la notion de raffinement pour une politique de synchronisation.

Rappelons que le langage CAOLAC a pour but de coordonner l'exécution d'objets concurrents au sein d'un système réparti. Dans notre cas le système réparti choisi est le système GUIDE [BBD+91]. Chaque objet de l'environnement est associé à une instance de comportement appelé méta-objet. Une classe de comportement fournit la synchronisation d'un ensemble de méthodes, tandis qu'une classe d'objets fournit les traitements effectifs (c'est à dire en dehors de toute synchronisation) associés à ces méthodes. La synchronisation est définie à l'aide de modèles états/transitions étendus dans lesquels plusieurs transitions peuvent s'exécuter simultanément. Les extensions proposées se traduisent par la définition de différentes sémantiques d'états. Ces sémantiques, présentées au paragraphe 6.3.3.2, assignent des règles pour le franchissement des transitions entrantes et sortantes. Nous proposons, dans ce chapitre, une formulation équivalente de ces règles en terme de logique temporelle d'actions.

Le paragraphe suivant introduit les notations dont nous nous servons, par la suite, pour définir la sémantique du modèle de synchronisation. Le paragraphe 7.2 interprète, en terme de logique temporelle d'actions, les cinq sémantiques définies au paragraphe 6.3.3.2. Chaque état est associé

à un couple sémantique entrante, sémantique sortante. L'interprétation temporelle complète d'un schéma de synchronisation s'obtient en rassemblant les couples de formules temporelles de tous les états. Le paragraphe 7.3 présente la construction de cette formule pour tout comportement CAOLAC. Nous illustrons, à l'aide d'un exemple, le processus de raffinement associé aux politiques de synchronisation. Finalement, le paragraphe 7.4 conclut ce chapitre.

7.1 Eléments de base

Dans cette partie, nous introduisons les notations utilisées dans la suite de ce chapitre pour définir la sémantique du modèle de synchronisation. Le paragraphe suivant définit le structure d'un comportement CAOLAC. Le paragraphe 7.1.2 définit les différentes variables utilisées dans l'écriture des formules TLA et illustre leur utilisation par un exemple.

7.1.1 Comportement

Un comportement CAOLAC est défini de manière formelle par le 4-uplet $C = \langle V, I, M, A\rangle$ où :

- V est un ensemble fini de variables typées,
- I est un ensemble fini d'invocations de méthodes pouvant être prises en compte par le comportement et constituant l'interface externe du comportement,
- M est un ensemble fini de méthodes de l'objet de base pouvant être appelées par le comportement et constituant l'interface interne du comportement,
- A est un modèle états/transitions.

Afin de simplifier l'exposé de la sémantique, nous ne différencions pas, dans l'ensemble V, les constantes des variables. En effet, d'un point de vue formel, la seule contrainte à vérifier au niveau des actions utilisant des constantes est que leur valeur ne peut pas être modifiée. Une invocation ou une méthode est caractérisée par un identificateur de message et une interface. Une interface est un ensemble fini de paramètres. Chaque paramètre est défini par un identificateur, un type et un mode de passage (entrée, entrée/sortie, sortie). L'ensemble M désigne les méthodes de l'objet de base associé au comportement CAOLAC. Le modèle états/transitions A d'un comportement CAOLAC est défini formellement par le 5-uplet $A = \langle E, D, T, L, Ei\rangle$ où :

- E est un ensemble fini d'états représentant des configurations de la politique de synchronisation,
- D est une fonction d'étiquetage d'états qui associe une sémantique à chaque élément de l'ensemble E,
- T est une fonction de transition entre les états de l'ensemble E,
- L est une fonction d'étiquetage de transitions qui associe une séquence finie d'actions à chaque élément de l'ensemble T,
- Ei est un ensemble fini d'états initiaux du comportement.

La fonction D associe à chaque état l'une des sémantiques définies au paragraphe 6.3.3.2. C'est un couple sémantique entrante sémantique sortante. Ces deux éléments sont choisis respectivement parmi les ensembles $DE = \{nulle, rendezvous\}$ et $DS = \{séquentielle, parallèle, tantque\}$. Dans le cas d'une sémantique sortante de type *tant que*, une clause booléenne définissant la période d'activation de l'état doit être fournie. Cette clause fait partie d'un ensemble $DACT$ d'expressions booléennes écrites à l'aide d'opérateurs arithmétiques et de compteurs de synchronisation. Formellement, la fonction D est donc l'union de trois fonctions : $De : E \rightarrow DE$, $Ds : E \rightarrow DS$ et $Dact : E \rightarrow DACT$. Elles associent à un état, respectivement, sa sémantique entrante, sa sémantique sortante et sa clause d'activation. Pour les états tels que la sémantique sortante est différente de *tant que*, la clause d'activation n'a aucune incidence.

La fonction T définit des transitions entre deux configurations d'un comportement CAOLAC. C'est donc une relation entre les éléments de l'ensemble E. Néanmoins, une relation de ce type n'est pas suffisante pour définir la fonction de transition. En effet, plusieurs transitions peuvent être spécifiées entre deux états identiques. On modélise ainsi des situations dans lesquelles différentes actions sont exécutées en parallèle (plusieurs transitions sont tirées concurremment) ou dans lesquelles des comportements alternants peuvent être définis (une seule transition est tirée). On choisit donc d'indicer les couples d'états mis en relation afin d'autoriser différentes transitions entre deux états identiques. Formellement, la fonction T est définie par $T : E \times \mathbb{N} \rightarrow E$.

La fonction L associe à chaque transition d'un comportement une séquence finie d'actions. Ces actions sont choisies parmi un ensemble ACT d'actions possibles. Cet ensemble comprend toutes les instructions du langage de base associé à CAOLAC (en l'occurrence GUIDE) augmentées des instructions `invocation`, `require`, `become` et `BASE` ajoutées par CAOLAC. Formellement, la fonction L est définie par $L : E \times \mathbb{N} \rightarrow 2^{ACT}$.

Finalement, l'ensemble Ei des états initiaux est un sous-ensemble de l'ensemble des configurations : $Ei \subseteq E$. Ce sont les états actifs à l'instanciation du comportement.

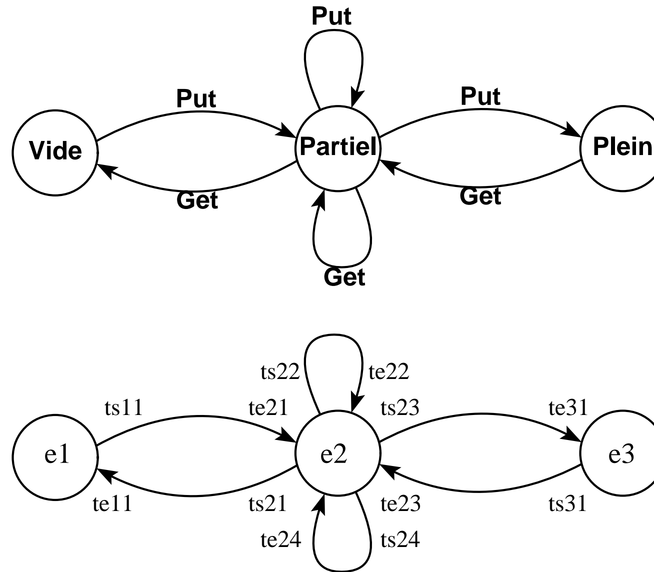
7.1.2 Variables propositionnelles

Dans le cas d'un comportement CAOLAC, l'ensemble *Var* des variables propositionnelles qui servent à l'écriture des formules de logique temporelle d'actions, comprend :

- l'ensemble V des variables et constantes du comportement,
- un ensemble $VarE = \{e_1, \ldots, e_n\}$ de variables booléennes associées aux états du comportement,
- deux ensembles $VarI = \{i_1, \ldots, i_n\}$ et $VarC = \{c_1, \ldots, c_n\}$ de variables associées, respectivement, aux invocations et aux conditions des gardes du comportement,
- deux ensembles $VarTE = \{te_1, \ldots, te_n\}$ et $VarTS = \{ts_1, \ldots, ts_n\}$ de variables associées aux transitions entrantes et sortantes du comportement.

Dans la suite de ce paragraphe, nous détaillons la définition de ces différents ensembles et nous illustrons leur utilisation à l'aide de l'exemple d'un buffer synchronisé de taille fixe (Cf figure 7.1).

Ainsi, chaque élément e_i de l'ensemble *VarE* est une variable booléenne associée à un état de l'ensemble E. La variable e_i est égale à vrai, si et seulement si, l'état correspondant est actif. Dans l'exemple de la figure 7.1, on retrouve les variables e_1, e_2 *et* e_3 associées, respectivement, aux états *Vide*, *Partiel* et *Plein*.




```
behaviour bdtf {
 invocations:
  Get : REF TElement;
  Put( IN e:REF TElement );
 methods:
  IsFull : Boolean;
  IsEmpty : Boolean;

 initial state : Vide;
 state Vide sequential {
  TPut {
   invocation( Put(e) );
   BASE.Put(e); return;
   become( Partiel );
 }}

 state Plein sequential {
  TGet {
   invocation( Get );
   return BASE.Get;
   become( Partiel );
 }}

 state Partiel sequential {
  TPut1 {
   invocation( Put(e) );
   BASE.Put(e); return;
   become( Partiel );
  }
  TPut2 {
   invocation( Put(e) );
   BASE.Put(e); return;
   become( Plein );
  }
  TGet1 {
   invocation( Get );
   return BASE.Get;
   become( Partiel );
  }
  TGet2 {
   invocation( Get );
   return BASE.Get;
   become( Vide );
}}}
```

FIG. 7.1 – *Variables propositionnelles associées au comportement bdtf*

VarI et *VarC* sont des ensembles d'ensembles de variables associées, respectivement, aux invocations et aux conditions des gardes du comportement. Leur cardinalité est égale à celle de l'ensemble E des états du comportement. Un élément i_{ik} de l'ensemble i_i est une variable entière associée à l'invocation de la garde de la k-ème transition sortante de l'état e_i. La valeur de la variable i_{ik} représente le nombre d'invocations de ce type présentes dans la file d'attente. Si aucune invocation n'est spécifiée, alors la variable n'apparaît pas dans les formules temporelles. Un élément c_{ik} de l'ensemble c_i est une variable booléenne associée à la condition de la k-ème transition sortante de l'état e_i. La variable c_{ik} est égale à vrai, si et seulement si, la condition est vraie. Si aucune condition n'est spécifiée alors la variable est toujours vraie. Dans l'exemple de la figure 7.1, on retrouve donc les variables entières i_{11}, i_{21}, i_{22}, i_{23}, i_{24}, i_{31} et les variables booléennes c_{11}, c_{21}, c_{22}, c_{23}, c_{24}, c_{31}. Les variables i_{11}, i_{22} et i_{23} représentent le nombre d'invocations de la méthode *Put* présentes dans la file d'attente du comportement. Les variables i_{21}, i_{24} et i_{31} concernent l'invocation *Get* et sont également toujours égales entre elles. Aucune condition n'est spécifiée dans les gardes. Les variables c_{11} à c_{31} sont donc égales à vrai.

VarTE et *VarTS* sont des ensembles d'ensembles de variables associées aux transitions entrantes et sortantes du comportement. Leur cardinalité est égale à celle de l'ensemble E des états du comportement. Les éléments te_i et ts_i des ensembles *VarTE* et *VarTS* sont, respectivement, les ensembles de transitions entrantes et sortantes associées à l'état e_i. Les éléments te_{ik} et ts_{ik} des ensembles te_i et ts_i sont des variables entières attachées respectivement à la k-ème transition entrante de l'état e_i et à la k-ème transition sortante de l'état e_i. Pour activer un état, il faut consommer un certain nombre de transitions : une instance d'une transition dans le cas d'une sémantique entrante *nulle*, et une instance de toutes les transitions dans le cas d'une sémantique *rendez-vous*. La variable te_{ik} représente le nombre d'instances consommables de la k-ème transition entrante de l'état e_i. La variable ts_{ik} représente, quant à elle, le nombre d'instances de cette transition qui peuvent être déclenchables. Chaque transition de l'ensemble T est donc associée à deux variables entières : te_{jl} et ts_{ik}. Il s'agit de la l-ème transition entrante d'un état e_j et la k-ème transition sortante d'un état e_i. Afin de désigner ces deux variables avec le même couple d'indice, nous définissons la fonction $TE :: E \times \mathbb{N} \rightarrow VarTE$. Ainsi, les variables $te_{jl} = TE(e_i, k)$ et ts_{ik} se rapportent à la même transition. Dans l'exemple de la figure 7.1, on obtient les variables entières $te_{11}, \ldots, te_{31}$ et $ts_{11}, \ldots, ts_{31}$. Les variables te_{ik} sont associées aux transitions entrantes, tandis que les variables ts_{jl} sont associées aux transitions sortantes. Ces variables sont mises en relation par la fonction *TE*. On obtient par exemple : $te_{31} = TE(ts_{23})$.

7.2 Sémantiques d'état

Nous avons montré, au paragraphe 6.3.3.2, que l'interprétation habituelle des états proposée par la théorie des automates à états finis est insuffisante pour décrire des comportements mettant en œuvre du parallélisme intra-objet. Nous avons donc défini cinq sémantiques d'état. L'interprétation d'un état est un couple sémantique entrante, sémantique sortante. La première définit des règles pour les transitions entrantes d'un état, tandis que la seconde définit des règles pour les transitions sortantes. Dans ce paragraphe, nous donnons une interprétation de ces évolutions en terme de logique temporelle d'actions.

7.2.1 Sémantiques entrantes

Deux sémantiques entrantes ont été définies au paragraphe 6.3.3.2 : *nulle* et *rendez-vous*. Elles définissent les règles d'activation des états. Rappelons qu'un état à sémantique entrante *nulle* est activé dès que l'une quelconque des transitions entrantes de cet état a été exécutée, et qu'un état à sémantique entrante *rendez-vous* est activé dès que toutes ses transitions entrantes ont été exécutées exactement une fois chacune.

On considère donc un état e_i muni de m transitions entrantes. Les variables propositionnelles utilisées pour définir la sémantique entrante de cet état sont la variable booléenne e_i associée à l'état, et les variables entières $te_{i1}, \ldots, te_{im}$ associées aux transitions entrantes de cet état. L'activation de l'état correspond au passage à vrai de la variable e_i. Dans le cas d'une sémantique entrante *nulle*, cette évolution est possible dès que l'une quelconque des transitions entrantes a été exécutée, c'est à dire, dès que l'une des variables propositionnelles $te_{i1}, \ldots, te_{im}$ est positive. Le comportement évolue alors en consommant la variable te_{ik} et en activant l'état e_i. On obtient la formule suivante :

$$\begin{array}{lcll} \mathcal{DE}_1(i,k) & \widehat{=} & & te_{ik} > 0 \\ & & \wedge & (te'_{ik} = te_{ik} - 1) \wedge (e'_i = true) \end{array}$$

La sémantique entrante *rendez-vous* requiert, quant à elle, l'exécution de toutes les transitions entrantes. Il est donc nécessaire que les m variables te_{ik} soient positives (ce qui s'exprime par $\bigwedge_{k \in [1,m]} te_{ik} > 0$). On décrémente alors toutes les variables te_{ik} et on active l'état e_i. On obtient la formule suivante :

$$\begin{array}{lcll} \mathcal{DE}_2(i,k) & \widehat{=} & & \bigwedge_{k \in [1,m]}(te_{ik} > 0) \\ & & \wedge & \bigwedge_{k \in [1,m]}(te'_{ik} = te_{ik} - 1) \wedge (e'_i = true) \end{array}$$

7.2.2 Sémantiques sortantes

Trois sémantiques sortantes ont été définies au paragraphe 6.3.3.2 : deux de contrôle (*séquentielle* et *parallèle*) et une orientée par les données (*tant que*). Elles définissent les règles de désactivation des états.

7.2.2.1 Séquentielle

Lorsqu'un état à sémantique *séquentielle* est atteint, toutes les gardes des transitions issues de cet état sont examinées. Dès qu'une garde est vérifiée, c'est à dire qu'un message de même type que celui de l'invocation est présent dans la file d'attente et que la condition s'évalue à vrai, alors l'invocation est retirée de la file d'attente et la transition est déclenchée. Les autres transitions cessent d'être examinées et l'état est désactivé. Si aucune garde ne convient, le processus de scrutation des transitions se poursuit jusqu'à ce que cela soit le cas. Si plusieurs gardes sont vraies simultanément, alors le système en choisit une et une seule de manière non déterministe.

On considère donc un état e_i muni de m transitions entrantes. Les variables propositionnelles utilisées pour définir la sémantique sortante de cet état sont la variable booléenne e_i associée à l'état, les variables entières $i_{i1}, \ldots, i_{im}$ et booléennes $c_{i1}, \ldots, c_{im}$ associées aux gardes de cet état. La sémantique *séquentielle* spécifie que si l'état est actif, si une invocation i_{ik} est présente dans la file d'attente et si l'expression c_{ik} est vérifiée, alors l'état est désactivé, l'invocation i_{ik} est retirée

de la file d'attente, les instructions $L(e_i, k)$ associées à la transition s'exécutent et la transition entrante de l'état conséquent est activée (c'est à dire la variable $TE(e_i, k)$ associée à la k-ème transition de e_i est incrémentée). On obtient la formule suivante :

$$
\begin{array}{lcll}
\mathcal{DS}_1(i,k) & \widehat{=} & & e_i = true \\
 & & \wedge & (i_{ik} > 0) \wedge (c_{ik} = true) \\
 & & & \\
 & & \wedge & (e_i^{'} = false) \wedge (i_{ik}^{'} = i_{ik} \Leftrightarrow 1) \\
 & & \wedge & L(e_i, k) \\
 & & \wedge & TE(e_i, k)^{'} = TE(e_i, k) + 1
\end{array}
$$

La formule complète traduisant l'évolution d'un état à sémantique *séquentielle* est obtenue en effectuant la disjonction d'expressions de ce type sur l'ensemble des m transitions sortantes.

7.2.2.2 Parallèle

Lorsqu'un état à sémantique *parallèle* est atteint, toutes les gardes des transitions issues de cet état sont examinées. Dès qu'une garde est vérifiée, c'est à dire qu'un message de même type que celui de l'invocation est présent dans la file d'attente et que la condition s'évalue à vrai, alors l'invocation est retirée de la file d'attente et la transition est exécutée. Toutes les autres transitions, sauf celle-ci, continuent à être examinées. Ce processus se poursuit jusqu'à ce que toutes les transitions aient été exécutées. Lorsque c'est le cas, l'état est désactivé. Si aucune garde ne convient, le processus de scrutation se poursuit jusqu'à ce que cela soit le cas.

La sémantique entrante *parallèle* s'exprime à l'aide les variables propositionnelles suivantes : la variable booléenne e_i associée à l'état, les variables entières $ts_{i1}, \ldots, ts_{im}$ associées aux transitions sortantes de cet état, les variables entières $i_{i1}, \ldots, i_{im}$ et booléennes $c_{i1}, \ldots, c_{im}$ associées aux gardes de cet état. La sémantique *parallèle* spécifie que, si l'état est actif, alors les m transitions sortantes sont déclenchables une fois chacune à la suite de quoi, l'état est désactivé. Si une invocation i_{ik} est présente dans la file d'attente et si l'expression c_{ik} est vérifiée, alors l'invocation i_{ik} est retirée de la file d'attente, les instructions $L(e_i, k)$ associées à la transition s'exécutent et la transition entrante de l'état conséquent est activée (c'est à dire la variable $TE(e_i, k)$ associée à la k-ème transition de e_i est incrémentée). On obtient la formule suivante :

$$
\begin{array}{lcllcll}
\mathcal{DS}_2(i,k) & \widehat{=} & & & & e_i = true & \\
 & & & & \wedge & (e_i^{'} = false) \wedge \bigwedge_{k \in [1,m]} (ts_{ik}^{'} = ts_{ik} + 1) & \\
 & & & & & & \\
 & & \vee & \bigvee_{k \in [1,m]} & (& ts_{ik} > 0 & \\
 & & & & \wedge & (i_{ik} > 0) \wedge (c_{ik} = true) & \\
 & & & & \wedge & (ts_{ik}^{'} = ts_{ik} \Leftrightarrow 1) \wedge (i_{ik}^{'} = i_{ik} \Leftrightarrow 1) & \\
 & & & & \wedge & L(e_i, k) & \\
 & & & & \wedge & TE(e_i, k)^{'} = TE(e_i, k) + 1 &)
\end{array}
$$

7.2.2.3 Tant que

Lorsqu'un état à sémantique *tant que* est atteint toutes les gardes des transitions issues de cet état sont examinées. Dès qu'une garde est vérifiée, c'est à dire qu'un message de même type

que celui de l'invocation est présent dans la file d'attente et que la condition s'évalue à vrai, alors l'invocation est retirée de la file d'attente et la transition est déclenchée. Toutes les transitions, y compris celle-ci, continuent à être examinées. Ce processus se poursuit tant que l'expression booléenne associée à l'état est vérifiée. Lorsque ce n'est plus le cas, l'état est désactivé. Si aucune garde ne convient, le processus de scrutation des transitions se poursuit selon la règle énoncée précédemment. Si plusieurs gardes sont vraies simultanément, alors le système les choisit dans un ordre non déterminé.

Pour exprimer la sémantique d'un état à sémantique *tant que* nous retenons les variables propositionnelles suivantes : la variable booléenne e_i associée à l'état, les variables entières $i_{i1}, \ldots, i_{im}$ et booléennes $c_{i1}, \ldots, c_{im}$ associées aux gardes de cet état. La sémantique *tant que* spécifie que si l'état est actif, si une la clause d'activation de l'état $Dact(e_i)$ est vérifiée, si une invocation i_{ik} est présente dans la file d'attente et si l'expression c_{ik} est vérifiée, alors l'état reste actif si et seulement si sa clause d'activation est vraie, l'invocation i_{ik} est retirée de la file d'attente, les instructions $L(e_i, k)$ associées à la transition s'exécutent et la transition entrante de l'état conséquent est activée (c'est à dire la variable $TE(e_i, k)$ associée à la k-ème transition de e_i est incrémentée). On obtient la formule temporelle suivante :

$$
\begin{array}{rcl}
\mathcal{DS}_3(i,k) & \widehat{=} & (e_i = true) \wedge Dact(e_i) = true \\
& \wedge & (i_{ik} > 0) \wedge (c_{ik} = true) \\
& & \\
& \wedge & (e_i^{'} = Dact(e_i)) \wedge (i_{ik}^{'} = i_{ik} \Leftrightarrow 1) \\
& \wedge & L(e_i, k) \\
& \wedge & TE(e_i, k)^{'} = TE(e_i, k) + 1
\end{array}
$$

7.3 Modèle états/transitions

Les définitions proposées au paragraphe précédent permettent d'interpréter les sémantiques d'état du langage CAOLAC à l'aide de formules de logique temporelle d'actions. Dans ce paragraphe, nous montrons comment ces formules peuvent être assemblées afin de fournir un programme temporel complet représentant l'ensemble des évolutions correctes d'un modèle états/transitions CAOLAC.

7.3.1 Formule générale

Le programme temporel doit spécifier qu'à l'instanciation l'état initial est actif, puis, que le comportement évolue constamment selon les règles spécifiées par les sémantiques entrantes et sortantes. On obtient donc un programme TLA (Cf. figure 7.2) composé :

- d'un ensemble *Var* de variables (Cf. paragraphe 7.1.2) comprenant l'ensemble *V* des constantes et variables d'instance du comportement, les ensembles *VarI* et *VarC* des invocations et conditions associées aux gardes des transitions et les ensembles *VarTE* et *VarTS* des variables associées aux transitions entrantes et sortantes,
- d'une expression d'initialisation $Init_\Phi$ qui affecte la valeur vrai à l'ensemble des variables associées aux états initiaux du comportement, affecte la valeur faux aux autres états et initialise à zéro l'ensemble des autres variables propositionnelles,

$$
\begin{array}{rcl}
\Phi & \widehat{=} & Init_\Phi \wedge \Box[\mathcal{N}]_{Var} \wedge F(Var) \\
\\
Var & \widehat{=} & V \cup VarE \cup VarI \cup VarC \cup VarTE \cup VarTS \\
\\
Init_\Phi & \widehat{=} & \wedge \ \bigwedge_{e \in Ei}(e = true) \\
 & & \wedge \ \bigwedge_{e \notin Ei}(e = false) \\
 & & \wedge \ \bigwedge_{(i,k,l) \in [1,n] \times [1,card(te_i)] \times [1,card(ts_i)]}(i_{ik} = 0 \wedge te_{ik} = 0 \wedge ts_{il} = 0) \\
\\
\mathcal{N} & \widehat{=} & \bigvee_{i \in [1,n]}(\mathcal{N}_1(i) \vee \mathcal{N}_2(i) \vee \mathcal{N}_3(i) \vee \mathcal{N}_4(i) \vee \mathcal{N}_5(i))
\end{array}
$$

$$
\begin{array}{rcllcl}
\mathcal{N}_1(i) & \widehat{=} & De(e_i) = nulle & \wedge & \bigvee_{k \in [1,card(te_i)]} \mathcal{DE}_1(i,k) \\
\mathcal{N}_2(i) & \widehat{=} & De(e_i) = rendezvous & \wedge & \bigvee_{k \in [1,card(te_i)]} \mathcal{DE}_2(i,k) \\
\mathcal{N}_3(i) & \widehat{=} & Ds(e_i) = sequentielle & \wedge & \bigvee_{k \in [1,card(ts_i)]} \mathcal{DS}_1(i,k) \\
\mathcal{N}_4(i) & \widehat{=} & Ds(e_i) = parallele & \wedge & \bigvee_{k \in [1,card(ts_i)]} \mathcal{DS}_2(i,k) \\
\mathcal{N}_5(i) & \widehat{=} & Ds(e_i) = tantque & \wedge & \bigvee_{k \in [1,card(ts_i)]} \mathcal{DS}_3(i,k)
\end{array}
$$

$$
F(Var) \ \widehat{=} \ \bigwedge_{(i,k) \in [1,n] \times [1,m]}(WF_{Var}\mathcal{DE}_1(i,k) \wedge \ldots \wedge WF_{Var}\mathcal{DS}_3(i,k))
$$

FIG. 7.2 – *Formule TLA associée à un comportement CAOLAC*

- d'une action $\mathcal{N}$ exécutée en permanence qui traduit les sémantiques entrantes et sortantes de l'ensemble des n états du comportement,

- d'une condition d'équité F sur la prise en compte et le tir de toutes les transitions du comportement. Rappelons que, dans un comportement concurrent, plusieurs actions (i.e. celles dont les gardes sont vraies) peuvent être entreprises simultanément. Néanmoins, le fait qu'elles puissent être entreprises ne signifie pas qu'elles le seront nécessairement. Par exemple, il se peut qu'une garde vraie ne soit jamais examinée par le moteur d'exécution, et donc, que son action ne soit jamais entreprise. Les conditions d'équité ont pour but de garantir que de telles situations indésirables ne se produisent. Par exemple, la condition d'équité faible *WF* (*weak fairness*) garantit que, soit l'action est entreprise, soit sa garde devient inévitablement fausse.

L'action $\mathcal{N}$ est décomposée en cinq formules $\mathcal{N}_1(i), \ldots, \mathcal{N}_5(i)$ traduisant les sémantiques des transitions entrantes et sortantes des n états. Chaque état possède $card(te_i)$ transitions entrantes et $card(ts_i)$ transitions sortantes. Les expressions $\mathcal{N}_1(i)$ et $\mathcal{N}_2(i)$ concernent les sémantiques entrantes. Elles testent, à l'aide de la fonction $De : E \rightarrow \{nulle, rendezvous\}$ définie au paragraphe 7.1.1, le statut de l'état e_i puis appliquent les formules $\mathcal{DE}_1(i,k)$ et $\mathcal{DE}_2(i,k)$ du paragraphe 7.2.1, pour chacune des m transitions entrantes. Les expressions $\mathcal{N}_3(i)$ à $\mathcal{N}_5(i)$ jouent, quant à elles, le même rôle pour les sémantiques sortantes. Ainsi, à chaque étape d'un programme et en fonction des possibilités d'évolution définies par le modèle, une ou plusieurs des $5*n$ formules de type $\mathcal{N}_j(i)$ sont exécutées. La condition F exprime que toutes ces actions sont exécutées avec une condition d'équité faible. Cela signifie qu'aucune transition sortante ne reste déclenchable indéfiniment sans être éventuellement exécutée à un instant futur, et qu'aucune transition entrante ne reste activée indéfiniment sans que l'état ne soit examiné à un instant futur.

7.3.2 Exemple

Dans ce paragraphe, nous appliquons la formule générique précédente à l'exemple du buffer synchronisé. Le paragraphe 7.3.2.2 raffine ce comportement. Finalement, le paragraphe 7.3.2.3 présente un certain nombre d'éléments de preuve associés à ce processus de raffinement.

7.3.2.1 Comportement buffer synchronisé

La figure 7.3 présente la formule TLA équivalente au comportement *bdtf* de la figure 7.1 page 156. Afin de simplifier les notations nous utilisons les identificateurs *Vide*, *Plein* et *Partiel* en lieu et place des trois variables d'état e_1, e_2 et e_3. Les valeurs des variables i_{11}, i_{22} et i_{23} représentent le nombre d'invocations *Put* présentes dans la file d'attente du comportement. Ces valeurs sont identiques. On les désigne à l'aide de l'identificateur *Put*. De même, l'identificateur *Get* est utilisé en lieu et place des variables i_{21}, i_{24} et i_{31}. Ce comportement ne contient que des états à sémantique entrante *nulle* et à sémantique sortante *séquentielle*. Nous omettons donc les variables ts_{ik} qui n'interviennent pas dans l'écriture d'une telle sémantique. Rappelons de plus, que la fonction $L : E \times \mathbb{N} \rightarrow 2^{ACT}$ associe à chaque transition une séquence d'instructions. Ainsi, les actions $L(Vide, 1)$, $L(Partiel, 2)$ et $L(Partiel, 3)$ correspondent à l'ajout effectif d'un élément dans le buffer. Les actions $L(Partiel, 1)$, $L(Partiel, 4)$ et $L(Plein, 1)$ correspondent, quant à elles, au retrait d'un élément.

7.3.2.2 Raffinement

Néanmoins, par rapport à la spécification d'un buffer de taille fixe, le programme Φ de la figure 7.3 est insuffisant. Par exemple, les formules $\mathcal{DS}_1(2,2)$ et $\mathcal{DS}_1(2,3)$ traduisent la prise en compte d'une invocation *Put* à partir de l'état *Partiel* et conduisent, respectivement, aux états *Partiel* et *Plein*. Une seule de ces actions doit être entreprise. Néanmoins, leurs gardes sont identiques et valent :

$$Enabled\,\mathcal{DS}_1(2,2) = Enabled\,\mathcal{DS}_1(2,3) = (Partiel = true) \wedge (Put > 0)$$

De ce fait ces formules sont exécutables simultanément. De même, une seule des formules $\mathcal{DS}_1(2,4)$ et $\mathcal{DS}_1(2,1)$ attachées à l'invocation *Get* doit être exécutée à partir de l'état *Partiel*.

La spécification initiale est légèrement incomplète. En effet, il est nécessaire de préciser la transition à activer suite à l'exécution de l'invocation *Put*. Deux types de solutions sont envisageables :

- une solution dite au plus tôt, qui consiste à faire remonter les conditions du choix de la transition dans les gardes,
- une solution dite au plus tard, qui consiste à effectuer le test après l'exécution de la transition, et à choisir l'état actif en fonction du résultat de ce test.

La solution au plus tôt semble, à première vue, plus satisfaisante, car elle précise, dès le début de transition, l'évolution future du comportement. Néanmoins, il n'est pas évident qu'une solution au plus tôt soit envisageable pour n'importe quel comportement. En effet, pour des actions complexes, la transformation de la condition peut ne pas être aisée. La solution au plus tard est, quant à elle, beaucoup plus algorithmique. C'est celle que nous retenons. Dans le cas du buffer de taille fixe, la

$$\Phi \;\widehat{=}\; Init_{\Phi} \wedge \Box[\mathcal{N}]_{Var} \wedge F(Var)$$

$$Var \;\widehat{=}\; \{Vide, Partiel, Plein, Put, Get, te_{11}, \ldots, te_{31}\}$$

$$\begin{array}{rcl}
Init_{\Phi} & \widehat{=} & Vide = true \\
 & & \wedge \quad (Partiel = false) \wedge (Plein = false) \\
 & & \wedge \quad (Put = 0) \wedge (Get = 0) \wedge (te_{11}, \ldots, te_{31} = 0)
\end{array}$$

$$\mathcal{N} \;\widehat{=}\; \mathcal{DE}_1(1,1) \vee \ldots \vee \mathcal{DE}_1(3,1) \vee \mathcal{DS}_1(1,1) \vee \ldots \vee \mathcal{DS}_1(3,1)$$

$$\begin{array}{rcl}
\mathcal{DE}_1(1,1) & \widehat{=} & (te_{11} > 0) \wedge (te_{11}^{'} = te_{11} \Leftrightarrow 1) \wedge (Vide^{'} = true) \\
\mathcal{DE}_1(2,1) & \widehat{=} & (te_{21} > 0) \wedge (te_{21}^{'} = te_{21} \Leftrightarrow 1) \wedge (Partiel^{'} = true) \\
\mathcal{DE}_1(2,2) & \widehat{=} & (te_{22} > 0) \wedge (te_{22}^{'} = te_{22} \Leftrightarrow 1) \wedge (Partiel^{'} = true) \\
\mathcal{DE}_1(2,3) & \widehat{=} & (te_{23} > 0) \wedge (te_{23}^{'} = te_{23} \Leftrightarrow 1) \wedge (Partiel^{'} = true) \\
\mathcal{DE}_1(2,4) & \widehat{=} & (te_{24} > 0) \wedge (te_{24}^{'} = te_{24} \Leftrightarrow 1) \wedge (Partiel^{'} = true) \\
\mathcal{DE}_1(3,1) & \widehat{=} & (te_{31} > 0) \wedge (te_{31}^{'} = te_{31} \Leftrightarrow 1) \wedge (Plein^{'} = true)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{DS}_1(1,1) & \widehat{=} & Vide = true \wedge Put > 0 \\
 & & \wedge \quad Vide^{'} = false \\
 & & \wedge \quad Put^{'} = Put \Leftrightarrow 1 \\
 & & \wedge \quad L(Vide, 1) \\
 & & \wedge \quad te_{21}^{'} = te_{21} + 1
\end{array}$$

$$\begin{array}{rcl}
\mathcal{DS}_1(2,2) & \widehat{=} & Partiel = true \wedge Put > 0 \\
 & & \wedge \quad Partiel^{'} = false \\
 & & \wedge \quad Put^{'} = Put \Leftrightarrow 1 \\
 & & \wedge \quad L(Partiel, 2) \\
 & & \wedge \quad te_{22}^{'} = te_{22} + 1
\end{array}$$

$$\begin{array}{rcl}
\mathcal{DS}_1(2,3) & \widehat{=} & Partiel = true \wedge Put > 0 \\
 & & \wedge \quad Partiel^{'} = false \\
 & & \wedge \quad Put^{'} = Put \Leftrightarrow 1 \\
 & & \wedge \quad L(Partiel, 3) \\
 & & \wedge \quad te_{31}^{'} = te_{31} + 1
\end{array}$$

$$\begin{array}{rcl}
\mathcal{DS}_1(2,4) & \widehat{=} & Partiel = true \wedge Get > 0 \\
 & & \wedge \quad Partiel^{'} = false \\
 & & \wedge \quad Get^{'} = Get \Leftrightarrow 1 \\
 & & \wedge \quad L(Partiel, 4) \\
 & & \wedge \quad te_{24}^{'} = te_{24} + 1
\end{array}$$

$$\begin{array}{rcl}
\mathcal{DS}_1(2,1) & \widehat{=} & Partiel = true \wedge Get > 0 \\
 & & \wedge \quad Partiel^{'} = false \\
 & & \wedge \quad Get^{'} = Get \Leftrightarrow 1 \\
 & & \wedge \quad L(Partiel, 1) \\
 & & \wedge \quad te_{11}^{'} = te_{11} + 1
\end{array}$$

$$\begin{array}{rcl}
\mathcal{DS}_1(3,1) & \widehat{=} & Plein = true \wedge Get > 0 \\
 & & \wedge \quad Plein^{'} = false \\
 & & \wedge \quad Get^{'} = Get \Leftrightarrow 1 \\
 & & \wedge \quad L(Plein, 1) \\
 & & \wedge \quad te_{23}^{'} = te_{23} + 1
\end{array}$$

$$\begin{array}{rcl}
F(\omega) & \widehat{=} & WF_{\omega}\mathcal{DE}_1(1,1) \wedge \ldots \wedge WF_{\omega}\mathcal{DE}_1(3,1) \wedge \\
 & & WF_{\omega}\mathcal{DS}_1(1,1) \wedge \ldots \wedge WF_{\omega}\mathcal{DS}_1(3,1)
\end{array}$$

FIG. 7.3 – *Formule TLA associée au comportement bdtf*

solution au plus tard revient à déclarer un sous-comportement *bdtf2* du comportement initial *bdtf*, dans lequel (Cf. figure 7.4) :

- les deux anciennes transitions *TPut1* et *TPut2* issues de l'état *Partiel* ont été supprimées,
- deux états *f1* et *f2* à sémantique *séquentielle* ont été introduits après l'ajout effectif d'un élément dans le buffer.

L'état *f1* comporte deux transitions *t1* et *t2* dont les gardes sont mutuellement exclusives. La première, déclenchable si le buffer n'est pas plein, conduit à l'état *Partiel*, tandis que la seconde, déclenchable dans les autres cas, conduit à l'état *Plein*. Le traitement de l'invocation *Get* s'effectue, de la même façon, en ajoutant un état séquentiel *f2* muni de deux transitions *t3* et *t4*. Les gardes de ces transitions sont associées, respectivement, aux expressions $IsEmpty = false$ et $IsEmpty = true$. Elles conduisent aux états *Partiel* et *Vide*.

La formule Ψ figure 7.5 définit le programme temporel associé au comportement *bdtf2*. Comme tout programme TLA, il comprend une expression d'initialisation $Init_\Psi$, un ensemble d'actions $\mathcal{M}$ et une condition d'équité G sur ces actions. L'ensemble *Var2* des variables utilisées dans ce programme comporte, en plus de l'ensemble *Var* des variables du programme Φ, les variables booléennes *f1* et *f2* associées aux deux nouveaux états introduits et les variables entières te_{41} et te_{51} associées aux transitions entrantes de ces deux états. La formule $Init_\Psi$ initialise ces variables. L'ensemble des actions $\mathcal{M}$ du programme Ψ comprend :

- les actions $\mathcal{DE}_1(1,1)$ à $\mathcal{DE}_1(3,1)$ héritées du programme Φ qui traduisent les sémantiques entrantes des états *Vide*, *Partiel* et *Plein* (bien que quatre des six anciennes transitions aient été redéfinies, le nouveau comportement *bdtf2* n'introduit pas de transition entrante supplémentaire pour ces états),
- les actions $\mathcal{DE}_1(4,1)$ et $\mathcal{DE}_1(5,1)$ qui traduisent la sémantique entrante des nouveaux états *f1* et *f2*,
- les actions $\mathcal{DS}_1(1,1)$ et $\mathcal{DS}_1(3,1)$ héritées du programme Φ traduisant la sémantique sortante des états *Vide* et *Plein*,
- les actions $\mathcal{M}_1$ à $\mathcal{M}_6$ qui définissent la sémantique sortante des états *Partiel*, *f1* et *f2*.

7.3.2.3 Eléments de preuve

Comme nous l'avons présenté au chapitre 2, la preuve qu'un programme TLA Ψ raffine un programme Φ revient à établir l'implication logique $\Psi \Rightarrow \Phi$. Nous ne donnons pas une démonstration complète de ce raffinement mais juste un certain nombre d'idées permettant d'en saisir le principe. Cette démonstration comprend trois étapes :

1. Il faut tout d'abord établir que les conditions initiales du programme raffiné entraînent celles du programme père, c'est à dire que $Init_\Psi \Rightarrow Init_\Phi$. La formule $Init_\Phi$ faisant partie de l'écriture de $Init_\Psi$, l'implication est évidente.
2. Dans un deuxième temps, il faut prouver que chaque action du programme Ψ raffine soit une action du programme initial Φ, soit une étape de bégaiement de ce programme. Les actions $\mathcal{DE}_1(1,1)$ à $\mathcal{DE}_1(3,1)$, ainsi que les actions $\mathcal{DS}_1(1,1)$ et $\mathcal{DS}_1(3,1)$ sont présentes dans les

```
behaviour bdtf2
 subbehav of bdtf {

 state Partiel {
  TPut redefines TPut1, TPut2 {
   invocation( Put(e) );
   BASE.Put(e);
   return;
   become( f1 );
  }
  TGet redefines TGet1, TGet2 {
   invocatinon( Get );
   return BASE.Get;
   become( f2 );
 }}

 state f1 {
  t1 {
   require( BASE.IsFull=false );
   become( Partiel );
  }
  t2 {
   require( BASE.IsFull=true );
   become( Plein );
 }}

 state f2 {
  t3 {
   require( BASE.IsEmpty=false );
   become( Partiel );
  }
  t4 {
   require( BASE.IsEMPTY=true );
   become( Vide );
}}}
```

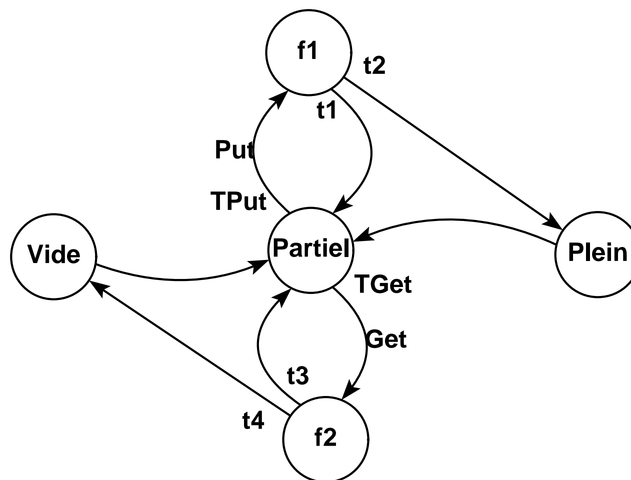


FIG. 7.4 – *Comportement bdtf2 raffinement du comportement bdtf*

$$\Psi \;\widehat{=}\; Init_{\Psi} \wedge \Box[\mathcal{M}]_{Var2} \wedge G(Var2)$$

$$Var2 \;\widehat{=}\; Var \cup \{f1, f2, te_{41}, te_{51}\}$$

$$\begin{aligned} Init_{\Psi} \;\widehat{=}\;& Init_{\Phi} \\ \wedge\;& (f1 = false) \wedge (f2 = false) \\ \wedge\;& (te_{41} = 0) \wedge (te_{51} = 0) \end{aligned}$$

$$\begin{aligned} \mathcal{M} \;\widehat{=}\;& \mathcal{DE}_1(1,1) \vee \ldots \vee \mathcal{DE}_1(3,1) \vee \mathcal{DE}_1(4,1) \vee \mathcal{DE}_1(5,1) \vee \\ & \mathcal{DS}_1(1,1) \vee \mathcal{DS}_1(3,1) \vee \mathcal{M}_1 \vee \ldots \vee \mathcal{M}_6 \end{aligned}$$

$$\mathcal{DE}_1(4,1) \;\widehat{=}\; (te_{41} > 0) \wedge (te_{41}^{'} = te_{41} \Leftrightarrow 1) \wedge (f1^{'} = true)$$
$$\mathcal{DE}_1(5,1) \;\widehat{=}\; (te_{51} > 0) \wedge (te_{51}^{'} = te_{51} \Leftrightarrow 1) \wedge (f2^{'} = true)$$

$$\begin{aligned} \mathcal{M}_1 \;\widehat{=}\;& Partiel = true \wedge Put > 0 \\ \wedge\;& Partiel^{'} = false \\ \wedge\;& Put^{'} = Put \Leftrightarrow 1 \\ \wedge\;& L(Partiel, 2) \\ \wedge\;& te_{41}^{'} = te_{41} + 1 \end{aligned}$$

$$\begin{aligned} \mathcal{M}_2 \;\widehat{=}\;& f1 = true \\ \wedge\;& IsFull = false \\ \wedge\;& f1^{'} = false \\ \wedge\;& te_{22}^{'} = te_{22} + 1 \end{aligned} \qquad \begin{aligned} \mathcal{M}_3 \;\widehat{=}\;& f1 = true \\ \wedge\;& IsFull = true \\ \wedge\;& f1^{'} = false \\ \wedge\;& te_{31}^{'} = te_{31} + 1 \end{aligned}$$

$$\begin{aligned} \mathcal{M}_4 \;\widehat{=}\;& Partiel = true \wedge Get > 0 \\ \wedge\;& Partiel^{'} = false \\ \wedge\;& Get^{'} = Get \Leftrightarrow 1 \\ \wedge\;& L(Partiel, 4) \\ \wedge\;& te_{51}^{'} = te_{51} + 1 \end{aligned}$$

$$\begin{aligned} \mathcal{M}_5 \;\widehat{=}\;& f2 = true \\ \wedge\;& IsEmpty = false \\ \wedge\;& f2^{'} = false \\ \wedge\;& te_{24}^{'} = te_{24} + 1 \end{aligned} \qquad \begin{aligned} \mathcal{M}_6 \;\widehat{=}\;& f2 = true \\ \wedge\;& IsEmpty = true \\ \wedge\;& f2^{'} = false \\ \wedge\;& te_{11}^{'} = te_{11} + 1 \end{aligned}$$

$$G(\omega) \;\widehat{=}\; F(\omega) \wedge WF_{\omega}\mathcal{DE}_1(4,1) \wedge WF_{\omega}\mathcal{DE}_1(5,1) \wedge WF_{\omega}\mathcal{M}_1 \wedge \ldots \wedge WF_{\omega}\mathcal{M}_6$$

FIG. 7.5 – *Formule TLA associée au comportement bdtf2*

deux programmes. Leur raffinement ne pose donc pas de problème. Les actions $\mathcal{DE}_1(4,1)$ et $\mathcal{DE}_1(5,1)$ manipulent uniquement, quant à elles, des variables introduites par le programme Ψ. Ce sont donc des raffinements d'étapes de bégaiement du programme Φ. Il reste à traiter les actions $\mathcal{M}_1$ à $\mathcal{M}_6$. Nous montrons, pour cela, que l'action $\mathcal{M}_1$ conduit nécessairement à l'action $\mathcal{M}_2$ ou à l'action $\mathcal{M}_3$, c'est à dire que soit $\mathcal{M}_1 \leadsto \mathcal{M}_2$ soit $\mathcal{M}_1 \leadsto \mathcal{M}_3$, et que de même, l'action $\mathcal{M}_4$ conduit nécessairement à l'action $\mathcal{M}_5$ ou à l'action $\mathcal{M}_6$. Puis nous montrons que ces quatre expressions sont des raffinements respectifs de $\mathcal{DS}_1(2,2)$, $\mathcal{DS}_1(2,3)$, $\mathcal{DS}_1(2,4)$ et $\mathcal{DS}_1(2,1)$.
L'exécution de l'action $\mathcal{M}_1$ provoque, en particulier, l'exécution de l'action $te_{41}^{'} = te_{41} + 1$. Cela a pour effet de rendre l'action $\mathcal{DE}_1(4,1)$ déclenchable. En effet, l'expression *Enabled* $\mathcal{DE}_1(4,1)$, qui vaut $te_{41} > 0$, devient vraie. Etant donné qu'aucune autre action de programme Ψ ne modifie la valeur de la variable te_{41}, les conditions d'équité $G(\omega)$ garantissent que l'action $\mathcal{DE}_1(4,1)$ est entreprise après l'action $\mathcal{M}_1$. En d'autres termes, on obtient $\mathcal{M}_1 \leadsto \mathcal{DE}_1(4,1)$. On constate alors que les gardes des actions $\mathcal{M}_2$ et $\mathcal{M}_3$ sont égales respectivement à :

$Enabled\,\mathcal{M}_2 = (f1 = true \wedge IsFull = false)$ et à

$Enabled\,\mathcal{M}_3 = (f1 = true \wedge IsFull = true)$

Comme l'exécution de l'action $\mathcal{DE}_1(4,1)$ provoque en particulier l'exécution de $f1^{'} = true$ et que la fonction *IsFull* retourne soit *true* soit *false*, nécessairement une des deux actions $\mathcal{M}_2$ ou $\mathcal{M}_3$ est exécutée. Par transitivité, on obtient donc soit $\mathcal{M}_1 \leadsto \mathcal{DE}_1(4,1) \leadsto \mathcal{M}_2$, soit $\mathcal{M}_1 \leadsto \mathcal{DE}_1(4,1) \leadsto \mathcal{M}_3$.
Par ailleurs, les actions de la formule $\mathcal{DS}_1(2,2)$ du programme Φ sont :

$$Partiel^{'} = false \wedge Put^{'} = Put \Leftrightarrow 1 \wedge L(Partiel, 2) \wedge te_{22}^{'} = te_{22} + 1$$

On constate que ces quatre sous-actions se retrouvent dans les actions $\mathcal{M}_1$ et $\mathcal{M}_2$, dont on a prouvé qu'elles étaient nécessairement exécutées en séquence. Les actions supplémentaires de cette séquence concernent les variables introduites dans le programme Ψ. Elles ne modifient donc pas le comportement du programme initial Φ. La séquence $\mathcal{M}_1 \leadsto \mathcal{DE}_1(4,1) \leadsto \mathcal{M}_2$ est donc un raffinement de l'action $\mathcal{DS}_1(2,2)$. De la même façon, on établit que les séquences $\mathcal{M}_1 \leadsto \mathcal{DE}_1(4,1) \leadsto \mathcal{M}_3$, $\mathcal{M}_4 \leadsto \mathcal{DE}_1(5,1) \leadsto \mathcal{M}_5$ et $\mathcal{M}_4 \leadsto \mathcal{DE}_1(5,1) \leadsto \mathcal{M}_6$ sont des raffinements respectifs des actions $\mathcal{DS}_1(2,3)$, $\mathcal{DS}_1(2,4)$ et $\mathcal{DS}_1(2,1)$.

3. Finalement, il faut prouver que le programme raffiné Ψ implique les conditions d'équité du programme Φ, c'est à dire que $\Psi \Rightarrow F(Var)$. De façon intuitive, on constate que toutes les actions ou toutes les séquences d'actions du programme Ψ raffinent des actions du programme Φ exécutées avec les mêmes conditions d'équités (c'est à dire une condition d'équité faible). Bien que nous ne nous soyons pas penché plus en détail sur les différents mécanismes de déduction associés à la logique temporelle d'actions (Cf. en particulier [Lam91, Lam94] pour une telle présentation), nous conjecturons que dans ce cas, l'implication peut être établie.

7.4 Conclusion

Dans ce chapitre, nous avons présenté une sémantique pour le modèle de synchronisation du langage CAOLAC. Nous avons choisi pour cela la logique temporelle d'actions [Lam91, Lam94]

de Lamport qui est une logique pour la description d'actions concurrentes. Chaque comportement du langage CAOLAC décrit une politique de synchronisation pour les méthodes d'une classe d'objets concurrents. Cette politique est définie à partir de modèles états/transitions. Afin de faciliter la description du parallélisme intra-objet, différentes sémantiques d'état ont été définies au chapitre 6. Chacune d'elles propose des règles pour la prise en compte des transitions entrantes (sémantique entrante) et le tir des transitions sortantes (sémantique sortante). Le paragraphe 7.2 de ce chapitre propose pour les cinq sémantiques (deux entrantes et trois sortantes) un ensemble de formules TLA traduisant ces règles. Le paragraphe 7.1 définit l'ensemble des variables utilisées dans l'écriture de ces formules. Elles décrivent la structure du comportement CAOLAC. Les principales variables sont les booléens e_i pour décrire l'activité des états, les variables entières i_k pour le nombre d'invocations de méthodes reçues, les variables entières te_{ik} pour le nombre d'instances de transitions entrantes consommables et les variables ts_{ik} pour le nombre d'instances de transitions sortantes exécutables. Chaque sémantique entrante et sortante est alors une action d'une formule TLA décrivant le comportement de synchronisation. La construction de cette formule est présentée au paragraphe 7.3.

Bien que la sémantique proposée en première approche dans ce chapitre soit, par certains aspects, incomplète (les points à préciser concernent, entre autres, la formalisation des historiques, le transfert des invocations entre un comportement et une classe ou entre deux comportements, la sémantique d'invocation, ...), nous pensons qu'elle permet de préciser formellement les règles d'évolution d'un comportement CAOLAC. De plus, le système de preuves associé à TLA permet de donner un support formel à la notion de raffinement. Bien que nous ne l'ayons pas complètement étudié, nous pensons qu'il permettra de vérifier des propriétés intéressantes sur le raffinements des hiérarchies de comportement. Nous avons commencé à illustrer ce mécanisme au paragraphe 7.3.2 à l'aide d'un exemple. Il est également envisageable que les mécanismes de preuves de propriétés de sûreté et de vivacité, qui se ramènent en TLA à des formules toujours vraies ou inévitablement vraies (Cf. paragraphe 2.2.2), soient applicables aux politiques de synchronisation. Néanmoins, de nombreux problèmes tant théoriques que pratiques, restent posés. Par exemple, les conditions d'équité spécifiées dans la sémantique soulèvent un certain nombre de questions. Comme pour tout problème de ce type, il n'est pas évident de garantir que l'implantation respecte les mêmes conditions d'équité que le modèle. En effet, les sources d'indéterminisme introduites par un système distribué sont telles qu'il est difficile de prévoir de façon certaine que l'exécution d'un programme est exactement conforme à son modèle (on peut citer, comme source d'indéterminisme, l'ordonnancement des processus et des fils d'activité exécutés en pseudo-parallélisme sur un même processeur, les accès concurrents à une mémoire ou à un système de fichiers partagés, les communications réseaux éventuellement non fiables, les délais de transmission non bornés, ...). Le second problème posé par la définition de cette sémantique TLA concerne la granularité des actions du modèle. En effet, selon que l'action associée à une transition est une méthode complète, un bloc d'instructions ou une simple opération, les possibilités d'entrelacements décrites par la formule TLA ne sont pas les mêmes. Un grain fin amène une sémantique relativement conforme à la réalité, mais produit des formules TLA lourdes à gérer. Inversement, un grain gros facilite l'exploitation des formules, mais pénalise la fidélité de représentation. Il est donc nécessaire de trouver un compromis entre entre la fidélité de représentation et la facilité d'exploitation. Une poursuite de ces travaux pourrait consister à comparer, pour un même algorithme, différents degrés de précision dans l'écriture des formules TLA et à étudier l'apport du raffinement.

Quatrième partie

Etudes de cas

Chapitre 8

Calcul d'arbres couvrants

Dans ce chapitre, nous appliquons les concepts présentés dans les parties deux et trois de ce mémoire, à l'exemple concret de la conception d'algorithmes répartis d'élection avec calcul d'arbre couvrant.

Nous nous intéressons plus particulièrement à la technique de parcours de réseau par vagues [Tel94][Gom95] et nous envisageons trois versions de l'algorithme d'élection avec calcul d'arbre couvrant : une dite à vagues inondantes issue des travaux de Laurent Bonnet [Bon94, BDFS96, BDFS97], une dite à régions et une dite à régions et retournement de branches. Nous montrons que ces trois versions peuvent être dérivées d'un même programme de niveau groupe. Après avoir rappelé la démarche de conception, nous présentons les algorithmes au paragraphe 8.2. Nous abordons, au paragraphe 8.3, la spécification de niveau groupe, à l'aide d'une hiérarchie de trois programmes. Le paragraphe 8.4 présente alors l'implantation de ces programmes au niveau objet. Le paragraphe 8.5 fournit le code du niveau méthode. Finalement, le paragraphe 8.6 conclut cette étude de cas.

8.1 Rappel de la démarche

La démarche de conception suggérée est un processus incrémental qui comprend trois niveaux méthodologiques: groupe, objet et méthode (Cf. paragraphe 4.1). L'idée de base est, dans un premier temps, de définir le ou les buts globaux de l'application (c'est à dire du groupe d'objets participant à sa réalisation). Dans un second temps, on propose de déduire, à partir de ces buts globaux, les buts locaux de chaque objet dans le groupe. Finalement, dans une troisième étape, le niveau méthode définit les algorithmes, c'est à dire le corps des méthodes, permettant d'atteindre ces buts locaux. Le processus est donc essentiellement descendant et procède par raffinements successifs. Chaque niveau précise les éléments suivants :

– le niveau groupe définit les objectifs de l'algorithme réparti. Pour cela, on utilise un formalisme épistémique à base d'opérateurs de connaissance. On traduit les buts globaux en niveaux de connaissances pour le groupe. On fournit les actions globales qui permettent d'atteindre ces niveaux. Ces actions manipulent des structures de données réparties. Ce sont par exemple des arbres, des files ou des anneaux dont les éléments sont répartis sur l'ensemble des objets du groupe. Les actions sont construites à partir de quatre structures de contrôle

génériques définies au chapitre 5 : la phase, la conditionnelle, l'itération et la récursion distribuées. Ces structures de contrôle généralisent à un niveau réparti les structures de base de l'algorithmique séquentielle. Les structures de données, les actions ainsi que les niveaux de connaissance sont exprimés à l'aide de la notion de programme à base de connaissance de niveau groupe définie au paragraphe 4.5.

- le niveau objet fournit la synchronisation des méthodes qui permet de réaliser le comportement de niveau groupe. Cette synchronisation est définie à l'aide du langage CAOLAC présenté au chapitre 6. Ce langage se présente sous la forme d' un protocole de niveau méta-objet et propose un formalisme à base d'états et de transitions pour contrôler les exécutions de méthodes au sein d'un objet concurrent.

- le niveau méthode fournit l'implantation des méthodes synchronisées.

8.2 Présentation des algorithmes

Les trois versions de l'algorithme d'élection avec calcul d'arbre couvrant que nous envisageons ont pour but de procéder à l'élection d'un site parmi l'ensemble des sites d'un réseau et d'organiser ce dernier en une structure arborescente telle que la racine de l'arbre soit l'élu, c'est à dire le site vainqueur de l'élection. Chaque site est représenté par un objet et le réseau est un groupe d'objets. Ces algorithmes manipulent une structure de données réparties de type arbre. Ils utilisent des structures de contrôle de type phase, itération distribuée et récursion distribuée (Cf. chapitre 5). Rappelons, brièvement, que l'itération distribuée permet de répéter un comportement de groupe, tandis que la récursion distribuée permet de parcourir récursivement un ensemble d'objets distribués. Dans la suite de ce chapitre, nous utilisons le terme vague pour désigner ce dernier comportement.

Le contrôle dans le groupe d'objets étant complètement décentralisé, plusieurs objets peuvent démarrer simultanément l'algorithme. Néanmoins, afin de faciliter la présentation, nous exposons d'abord le cas où un seul site démarre la construction. Cette description est valable pour les trois versions de l'algorithme. Puis, au paragraphe 8.2.2, nous présentons le cas où plusieurs lancements sont effectués simultanément. Nous introduisons alors les comportements des trois versions. Bien entendu, la présentation avec un seul initiateur est un cas particulier de celle avec plusieurs initiateurs.

8.2.1 Algorithmes avec un seul initiateur

Un site qui décide d'initialiser la construction de l'arbre est appelé initiateur. Il émet une requête en direction de tous ses voisins pour les informer qu'ils sont ses fils potentiels. Cet algorithme est dit à vague, car la requête est propagée de proche en proche sur tous les sites du réseau. Un fils potentiel reçoit éventuellement plusieurs propositions et décide d'en accepter une. Cela peut être, par exemple, la première ou la meilleure selon un critère à spécifier. L'émetteur de la proposition acceptée devient le père du site accepteur. Celui-ci est donc considéré comme un de ses fils. Le lien entre le père et le fils est incorporé à l'arbre.

Chaque fils décide de propager la vague à l'ensemble de ses voisins hormis son père. Une requête atteignant un site connaissant déjà un père est rejetée. Lorsqu'il n'y a plus de voisin à atteindre

ou lorsque tous ont rejeté la proposition, le site fils retourne la vague à son père. La vague reflue donc de proche en proche vers l'initiateur de la construction.

Exemple

La figure 8.1 présente l'exemple du déroulement d'un vague. Le site 1 est initiateur. Il propage la vague vers ses fils 2, 3 et 4. Chacun d'entre eux repropage la valeur vers leurs propres voisins moins leur père, c'est à dire 1. Ainsi 2 propage vers 5 et 3 propage vers 5 et 6. 4 ne propage vers aucun site et retourne simplement la vague vers 1. 5 reçoit donc deux propositions de vague : une de 2 et une de 3. Il en accepte une seule, par exemple celle de 2, et retourne l'autre à son émetteur. Il repropage alors la vague vers 7. La vague continue jusqu'à ce qu'il n'y ait plus de site à visiter. Elle reflue alors vers l'initiateur 1. On crée ainsi des fronts de vague qui se propagent et refluent au sein du réseau. Selon les temps de propagation dans les différents liens et selon les temps de traitement des différents sites, deux exécutions à partir d'un même initiateur peuvent construire deux arbres différents. Par exemple, la figure 8.2 présente une deuxième exécution possible de la vague avec une même topologie de réseau. Dans cet exemple, le site 8 est atteint par la proposition issue de 10 avant celle issue de 6. Il enregistre donc 10 comme son père et rejette 6.

8.2.2 Algorithmes avec plusieurs initiateurs

Pour compléter cet algorithme, il faut prendre en compte le cas où plusieurs initiateurs décident de construire simultanément un arbre couvrant. On suppose pour cela que chaque site possède un identifiant unique et que les identifiants sont totalement ordonnés. Deux vagues se rencontrent lorsqu'un site appartenant à un initiateur reçoit une proposition d'un autre initiateur. Plusieurs solutions sont alors envisageables : la vague gagnante peut recouvrir la vague perdante (version vagues inondantes), les deux vagues peuvent refluer (version vagues à régions) ou la vague gagnante peut retourner une partie de la vague perdante (version vagues à régions et retournements).

8.2.2.1 Version à vagues inondantes

Cette version de l'algorithme est la plus simple des trois. Lorsque deux vagues se rencontrent, la vague perdante reflue et la vague gagnante la recouvre. La vague gagnante reconstruit donc un arbre dans la partie visitée par la vague perdante. Quand un site S appartenant à l'arbre d'un initiateur I_1 reçoit une requête issue d'un initiateur I_2, il compare les identificateurs des deux initiateurs. Si celui de I_2 est inférieur à celui de I_1, alors la vague initiée par I_2 perd et S retourne une notification indiquant que la vague I_2 a perdu. Inversement, si I_2 est supérieur à I_1, alors S enregistre I_2 comme étant son nouveau père et propage la vague initiée par I_2. De ce fait, toutes les vagues perdantes refluent vers leur initiateur respectif, tandis que la vague gagnante est la seule à recouvrir le réseau complet.

Exemple

Les figures 8.3 et 8.4 présentent le déroulement d'une vague avec deux initiateurs : les sites 1 et 10. Dans l'étape représentée, 6 appartient à la vague issue de 1. Il reçoit, par l'intermédiaire de 8, une proposition de 10. Il choisit donc de s'affilier à cette dernière proposition. Il enregistre 8 comme son nouveau père et propage la proposition à tous ses voisins hormis 8. En particulier, il

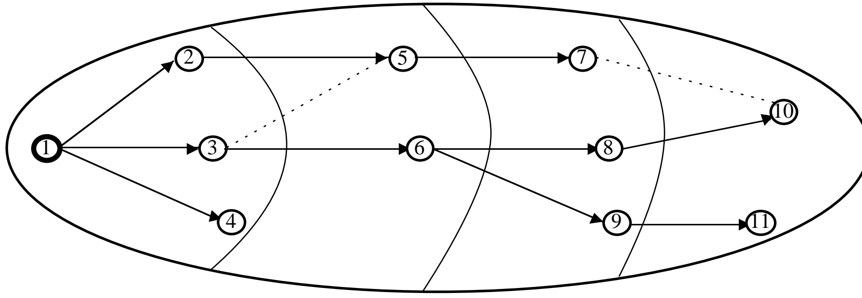


FIG. 8.1 – *Exécution de la vague avec un seul initiateur*

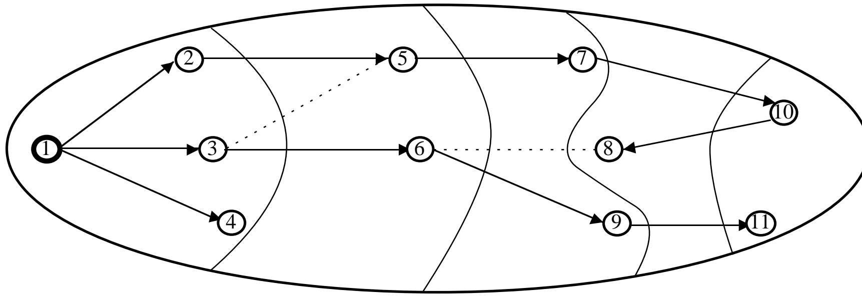


FIG. 8.2 – *Deuxième exécution possible de la vague de la figure 8.1*

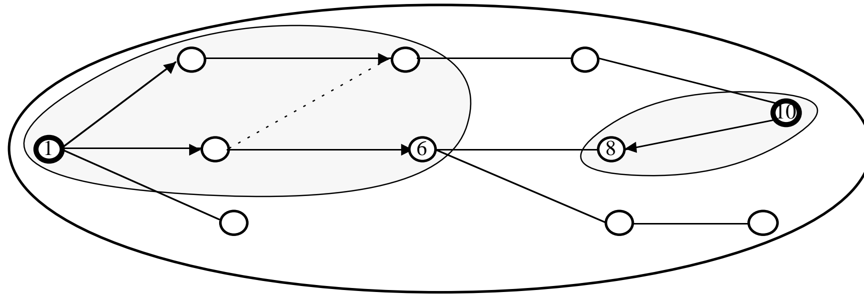


FIG. 8.3 – *Exécution partielle de la vague avec deux initiateurs*

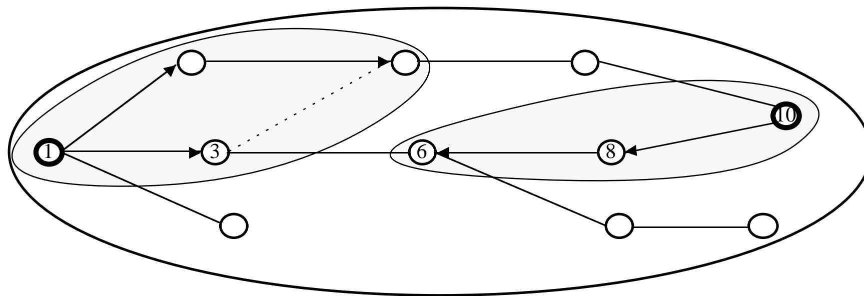


FIG. 8.4 – *La vague issue de 10 recouvre celle issue de 1*

la transmet à 3 qui se retrouve dans la même situation : il choisit 10 comme son nouveau père et recouvre l'arbre issu de 1.

8.2.2.2 Version à régions

Dans cette version, lorsque deux vagues se rencontrent, elles refluent vers leur initiateur respectif. Chaque phase de reflux véhicule l'identité de l'initiateur de la vague rencontrée. Chaque initiateur reçoit donc les identités de toutes les vagues concurrentes rencontrées. Cet algorithme est dit à région, car tous les sites visités par une vague forment une région qui est marquée par son initiateur. On dit alors que les sites sur lesquels deux vagues se sont rencontrées sont sur la frontière.

Exemple

La figure 8.5 présente une exécution de l'algorithme à régions avec quatre initiateurs : 1, 5, 6 et 10. Chaque initiateur développe sa région en incorporant un certain nombre de sites. A la fin de la phase de propagation/reflux des quatre vagues, la zone du site 1 est adjacente avec les zones des sites 5 et 6, celle de 5 l'est avec celles de 1 et 10, celle de 6 avec celles de 1 et 10 et celle de 10 avec celles de 5 et 6 (Cf. figure 8.6).

Poursuite de l'algorithme

A la fin de cette première phase de propagation/reflux, chaque initiateur connaît un ensemble d'initiateurs avec qui il possède une frontière. On constitue alors un réseau virtuel en prenant comme sites les initiateurs de chaque région, et comme liens, les liens entre régions adjacentes. On relance alors l'algorithme sur ce réseau virtuel. Chaque site détermine s'il est initiateur en fonction de l'identificateur de ses voisins. Seuls les sites dont tous les voisins ont un identificateur inférieur au leur se déclarent initiateur. Dans l'exemple de la figure 8.6, il ne reste plus qu'un initiateur (le site 10). Il propage une vague et incorpore les sites 1, 5 et 6.

Le cas échéant, on peut obtenir des réseaux virtuels avec plusieurs initiateurs. On construit alors de nouvelles régions et on itère ce processus jusqu'à obtenir un réseau virtuel avec un seul initiateur.

Comme la version par vagues inondantes, l'algorithme par régions élit l'initiateur d'identificateur le plus élevé. Néanmoins, l'arbre couvrant construit est à plusieurs niveaux. Le premier niveau correspond à la dernière phase de propagation/reflux exécutée. Chaque site de ce niveau est un initiateur de la phase précédente. Il est donc associé à un sous-arbre couvrant. Eventuellement, chaque site de ce sous-arbre est lui-même un initiateur d'une phase antérieur. L'arbre couvrant correspondant à l'exécution des figures 8.5 et 8.6 comprend deux niveaux. Il est représenté figure 8.7.

8.2.2.3 Version par régions et retournement de branches

Comme dans la version précédentes, lorsque deux vagues de l'algorithme par régions et retournement de branches se rencontrent, elles refluent vers leur initiateur respectif. Dans cette version, le site de la vague perdante situé sur la frontière initie une phase de reflux qui retourne la branche empruntée lors de la phase de propagation. Ainsi, dans cette branche, les pères deviennent des fils

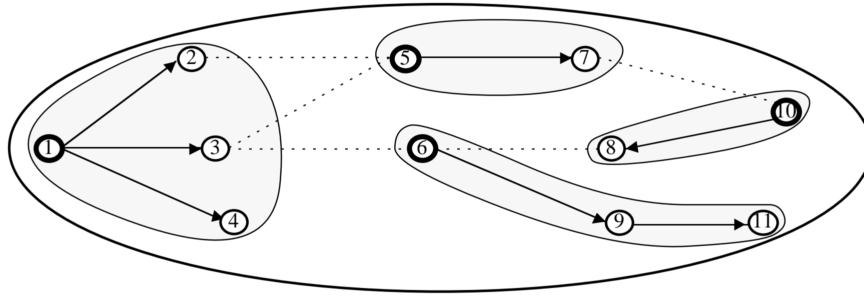


FIG. 8.5 – *Algorithme de calcul d'arbre couvrant par régions*

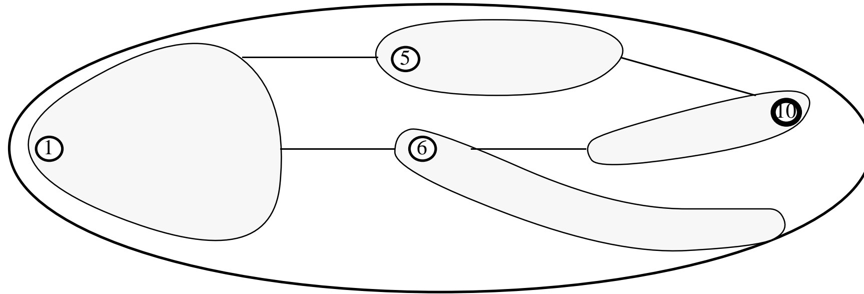


FIG. 8.6 – *Régions adjacentes après une phase de propagation/reflux*

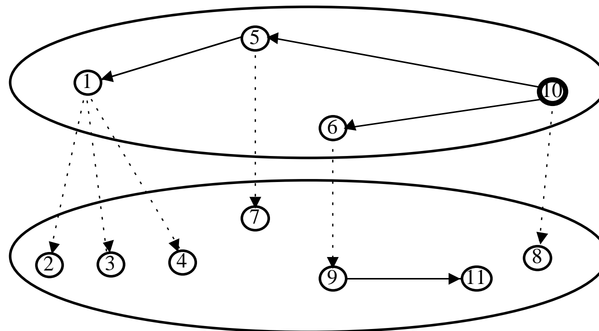


FIG. 8.7 – *Arbre couvrant après deux phases de propagation/reflux*

et les fils deviennent des pères. Le retournement se termine à la fin du reflux, c'est à dire lorsque la branche a été remontée jusqu'à l'initiateur.

Exemple

La figure 8.8 présente l'exécution de cette version avec quatre initiateurs : 1, 5, 6 et 10. La figure 8.9 donne une exécution possible des phases de reflux des vagues perdantes. Par exemple, la branche entre 5 et 7 appartient à la région issue de 5. Cette région est adjacente à la région issue de 10. La région issue de 5 est donc perdante et la branche entre 5 et 7 est retournée. De même pour la branche entre 1 et 2.

Néanmoins, il se peut que dans une même région, deux ou plusieurs retournements concurrents soient opérés. C'est le cas, par exemple, des branches 1-2 et 1-3 de la région 1. En effet, celle-ci est en contact avec les zones 5 et 6 qui lui imposent, toutes les deux, un retournement. Dans ce cas, lorsqueles deux retournements aboutissent à l'initiateur, celui-ci choisit de n'en conserver qu'un (par exemple le premier), et d'annuler ses liens avec les derniers sites des autres retournements. Ainsi, dans l'exemple de la figure 8.9, 1 accepte le retournement provenant de 2 et détruit dans l'arbre, son lien avec 3.

8.3 Spécification de niveau groupe

Dans ce paragraphe, nous présentons la spécification en terme de connaissances des trois versions de l'algorithme d'élection avec calcul d'arbre couvrant. Nous définissons, tout d'abord, les hypothèses nécessaires au bon déroulement de ces algorithmes ainsi que le but global à atteindre. Puis, nous exprimons, à l'aide de programmes à base de connaissances, trois raffinements du comportement de groupe à mettre en place.

8.3.1 Hypothèses

Le réseau est représenté par un graphe orienté $G = (S, L)$. S est un ensemble de sites et L un ensemble de liaisons. L'orientation permet de déterminer le sens de circulation des données dans le réseau. On suppose, comme c'est le cas dans la plupart des réseaux actuels, que les liens de communication sont bidirectionnels. Ils sont donc représentés par deux arcs.

Pour que l'algorithme se déroule de façon correcte, il faut que le réseau vérifie un certain nombre de propriétés que nous désignons sous le terme d'hypothèses de bon fonctionnement. Ce sont les conditions minimales qui doivent être respectées pendant toute la durée de l'exécution répartie. Nous les notons *HR* (acronyme de hypothèses de réseau). Elles comprennent six prédicats épistémiques ($HR \mathrel{\widehat{=}} H_1 \wedge \ldots \wedge H_6$) définis ci-après. Nous utilisons deux fonctions *id* et *voisins*. Elles associent à chaque site du réseau, respectivement, un identificateur et un ensemble de voisins.

- Le réseau est défini localement par la connaissance des voisinages, donc tous les sites connaissent un ensemble de voisins :
  $H_1 \mathrel{\widehat{=}} \forall i \in S,\ K_i(voisins(i))$
- Les liens de communication sont symétriques, donc tout site est également voisin de ses voisins :
  $H_2 \mathrel{\widehat{=}} \forall i, j \in S,\ j \in voisins(i) \Rightarrow i \in voisins(j)$

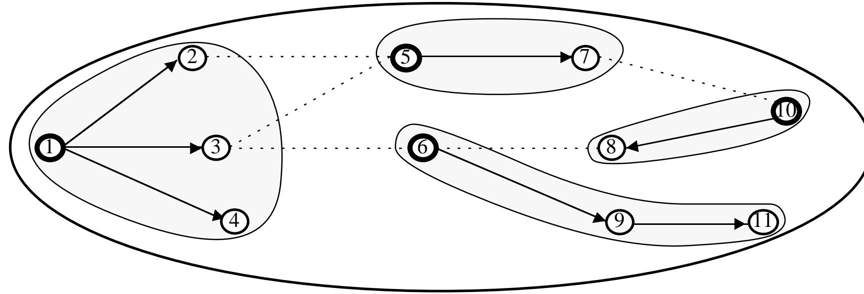


FIG. 8.8 – *Algorithme de calcul d'arbre couvrant par régions et retournement de branches*

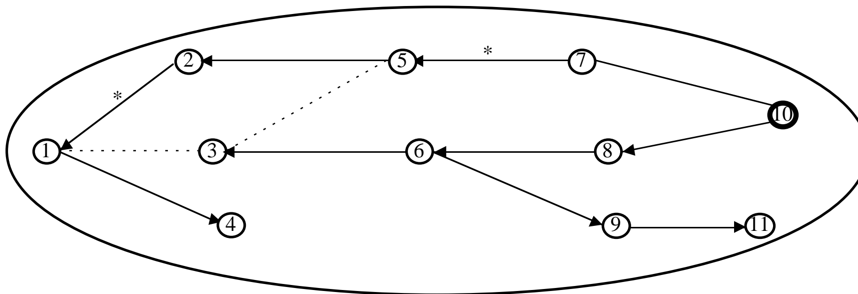


FIG. 8.9 – *Retournement de branches*

- Les identificateurs de sites sont totalement ordonnés :
  $H_3 \hat{=} \forall i, j \in S,\ id(i) = id(j) \vee id(i) > id(j) \vee id(i) < id(j)$

- Chaque site connaît son identité et l'identité de ses voisins :
  $H_4 \hat{=} \forall i \in S,\ K_i(id(i)) \wedge \forall j \in voisins(i), K_i(id(j))$

- Il n'y a pas d'homonyme :
  $H_5 \hat{=} \forall i, j \in S,\ i \neq j \Rightarrow id(i) \neq id(j)$

- Le réseau est connexe, c'est à dire que l'on peut atteindre tous les sites à partir de n'importe lequel d'entre eux. La notation $voisins^k(i)$ désigne l'ensemble des voisins à distance k du site i :
  $H_6 \hat{=} \forall i \in S,\ \bigcup_{k=0}^{\infty} voisins^k(i) = S$

Le prédicat HR constituent donc un invariant du comportement global. Nous le notons dans la section **environment** du programme à base de connaissances de niveau groupe. On utilise comme structure de données répartie un ensemble constant S d'objets.

group vars
  const S : **set of object**
environment
  $HR \hat{=} H_1 \wedge \ldots \wedge H_6$

8.3.2 But global

Le but de l'algorithme est de construire un arbre couvrant d'un réseau respectant les hypothèses HR. L'arbre couvrant $Arbre = (Sites, Liaisons)$ est donc un sous-graphe du graphe complet G et doit être tel que :

- $Sites = S$: l'arbre couvre la totalité du réseau,

- $Liaisons \subseteq L$: les liaisons retenues font partie du graphe initial,

- $|Sites| = |Liaisons| + 1$: il y a un sites de plus que de liaisons (c'est la définition d'un arbre).

Expression du but global en terme de connaissances

Dans un arbre, chaque site a deux types de voisins : un père et des fils. Le père désigne le site de niveau supérieur dans l'arbre, tandis que les fils sont les sites de niveau inférieur. On note *père* et *fils* les fonctions qui, respectivement, associent un père et un ensemble de fils à chaque site de l'arbre.

De façon globale, on peut distinguer trois types de sites : la racine, les feuilles et les nœuds.

- la racine est unique et n'a pas de site père,

- les feuilles sont les sites terminaux et n'ont pas de sites fils,

- les nœuds sont les sites intermédiaires et ont un unique père et un ou plusieurs fils.

Afin de compléter l'écriture des fonctions *père* et *fils*, on convient que le père de la racine est égal à la valeur nulle et que les ensembles de fils des feuilles sont vides.

Exprimé en terme de connaissance, le but global de l'algorithme est, d'une part, de calculer un arbre couvrant, c'est à dire de faire en sorte que tous les sites connaissent un père et des fils, et d'autre part, d'élire la racine, c'est à dire de faire en sorte que tous les sites connaissent la racine. Il faut, de plus, que les ensembles de fils soient disjoints, c'est à dire qu'un site ne soit pas fils de deux pères différents. En notant $ArbreConnu$ le prédicat épistémique représentant le but global de l'algorithme, on obtient :

$$ArbreConnu \;\widehat{=}\; (\forall i \in Sites,\ K_i(p\grave{e}re(i)) \wedge K_i(fils(i))) \wedge (E_{Sites}(racine)) \wedge (\bigcup_{i \in Sites} fils(i) = \oslash)$$

8.3.3 Raffinement du but global

Plutôt que d'envisager un algorithme qui calcule un arbre couvrant puis qui se termine, on souhaite pouvoir reconstruire continuellement un couverture du réseau. Ainsi l'algorithme doit pouvoir être relancé après, par exemple, un changement de topologie ou une panne de site. On va donc construire plusieurs arbres couvrants du réseau. Le but intermédiaire de l'algorithme est donc de construire un arbre couvrant d'indice n noté $Arbre_n = (Sites_n, Liaisons_n)$, et dont la connaissance est représentée par le prédicat $ArbreConnu_n$ suivant :

$$ArbreConnu_n \;\widehat{=}\; (\forall i \in Sites_n,\ K_i(p\grave{e}re_n(i)) \wedge K_i(fils_n(i))) \wedge (E_{Sites_n}(racine_n)) \wedge (\bigcup_{i \in Sites_n} fils_n(i) = \oslash)$$

8.3.4 Premier programme de niveau groupe

La figure 8.10 présente une première version du programme de niveau groupe. Nous notons *Arbre* la méthode qui réalise la construction de l'arbre. Nous utilisons une variable positive *Epoque* pour distinguer les constructions d'arbre successives. L'algorithme étant exécuté en permanence, on utilise une boucle *while* infinie. Avant toute construction, on connaît les arbres d'époques inférieures à l'époque courante. La construction d'époque courante est réalisée par la méthode *Arbre* qui est une phase gardée par la condition *Demande*. Celle-ci vaut vraie dès qu'un ou plusieurs sites décident d'initier une construction dans l'époque courante. A la fin de la méthode, tous les arbres d'époques inférieures ou égales à l'époque courante sont connus. Finalement, avant de réitérer ce comportement, on incrémente le numéro d'époque courante. De façon théorique, à la fin de la boucle, on connaît une infinité d'arbres.

Ce premier programme est très général. Il est commun à de nombreuses variantes d'algorithmes de construction d'arbres. En particulier, il recouvre les trois versions (par vagues inondates, par régions, par régions et retournement de branches) évoquées précédemment.

8.3.5 Premier raffinement du programme de niveau groupe

Le premier raffinement du programme de niveau groupe précédent consiste à introduire le comportement de la méthode *Arbre*. Nous avons vu que les trois versions de l'algorithme utilisent des vagues. Chaque vague parcourt récursivement un ensemble de sites et comprend une phase de propagation et une phase de reflux. L'idée de base est d'utiliser un tel parcours pour la phase correspondant à la méthode *Arbre*.

group vars
  const S : **set of object** ;
  Epoque : **Positive** = 0
environment
  $HR \mathrel{\hat=} H_1 \wedge \ldots \wedge H_6$

method Main
begin
  {}
  while true **do**
    $\{\forall n < Epoque,\ ArbreConnu_n\}$
    choice
      Demande = true : self.**Arbre**
    endchoise
    $\{\forall n \leq Epoque,\ ArbreConnu_n\}$
    Epoque := Epoque + 1
  enddo
  $\{\forall n,\ ArbreConnu_n\}$
end

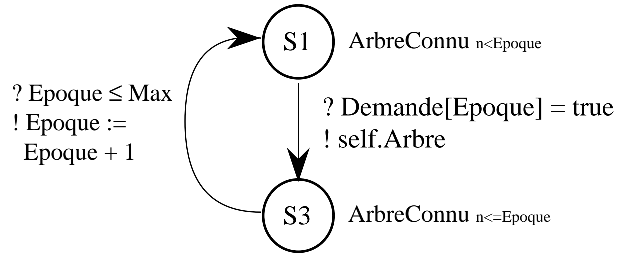


FIG. 8.10 – *Premier programme de niveau groupe*

Selon les versions de l'algorithme, une ou plusieurs vagues sont utilisées pour construire l'arbre et élire la racine. Dans tous les cas, chaque site a été visité par une phase de propagation et une phase de reflux. Néanmoins, à la fin du reflux, seule la racine sait que la construction de l'arbre est terminée. En effet :

- dans la version par vagues inondantes, les autres sites attendent une éventuelle meilleure proposition de vague,
- dans la version par régions, les autres sites ne savent pas si une nouvelle élection sur un réseau virtuel est ou non en train de se dérouler,
- dans la version par régions et retournement, les autres sites ne savent pas si tous les retournements ont été opérés.

Afin d'informer l'ensemble des sites de la fin de l'algorithme, on ajoute à la vague de construction proprement dite, une vague de terminaison. Celle-ci est initiée par la racine et parcourt l'arbre construit. La méthode *Arbre* comprend donc deux phases exécutées en séquence : une phase *Construire* pour construire l'arbre et une phase *Terminer* pour informés tous les sites de la terminaison de l'algorithme. On note le prédicat intermédiaire à ces deux phases $ConstructionFinie_n$. C'est le même que celui de fin d'algorithme moins le terme $E_{Sites_n}(racine_n)$ exprimant la connaissance de tous de la racine :

$$ConstructionFinie_n \quad \hat{=} \quad (\forall i \in Sites_n,\ K_i(p\grave{e}re_n(i)) \ \wedge\ K_i(fils_n(i))) \wedge \\ (\bigcup_{i \in Sites_n} fils_n(i) = \emptyset)$$

Avant l'exécution de la méthode *Construire*, le niveau de connaissance est le même que celui précédent la méthode *Arbre* : tous les arbres d'époques inférieurs à l'époque courante sont connus.

Après la construction de l'arbre, le prédicat de connaissance de groupe indique que la construction d'époque courante est finie. Finalement, après la phase de terminaison, tous les d'époques inférieurs ou égales à l'époque courante sont connus. Le programme de niveau groupe associée à la méthode *Arbre* s'écrit donc :

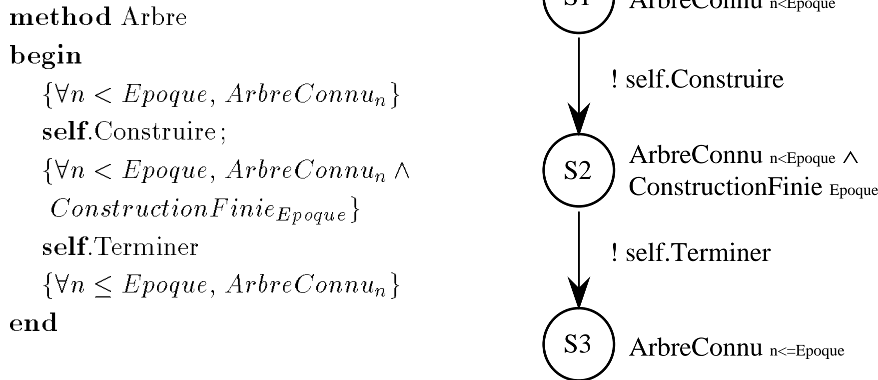


FIG. 8.11 – *Méthode Arbre du programme de niveau groupe*

8.3.6 Deuxième raffinement du programme de niveau groupe

Le second raffinement du comportement de niveau groupe consiste à introduire les comportements des méthodes *Construire* et *Terminer*. Ce sont des vagues de parcours construite sur le modèle du gabarit récursion distribuée (Cf paragraphe 5.4). Nous commençons par exposer la méthode *Terminer* qui est la plus simple des deux.

8.3.6.1 Méthode *Terminer*

La méthode *Terminer* parcourt l'arbre construit par la méthode *Construire*. La racine est le seul initiateur de ce parcours. Chaque site propage l'information vers ses fils.

La figure 8.12 présente le programme de niveau groupe associé à ce comportement. Les méthodes *Terminer*, *TerminerRec* et *TerminerPoursuivre* jouent les rôles des méthodes *Main* et *Rec* du programme décrit au paragraphe 5.4. La méthode *TerminerSitesAjoutés* sélectionne, à chaque étape de la récursion, un sous-ensemble des fils des sites visités. Finalement, la méthode *TerminerPré-traitement* informe les sites ajoutés de la terminaison de l'algorithme (en modifiant, par exemple, une variable de leur état local), tandis que la méthode *TerminerPost-traitement* ne fait rien.

8.3.6.2 Méthode *Construire*

Les trois versions de l'algorithme diffèrent dans leur façon de construire l'arbre. Elles utilisent toutes les trois un schéma de parcours de type récursion distribuée. Nous le notons *ConstruireRec*. Il est identique à la méthode *TerminerRec*.

```
method Terminer
 Initiateurs : set of object = {racine_n}
begin
 self.TerminerRec( Initiateurs )
end

method TerminerRec( in Visités : set of object )
 Ajout : set of object
begin
 if self.TerminerPoursuivre( Visités ) then
  Ajout := self.TerminerSitesAjoutés( Visités ) ;
  self.TerminerPré-traitement ;
  self.TerminerRec( Visités ∪ Ajout ) ;
  self.TerminerPost-traitement
 endif
end

method TerminerPoursuivre( in Visités : set of object ) : boolean
begin
 return Visités ⊂ Sites
end

method TerminerSitesAjoutés( in Visités : set of object ) : set of object
 FilsVisités, Ajout : set of object
begin
 FilsVisités = {s|∃i ∈ Visité, fils_Epoque(i) = s} ;
 Ajout ⊆ {s|s ∉ FilsVisités ∧ ∃i ∈ Visités, d(i, s) = 1} ;
 return Ajout
end

method TerminerPré-traitement
begin
 // Informer les sites ajoutés de la terminaison
end

method TerminerPost-traitement
begin
end
```

FIG. 8.12 – *Programme de niveau groupe de la phase Terminer*

Version par vagues inondantes

Dans cette version, la vague gagnante parcourt la totalité de l'arbre. De ce fait, certains sites sont visités éventuellement plusieurs fois. La phase de construction se compose donc de plusieurs sous-phase exécutées en parallèle. Chaque sous-phase correspond a une vague de parcours. Pour une construction donnée, il y a donc autant de sous-phases que d'initiateurs différents.

```
method Construire    // version à vagues inondantes
begin
  co_begin
    c1 : self.ConstruireRec( {Initiateur1} )
    c2: self.ConstruireRec( {Initiateur2} )
    ...
    cn: self.ConstruireRec( {Initiateurn} )
  co_end
end
```

Dès que deux vagues se rencontrent, c'est à dire dès qu'une sous-phase de construction atteint un site déjà visité, elles testent la valeur de leurs identificateurs respectif. La vague perdante stoppe sa récursion et reflue. La vague gagnante poursuit sa récursion. Dans ce cas, la phase de *Pré-traitement* enregistre, sur chaque nouveau site visité, le père proposé et l'identité de l'initiateur. La phase de *Post-traitement* enregistre les fils.

Version par régions

La version par régions nécessite d'itérer un parcours récursif jusqu'à ce qu'il n'y ait plus qu'un seul initiateur dans le réseau. De plus, elle nécessite une adaptation de la méthode *ConstruireRec*: la topologie du réseau sur lequel s'effectue le parcours change à chaque itération (Cf. paragraphe 8.2.2.2).

```
method Construire    // version à régions
  Niveau : Integer = 0
begin
  while |InitiateursNiveau| > 1
    self.ConstruireRec( {InitiateursNiveau} ) ;
    Niveau := Niveau+1
  enddo
  self.ConstruireRec( {InitiateursNiveau} )
end
```

Les phases de *Pré-traitement* et de *Post-traitement* sont identiques à celles de la version précédente. La première enregistre le père et l'initiateur, tandis que la seconde enregistre les fils.

Version par régions et retournement de branches

La version par régions est celle qui génère le plus petit nombre d'interactions entre les objets. Néanmoins, c'est la plus délicate à gérer du point de vue de la mise à jour des pères et des fils. Comme dans les deux versions précédentes la phase de *Pré-traitement* enregistre le père et l'initiateur. La phase de *Post-traitement* enregistre les fils, et en cas de retournement, change le

père. De plus, si plus d'un retournement atteint un initiateur perdant, elle fait en sorte de ne garder qu'un seul lien.

```
method Construire    // version à régions et retournements
begin
  self.ConstruireRec( {Initiateurs} )
end
```

8.3.7 Résumé

Rappelons que, dans cet algorithme, chaque objet désirant construire un arbre couvrant, se déclare initiateur et appelle la méthode *Arbre*. Cette méthode comprend deux phases : *Construire* et *Terminer*. La première construit l'arbre proprement dit, et la seconde informe tous les objets de la terminaison de la construction. Ce sont toutes les deux des instances du gabarit récursion distribué. A la fin de la méthode *Arbre*, le groupe passe dans une époque suivante et une nouvelle construction peut être entreprise.

Nous avons donc proposé trois niveaux de raffinement pour les programmes de groupe. Les deux premiers définissent les schémas de phasage, tandis que le troisième introduit les parcours récursifs. Les automates états/transitions qui dénotent ces programmes peuvent être exprimés à l'aide de comportements du langage CAOLAC (Cf. figure 8.13). Le comportement *ACGroupe* est associé au premier programme. Le comportement *ACDeuxPhases* est un sous-comportement du précédent et est associé au second programme.

Les comportements CAOLAC fournissent un canevas d'implantation qui est réutilisé au niveau objet. Cette réutilisation est possible car il y a une similitude forte entre la représentation en terme d'états et de transitions d'une phase au niveau groupe et son implantation sur chacun des objets. Ainsi, chaque phase définie au niveau groupe correspond à une phase au niveau objet, et chaque phase du niveau objet fait partie d'une phase de groupe. La situation n'est pas la même pour la récursion distribuée. Comme nous le présentons au paragraphe suivant, il n'y a pas la même similitude entre l'automate de niveau groupe et celui de niveau objet. De ce fait, la récursion s'exprime dans le langage CAOLAC par l'intermédiaire d'un schéma d'implantation type qui n'a pas de similitude "graphique" avec le programme de niveau groupe et son automate associé.

8.4 Spécification de niveau objet

Dans ce paragraphe, nous fournissons les comportements des objets qui implantent les comportements de groupe décrits précédemment. Nous utilisons pour cela le protocole méta-objet de synchronisation CAOLAC présenté au chapitre 6 et le langage GUIDE [BBD+91]. Nous développons la version de l'algorithme à base de vagues inondantes.

8.4.1 Variables

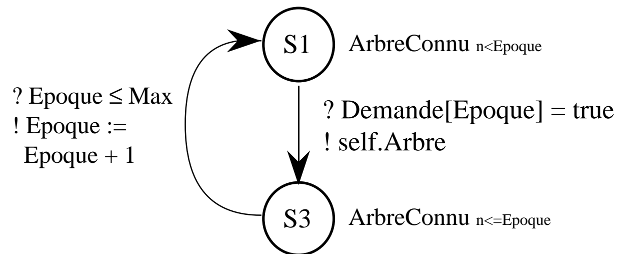
Chaque objet du groupe exécutant l'algorithme gère les informations suivantes : il tient à jour son numéro d'époque courant (variable *époque*), elle possède une référence sur l'élu qui est aussi la racine de l'arbre (variable *élu*), sur son père (variable *père*) et sur un groupe de fils (variable *fils*).

```
behaviour ACGroupe {

  variables : Epoque : Integer = 0;
  invocations : Arbre;
  methods : Demande : Boolean;

  initial state : S1;
  state S1 {
    TArbre {
      require( self.Demande = true );
      self.Arbre;
      become( S3 );
  } }

  state S3 {
    TNouvelleEpoque {
      Epoque := Epoque+1;
      become( S1 );
  } }
}
```




```
behaviour ACGroupeDeuxPhases
  subbehav of ACGroupe {

  invocations :
    Construire;
    Terminer;

  state S1 {
    TConstruction redefines TArbre {
      self.Construire;
      become( S2 );
  } }

  state S2 {
    TTerminaison {
      self.Terminer;
      become( S3 );
  } }
}
```

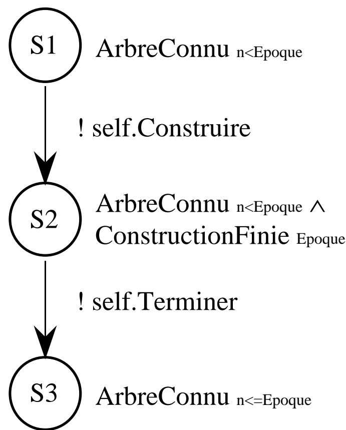


Fig. 8.13 – *Traduction en terme de comportements CAOLAC des programmes de groupe*

époque : **Integer** = 0;
élu, père : **ref** ac;
fils : **ref** group **of ref** ac;

8.4.2 Invocations

Chaque objet accepte deux type d'invocations : *Construire* et *Terminer*. La phase de construction propage trois informations : le numéro d'époque auquel elle appartient (c'est à dire l'époque proposée), l'identité de son initiateur (c'est à dire l'élu proposé) et l'identité du site qui l'a transmise en dernier (c'est à dire le père proposé). Elle retourne un code indiquant si elle est gagnante (entier *vaguegagnante*) ou perdante (entier *vagueperdante*). De plus, une vague peut arriver en retard par rapport à la construction courante (Cf. paragraphe 8.4.4.1) ou revenir sur un site déjà visité (Cf. paragraphe 8.4.4.2). On prévoit donc deux codes pour ces situations : *vagueretard* et *vaguebouclage*. La méthode *Terminer* ne transmet pas de paramètre. On obtient la déclaration suivante :

variables :
const vagueperdante : **Integer** = 0;
const vaguegagnante : **Integer** = 1;
const vagueretard : **Integer** = 2;
const vaguebouclage : **Integer** = 3;

invocations :
Construire(**in** époqueproposée : **Integer** ; **in** éluproposé, pèreproposé : **ref ac**) : **Integer**;
Terminer;

8.4.3 Comportement CAOLAC

Rappelons que le programme d'élection avec calcul d'arbre couvrant comprend deux phases principales : une de calcul et une de changement d'époque. La phase de calcul est raffinée en une phase de construction et une phase de terminaison. Nous avons vu que la structure en termes d'états et de transitions de ces phases peut être exprimées à l'aide du comportement CAOLAC *ACGroupe* et de son sous-comportement *ACGroupeDeuxPhases*. Sur chaque objet du groupe, l'algorithme est implanté par le comportement *ArbreCouvrant* (Cf. figure 8.14). C'est un sous-comportement de *ACGroupeDeuxPhases*. Il fournit la synchronisation des phases de construction (méthode *Construire*) et des phases de terminaison (méthode *Terminer*). Il comprend les déclarations de variables et d'invocations décrites précédemment.

La phase de construction commence avec l'invocation de la méthode *Construire*. Chaque objet peut recevoir une proposition de construction tant qu'il n'a pas reçu une demande de terminaison. La sémantique de l'état *S1* est donc de type *server*. Il reste actif tant que la phase de terminaison n'a pas débutée sur l'objet. Cette condition s'exprime avec le compteur *StartedTransition* sur la transition *TTerminaison* héritée du comportement *ACDeuxPhases*. La phase de terminaison est déclenchée par l'initiateur gagnant de la construction (transition de droite entre les états *S2* et *S3* dans l'automate de la figure 8.14).

```
behaviour ArbreCouvrant
 subbehav of ACGroupeDeuxPhases {

 // les déclarations de variables et d'invocations des paragraphes précédents

 state S1 server while StartedTransition(ACDeuxphases::TTerminaison) = 0 {
  TConstruire redefines TConstruction {
   invocation( Construire( époqueproposée, éluproposé, pèreproposé ) );
   retour : integer;
   retour := BASE.Construire( époqueproposée, éluproposé, pèreproposé );
   return retour;
   become( S2 );
 } }

 state S2 {
  TTerminerInitiateur redefines TTerminaison {
   require( retour=vaguegagnante and élu = self);
   BASE.Terminer;
   become( S3 );
  }
  TTerminerAutre redefines TTerminaison {
   invocation( Terminer );
   require( retour=vaguegagnante );
   BASE.Terminer;
   become( S3 );
 } }
}
```

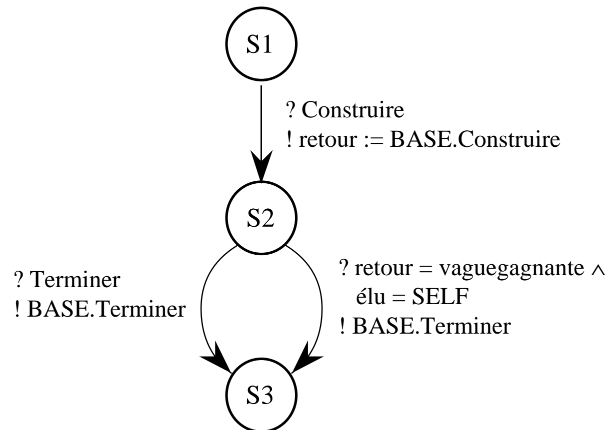


FIG. 8.14 – *Comportement CAOLAC implantant l'algorithme sur un objet*

8.4.4 Synchronisations additionnelles pour la phase de construction

Deux synchronisations supplémentaires sont à mettre en place lorsqu'un objet reçoit une proposition de construction. Tout d'abord, on doit contrôler que la proposition appartient bien à l'époque courante de l'objet. Puis, on doit déterminer si la proposition provient ou non, d'un meilleur initiateur que celui que l'on est en train de traiter. Ces synchronisations sont réalisées par une tour de deux méta-niveaux *ContrôleEpoque* et *ContrôleInitiateur* (Cf. figure 8.15).

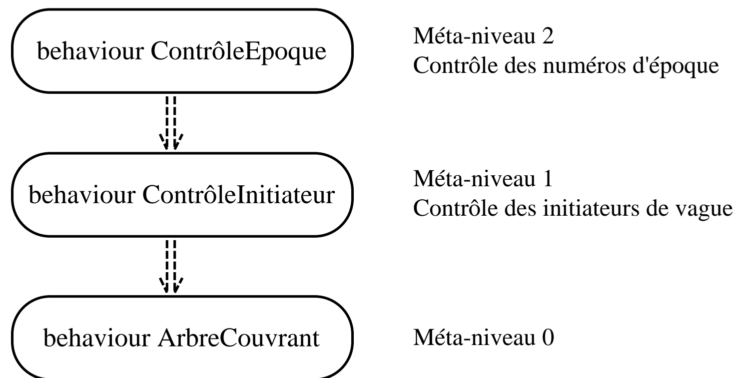


FIG. 8.15 – *Méta-niveaux pour la synchronisation de la phase de construction*

8.4.4.1 Méta-niveau 2 : contrôle des numéros d'époque

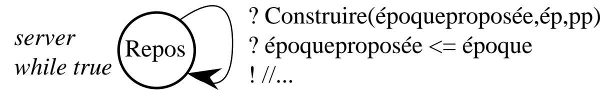
En ce qui concerne le numéro d'époque véhiculé par la vague, trois situations peuvent être rencontrées : il est, soit supérieur, soit égal, soit inférieur, au numéro d'époque courant de l'objet.

- le numéro d'époque proposé est supérieur au numéro courant : cela correspond à une demande de construction qui est en avance. L'objet n'a pas encore fini la construction courante. Il faut donc qu'il retarde la proposition jusqu'à ce qu'il ait atteint l'époque proposée.

- le numéro d'époque proposé est égal au numéro courant : c'est la situation normale. L'objet accepte la proposition. Il transmet la proposition au niveau inférieur dans la tour méta.

- le numéro d'époque proposé est inférieur au numéro courant : cela correspond à une demande de construction qui est en retard. Il faut indiquer à l'initiateur de cette ancienne demande qu'une nouvelle époque a commencé et qu'il peut abandonner sa construction puisqu'une nouvelle est en cours.

La figure 8.16 présente le comportement CAOLAC qui réalise cette synchronisation. Il comprend un seul état, *Repos*, actif en permanence (sa sémantique est *server while true*), et une seule transition (*TConstruction*). Elle est déclenchée lorsque, une invocation de type *Construire* est présente et que le numéro d'époque propagée par la vague est inférieur ou égal à l'époque courante (*époqueproposée* $<=$ *époque*). Si la vague est en retard, alors le code *vagueretard* est retourné, sinon l'invocation est transmise, pour traitement, au niveau inférieur dans la tour méta (*BASE.Construire*).




```
behaviour ContrôleEpoque
  meta of ContrôleProposition {
 initial state : Repos;
 state Repos server while true {
   TConstruction {
    invocation( Construire(époqueproposée,ép,pp) );
    require( époqueproposée <= époque );
    if époqueproposée < époque
    then return vagueretard;
    else return BASE.Construire(epoqueproposée,ép,pp);
    end;
   }
 }
}
```

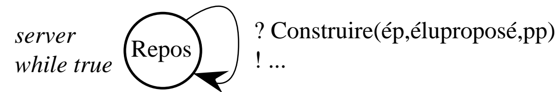
FIG. 8.16 – *Contrôle des numéros d'époque*

8.4.4.2 Méta-niveau 1 : contrôle des initiateurs de vagues

Chaque vague véhicule l'identité de son initiateur. Trois situations peuvent être rencontrées lorsque cette proposition arrive sur un site : il est, soit supérieur, soit égal, soit inférieur, à l'initiateur courant de l'objet.

- l'initiateur proposé est supérieur à l'initiateur courant : cela correspond à une vague gagnante par rapport à la vague courante. On prend en compte cette meilleure proposition.
- l'initiateur proposé est égal à l'initiateur courant : cela correspond à une vague qui revient sur le même site suite à une boucle dans le réseau. Cette proposition ne doit pas être prise en compte.
- l'initiateur proposé est inférieur à l'initiateur courant : cela correspond à une vague perdante. Il faut stopper la récursion de cette vague en retournant la proposition.

La figure 8.17 présente le comportement CAOLAC qui réalise cette synchronisation. Comme pour le comportement *ContrôleEpoque*, il comprend un seul état, *Repos*, actif en permanence (sa sémantique est *server while true*), et une seule transition (*TConstruction*). Elle est déclenchée lorsqu'une invocation de type *Construire* est présente. Si l'élu proposé est inférieur à l'élu courant (*self.Inf(éluproposée,élu) = true*), alors le code *vagueperdante* est retourné. Si, au contraire, il est supérieur (*self.TesttosupAndSet(éluproposé,élu) = true*), alors la proposition est prise en compte. La méthode *TesttosupAndSet* implante une opération atomique qui consiste, à tester si la première référence d'objet passée en paramètre est supérieure à la seconde, et si c'est le cas, à affecter la première à la seconde. De ce fait, si l'initiateur proposé est meilleur que l'initiateur courant, on l'enregistre et on transmet la proposition au niveau inférieur (*BASE.Construire*). Si-




```
behaviour ContrôleInitiateur
  meta of ac {
 methods:
  Inf( in arg1, arg2 : ref ac ) : Boolean;
  TesttosupAndSet( in arg1, arg2 : ref ac ) : Boolean;
 initial state : Repos;
 state Repos server while true {
  TConstruction {
   invocation( Construire(ép,éluproposé,pp) );
   if self.Inf( éluproposée , élu ) = true
   then   return vagueperdante;
   else   if self.TesttosupAndSet( éluproposé , élu ) = true
          then return BASE.Construire(ép,éluproposé,pp);
          else return vaguebouclage;
          end;
   end;
   become( Repos );
  }
 }
}
```

Fig. 8.17 – *Contrôle des initiateurs de vague*

non, si les identificateurs sont égaux, alors on est en présence d'un bouclage et on retourne le code *vaguebouclage*.

En cas de meilleure proposition, les opérations de test et de mise à jour de la variable *élu* doivent être réalisées de façon atomique. En effet, si deux propositions d'élus supérieurs à l'élu courant sont traitées de façon concurrente, il ne faut pas que l'entrelacement des tests et des mises à jour amène un état incohérent. De ce fait, on utilise une opération atomique de type *TestAndSet*.

8.4.5 Résumé

La spécification de niveau objet fournit la synchronisation des objets qui participent à la réalisation de l'algorithme. Le comportement de chaque objet reflète le comportement défini au niveau groupe. De ce fait, les schémas de phasage *ACGroupe* et *ACGroupeDeuxPhases* qui définissent, respectivement, l'exécution en boucle de l'algorithme et les deux phases de construction et de terminaison qui le composent, sont héritées par le comportement *ArbreCouvrant* de niveau objet. On établit ainsi un lien sémantique entre un schéma global de conception et son implantation locale. Ce lien est traduit dans le langage CAOLAC par la relation sous-comportementale qui permet à un automate d'hériter de la structure, c'est à dire les états, les transitions et le code attaché aux transitions, d'un comportement parent.

Dans un second temps, le comportement *ArbreCouvrant* est associé à deux méta-comportements *ContrôleEpoque* et *ContrôleInitiateur*. La relation méta-comportementale permet de modulariser la conception de la politique de synchronisation. Ainsi, chaque comportement dans la tour méta est indépendant des autres et implante une partie de la politique. Cette démarche facilite la conception de niveau objet en scindant un problème de synchronisation complexe en plusieurs problèmes de taille réduite. Par exemple, *ContrôleEpoque* contrôle uniquement les numéros d'époque des propositions transmises. *ContrôleInitiateur* s'intéresse quant à lui, à l'identité des initiateurs de vague. Finalement, *ArbreCouvrant* détaille l'enchaînement des phases de construction et de terminaison.

La figure 8.18 résume les liens entre les différents modèles CAOLAC utilisés pour la version à vagues inondantes de l'algorithme d'élection avec calcul d'arbre couvrant. Elle reflète une des caractéristiques du processus de développement que nous proposons. La relation sous-comportementale est utilisée, en général, pour transmettre du niveau groupe au niveau objet un schéma de phasage. Elle permet donc de faire le lien entre ces deux niveaux méthodologiques. La relation méta-comportementale est, quant à elle, utilisée au niveau objet pour scinder une politique de synchronisation en composants autonomes. Chaque composant au sein d'une tour méta résout, indépendamment des autres, un problème qui lui est propre.

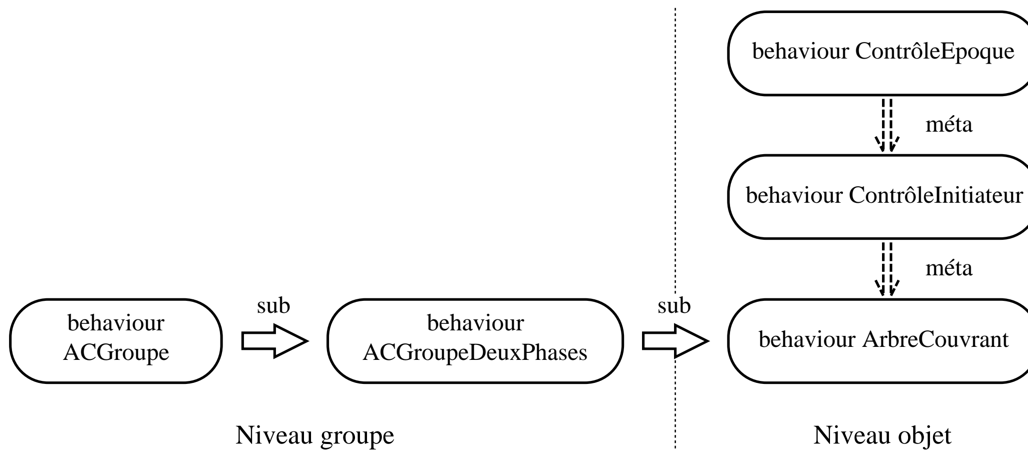


FIG. 8.18 – *Diagramme de réutilisation des comportements CAOLAC*

8.5 Spécification de niveau méthode

Au niveau de base, la méthode *Construire* enregistre le père proposé, propage la vague sur les voisins (moins le père) et enregistre les fils. Ces opérations sont réalisées, respectivement, par *ConstruirePré-traitement*, *ConstruirePropager* et *ConstruirePost-traitement* (Cf. figure 8.19). La méthode *Terminer* enregistre la terminaison et propage l'information à l'ensemble des fils du site. Ces opérations sont réalisées par *TerminerPré-traitement* et *TerminerPropager*.

En ce qui concerne la méthode *Construire*, notons cependant, que deux invocations peuvent être délivrées concurremment. En effet, il se peut que, pendant que le comportement *ContrôleInitiateur* reçoit une meilleure proposition de vague, l'enregistre avec l'opération *TesttosupAndSet* et la délivre au niveau inférieur (*BASE.Construire*), une seconde proposition de meilleur initiateur

arrive. Dans ce cas, cette dernière est également enregistrée et délivrée. L'objet de classe *ac* peut donc avoir à traiter simultanément plusieurs constructions. Or, cette situation est indésirable. En effet, l'entrelacement de ces différentes constructions peut amener des mises à jour incohérentes du père et des fils. De ce fait, l'exécution de la méthode *Construire* est protégée par une exclusion mutuelle (clause de contrôle *EXCLUSIVE* dans la classe *ac*). De plus, un test (*éluproposé* = *élu*) est effectué au début de cette méthode pour vérifier si une meilleure proposition n'a pas été enregistrée depuis le début de la proposition courante. Si c'est le cas, celle-ci est perdante et le code *vagueperdante* est retourné.

La méthode *ConstruirePropager* appelle en parallèle les méthodes *Construire* de tous les voisins (moins le père) du site courant. Elle récupère donc un tableau de code retour. Si au moins un voisin indique que la construction courante est en retard (code retour *vagueretard*), alors cela signifie qu'une nouvelle construction à débuter. Le site courant renvoie donc, lui aussi, le code *vagueretard*. De même, si au moins un voisin appartient à une meilleure construction, il faut indiquer que la construction courante est perdante. Le site retourne donc *vagueperdante*. Dans tous les autres cas, c'est dire si tous les sites ont répondu *vaguegagnante* ou *vaguebouclage*, la construction courante est gagnante. En effet, le code *vaguebouclage* n'influence pas le statut de la construction courante. Il indique juste que la vague courante a revisité un site qu'elle avait déjà visité. Néanmoins, seuls les voisins ayant répondu *vaguegagnante* sont pris en compte comme fils.

8.6 Conclusion

```
class ac with behaviour ArbreCouvrant is

  method Construire( in époqueproposée : Integer ; in éluproposé, pèreproposé : ref ac ) : Integer;
   retour : Integer;
  begin
   if éluproposé = élu then
     ConstruirePré-traitement( éluproposé, pèreproposé );
     retour := ConstruirePropager;
     ConstruirePost-traitement( retour );
     return retour;
   else return vagueperdante;
   end;
  end Construire;

  procedure ConstruirePré-traitement( in éluproposé, pèreproposé : ref ac );
  begin
   // Enregistrement du nouveau père
   père := pèreproposé;
  end ConstruirePré-traitement;

  procedure ConstruirePropager : Integer;
  begin
   // Diffusion de la proposition à tous les voisins moins le père
  end ConstruirePropager;

  procedure ConstruirePost-traitement( in coderetour : Integer );
  begin
   // Enregistrement comme fils de tous les sites qui ont répondu que cette vague est gagnante
  end ConstruirePost-traitement;

  method Terminer;
  begin
   TerminerPré-traitement;
   TerminerPropager;
  end Terminer;

  procedure TerminerPré-traitement;
  begin
   // Enregistrement de la terminaison
  end TerminerPré-traitement;

  procedure TerminerPropager;
  begin
   // Propagation à tous les fils
  end TerminerPropager;

  control
   EXCLUSIVE( Construire );

end ac.
```

FIG. 8.19 – *Classe implantant l'algorithme sur chaque site*

Chapitre 9

Protocole transactionnel

Dans ce chapitre, nous présentons la conception d'un protocole transactionnel de validation à deux phases. Nous appliquons le processus de développement en trois niveaux (groupe, objet, méthode) défini dans cette thèse. En particulier, les programmes de niveau groupe utilisent des instances des gabarits phase et conditionnelle distribuée (Cf. paragraphe 9.2), le niveau objet implante la synchronisation de ces programmes à l'aide du langage CAOLAC (Cf. paragraphe 9.3) et le niveau méthode fournit, en langage GUIDE, le code séquentiel (Cf. paragraphe 9.5). Finalement, le paragraphe 9.5 conclut cette étude de cas.

9.1 Présentation du protocole

Le protocole de validation à deux phases [LS76] (en anglais *Two Phases Commit Protocol*) a pour but de garantir l'exécution atomique d'une transaction répartie. Ce protocole, que nous présentons au paragraphe 9.1.2, fait partie des protocoles dits de validation atomique.

9.1.1 Protocoles de validation atomique

Les protocoles de validation atomique (en anglais ACP pour *Atomic Commitment Protocol*) mettent en jeu deux types d'entités : le coordinateur de la transaction et deux ou plusieurs participants. Elle garantit que, soit le coordinateur et tous les participants valident la transaction, soit ils l'abandonnent et reviennent à l'état antérieur à la transaction. Plus précisement, Bernstein dans [BHG87] exprime cela de la façon suivante : chaque entité (coordinateur ou participant) peut émettre seulement deux votes *Oui* ou *Non*, et chaque entité peut prendre seulement deux décisions *Valide* ou *Abandonne*. Un protocole de validation atomique vérifie alors les cinq règles suivantes :

1. toutes les entités qui prennent une décision prennent la même,
2. une entité ne peut pas revenir sur sa décision après l'avoir prise,
3. la décision *Valide* peut seulement être prise si toutes les entités ont voté *Oui*,
4. s'il n'y a pas de pannes et si toutes les entités votent *Oui*, alors la décision *Valide* sera prise,
5. en ne considérant que les pannes tolérées par l'algorithme, quel que soit l'avancement de l'exécution, si toutes ces pannes sont compensées et si aucune autre panne ne survient pendant

un intervalle de temps suffisamment long, alors toutes les entités prendront inévitablement une décision.

9.1.2 Protocole de validation à deux phases

Le protocole de validation à deux phases est la forme la plus simple de protocole de validation atomique. Il ne tolère ni les pertes de messages, ni les pannes de sites. Il est donc dit bloquant. Une présentation d'un protocole de validation non bloquant (c'est à dire tolérant les pertes de message et les pannes de sites) comportant trois phases et dû à [Ske82], est fournie dans [BHG87]. Le protocole de validation à deux phases comprend les quatre étapes suivantes (Cf. figure 9.1) :

1. le coordinateur envoie une demande de vote à tous les participants.

2. un participant recevant une demande de vote détermine s'il peut ou non effectuer la transaction et répond au coordinateur *Oui* ou *Non*.

3. le coordinateur collecte les votes de tous les participants. Si tous ont voté *Oui*, alors il envoie un ordre de validation à tous les participants. Si un seul participant a voté *Non*, alors il envoie un ordre d'annulation.

4. chaque participant attend un ordre de validation ou d'annulation et décide d'entreprendre l'action correspondante. Lorsque celle-ci est terminée, il envoie un acquittement de fin au coordinateur.

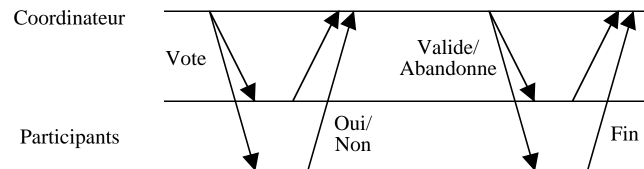


FIG. 9.1 – *Protocole de validation à deux phases avec deux participants*

9.2 Spécification de niveau groupe

Dans ce paragraphe, nous présentons la spécification de niveau groupe du protocole transactionnel de validation à deux phases. Nous définissons tout d'abord, les hypothèses de bon fonctionnement de cet algorithme, puis nous exprimons, en terme de connaissance, son but but global. Nous présentons alors le programme de niveau groupe qui permet d'atteindre cet objectif, et nous le traduisons, en vue de son implantation au niveau objet, par un modèle états/transitions exprimé dans la syntaxe du langage CAOLAC.

9.2.1 Hypothèses

Deux types d'entités prennent part à ce protocole : un coordinateur $Coord$ et un ensemble $Part = \{Part_1, \ldots, Part_n\}$ de deux ou plusieurs participants. On représente ce groupe par un ensemble $S = \{Coord\} \cup Part$ d'objets distribués.

L'hypothèse minimale de bon fonctionnement pour cet algorithme est que le coordinateur connaisse tous les participants et tous les participants connaissent le coordinateur. Cela s'exprime par le prédicat épistémique suivant : $\forall i \in Part,\ K_{Coord}(i) \wedge K_i(Coord)$. Ce prédicat constitue un invariant du comportement global que nous notons dans la section **environment** du programme de niveau groupe :

group vars
  const Coord : **object** ;
  const Part : **set of object** ;
  const S : **set of object** = $\{Coord\} \cup Part$
environment
  $\forall i \in Part,\ K_{Coord}(i) \wedge K_i(Coord)$

9.2.2 But global

Le but de l'algorithme est que tous les participants exécutent, de façon atomique, la transaction. Ils doivent donc, soit tous l'exécuter, soit tous l'abandonner. Nous notons *Statut* la variable de groupe qui représente le statut de la transaction. Chaque membre i du groupe possède une copie, notée *i.Statut*, de cette variable. *Statut* peut prendre deux valeurs : *validé* ou *abandonné*. Le but de l'algorithme est de faire en sorte qu'à la fin de la transaction, tous les objets connaissent ce statut, et que celui-ci soit le même pour tous. En notant *StatutConnu* le prédicat épistémique représentant ce but global, on obtient :

$$StatutConnu \quad \widehat{=} \quad E_S(statut) \wedge \forall i, j \in S, i.Statut = j.Statut$$

9.2.3 Programme de niveau groupe

La figure 9.2 présente le programme de niveau groupe correspondant au protocole de validation à deux phases. Nous utilisons une variable booléenne *Res* pour représenter le vote des participants. Elle vaut *true* si tous les participants votent *Oui*, et *false* sinon. Ce protocole comprend trois phases : celle de vote (notée *Voter*), celle de validation (notée *Valider*) et celle d'abandon (notée *Abandonner*). A la fin de la phase de vote et avant la phase de validation ou d'abandon, le coordinateur est le seul à connaître le statut de la transaction. Le prédicat associé à l'état *S2* est donc $K_{Coord}(statut)$. Finalement, à la fin du protocole (états *S3* ou *S4*), le prédicat *StatutConnu* est atteint.

9.2.4 Modèle états/transitions

La figure 9.3 présente le schéma états/transitions et le comportement CAOLAC *Valide2Phases* associés à ce programme de groupe. Les trois actions sont définies en tant qu'invocations. La variable *Res* reçoit le résultat du vote des participants. L'invocation *Voter* retourne un booléen qui vaut *true* si tous les participants peuvent effectuer la transaction et *false* sinon. Les invocations *Valider* et *Abandonner* ne retournent rien. L'état *S1* est l'état initial et représente la situation avant le vote. Dans l'état *S2*, le vote a été effectué mais la validation ou l'abandon ne l'a pas encore été. Finalement, les états *S3* et *S4* sont les états finaux, suite respectivement à la phase de validation et à celle d'abandon. Dans cet exemple simple, la sémantique des états est systématiquement de type

group vars
  const Coord : **object** ;
  const Part : **set of object** ;
  const S : **set of object** = $\{Coord\} \cup Part$;
  Res : **Boolean**
environment
  $\forall i \in Part,\ K_{Coord}(i) \wedge K_i(Coord)$

method Main
begin
  {}
  Res := **self**.Voter ;
  $\{K_{Coord}(statut)\}$
  if Res = **true**
  then self.Valider
  else self.Abandonner
  endif
  $\{StatutConnu\}$
end

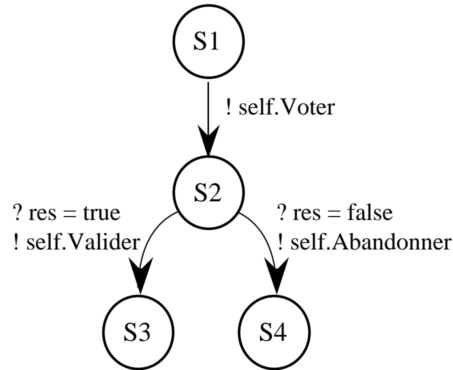


FIG. 9.2 – *Programme de niveau groupe de la validation à deux phases*

séquentielle. En effet, toutes les transitions sont exécutées en séquence et aucun état ne nécessite l'exécution de plus d'une transition pour être activé.

9.3 Spécification de niveau objet

Dans ce paragraphe, nous fournissons les comportements de niveau objet qui implantent les comportements de groupe définis précédemment. Les états définis à ce niveau sont des versions locales des états du niveau groupe. Le protocole de validation à deux phases définit deux rôles différents : celui du coordinateur et celui d'un participant. Ils prennent part tous deux au même comportement de groupe, et donc, à ce titre, dérivent le comportement *Valide2Phases*.

La figure 9.4 présente les deux comportements CAOLAC *Coord* et *Part*. Ils héritent de tous les éléments (variables, invocations, états et transitions) définis dans le comportement *Valide2Phases*. Dans cet exemple, le comportement de l'entité coordinatrice est identique au comportement de niveau groupe. En effet, dans cet exemple, le contrôle du groupe d'objets est réalisé en totalité par l'entité coordinatrice. Le comportement *Coord* n'ajoute donc aucun élément au comportement *Valide2Phases*. Le comportement *Part* conserve la structure imposée par la coordination inter-objets mais redéfinit les trois phases *Vote*, *Valide* et *Abandonne*. De façon générale, ces redéfinitions ont pour but de préciser au début de chaque phase que la transition associée ne peut être franchie que si l'invocation correspondante a été reçue.

```
behaviour Valide2Phases {

 variables:
  Res : Boolean;
 invocations:
  Voter : Boolean;
  Valider;
  Abandonner;

 initial state: S1;
 state S1 sequential {
  TVote {
   Res := BASE.Voter;
   become( S2 );
 }}

 state S2 sequential {
  TValide {
   require( Res = true );
   BASE.Valider;
   become( S3 );
  }
  TAbandonne {
   require( Res = false );
   BASE.Abandonner;
   become( S4 );
 }}

 state S3 sequential {}
 state S4 sequential {}
}
```

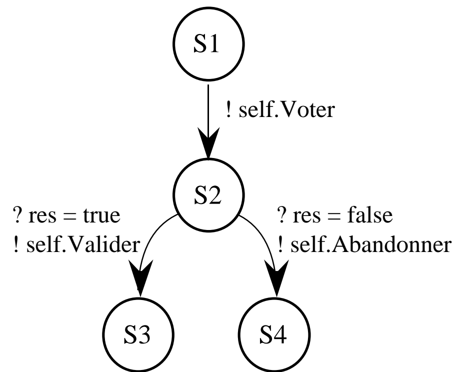


FIG. 9.3 – *Modèle états/transitions CAOLAC du programme de niveau groupe*

```
behaviour Coord subbehav of Valide2Phases {
}

behaviour Part subbehav of Valide2Phases {

 state S1 sequential {
  TVote {
   invocation( Vote );
   return BASE.Vote;
   become( S2 );
 }}

 state S2 sequential {
  TValide {
   invocation( Valide );
   BASE.Valide; return;
   become( S3 );
  }
  TAbandonne {
   invocation( Abandonne );
   BASE.Abandonne; return;
   become( S4 );
}}}
```

FIG. 9.4 – *Modèle états/transitions CAOLAC du comportement de niveau objet*

9.4 Spécification de niveau méthode

Le niveau de base fournit les traitements effectifs (c'est à dire en dehors de toute synchronisation) correspondant aux invocations des niveaux précédents.

La figure 9.5 présente les classes *Coordinateur* et *Participant* correspondant respectivement aux rôles coordinateur et participant. La classe *Coordinateur* utilise le comportement *Coord*. Elle possède les références des deux objets participants *part1* et *part2*. La phase de vote consiste à invoquer la méthode *Vote* de chacun de ces objets et à retourner la synthèse du résultat. Les phases de validation et d'annulation consistent en un appel des méthodes *Valide* et *Abandonne*. Rappelons que ces exécutions vont en permanence être coordonnées par les niveaux groupe et objet. Ainsi, l'architecture mise en place du côté coordinateur comprend un objet appartenant à la classe *Coordinateur* et un méta-objet appartenant au comportement *Coord* (lui-même héritant sa structure du comportement *Valide2Phases*). L'exécution de ce rôle commence alors au niveau méta par la transition *TVote* de l'état *S1*. De même, le rôle de chaque participant est assuré par un objet appartenant à la classe *Participant* et un méta-objet appartenant au comportement *Part* (lui-même héritant sa structure du comportement *Valide2Phases*).

```
class Coordinateur with behaviour Coord is
 part1, part2 : REF Participant;

 method Vote : Boolean;
 begin
  return part1.Vote and part2.Vote;
 end Vote;

 method Valide;
 begin
  part1.Valide;
  part2.Valide;
  return;
 end Valide;

 method Abandonne;
 begin
  part1.Abandonne;
  part2.Abandonne;
  return;
 end Abandonne;

end Coordinateur.

class Participant with behaviour Part is

 method Vote : Boolean;
 begin
  // Détermine si la transaction est réalisable
 end Vote;

 method Valide;
 begin
  // Valide la transaction
 end Valide;

 method Abandonne;
 begin
  // Abandonne la transaction
 end Abandonne;

end Participant.
```

FIG. 9.5 – *Niveau de base pour le protocole de validation à deux phases*

9.5 Conclusion

Dans ce chapitre, nous avons appliqué la démarche de conception présentée dans cette thèse, à l'exemple simple d'un protocole transactionnel de validation à deux phases. Ce protocole appartient à la classe des protocoles de validation atomique. Il met en jeu un ensemble de participants répartis sur les différents nœuds physiques d'un système. Il permet d'exécuter de manière atomique (c'est à dire en totalité ou pas du tout) une transaction sur cet ensemble de participants.

Le paragraphe 9.2 présente, en terme de connaissances, les buts globaux de cet algorithme. Leur réalisation met en jeu une structure de contrôle de type conditionnelle distribuée. En effet, le protocole comporte trois phases (*Voter*, *Valider* et *Abandonner*) et si tous les participants votent *Oui* au cours de la phase de vote, alors la phase de validation est entreprise, sinon c'est celle d'abandon qui l'est. Nous avons fourni le programme de niveau groupe correspondant à ce comportement ainsi que sa traduction en terme de modèle états/transitions CAOLAC. Le paragraphe 9.3 présente la synchronisation des objets qui réalisent ce comportement. Les modèles CAOLAC de ce niveau héritent leur structure du modèle de niveau groupe. On distingue ainsi la synchronisation de l'objet coordinateur et celle des objets participants. Finalement, le niveau de base fournit les traitements effectifs (c'est à dire en dehors de toute synchronisation) des phases de vote, de validation et d'annulation.

Conclusion

L'objectif de cette thèse était de fournir des outils pour aider les concepteurs à développer et à mettre en place des applications à base d'objets distribués. Nous nous étions proposés de définir une démarche systématique pour l'algorithmique répartie, de mettre en place un support méthodologique de développement, d'explorer la réutilisation de schémas types de comportement et de collaboration, de fournir un mécanisme de haut niveau permettant de décrire les échanges d'informations et, enfin, de gérer la concurrence des objets implantant l'application distribuée.

Finalement, notre contribution est organisée autour de deux axes complémentaires : la coordination inter-objets, qui s'intéresse à la spécification comportementale au sein de groupes d'objets distribués, et la coordination intra-objet, qui s'intéresse à la synchronisation des activités au sein d'un objet appartenant à un groupe. Pour cela, nous proposons :

- une démarche méthodologique pour la conception d'applications réparties,
- des structures de contrôle réparties,
- un langage de synchronisation pour des objets concurrents.

Ces éléments sont mis en pratique dans la quatrième partie de cette thèse à l'aide de l'étude d'un algorithme réparti de calcul d'arbres couvrants (Cf. chapitre 8) et d'un protocole transactionnel de validation à deux phases (Cf. chapitre 9).

Démarche méthodologique pour la conception d'applications réparties

Au cours de ce travail, nous avons ressenti le besoin d'intégrer les aspects liés à la répartition dans le processus méthodologique de développement d'une application distribuée. Nous avons donc proposé un processus de développement comprenant trois étapes méthodologiques : ce sont les niveaux de conception groupe, objet et méthode. Au niveau groupe, nous suggérons de décrire le comportement global de l'ensemble d'objets distribués implantant l'application. Le niveau objet se focalise sur le comportement d'un objet particulier au sein du groupe. Finalement, le niveau méthode fournit la structure des méthodes de chaque objet du groupe.

Le processus de développement est donc de type linéaire descendant. On commence par exprimer le ou les buts globaux poursuivis par le groupe d'objets. On s'intéresse alors aux buts locaux assignés à chaque objet et permettant d'atteindre ces buts globaux. Finalement, on conçoit, au niveau méthode, les comportements permettant de mettre en œuvre ces buts locaux. Cette démarche est donc fortement influencée par l'algorithmique répartie. Bien que ce travail se soit également inspiré du domaine des systèmes multi-agents (avec, par exemple, la notion d'opérateurs épistémiques de connaissance), nous n'en avons pas retenu la démarche de conception. En effet, celle-ci est fondée sur l'idée que chaque agent (implanté, par exemple, par un objet) est hautement autonome et poursuit des buts locaux, et que la réalisation d'une tâche collective émerge de la collaboration des agents. Nous pensons que cette démarche, qui peut s'avérer extrêmement fructueuse pour certains problèmes, présente deux inconvénients majeurs pour ce qui concerne les applications informatiques actuelles : c'est, d'une part, l'absence de but global clairement identifié, et d'autre part, l'inadéquation du concept d'émergence avec la construction rigoureuse de programmes. Ce sont les deux aspects que nous voulons mettre en avant dans notre processus de développement. En effet, dans l'optique d'un système formel pour des applications distribuées, il est, tout d'abord, nécessaire de pouvoir vérifier qu'un programme est conforme à sa spécification (c'est à dire qu'il réalise bien ce que l'on attend de lui). Par ailleurs, il est également souhaitable de disposer d'une

démarche incrémentale permettant d'aborder, petit à petit, les différents aspects d'une application. On incorpore ainsi, au fur et à mesure, de plus en plus de détails dans la spécification initiale et on raffine le but global. Dans l'approche que nous proposons, le raffinement peut aussi bien se faire au sein d'un même niveau (par exemple, en fournissant une spécification de niveau groupe plus détaillée qu'une autre spécification de niveau groupe), qu'entre niveaux différents (par exemple, en fournissant une spécification de niveau objet qui implante le comportement défini au niveau groupe).

Dans l'état actuel de nos travaux, les applications sont décrites et implantées dans ces trois niveaux, à l'aide, respectivement, de la notion de programme à bases de connaissance de niveau groupe, du langage CAOLAC et du langage objet du système distribué GUIDE [BBD+91]. De façon globale, les programmes de niveau groupe traitent les aspects liés à la distribution, le langage CAOLAC gère la concurrence et le langage GUIDE fournit le code des traitements. Comme nous l'évoquons ci-dessous dans les perspectives, il serait souhaitable, qu'à terme, ils soient tous trois fondus dans un seul et même langage. Actuellement, ces trois langages offrent les avantages suivants :

- la notion de programme à base de connaissances de niveau groupe (Cf. chapitre 4) peut être vu comme une extension des programmes à base de connaissance introduits par Fagin, Halpern, Moses et Vardi (Cf. chapitre 3). Ils décrivent les actions entreprises par le groupe d'objets réalisant l'application. De même que dans un système d'axiomatisation de type Hoare, les actions d'un programme séquentiel sont dénotées par des pré et des post-conditions, les actions de nos programmes de niveau groupe sont dénotées par des prédicats. Ceux-ci expriment les différents degrés de connaissance atteints par le groupe d'objets, avant et après chaque action globale. Le prédicat final représente le but global assigné au programme. L'expression de prédicats et la détection de propriétés étant un problème délicat en environnement réparti, nous avons ressenti le besoin d'une logique plus expressive que celle du premier ordre pour les prédicats des programmes de groupe. Dans la lignée de notre extension du formalisme de Fagin et al., notre choix s'est porté sur la logique épistémique (Cf. chapitre 2). C'est une logique modale qui permet d'exprimer à l'aide de différents opérateurs (K, D, S, E, C), la façon dont un fait est réparti parmi les éléments d'un groupe. Ainsi, elle permet d'abstraire les détails d'implantation, et plutôt que de raisonner sur des variables et des valeurs, elle permet de raisonner en terme de connaissances manipulées et échangées par un programme ce qui est plus adapté à une programmation où l'interaction joue un rôle majeur.

- pour le niveau objet, nous proposons le langage CAOLAC (Cf. chapitre 6). Ce langage décrit, à l'aide d'un formalisme à base d'états et de transitions, le but local poursuivi par un objet et la synchronisation interne nécessaire à la réalisation de ce but. Ce langage a été implanté au-dessus du système distribué orienté objet GUIDE. Sa sémantique (Cf. chapitre 7) est définie, en partie, à l'aide de la logique temporelle d'actions de Lamport [Lam91, Lam94].

- au niveau méthode, nous n'introduisons pas de formalisme particulier et nous décrivons, sous forme de pseudo-code, les structures algorithmiques de chaque méthode. Ces structures peuvent être dénotées par des pré- et des post-conditions. On obtient alors une représentation sous forme d'états et de transitions. Nous implantons ce niveau à l'aide du langage objet du système distribué GUIDE.

La représentation des comportements à l'aide d'états et de transitions constitue le fil conducteur de ce processus de développement. Les états représentent des niveaux de connaissance de groupe, des points d'avancement du processus de synchronisation intra-objet et des points d'avancement d'un algorithme séquentiel. Les transitions sont associées respectivement, à des actions globales, des actions de synchronisation et des actions locales. Ainsi, chaque état défini au niveau groupe doit correspondre à un point d'avancement du processus de synchronisation. Chaque niveau introduisant plus de détails que le précédent, la réciproque n'est pas vraie. De même, chaque point d'avancement du processus de synchronisation doit correspondre à un point d'avancement du niveau méthode. Ainsi, les prédicats définis en terme de connaissance globale sont traduits en terme de variables de synchronisation, puis en terme de variables locales.

Une certaine dualité apparaît, dans chacun des trois niveaux, entre un style de programmation impératif et un style sous-forme d'états et de transitions. Ainsi, la notion de programme à base de connaissance de niveau groupe est clairement impérative, bien que nous suggérions de l'associer à une forme états/transitions afin de faire apparaître les différents niveaux de connaissance. Au niveau objet, le langage CAOLAC est plutôt de la forme états/transitions. Finalement, au niveau méthode, le langage de base GUIDE est impératif. Bien que ces deux styles de programmation soient équivalents, et que leur présence laisse aux concepteurs un éventail de possibilité plus large, il conviendrait certainement, d'une part de choisir l'un plutôt que l'autre, et d'autre part d'unifier les trois niveaux dans un seul et même langage. En ce qui concerne le premier objectif, nous n'avons pas, pour l'instant, suffisamment d'arguments en faveur de l'un ou de l'autre pour pouvoir trancher. Néanmoins, de façon tout à fait subjective, le style impératif semble être plus adapté. Le second objectif (c'est à dire l'unification des trois niveaux) demande, quant à lui, une poursuite importante de notre effort de recherche et de développement.

Structures de contrôle réparties

Afin de guider les concepteurs, nous avons défini, au chapitre 5 et dans [SD96], quatre structures de contrôle qui, selon nous, apparaissent de façon fréquente dans les applications distribuées. Notre démarche est, ici, proche de celle de la communauté des gabarits de conception (*design patterns* en anglais) [GHJV95]. Ces gabarits, qui connaissent actuellement un certain succès chez les concepteurs et les programmateurs objet, sont des schémas qui apparaissent fréquement et de façon récurrente dans de nombreuses applications. L'ouvrage fondateur de ce domaine [GHJV95] en définit une vingtaine pour les applications orientées objet en univers centralisé et séquentiel. Par exemple, le gabarit *itérateur* définit le parcours d'une liste d'objets, le gabarit *observateur* définit des dépendances entre objet de façon telle qu'un changement d'état dans l'un soit répercuté automatiquement dans les autres. Le bénéfice attendu d'un telle démarche est que l'étude systématique et la définition de schéma de conception d'implantation types permettent d'en faciliter la réutilisation.

Dans cette optique, il nous est apparu qu'au moins quatre structures de contrôle apparaissent de façon fréquente dans un nombre important d'applications distribuées. Ce sont le schéma de phasage, la conditionnelle distribuée, l'itération distribuée et la récursion distribuée. Ces schémas peuvent être vus comme des extensions à un niveau distribué des structures algorithmiques de base telles que la séquence, les conditionnelles *case* et *if*, les boucles *while* et le schéma de parcours arborescent. Ces structures ont été retenues pour leur caractère générique. Ce sont des briques de base qui peuvent être composées et assemblées au sein d'un programme à base de connaissance de

niveau groupe. Nous rappelons que :

- la phase définit une action globale entreprise par tous les objets d'un groupe. C'est, par exemple, la phase d'engagement dans un protocole transactionnel de validation à deux phases [BHG87]. Elle traduit une transition entre deux niveaux de connaissance de groupe. Elle peut être composée d'au moins trois façons. Ainsi, nous avons défini la séquence, le constructeur de parallélisme et les phases gardées. Dans le premier cas, l'exécution successive de deux phases fournit une séquence équivalente, au niveau distribué, à l'opérateur point-virgule des langages de programmation. Le constructeur de parallélisme permet de composer plusieurs phases exécutées en parallèle au sein du groupe. Finalement, les phases gardées fournissent un mécanisme de commandes gardées [Dij75, Dij76] identique à l'instruction *select* du langage ADA [DoD80, Bar89].

- la conditionnelle distribuée définit un choix entre plusieurs actions globales selon une condition sur l'état global du groupe. Par exemple, un processus de validation à deux phases peut être vu comme une condition distribuée. En effet, le site coordinateur déclenche le vote, puis choisit d'entreprendre l'une des deux actions globales *Engagement* ou *Annulation* en fonction du résultat du vote de chaque participant, c'est à dire en fonction d'une condition sur l'état du groupe des participants. Dans un cas plus général, la condition peut tester un type quelconque et impliquer plus de deux actions globales.

- l'itération distribuée permet d'itérer un comportement tant qu'une condition sur l'état global du groupe est vérifiée. Par exemple, de nombreux algorithmes réseau de routage sont exécutés tant que le calcul n'a pas atteint un résultat stable. Nous proposons deux versions de cette structure : l'une synchrone, dans laquelle tous les objets du groupe exécutent de façon synchrone chaque itération, et l'autre asynchrone, dans laquelle chaque objet exécute ses itérations indépendamment des autres. Cette dernière variante caractérise, entre autres, le comportement des algorithmes répartis dits auto-stabilisants [Dij74][Tel94] qui possèdent la propriété de converger vers un état stable quel que soit l'état de départ.

- la récursion distribuée permet de parcourir un groupe d'objets distribués sur un réseau dont la topologie est quelconque et n'est pas connue de façon centralisée. Par exemple, cette structure sert à collecter l'état global d'un groupe d'objets. Dans ce cas, l'hypothèse de départ d'une telle structure est que le réseau est défini par les voisinages, c'est à dire que chaque objet connaît seulement un ensemble de voisins avec qui il est capable d'interagir. La structure de contrôle récursion distribuée initie alors une (récursion séquentielle) ou plusieurs vagues (récursion parallèle) qui parcourent l'ensemble des objets avant de refluer vers leur initiateur.

Langage de synchronisation pour des objets concurrents

Le langage CAOLAC (Cf. chapitre 6 et [Sei96, Sei97b, SDF97]) permet de définir des politiques de synchronisation pour les méthodes concurrentes d'un langage orienté objet. Nous l'utilisons pour gérer les aspects liés à la concurrence dans les programmes et les structures de contrôle de niveau groupe. Nous en avons réalisé un prototype pour le langage du système distribué GUIDE [BBD+91]. Néanmoins, les concepts introduits dans CAOLAC sont suffisamment généraux pour que sa conception dépende le moins possible du langage de base choisi. Afin de le

vérifier, nous prévoyons d'en réaliser une implantation pour le langage C++ sur une plateforme CORBA.

Le langage CAOLAC se présente comme un protocole de niveau méta-objet [KdRB91]. Ce terme désigne une forme d'architecture réflexive dans laquelle chaque objet d'un langage est géré par un méta-objet. Celui-ci intercepte toutes les invocations de méthodes destinées à l'objet de base, les traite, puis éventuellement, les lui délivre. Le code du traitement associé à la méthode est alors exécuté, puis les paramètres sont retournés au méta-objet qui les retransmet à l'appelant. Une telle architecture permet de gérer facilement et de façon transparente de nombreux mécanismes. Par exemple, la sécurisation des appels de méthodes peut être traitée par un niveau méta qui réalise l'authentification des correspondants et encrypte les données. Le niveau de base gère, quant à lui, uniquement le mécanisme d'invocation standard. De même, la gestion de la réplication, de la localisation ou des appels de procédures à distance peut être réalisée par un niveau méta. De plus, plusieurs niveaux peuvent être assemblés lorsque l'on a besoin, par exemple, d'effectuer des appels à distance sécurisés.

Le langage CAOLAC permet de définir des classes de méta-objets qui gèrent la synchronisation des méthodes concurrentes. Son originalité est constituée par la définition à l'aide de modèles états/transitions du code des méta-classes. Dans CAOLAC, chaque méta-classe s'appelle un comportement. Chaque comportement est associé de façon statique (c'est à dire à la compilation) à une classe GUIDE dont il synchronise l'exécution. Chaque état d'un comportement représente une configuration de la politique de synchronisation. Les transitions entre états définissent les évolutions possibles de cette politique. Chaque transition comporte une garde et un ensemble d'instructions GUIDE. La garde est composée d'une invocation et d'une expression booléenne. La transition est déclenchable et les instructions sont exécutées, lorsque l'invocation correspondante est présente dans la file d'attente et que la condition s'évalue à vrai. Afin d'autoriser le parallélisme intra-objet, plusieurs transitions doivent pouvoir s'exécuter concurremment. Pour cela, nous proposons différentes sémantiques d'états. Ainsi, chaque état possède un ensemble de règles pour la prise en compte de ses transitions entrantes (sémantique entrante) et un ensemble de règles pour le tir de ses transitions sortantes (sémantique sortante). Nous proposons deux types de sémantiques entrantes (*nulle* et *rendez-vous*) et trois types de sémantiques sortantes (*séquentielle*, *parallèle* et *tant que*). Ils permettent d'implanter les mécanismes habituels tels que les rendez-vous de transitions, les constructeurs de parallélisme de type *fork/join* ou les activations périodiques de transitions. Finalement, un mécanisme d'héritage de comportement permet de réutiliser les politiques de synchronisation.

Afin de préciser le fonctionnement du langage CAOLAC, nous définissons, au chapitre 7, une sémantique du modèle états/transitions en terme de logique temporelle d'actions (TLA pour *Temporal Logic of Actions*) [Lam91, Lam94]. Nous n'avons pas formalisé la totalité des fonctionnalités de CAOLAC (par exemple, les mécanismes d'héritage de comportements, d'association entre un comportement et une classe, d'invocation ne sont pas formalisés). Le but de ce chapitre est de préciser le fonctionnement du modèle états/transitions. En particulier, nous définissons pour chaque sémantique (entrante ou sortante), une formule TLA équivalente à son comportement. Ces formules traduisent, par exemple, le nombre de transitions exécutées nécessaires à l'activation d'un état, ou le nombre de transitions devant être exécutées avant de désactiver un état. La formule générale traduisant la synchronisation complète s'obtient alors en réunissant les formules correspondant aux entrées et sorties de tous les états. On est alors en mesure de donner un support formel à la notion de raffinement de comportement : c'est une implication logique entre deux for-

mules TLA. En effet, une formule Ψ raffine une formule φ, si et seulement si $\Psi \Rightarrow \varphi$. Néanmoins, nous n'avons pu donner, au chapitre 7, qu'une idée de ce type de preuve. Il nous reste un travail important à réaliser afin de maîtriser pleinement les mécanismes complexes d'axiomatisation et de déduction de TLA. De même, nous pensons qu'il est possible d'effectuer des preuves de propriétés de sûreté et de vivacité sur les comportements CAOLAC. En effet, celles-ci se ramènent, en TLA, à des preuves de propriétés toujours vraies et inévitablement vraies.

Perspectives

L'objectif majeur proposé par cette thèse consistait à mettre en place une démarche algorithmique systématique pour la conception, la preuve et l'implantation d'applications distribuées. Une première étape a été franchie grâce à différents outils. Rappelons, entre autres, la définition de la notion de programme à base de connaissances de niveau groupe, la définition du langage CAOLAC et l'utilisation d'un langage orienté objets répartis tel que GUIDE, pour conduire les descriptions comportementales respectivement, aux niveaux groupe, objet et méthode. Néanmoins, l'aspect le plus important à traiter reste certainement l'unification de ces trois outils dans un seul et même formalisme.

Ce travail peut donc être poursuivi de multiples façons. Un certain nombre d'objectifs, d'une part à court terme, et d'autre part à moyen et long terme, peuvent être mentionnés. La quantité de travail à mettre en œuvre se chiffre en semaines pour les premiers, tandis qu'elle se chiffre certainement en mois pour les seconds.

Perspectives à court terme

Les perspectives à court terme concernent, essentiellement, l'implantation du langage CAOLAC pour une plateforme autre que le système GUIDE, la définition de nouvelles structures de contrôle, la poursuite de la sémantique de CAOLAC et l'étude de la conception d'autres algorithmes avec notre approche.

L'un des premiers objectifs à court terme concerne l'implantation du langage CAOLAC au-dessus d'une autre plateforme que le système GUIDE. Ce système comporte de nombreux concepts novateurs et nous a donné entière satisfaction. Néanmoins, il semble qu'actuellement les normes d'interopérabilité pour systèmes ouverts du style CORBA ou DCE prennent le pas sur les systèmes répartis fermés comme GUIDE, Arjuna ou Amoeba. De ce fait, une plateforme envisageable pour un tel portage pourrait être constituée par CORBA associée au langage C++ ou Java. Cela nous permettrait, d'une part de comparer les implantations avec le langage GUIDE à celles avec des langages comme C++ ou Java, et d'autre part, à comparer CAOLAC avec d'autres protocoles méta-objet comme Open C++ [Chi95], PC++ [WSMB95] ou MetaJava [GK97a, GK97b]. Ce travail d'implantation pure ne pose pas de difficultés théoriques majeures. Une grande partie du code déjà développé doit pouvoir être réutilisé. Il peut, certainement, être confié à un étudiant dans le cadre d'un stage d'ingénieur ou de DEA.

Une seconde possibilité de poursuite concerne la définition de structures de contrôle supplémentaires. Les quatre structures proposées au chapitre 5 (le schéma de phasage, la conditionnelle, l'itération et la récursion) couvrent, selon nous, les comportements de base de nombreuses applications. On peut alors envisager de développer cet axe en recherchant de nouvelles structures. Il est probable, par exemple, que des comportements types tels que la collecte d'état global ou des

processus plus compliqués comme la gestion de cohérences mémoires ou de diffusions sur groupe puissent être définis sous forme de structures de contrôle. Par exemple, les travaux menés au sein de notre équipe (Cf. [Cos95, CDFS96]) ont permis de dégager deux sémantiques principales pour l'invocation de méthode à distance sur un groupe d'objets : une première sémantique dite ordre de soumission et une seconde dite ordre d'exécution. Ces mécanismes ainsi que d'autres (on peut envisager par exemple l'adaptation des ordres de diffusion causaux et totaux du mode message à un univers d'invocation de méthode) constituent certainement des comportements suffisamment courant pour justifier leur définition sous forme de structure de contrôle.

La sémantique en terme de logique temporelle d'actions du langage CAOLAC est également une activité qui demande à être développée. Comme nous l'avons signalé dans la première partie de la conclusion, la majeure partie du travail réalisé jusqu'à présent concerne l'interprétation des sémantiques d'états. Il semble nécessaire de se pencher, entre autres, sur les mécanismes d'héritage d'états et sur les mécanismes d'associations entre un comportement et une classe. Ce travail devrait permettre de consolider l'assise formelle du langage CAOLAC. Il devrait déboucher également sur une étude plus approfondie des mécanismes d'axiomatisation et de déduction de TLA.

Perspectives à moyen et long termes

Parmi les perspectives de développement à moyen et long termes, il peut être intéressant d'introduire, dans notre démarche réflexive, des aspects autres que la coordination ou la synchronisation, comme par exemple la sécurité et le déverminage. Néanmoins, la perspective majeure de développement de ce travail reste la mise en place d'une démarche unifiée d'algorithmique répartie.

Dans cette thèse, nous avons montré que la coordination et la synchronisation d'un ensemble d'objets distribués pouvaient être exprimées de façon simple au niveau méta d'un langage objet. Néanmoins, l'intérêt d'une démarche réflexive ne se limite pas seulement à ces deux aspects. Par exemple, l'introduction de mécanismes de déverminage à la volée et d'observation d'exécution à un niveau réflexif semble être prometteur. Ainsi, l'instrumentation du code source peut se faire indépendamment du code effectif à l'aide de méta-objets.

Dans un second temps, la refonte de la notion de programme à base de connaissances de niveau groupe, du langage CAOLAC et d'un langage comme GUIDE, dans un seul et même ensemble semble souhaitable. On espère ainsi être en mesure d'effectuer une construction rigoureuse de programmes distribués et pouvoir appliquer des mécanismes de preuve. Pour cela, deux voies semblent envisageables. Cet ensemble peut prendre la forme d'un système formel permettant, à la manière de la méthode B [Abr96], d'implanter, plus ou moins directement, les modèles suffisamment raffinés. De façon alternative, cet ensemble peut être directement un langage exécutable associé à une sémantique formelle. Dans les deux cas, la formalisation devra rassembler dans un même système une grande partie des notions évoquées dans cette thèse. Par exemple, dans l'état actuel de ce travail, les modalités épistémiques sont utilisées pour décrire les aspects de distribution, tandis que les modalités temporelles décrivent la concurrence. Il faudrait les intégrer pour en faire, par exemple, une logique épistémique et temporelle d'actions. Ce travail demande, certainement, de nombreux efforts. De plus, le système obtenu risque d'être complexe. En effet, l'axiomatisation de TLA est déjà, en soi, complexe. Par la suite, le choix d'un système d'axiomatisation, parmi les nombreux systèmes existants, pour la logique épistémique (Cf. par exemple [MvdH95] pour une présentation de ces systèmes) n'est pas forcément trivial. Finalement, l'intégration des axiomes épistémiques et temporels devra être étudiée. De ce fait, la mise en place d'une démarche systéma-

tique de construction d'algorithmes répartis demande encore de nombreux travaux de recherche et de développement.

Annexe

Annexe A

Langage CAOLAC

Cette annexe est consacrée à une présentation de la syntaxe du langage CAOLAC (paragraphe A.1), à la façon dont il est traduit en langage GUIDE (paragraphe A.2).

A.1 Présentation du langage CAOLAC

Le paragraphe A.1.1 présente les principes du langage CAOLAC. Nous commençons par les concepts de base (paragraphe A.1.1.1) et la façon dont on associe un comportement CAOLAC à une classe GUIDE (paragraphe A.1.1.2). Nous les illustrons au paragraphe A.1.1.3. Nous présentons alors certains concepts avancés (réutilisation de comportements, mécanismes de gestion d'historiques) au paragraphe A.1.1.4. Finalement, le paragraphe A.1.2 est consacré à la BNF du langage CAOLAC.

A.1.1 Définition de comportements

A.1.1.1 Concepts de base

Le langage CAOLAC permet de définir des classes de synchronisation que nous appelons comportements. La figure A.1 illustre, de façon schématique, les concepts de base d'un comportement CAOLAC.

Rappelons tout d'abord que le langage CAOLAC est défini pour exprimer des comportements de synchronisation entre méthodes par une une approche de protocole méta-objet. Le langage est orienté états/transitions. Nous avons choisi de distinguer les classes de base du langage GUIDE préfixées par le mot clé `class`, des méta-classes, appelées comportements dans CAOLAC, utilisées pour la synchronisation et prefixées par le mot clé `behaviour`. Un comportement possède un identificateur et ses instructions sont comprises entre accolades. Son corps comprend deux parties : la partie déclaration et la partie définition des états.

La partie déclaration définit les variables d'instances du comportement, les invocations de méthodes qu'il peut traiter, les méthodes de la classe GUIDE qu'il peut appeler et son état initial. Ces différents éléments se définissent de la façon suivante :

- le mot clé `variables` suivi de deux points permet de commencer la déclaration des variables d'instances du comportement. Celles-ci se déclarent comme pour celles d'une classe GUIDE à l'aide d'un identificateur, d'un type et éventuellement, d'une expression d'initialisation.

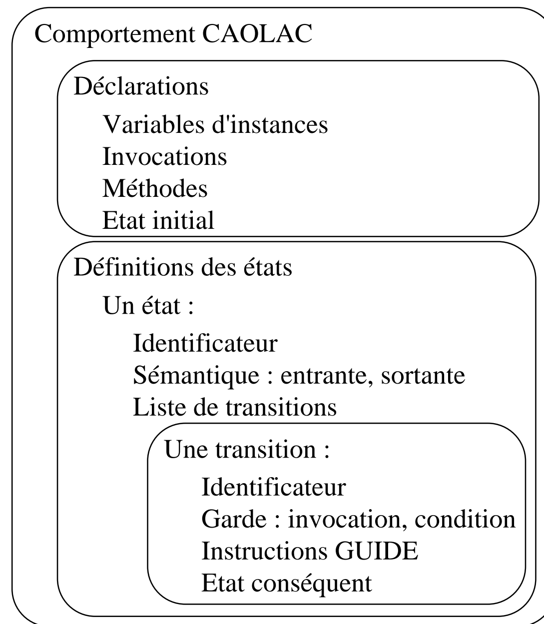


FIG. A.1 – *Concept de base d'un comportement CAOLAC*

- la déclaration des invocations de méthodes pour lesquelles un schéma de synchronisation est défini par le comportement commence par le mot clé `invocations` et est suivie de deux points. Ces méthodes sont en fait visibles par un utilisateur de la classe comportementale. Le profil de chaque invocation est défini de la même façon que celui d'une méthode du langage GUIDE. Ainsi dans l'exemple de la figure A.2, *Put(IN e : REF TElement)* désigne une invocation d'identificateur *Put* et acceptant en paramètre d'entrée un élément *e* de type *REF TElement*.

- les méthodes des classes GUIDE pouvant être appelées par un comportement CAOLAC sont déclarées dans la section commençant par le mot clé `methods`. Leur profil se définit comme celui des méthodes GUIDE. Cette déclaration permet d'assurer le lien entre le méta-objet comportemental et l'ensemble des méthodes de l'objet qu'il utilise.

- finalement, le dernier élément de la partie déclaration d'un comportement concerne l'état initial (mot clé `initial state` suivi de deux points). C'est l'état actif à l'instanciation du comportement.

La définition d'un état dans un comportement CAOLAC débute par le mot clé `state` suivi d'un identificateur. Chaque identificateur doit être unique au sein du comportement. Par convention, on désigne un état en le préfixant par l'identificateur de son comportement et par deux fois le symbole deux points. Ainsi *BufferDeTailleFixe::Plein* désigne l'état *Plein* du comportement *BufferDeTailleFixe*. Chaque état possède une sémantique et définit, entre accolades, un ensemble de transitions.

- la sémantique d'état comprend une sémantique pour les transitions entrantes et une sémantique pour les transitions sortantes (Cf. paragraphe 6.3.3.2). Elles sont séparées par une virgule. La sémantique entrante peut être : *nulle* (pas de mot clé) ou *rendez-vous* (mot clé

```
behaviour BufferDeTailleFixe {

 // Déclarations

 variables:
  // Variables d'instance
  // du comportement

 invocations:
  // Profil des invocations
  Put( IN e : REF TElement );
  Get : REF TElement;

 methods:
  // Profil des méthodes
  IsFull : Boolean;
  IsEmpty : Boolean;

 // Etat initial
 initial state : Vide;


 // Définition des états

 // Identificateur + sémantique
 state Vide sequential {

  // Transitions de l'état

  TPut {

   // Garde de la transition
   // invocation + condition
   invocation( Put(e) );
   require( true );

   // Instructions GUIDE à exécuter
   // BASE : réf à l'objet à synchroniser
   BASE.Put(e);
   return;

   // Etat conséquent
   become( Partiel );
  }
 }

 state Partiel sequential {
  TPut {
   invocation( Put(e) );
   require( true );
   BASE.Put(e); return;
   if BASE.IsFull = true
   then become( Plein );
   else become( Partiel );
   end;
  }
  TGet {
   invocation( Get );
   require( true );
   return BASE.Get;
   if BASE.IsEmpty = true
   then become( Vide );
   else become( Partiel );
   end;
 }}

 state Plein sequential {
  TGet {
   invocation( Get );
   require( true );
   return BASE.Get;
   become( Partiel );
}}}
```

FIG. A.2 – *Définition d'un comportement CAOLAC*

`join`). La sémantique sortante peut être : *séquentielle* (mot clé `sequential`), *parallèle* (mot clé `parallel`) ou *tant que* (mot clé `server while` *condition*) et une sémantique pour les transitions entrantes. Par exemple, la définition *state Vide sequential* figure A.2, désigne un état dont l'identificateur est *Vide*, de sémantique entrante *nulle* et de sémantique sortante *séquentielle*. De même, *state essai join, server while (i=10)* désigne un état dont l'identificateur est *essai*, de sémantique entrante *rendez-vous* et qui reste actif tant que la condition *i=10* est vérifiée.

- chaque transition possède un identificateur qui doit être unique au sein de l'état. Par convention, on désigne une transition en la préfixant par l'identificateur de son comportement, suivi par deux fois le symbole deux points, suivi de l'identificateur de son état, suivi d'un point. Ainsi *BufferDeTailleFixe::Plein.TGet* désigne la transition *TGet* de l'état *Plein* du comportement *BufferDeTailleFixe*. Le corps d'une transition comprend une garde, un ensemble d'instructions GUIDE et un état conséquent. Ces trois éléments se définissent de la façon suivante :

  - la garde comprend deux parties : une invocation de méthode à prendre en compte (mot clé `invocation` suivi d'un type d'invocation entre parenthèses) et une condition booléenne à évaluer (mot clé `require` suivi d'une expression entre parenthèses). L'invocation doit avoir été déclarée dans la section `invocations` du comportement. Si elle comporte des paramètres, ceux-ci doivent être présents. La sémantique d'une transition est définie de la façon suivante : si l'état antérieur est actif, si l'invocation spécifiée par l'instruction `invocation` est présente dans la file d'attente, si la condition de l'invocation `require` s'évalue à vrai, alors l'invocation est retirée de la file d'attente et les instructions de la transition sont exécutées. L'invocation à prendre en compte et/ou la condition à évaluer peuvent être omises.
  - les instructions sont exécutées lorsque la garde de la transition est vérifiée. Toutes les instructions algorithmiques (boucles, conditionnelles, affectations, retour de paramètres, ...) du langage GUIDE peuvent être utilisées. Des déclarations de variables locales peuvent être insérées n'importe où dans ce code. Néanmoins, on ne peut déclarer, au sein des instructions d'une transition, ni types, ni classes, ni méthodes, ni procédure. Le langage CAOLAC fournit l'instruction `BASE` qui est une référence à l'objet de base associée au comportement. Elle permet, soit d'appeler des méthodes de l'objet de base, soit de délivrer des invocations de méthode à cet objet de base.
  - l'état conséquent à la transition est spécifié par l'instruction CAOLAC `become` suivi d'un identificateur d'état entre parenthèses. Lorsque plusieurs états conséquents sont envisageables pour une même transition, des tests peuvent être associés à l'instruction `become`. Par exemple, la transition *BufferDeTailleFixe::Partiel.TPut* figure A.2, utilise le test *if BASE.IsFull then become(Plein); else become(Partiel)*. Si l'appel de la méthode *IsFull* sur l'objet de base retourne vrai, alors l'état *Plein* est activé, sinon c'est l'état *Partiel* qui l'est.

A.1.1.2 Association entre un comportement CAOLAC et une classe GUIDE

La figure A.2 fournit le code du comportement *BufferDeTailleFixe* qui synchronise l'ajout (invocation *Put*) et le retrait d'éléments (invocation *Get*) dans un tampon de taille fixe. Il ne

donne pas la façon dont un élément est effectivement ajouté ou retiré du tampon. Il spécifie seulement qu'un élément ne peut être ajouté que si le tampon n'est pas plein, et qu'il ne peut être retiré que si le tampon n'est pas vide. Ces traitements effectifs sont définis dans une classe GUIDE.

Par exemple, la figure A.3 fournit le code effectif de la classe *File*. Elle implante une politique dernier entré, premier sorti pour le tampon de taille fixe. Nous avons ajouté aux mots clés du langage GUIDE, les mots clés `with behaviour` pour indiquer que la classe implante un comportement CAOLAC (ici *BufferDeTailleFixe*). La classe *File* fournit le code effectif (i.e. en dehors de toute synchronisation) de toutes les invocations de méthodes prises en compte par le comportement (ici *Put* et *Get*) et de toutes les méthodes appelées par le comportement (ici *IsFull* et *IsEmpty*).

Remarque 1 : plusieurs classes GUIDE peuvent utiliser le même schéma de synchronisation (i.e. le même comportement CAOLAC). Par exemple, on peut envisager de synchroniser une classe *Pile* avec *BufferDeTailleFixe*. Il suffit, pour cela, que la classe Pile fournisse une implantation de toutes les invocations et les méthodes (i.e. *Put*, *Get*, *IsFull*, *IsEmpty*), et qu'au moment de la traduction en GUIDE, le traducteur connaisse le comportement BufferDeTailleFixe (i.e. soit le comportement et la classe sont inclus dans le même fichier source, soit ils sont dans deux fichiers différents passés en paramètre au traducteur).

Remarque 2 : rappelons que la synchronisation intra-objet en GUIDE est définie par des clauses comportementales de type commandes gardées. Celles-ci sont appliquées après les règles définies par le comportement CAOLAC (i.e. les invocations de méthodes sont d'abord synchronisées par le comportement, puis par les clauses de la classe). Néanmoins, l'emploi de clauses comportementales fait double emploi avec les mécanismes que nous proposons et va à l'encontre de l'objectif de séparation de la synchronisation et des traitements effectifs que nous nous sommes fixés. Nous ne recommandons donc pas leur emploi conjointement avec un comportement CAOLAC.

A.1.1.3 Fonctionnement d'un comportement

D'une manière générale, plusieurs états d'un comportement CAOLAC peuvent être actifs simultanément (à l'instanciation du comportement, seul l'état initial l'est). Les invocations de méthodes à destination de l'objet de base sont détournées vers l'instance de comportement CAOLAC (appelée méta-objet). Le fonctionnement de l'ensemble, lorsqu'une invocation de méthode arrive, est illustré figure A.4 et décrit dans la suite de ce paragraphe. Nous considérons que l'état Partiel du comportement *BufferDeTailleFixe* est actif et qu'une invocation *Put* arrive.

1. Evaluation de la garde des transitions
   Si une transition issue d'un état actif prend en compte une invocation de même type que l'invocation arrivante (par exemple la transition *BufferDeTailleFixe::Partiel.TPut* prend en compte une invocation *Put* par l'intermédiaire de l'instruction *invocation(Put(e))*), et
   si la condition de la garde est vraie (ici *require(true)* est toujours vraie), alors l'invocation est retirée de la file d'attente et selon la sémantique sortante de l'état antérieur, l'activation de cet état évolue.

```
class File
 with behaviour BufferDeTailleFixe
is

 const Max : Integer = 50;
 buf : Array [Max] OF REF TElement;
 ptrEntree, ptrSortie : Integer = 0;


 method IsFull : Boolean;
 begin
  return (ptrEntree-ptrSortie) mod Max
           = Max-1;
 end IsFull;

 method IsEmpty : Boolean;
 begin
  return ptrEntree=ptrSortie;
 end IsEmpty;

 method Put( IN e : REF TElement );
 begin
  buf[ptrEntree] := e;
  ptrEntree := (ptrEntree+1) mod Max;
 end Put;

 method Get : REF TElement;
  e: REF TElement;
 begin
  e := buf[ptrSortie];
  ptrSortie := (ptrSortie+1) mod Max;
  return e;
 end Get;

end File.
```

FIG. A.3 – *Définition d'une classe GUIDE associée à un comportement CAOLAC*

Si l'état antérieur est à sémantique sortante :

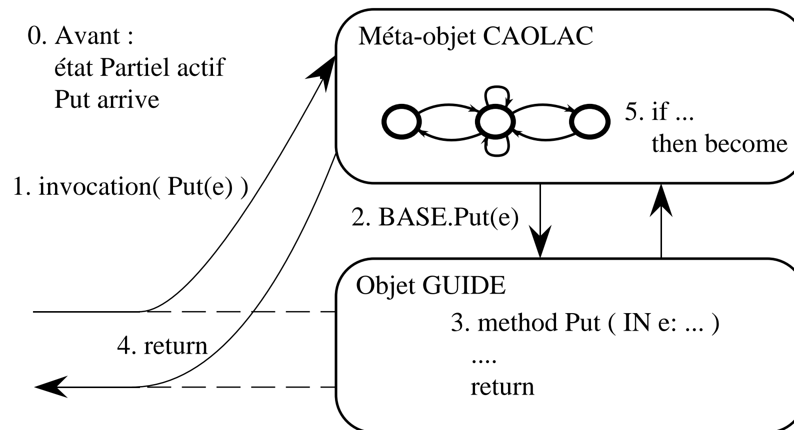
- *séquentielle* (ce qui est le cas ici) : l'état antérieur est désactivé,
- *parallèle* : l'état antérieur n'est désactivé que lorsque toutes les transitions sortantes ont été déclenchées une fois chacune,
- *tant que* : l'état antérieur reste actif tant que la condition fournie par le programmeur est vraie.

2. Exécution des instructions de la transition
   Lorsque une garde est vérifiée, les instructions GUIDE qui la suivent sont exécutées. Si une instruction *BASE.methode* est rencontrée, l'invocation de méthode est délivrée à l'objet de base. Cette commande agit comme un appel de méthode entre le comportement et l'objet de base. Ainsi, *BASE.Put(e)* provoque un appel de la méthode *Put*.

3. Exécution de la méthode dans l'objet de base GUIDE
   La méthode de l'objet de base est exécutée jusqu'à ce qu'un `return` soit rencontré. Cette instruction provoque alors un retour au méta-objet définissant le comportement.

4. Retour à l'appelant
   Les instructions suivant l'appel de la méthode de base sont exécutées. Notons que les instructions d'une transition peuvent comporter zéro, un ou plusieurs instructions quelconques ainsi que des appels à une méthode de base. Lorsqu'une instruction `return` est rencontrée dans le comportement, l'invocation est retournée à son appelant. Si cette instruction est absente, l'exécution se poursuit dans l'état conséquent spécifié par la transition.

5. Etat conséquent
   L'instruction `become` désigne le prochain état à activer. Elle peut se trouver après une ins-




```
state Partiel sequential {
  TPut {
    invocation( Put(e) );
    require( true );
    BASE.Put(e);
    return;
    if BASE.IsFull = true
    then become( Plein );
    else become( Partiel );
    end;
  }
}
```

FIG. A.4 – *Fonctionnement du comportement BufferDeTailleFixe pour la transition Partiel.TPut*

truction `return` (dans ce cas, le comportement renvoie l'invocation à l'appelant, désigne le prochain état à activer et attend une autre invocation), ou avant (dans ce cas, le comportement désigne le prochain état à activer et examine les gardes des transitions issues de cet état).

Si la sémantique entrante de l'état à activer est :

- *nulle* : l'état est activé immédiatement,
- *rendez-vous* : toutes les transitions entrantes doivent avoir été exécutées une fois chacune avant que l'état soit activé.

A.1.1.4 Concepts avancés

Dans ce paragraphe, nous apportons un certain nombre de précisions sur trois concepts avancés du langage CAOLAC : la réutilisation de comportement, les invocations génériques, les compteurs d'événements et les historiques d'événements.

A.1.1.4.1 Réutilisation de comportements

Le langage CAOLAC a pour but de faciliter le développement d'applications en réutilisant des comportements de synchronisation préexistants. Nous avons défini pour cela deux relations : une dite méta-comportementale et une seconde dite sous-comportementale.

La première relation permet de créer des tours métas de comportements CAOLAC. Le comportement de plus haut niveau dans la tour prend en compte les invocations de méthode. Chaque comportement applique une synchronisation et délivre les invocations au comportement immédiatement inférieur dans la tour. Le traitement de synchronisation est ainsi séparé en plusieurs modules. Ceux-ci peuvent être conçus et testés de manière indépendante. Cette approche facilite la lisibilité et la compréhension de l'ensemble. L'association entre un comportement et son méta peut se faire de deux façons. Dans le premier cas, le comportement de niveau inférieur déclare avec le mot clé `with behaviour` qu'il doit être synchronisé par un méta (ce mécanisme est identique à celui utilisé par une classe qui s'associe à un comportement). Dans le second cas, le comportement de niveau supérieur déclare avec le mot clé `meta of` qu'il synchronise un autre comportement.

behaviour ComportementSuperieur **meta of** ComportementIntermediaire { /* ... */ }
behaviour ComportementIntermediaire { /* ... */ }
behaviour ComportementInferieur **with behaviour** ComportementIntermediaire { /* ... */ }

FIG. A.5 – *Relations méta entre comportements CAOLAC*

La seconde relation de réutilisation de comportement est dite sous-comportementale. Elle permet de créer de nouveaux comportements en héritant la structure états/transitions de comportements existants. Dans la déclaration d'un sous-comportement, le mot clé `subbehav of` désigne le sur-comportement associé. L'ensemble des états, des transitions et du code des transitions du sur-comportement est repris dans le sous-comportement. Les états peuvent être redéfinis ou partitionnés. Les transitions peuvent être redéfinies. Finalement, de nouveaux états et de nouvelles transitions peuvent être ajoutés.

A.1.1.4.2 Invocations génériques

Dans certains comportements, le même type de synchronisation doit pouvoir être appliqué à plusieurs invocations. Par exemple, un comportement *Verrou* permettant de bloquer temporairement l'accès à une classe fournit deux méthodes *Lock* et *Unlock* pour, respectivement, bloquer et débloquer la classe. Quelles que soient les autres invocations envisagées, elles ont toutes, vis à vis du comportement *Verrou*, la même synchronisation : dans l'état bloqué, aucune n'est acceptée, tandis que dans l'état débloqué, toutes le sont. On peut alors les rassembler sous une dénomination générique. Les invocations génériques sont définies, à l'aide du mot clé `generic invocations`, dans la partie déclaration d'un comportement CAOLAC. Chaque invocation possède un identificateur (*Any* dans l'exemple de la figure A.7). Chaque invocation générique est alors associée, par la clause `where`, à une ou plusieurs invocations concrètes lors de la définition de la classe (dans l'exemple, *Any* est associée à *Put* et à *Get*).

```
behaviour SurComportement {

  state Etat1 { /* ... */ }
  state Etat2 { /* ... */ }
  state Etat3 {
    Transition1 { /* ... */ }
    Transition2 { /* ... */ }
  }
  state Etat4 { /* ... */ }
}

behaviour SousComportement subbehav of SousComportement {
  /* ... */
  state NouvelEtat { /* un nouvel état */ }

  /* Etat1 hérité */

  state Etat2 {
    /* toutes les transitions de Etat2 sont héritées, TNouvelleTransition est ajoutée */
    TNouvelleTransition { /* une nouvelle transition */ }
  }

   state Etat3 {
    /* toutes les transitions de Etat3 sont héritées, Transition1 et Transition2 sont redéfinies */
    Transition1 { /* ... */ }
    Transition redefines Transition2 { /* ... */ }
  }

  state Etat4 {
    /* toutes les transitions de Etat4 sont héritées, Etat4 est partitionné */
    expr1 : state SousEtat1 { /* ... */ }
    expr2 : state SousEtat2 { /* ... */ }
    /* ... */
    else : state SousEtatn { /* ... */ }
  }
}
```

FIG. A.6 – *Héritage de comportements*

```
behaviour Verrou {
  /* ... */
  generic invocations: Any;
  /* ... */
}

class BufferVerrouille with behaviour Verrou
  where Any => Put( IN e:REF TElement ), Get is
  // ...
end BufferVerrouille.
```

FIG. A.7 – *Invocations génériques*

A.1.1.4.3 Compteurs d'événements

Les compteurs d'événements du langage CAOLAC étendent ceux du langage GUIDE aux éléments de synchronisation ajoutés par CAOLAC. Ainsi, GUIDE permet de compter le nombre d'instances arrivées, commencées, terminées, en attente, en cours d'exécution, d'une méthode dans une classe concurrente. Le langage CAOLAC reprend ces cinq compteurs concernant les invocations de méthodes et introduit des compteurs pour les états et les transitions. Ainsi, trois compteurs sont disponibles pour les transitions : *StartedTransition*, *CompletedTransition* et *CurrentTransition*. Ils enregistrent le nombre d'instances, respectivement, commencées, terminées, en cours d'exécution d'une transition. Finalement, le compteur *CurrentState* retourne un booléen qui vaut vrai lorsque l'état est actif.

A.1.1.4.4 Historiques d'événements

Les historiques ont pour but d'enregistrer une suite d'événements en vue de la construction de relations d'ordre dans les applications distribuées. On veut, par exemple, être en mesure de fournir l'ordre des invocations de méthodes correspondant à une exécution donnée, c'est à dire l'ordre dans lequel les différentes méthodes des objets de l'application ont été exécutées. Cet ordre est utile lorsque, par exemple, on essaie de reconstituer un état global cohérent en vue du déverminage ou pour constituer des points à partir desquels l'application peut être reprise.

De nombreux travaux ont été effectués pour cela dans le domaine des applications répartis conçues selon des ensembles de processus communiquant par envoi de messages asynchrones (Cf. [SM92] pour la référence majeure dans ce domaine). Un ensemble de travaux réalisés dans notre équipe [Pla94, PDFS95b, PDFS95a], ont étendu cette approche aux environnements objets répartis et ont identifié, dans ce contexte, quatre sources d'ordre. Plutôt que d'essayer de traduire, de façon brute, les dépendances entre blocs de code provenant des seules opérations de communication, comme c'est le cas dans les approches précédentes, ces quatre sources d'ordre ont pour but de traduire des dépendances plus proches de la sémantique des programmes. Dans ce contexte, les historiques du langage CAOLAC ont pour but d'enregistrer les occurrences d'événements permettant de reconstituer trois de ces quatre sources d'ordre. Ainsi, on enregistre[1] :

– les opérations d'appel de méthodes et les opérations d'appel de méthodes en parallèle (par

1. L'enregistrement n'est pas systématique. Afin de ne pas ralentir inutilement l'exécution, il n'est effectué qu'à la demande.

un constructeur `CO_BEGIN/CO_END`) pour reconstituer l'ordre d'appel de méthodes,

- les événements d'arrivée, de début et de fin de méthodes afin de reconstituer l'ordre de synchronisation intra-objet,
- les opérations de lecture et d'écriture de variables partagées pour construire l'ordre transactionnel.

Ce mécanisme d'enregistrement des historiques n'est qu'une première tentative visant à faire converger les travaux concernant, d'une part, les relations d'ordre et le déverminage d'applications réalisés au sein de notre équipe, et notre travail de spécification de comportements. Il nous semble nécessaire de poursuivre cette démarche, d'une part, afin de compléter l'enregistrement des informations nécessaires aux quatre sources d'ordre et, d'autre part, afin d'évaluer le bien-fondé et l'adéquation des choix réalisés.

A.1.2 Syntaxe

Nous présentons dans ce paragraphe la syntaxe du langage CAOLAC. Les éléments lexicaux du paragraphe A.1.2.1 sont traités à l'aide de l'analyseur lexical `flex` (compatible `lex`). La BNF du paragraphe A.1.2.2 est traitée à l'aide de l'analyseur grammatical `bison` (compatible `yacc`).

A.1.2.1 Eléments lexicaux

Identificateurs

Les identificateurs de comportement, d'état, de transition, de variable, d'invocation et de méthode doivent être alphanumériques. Toute combinaison de lettres majuscules ou minuscules, de chiffres et du caractère _ est un identificateur valide sauf :

- si elle commence par un chiffre ou par le caractère _,
- si elle commence par l'un des préfixes `c_`, `g_`, `i_`, `p_`, `s_`, `t_` ou `v_`,
- si elle appartient à la liste des mots clés définie ci-dessous.

Mots clés

Le langage CAOLAC n'est pas sensible à la casse des mots clés. Ceux-ci peuvent être écrits indifféremment en majuscules ou en minuscules.

```
BASE                    EVINVOKED        INVOCATIONS
BECOME                  EVPARCALL        INVOKEDINVOCATION
COMPLETEDINVOCATION     EVREAD           JOIN
COMPLETEDTRANSITION     EVSTARTED        METABEHAV
CURRENTINVOCATION       EVWRITE          METHODS
CURRENTTRANSITION       GENERIC          PAR
CURRENTNUMORDER         HISTORY          PARALLEL
CURRENTSTATE            INITIAL          PENDINGINVOCATION
EVCALL                  IN               PREDICATE
EVCOMPLETED             INVOCATION       POST
```

`REDEFINES`
`REQUIRE`
`SEQUENTIAL`
`SERVER`
`STARTEDINVOCATION`
`STARTEDTRANSITION`
`STATE`
`SUBBEHAV`
`TRACED`
`VARIABLES`
`WHERE`
`WITH`

La BNF présentée ci-dessous utilise, en plus des mots clés précédents spécifiques au langage CAOLAC, les mots clés suivants du langage GUIDE.

`CASE`
`CO_BEGIN`
`CO_END`
`DO`
`ELSE`
`END`
`FOR`
`IF`
`INTEGER`
`NOT`
`OF`
`OTHERS`
`REAL`
`REF`
`REPEAT`
`RETURN`
`SELF`
`STRING`
`SUPER`
`THEN`
`TO`
`UNTIL`
`WHILE`

Littéraux

Les chaînes de caractères sont écrites entre guillemets. On les désigne, ci-dessous dans la BNF, par le jeton STRING. Les caractères sont entre apostrophes et sont désignés par le jeton CHAR. Les valeurs numériques entières et flottantes sont désignées par le jeton NUM.

Commentaires

Les commentaires dans un code source CAOLAC suivent les mêmes règles que celles du langage GUIDE. Tout texte compris entre les délimiteurs `/*` et `*/` constitue un commentaire et est de ce fait ignorée par le compilateur. De même les caractères compris entre le délimiteur `//` et la fin de ligne sont considérés comme des commentaires.

A.1.2.2 BNF

Cette partie présente la syntaxe complète du langage CAOLAC. Les mots clés sont écrits en fonte Typewriter (par exemple `BEHAVIOUR`). ident désigne un identificateur valide défini selon les règles données ci-dessus. Les caractères entre apostrophes font partie des signes de ponctuation du langage. Les signes | * et "" représentent respectivement l'alternative, la répétition (de *0* à *n* occurrences) et l'absence de syntaxe.

<comportement> ::= `BEHAVIOUR` ident <comportements hérités> '{'
<variables>
<invocations génériques>
<invocations>
<méthodes>
<états>
'}'

<variables> ::= `VARIABLES` ':' (<identificateurs> ':' <type> <initialisation> ';' <tracé>)*
<identificateurs> ::= ident (',' ident)*

```
<type> ::= INTEGER
         | REAL
         | STRING
         | STRING'[' nombre ']'
         | REF ident
         | REF ident OF <type>
         ;
<initialisation> ::= '=' <expression>
                   | ""
                   ;
<tracé> ::= TRACED ';'
          | ""
          ;

<invocations génériques> ::= GENERIC INVOCATIONS ':' ( ident ';' )*

<invocations> ::= INVOCATIONS ':' ( <interface> ';' <tracé> )*
<interface> ::= ident <paramètres> <retour>
<paramètres> ::= '(' <paramètre> ( ';' <paramètre> ) * ')'
               | ""
               ;
<paramètre> ::= <mode> <identificateurs> ':' <type>
<mode> ::= IN | IN_OUT | OUT | ""
<retour> ::= ':' <type>
           | ""
           ;

<méthodes> ::= METHODS ':' ( <interface> ';' )*

<états> ::= INITIAL STATE ':' ident ';' ( <état simple> | <état partionné> )*

<état simple> ::= STATE ident <prédicat> <sémantique> '{' <transition>* '}'
<prédicat> ::= PREDICATE '(' <expression> ')'
             | ""
             ;
<sémantique> ::= <sémantique entrante> <sémantique sortante>
<sémantique entrante> ::= JOIN ','
                        | ""
                        ;
<sémantique sortante> ::= SEQUENTIAL
                        | PARALLEL
                        | SERVER WHILE '(' <expression ')'
                        | ""
                        ;

<état partitionné> ::= STATE ident '{' <partition>* ELSE ':' <état simple> '}'
<partition> ::= <expression> ':' <état simple>
```

```
<transition> ::= ident <transition redéfinie> '{' <garde> <action>* '}'
<transition redéfinie> ::= REDEFINES ident
                          | ""
                          ;

<garde> ::= <invocation gardée> <condition gardée>
<invocation gardée> ::= INVOCATION '(' <interface> ')' ';'
                        | ""
                        ;
<condition gardée> ::= REQUIRE '(' <expression> ')' ';'
                       | ""
                       ;

<actions> ::= <action>*
<action> ::= <action élémentaire>
           | IF <expression> THEN <actions> ELSE <actions> END ';'
           | IF <expression> THEN <actions> END ';'
           | WHILE <expression> DO <actions> END ';'
           | FOR <terme> ':' '=' <expression> TO <expression> DO <actions> END ';'
           | CASE <terme> OF (<expression> ':' <actions> END ';')*
             OTHERS ':' <actions> END ';'
           | CASE <terme> OF (<expression> ':' <actions> END ';')* END ';'
           | REPEAT <actions> UNTIL <expression> ';'
           | CO_BEGIN ( ident ':' <terme> ':' '=' <expression> ';' | ident ':' <terme> ';' )*
             CO_END <expression> ';'
           | CO_BEGIN ( ident ':' <terme> ':' '=' <expression> ';' | ident ':' <terme> ';' )*
             CO_END ';'
           | PAR <ident> IN <ident> DO <actions> END ';'
           ;
<action élémentaire> ::= <identificateurs> ':' <type> <initialisation> ';' <tracé>
                       | <terme> ':' '=' <expression> ';'
                       | <terme> ';'
                       | RETURN <expression> ';'
                       | RETURN ';'
                       | BECOME '(' <état conséquent> ')'
                       ;
<état conséquent> ::= ident
                    | ident '.' ident
                    ;

<expression> ::= <expression> '+' <expression>
               | <expression> '-' <expression>
               | <expression> '*' <expression>
               | <expression> '/' <expression>
               | NOT '(' <expression> ')'
               | '(' <expression> ')'
```

```
                    | <terme>
                    | <terme de synchronisation>
                    | STR
                    | CAR
                    | NUM
                    | TRUE
                    | FALSE
                    ;
<terme> ::= <terme> '.' <terme>
          | ident '(' <expressions> ')'
          | ident '[' <expressions> ']'
          | SELF
          | SUPER
          | BASE
          | ident
          | <terme d'historique>
          ;
<terme d'historique> ::= HISTORY '.' SEARCHFORLASTEV '(' ident ',' <type événement> ')'
                       | HISTORY '.' PRINT '(' ident ')'
                       ;
<type événement> ::= EVINVOKED
                   | EVSTARTED
                   | EVCOMPLETED
                   | EVCALL
                   | EVPARCALL
                   | EVREAD
                   | EVWRITE
                   ;
<terme de synchronisation> ::= INVOKEDINVOCATION '(' <identificateur invocation> ')'
                             | STARTEDINVOCATION '(' <identificateur invocation> ')'
                             | COMPLETEDINVOCATION '(' <identificateur invocation> ')'
                             | PENDINGINVOCATION '(' <identificateur invocation> ')'
                             | CURRENTINVOCATION '(' <identificateur invocation> ')'
                             | STARTEDTRANSITION '(' <identificateur transition> ')'
                             | COMPLETEDTRANSITION '(' <identificateur transition> ')'
                             | CURRENTTRANSITION '(' <identificateur transition> ')'
                             | CURRENTSTATE '(' <identificateur état> ')'
                             | CURRENTNUMORDER
                             ;
<identificateur invocation> ::= ident
                              | ident ':' ':' ident
                              ;
<identificateur transition> ::= ident
                              | ident '.' ident
                              | ident ':' ':' ident '.' ident
```

;
<identificateur état> ::= ident
| ident ':' ':' ident
;

<comportements hérités> ::= <sur-comportement> <méta-comportement>
<sur-comportement> ::= `SUBBEHAV OF` ident
| ""
;
<méta-comportement> ::= `WITH METABEHAV` ident
| `META BEHAV OF` ident
| ""
;

A.2 Traduction du langage CAOLAC en GUIDE

Dans ce paragraphe, nous présentons la façon dont le langage CAOLAC est traduit en GUIDE. La traduction des éléments de base est illustré au paragraphe A.2.1 à l'aide de l'exemple du tampon synchronisé de taille fixe. Les codes du comportement *BufferDeTailleFixe* et de la classe *File* associée sont rappelés pages 227 et 228. Le résultat de leur traduction en GUIDE est présenté pages 229 à 231. Le paragraphe A.2.2 présente la traduction des sémantiques d'état à l'aide d'un exemple test dont le code est fourni aux pages A.2.2.2 à A.2.2.2. Finalement, le paragraphe A.2.3 fournit aux pages 239 et 240 le code de classes auxiliaires utilisées par le résultat de la traduction CAOLAC vers GUIDE.

A.2.1 Traduction des éléments de base

Chaque fichier source peut contenir indifféremment des comportements CAOLAC ou des classes GUIDE. Chaque comportement est traduit par une classe : ici *BufferDeTailleFixe* est traduit par la classe *b_FileBufferDeTailleFixe*, et la classe *File* originale est devenue *b_File*. *b_FileBufferDeTailleFixe* fournit une méthode pour chaque invocation (ici *Put* et *Get*). De plus, c'est une sous-classe de *b_File*. Ainsi, chaque invocation de méthode est d'abord pris en compte par le niveau de synchronisation, puis remonte la hiérarchie d'héritage avant d'arriver au niveau des traitements effectifs. En plus du code et des déclarations de la classe *File* originale, celui-ci définit :

- autant de constantes entières, égales respectivement aux valeurs allant de 0 à $n \Leftrightarrow 1$, qu'il y a d'état dans le comportement (ici *s_BufferDeTailleFixeVide*, *s_BufferDeTailleFixePartiel* et *s_BufferDeTailleFixePlein*),
- une constante pour le nombre d'états (*s_BufferDeTailleFixeNbStates*),
- un vecteur de booléen de taille n (*c_BufferDeTailleFixeStates*); chaque élément de ce vecteur vaut vrai si l'état correspondant est actif.

La classe *b_FileBufferDeTailleFixe* fournit une méthode pour chaque invocation (ici *Put* et *Get*), plus une méthode *Init* et une méthode *Stop*. La première initialise le vecteur des états et

```
while true do
  /* on inspecte tous les états qui possèdent */
  /* une transition sortante prenant en compte cette invocation */
  if /* état1 actif */ then
    if /* la condition de la garde est vraie */ then
      /* désactiver l'état */
      /* exécuter les instructions de la transition */
      /* retourner l'invocation */
      /* activer l'état conséquent */
    end;
  end;
  if /* état2 actif */ then
    /* Même traitement */
  end;
  /* ... */
end;
```

FIG. A.8 – *Schéma de traduction d'une invocation CAOLAC en GUIDE*

active l'état initial. La seconde détruit ce vecteur. Le code de chaque invocation suit le schéma suivant de la figure A.8.

Illustrons ce procédé avec la méthode *Get* de la classe *b_FileBufferDeTailleFixe* (Cf. page 230). Deux états (*Plein* et *Partiel*) possèdent une transition sortante prenant en compte *Get* (on obtient donc deux branches `if` dans la boucle `while`). Le test d'activité de ces états se fait à l'aide de la méthode *BelongAndRemove* du vecteur d'état *c_BufferDeTailleFixeStates* (le code complet de la classe de cette variable est fourni page 239). Cette méthode exécute de façon atomique (i.e. en exclusion mutuelle) le test d'activation de l'état et la désactivation [2]. Ici la garde n'a pas de condition, donc il n'y a pas de second test [3].

Les instructions de la transition sont reproduites telles quelles sauf :

- pour la référence d'objet `BASE` qui est transformée en `SUPER`. En effet, lorsqu'on associe un comportement à une classe, le comportement est traduit par une sous-classe de cette classe. Donc, un appel entre un comportement et une classe est traduit par un appel d'une méthode "supérieure" (d'où l'emploi du mot clé `SUPER`).
- pour les valeurs retournées par l'instruction `return` : elles sont évaluées avant le passage à l'état conséquent.

Finalement, l'activation de l'état conséquent se fait par appel de la méthode *Plus* de la classe *c_VectStates*. Comme on peut le constater page 239, toutes les méthodes de cette classe s'exécutent en exclusion mutuelle pour garantir la cohérence du vecteur d'états en cas d'accès multiples concurrents.

2. L'atomicité du test et de la désactivation est nécessaire pour garantir qu'une invocation concurrente de *Get* n'effectue pas simultanément ce même test.

3. Lorsque la garde possède une condition et qu'elle s'évalue à faux, l'état est réactivé en appelant la méthode *Plus* de la classe *c_VectStates*.

Eléments additionnels introduits dans la traduction

Nous avons vu que chaque comportement est traduit par une classe (ici *BufferDeTailleFixe* génère *b_FileBufferDeTailleFixe*) et que chaque classe implantant un comportement est une sur-classe de la classe associée à ce comportement (ici *b_File* est une sur-classe de *b_FileBufferDeTailleFixe* et correspond à la classe *File* originale). Trois autres éléments sont générés lors de cette traduction :

- un type *File* de même nom que la classe originale qui comprend les interfaces de toutes les invocations (ici *Put* et *Get*), de toutes les méthodes (ici *IsFull* et *IsEmpty*) et des méthodes d'initialisation et de destruction *Init* et *Stop*.
- une classe *b_FileFile*, sur-classe de *b_File*, comprenant l'ensemble de ces invocations et méthodes et fournissant un corps vide pour chacune d'elles. Cette classe est nécessaire lorsque le comportement *BufferDeTailleFixe* (et donc la classe *b_FileBufferDeTailleFixe*) définit une invocation qui n'est pas reprise par la classe *File* (i.e. la classe *b_File*). Dans ce cas, la classe n'est pas conforme au type *File*. La classe *b_FileFile* fournit donc un corps vide pour toutes les invocations et méthodes afin de garantir que la classe est conforme au type.
- une classe vide *File*, sous-classe de *b_FileBufferDeTailleFixe*, est générée afin d'obtenir une uniformité du nommage. Ainsi, l'instanciation d'une classe *File* fournit bien un objet file synchronisé par le comportement *BufferDeTailleFixe*.

Code CAOLAC du tampon synchronisé de taille fixe

buffer.met Page 1/2

```
behaviour BufferDeTailleFixe {

  invocations:
    Get : REF TElement;
    Put( IN e:REF TElement );

  methods:
    IsFull : Boolean;
    IsEmpty : Boolean;

  initial state : Vide;

  state Vide sequential {
    TPut {
      invocation( Put(e) );
      BASE.Put(e);
      return;
      become( Partiel );
  } }

  state Partiel sequential {
    TPut {
      invocation( Put(e) );
      BASE.Put(e);
      return;
      if BASE.IsFull = true
      then become( Plein );
      else become( Partiel );
      end;
    }
    TGet {
      invocation( Get );
      return BASE.Get;
      if BASE.IsEmpty = true
      then become( Vide );
      else become( Partiel );
      end;
  } }

  state Plein sequential {
    TGet {
      invocation( Get );
      return BASE.Get;
      become( Partiel );
} } }

class File with behaviour BufferDeTailleFixe is

  const Max : REF TElement = 50;
  buf : Array [Max] OF REF TElement;
  ptrEntree, ptrSortie : REF TElement = 0;

  method Put( IN e:REF TElement );
  begin
    buf[ptrEntree] := e;
    ptrEntree := (ptrEntree+1) mod Max;
  end Put;

  method Get : REF TElement;
    e : REF TElement;
  begin
    e := but[ptrSortie];
    ptrSortie := (ptrSortie+1) mod Max;
    return e;
  end Get;

  method IsFull : Boolean;
  begin
    return (ptrEntree-ptrSortie) mod Max;
```

buffer.met Page 2/2

```
  end IsFull;

  method IsEmpty : Boolean;
  begin
    return ptrEntree=ptrSortie;
  end IsEmpty;

end File.
```

Traduction en GUIDE du tampon synchronisé de taille fixe

buffer.gui Page 1/3

```
TYPE File IS


  METHOD Init;
  METHOD Stop;

  METHOD Get : REF TElement;
  METHOD Put( IN e:REF TElement );
  METHOD IsFull : Boolean;
  METHOD IsEmpty : Boolean;


END File.

CLASS b_FileFile IMPLEMENTS File IS

  ATTRIBUTE UNMOVABLE;

  METHOD Init;
  BEGIN
  END Init;

  METHOD Stop;
  BEGIN
  END Stop;

  METHOD Get : REF TElement;
  BEGIN
  END Get;

  METHOD Put( IN e:REF TElement );
  BEGIN
  END Put;

  METHOD IsFull : Boolean;
  BEGIN
  END IsFull;

  METHOD IsEmpty : Boolean;
  BEGIN
  END IsEmpty;



END b_FileFile.

CLASS b_File SUBCLASS OF b_FileFile IMPLEMENTS File IS

  CONST s_BufferDeTailleFixeVide : Integer = 0;
  CONST s_BufferDeTailleFixePartiel : Integer = 1;
  CONST s_BufferDeTailleFixePlein : Integer = 2;
  CONST c_BufferDeTailleFixeNbStates : Integer = 3;

  c_BufferDeTailleFixeStates : REF c_Vect;


  CONST Max : REF TElement = 50;

  buf : Array[Max] OF REF TElement;

  ptrEntree,ptrSortie : REF TElement = 0;

  METHOD Put( IN e:REF TElement );
  BEGIN
    buf[ptrEntree] := e;
    ptrEntree := (ptrEntree+1) mod Max;
  END Put;

  METHOD Get : REF TElement;
```

buffer.gui Page 2/3

```
    e : REF TElement;
  BEGIN
    e := but[ptrSortie];
    ptrSortie := (ptrSortie+1) mod Max;
    RETURN e;
  END Get;

  METHOD IsFull : Boolean;
  BEGIN
    RETURN (ptrEntree-ptrSortie) mod Max;
  END IsFull;

  METHOD IsEmpty : Boolean;
  BEGIN
    RETURN ptrEntree=ptrSortie;
  END IsEmpty;

END b_File.

CLASS b_FileBufferDeTailleFixe SUBCLASS OF b_File IMPLEMENTS File IS

  ATTRIBUTE UNMOVABLE;

  METHOD Init;
  BEGIN
    c_BufferDeTailleFixeStates := c_VectStates[c_BufferDeTailleFixeNbStates].New
;
    c_BufferDeTailleFixeStates.Init;
    c_BufferDeTailleFixeStates.Plus( s_BufferDeTailleFixeVide );
    SUPER.Init;
  END Init;

  METHOD Stop;
  BEGIN
    c_BufferDeTailleFixeStates.Destroy;
    SUPER.Stop;
  END Stop;

  METHOD Get : REF TElement;
    c_ret : REF TElement;
  BEGIN
    WHILE true DO
      IF c_BufferDeTailleFixeStates.BelongAndRemove( s_BufferDeTailleFixePlein )
 = true THEN
          c_ret := SUPER.Get;
          c_BufferDeTailleFixeStates.Plus( s_BufferDeTailleFixePartiel );
          RETURN c_ret;
      END;
      IF c_BufferDeTailleFixeStates.BelongAndRemove( s_BufferDeTailleFixePartiel
 ) = true THEN
          c_ret := SUPER.Get;
          IF SUPER.IsEmpty=true THEN
            c_BufferDeTailleFixeStates.Plus( s_BufferDeTailleFixeVide );
            RETURN c_ret;
          ELSE
            c_BufferDeTailleFixeStates.Plus( s_BufferDeTailleFixePartiel );
            RETURN c_ret;
          END;
      END;
    END;
  END Get;

  METHOD Put( IN e:REF TElement );
  BEGIN
    WHILE true DO
      IF c_BufferDeTailleFixeStates.BelongAndRemove( s_BufferDeTailleFixePartiel
 ) = true THEN
          SUPER.Put(e);
          IF SUPER.IsFull=true THEN
```

buffer.gui Page 3/3

```
            c_BufferDeTailleFixeStates.Plus( s_BufferDeTailleFixePlein );
            RETURN;
          ELSE
            c_BufferDeTailleFixeStates.Plus( s_BufferDeTailleFixePartiel );
            RETURN;
          END;
      END;
      IF c_BufferDeTailleFixeStates.BelongAndRemove( s_BufferDeTailleFixeVide )
= true THEN
          SUPER.Put(e);
          c_BufferDeTailleFixeStates.Plus( s_BufferDeTailleFixePartiel );
          RETURN;
      END;
    END;
  END Put;

END b_FileBufferDeTailleFixe.

CLASS File SUBCLASS OF b_FileBufferDeTailleFixe IMPLEMENTS File IS

  ATTRIBUTE UNMOVABLE;

END File.
```

A.2.2 Traduction des sémantiques d'états

Nous illustrons la traduction des sémantiques d'état à l'aide d'un programme de test fourni page A.2.2.2. Le résultat de la traduction de ce programme est fourni aux pages A.2.2.2 à A.2.2.2.

A.2.2.1 Sémantique sortante

Séquentielle

La désactivation d'un état à sémantique sortante séquentielle se fait par appel de la méthode *BelongAndRemove* de la classe *c_VectStates* (Cf. la désactivation de l'état s_1 au début de l'invocation Go_1 page 237).

Parallèle

Lorsque la sémantique sortante est de type *parallèle*, on utilise en plus une variable vecteur de transitions (*c_VectTransitions*) de dimension égale au nombre de transitions du comportement [4]. Cette variable (ici *c_TestTransitionsOut* page 236) est un vecteur d'entiers. Chaque élément est associé à une transition et représente le nombre d'instances de cette transition qui peuvent être exécutées.

A chaque activation d'un état à sémantique sortante *parallèle*, les éléments correspondant aux transitions sortantes sont augmentés de 1. Par exemple, l'activation de l'état s_2 à la fin de l'invocation Go_1 page 237, se fait en activant l'état s_2 (*c_TestStates.Plus(s_Tests2)*) et en incrémentant les éléments correspondant aux transitions $s_2.t_2$ et $s_2.t_3$ (*c_TestTransitionsOut.Plus(t_Tests2t_Testt2)* et *c_TestTransitionsOut.Plus(t_Tests2t_Testt3)*).

Chaque fois qu'une transition d'un état à sémantique sortante *parallèle* est déclenché, son élément correspondant dans le vecteur des transitions est décrémenté. Cela se fait à l'aide de l'appel de la méthode *c_TestBelongAndRemove* page 237 (pour l'appel, Cf. par exemple la prise en compte de la transition $s_2.t_2$ au début de l'invocation Go_2 page 237). Cette méthode désactive l'état lorsque les éléments correspondant aux transitions sortant de l'état (ici $s_2.t_2$ et $s_2.t_3$) valent 0.

Tant que

Pour les états à sémantique sortante *tant que*, une méthode supplémentaire *UpdateServerStates* est générée dans la classe correspondant au comportement (ici *b_monTest* page 237). Elle évalue les conditions d'activation de tous les états du comportement qui ont une telle sémantique. Elle est appelée à chaque fin de transition.

A.2.2.2 Sémantique entrante

Nulle

L'activation d'un état à sémantique entrante *nulle* se fait par appel de la méthode *Plus* de la classe *c_VectStates* (Cf. l'activation de l'état s_1 au début de l'invocation Go_4 page 236 : *c_TestStates.Plus(s_Tests1)*).

4. On pourrait se limiter au nombre de transitions sortantes, mais cela impliquerait de gérer autant de variables qu'il y a d'états à sémantique sortante parallèle.

Rendez-vous

Lorsque la sémantique entrante est de type *rendez-vous*, on utilise en plus une variable vecteur de transitions (*c_VectTransitions*) de dimension égale au nombre de transitions du comportement. Cette variable (ici *c_TestTransitionsIn* page 236) est un vecteur d'entiers. Comme pour le vecteur *c_TestTransitionsOut* utilisé par la sémantique sortante *parallèle*, chaque élément est associé à une transition. Dans ce cas, l'élément représente le nombre d'instances de cette transition qui ont été exécutées avant que l'état conséquent ne soit activé.

Chaque fois qu'une transition d'un état à sémantique entrante *rendez-vous* est terminée, son élément correspondant dans le vecteur des transitions est incrémenté. Cela se fait à l'aide de l'appel de la méthode *c_TestPlusJoin* page 237 (pour l'appel, Cf. par exemple la fin de la transition $s_2.t_3$ à la fin de l'invocation Go_3 page 237). Cette méthode active l'état lorsque tous les éléments correspondant aux transitions entrant dans l'état (ici $s_2.t_2$ et $s_2.t_3$) sont différents de 0.

Code CAOLAC d'un programme de test des sémantiques d'état

test.met Page 1/1

```
behaviour Test {

  variables:
    var : Integer;

  invocations: Go1; Go2; Go3; Go4;

  initial state : s1;

  state s1 sequential {
    t1 {
      invocation( Go1 );
      require( var=10 );
      output.WriteString( "Transition Test::s1.t1\n" );
      return;
      become( s2 );
  } }

  state s2 parallel {
    t2 {
      invocation( Go2 );
      output.WriteString( "Transition Test::s2.t2\n" );
      return;
      become( s3 );
    }
    t3 {
      invocation( Go3 );
      output.WriteString( "Transition Test::s2.t3\n" );
      return;
      become( s3 );
  } }

  state s3 join, server while (true) {
    t4 {
      invocation( Go4 );
      output.WriteString( "Transition Test::s3.t4\n" );
      return;
      become( s1 );
} } }

class monTest with behaviour Test is

  method Go1;
  begin
    output.WriteString( "Invocation Go1\n" );
  end Go1;

  method Go2;
  begin
    output.WriteString( "Invocation Go2\n" );
  end Go2;

  method Go3;
  begin
    output.WriteString( "Invocation Go3\n" );
  end Go3;

  method Go4;
  begin
    output.WriteString( "Invocation Go4\n" );
  end Go4;

end monTest.
```

Traduction en GUIDE du programme de test des sémantiques d'état

test.gui Page 1/4

```
TYPE monTest IS


  METHOD Init;
  METHOD Stop;

  METHOD Go1;
  METHOD Go2;
  METHOD Go3;
  METHOD Go4;

  METHOD c_TestPlusJoin( IN ind,tran : Integer );
  METHOD c_TestBelongAndRemove( IN ind,tran : Integer ) : Boolean;
  METHOD c_TestUpdateServerStates;

END monTest.

CLASS b_monTestmonTest IMPLEMENTS monTest IS

  ATTRIBUTE UNMOVABLE;

  METHOD Init;
  BEGIN
  END Init;

  METHOD Stop;
  BEGIN
  END Stop;

  METHOD Go1;
  BEGIN
  END Go1;

  METHOD Go2;
  BEGIN
  END Go2;

  METHOD Go3;
  BEGIN
  END Go3;

  METHOD Go4;
  BEGIN
  END Go4;


  METHOD c_TestPlusJoin( IN ind,tran : Integer );
  BEGIN
  END c_TestPlusJoin;

  METHOD c_TestBelongAndRemove( IN ind,tran : Integer ) : Boolean;
  BEGIN
  END c_TestBelongAndRemove;

  METHOD c_TestUpdateServerStates;
  BEGIN
  END c_TestUpdateServerStates;


END b_monTestmonTest.

CLASS b_monTest SUBCLASS OF b_monTestmonTest IMPLEMENTS monTest IS

  CONST s_Tests1 : Integer = 0;
  CONST s_Tests2 : Integer = 1;
  CONST s_Tests3 : Integer = 2;
  CONST c_TestNbStates : Integer = 3;

  CONST t_Tests1t_Testt1 : Integer = 0;
```

test.gui Page 2/4

```
  CONST t_Tests2t_Testt2 : Integer = 1;
  CONST t_Tests2t_Testt3 : Integer = 2;
  CONST t_Tests3t_Testt4 : Integer = 3;
  CONST c_TestNbTransitions : Integer = 4;

  c_TestStates, c_TestTransitionsIn, c_TestTransitionsOut : REF c_Vect;

  var : Integer;

  METHOD Go1;
  BEGIN
    output.WriteString("Invocation Go1\n");
  END Go1;

  METHOD Go2;
  BEGIN
    output.WriteString("Invocation Go2\n");
  END Go2;

  METHOD Go3;
  BEGIN
    output.WriteString("Invocation Go3\n");
  END Go3;

  METHOD Go4;
  BEGIN
    output.WriteString("Invocation Go4\n");
  END Go4;

END b_monTest.

CLASS b_monTestTest SUBCLASS OF b_monTest IMPLEMENTS monTest IS

  ATTRIBUTE UNMOVABLE;

  METHOD Init;
  BEGIN
    c_TestStates := c_VectStates[c_TestNbStates].New;
    c_TestStates.Init;
    c_TestStates.Plus( s_Tests1 );
    c_TestTransitionsIn := c_VectTransitions[c_TestNbTransitions].New;
    c_TestTransitionsIn.Init;
    c_TestTransitionsOut := c_VectTransitions[c_TestNbTransitions].New;
    c_TestTransitionsOut.Init;
    SUPER.Init;
  END Init;

  METHOD Stop;
  BEGIN
    c_TestStates.Destroy;
    c_TestTransitionsIn.Destroy;
    c_TestTransitionsOut.Destroy;
    SUPER.Stop;
  END Stop;

  METHOD Go4;
  BEGIN
    WHILE true DO
      IF c_TestStates.CurrentState( s_Tests3 ) = true THEN
          output.WriteString("Transition Test::s3.t4\n");
          c_TestStates.Plus( s_Tests1 );
          SELF.c_TestUpdateServerStates;
          RETURN;
      END;
    END;
  END Go4;

  METHOD Go3;
  BEGIN
```

test.gui Page 3/4

```
    WHILE true DO
      IF SELF.c_TestBelongAndRemove( s_Tests2 , t_Tests2t_Testt3 ) = true THEN
          output.WriteString("Transition Test::s2.t3\n");
          SELF.c_TestPlusJoin( s_Tests3 , t_Tests2t_Testt3 );
          SELF.c_TestUpdateServerStates;
          RETURN;
      END;
    END;
  END Go3;

  METHOD Go2;
  BEGIN
    WHILE true DO
      IF SELF.c_TestBelongAndRemove( s_Tests2 , t_Tests2t_Testt2 ) = true THEN
          output.WriteString("Transition Test::s2.t2\n");
          SELF.c_TestPlusJoin( s_Tests3 , t_Tests2t_Testt2 );
          SELF.c_TestUpdateServerStates;
          RETURN;
      END;
    END;
  END Go2;

  METHOD Go1;
  BEGIN
    WHILE true DO
      IF c_TestStates.BelongAndRemove( s_Tests1 ) = true THEN
        IF var=10 THEN
          output.WriteString("Transition Test::s1.t1\n");
          c_TestStates.Plus( s_Tests2 );
          c_TestTransitionsOut.Plus( t_Tests2t_Testt2 );
          c_TestTransitionsOut.Plus( t_Tests2t_Testt3 );
          SELF.c_TestUpdateServerStates;
          RETURN;
        END;
        c_TestStates.Plus( s_Tests1 );
      END;
    END;
  END Go1;

  METHOD c_TestPlusJoin( IN ind,tran : Integer );
  BEGIN
    c_TestTransitionsIn.Plus(tran);
    IF ind=s_Tests3 AND
       c_TestTransitionsIn.CurrentState(t_Tests2t_Testt3)=true AND
       c_TestTransitionsIn.CurrentState(t_Tests2t_Testt2)=true THEN
      c_TestStates.Plus( s_Tests3 );
      c_TestTransitionsIn.Minus(t_Tests2t_Testt3);
      c_TestTransitionsIn.Minus(t_Tests2t_Testt2);
    END;
  END c_TestPlusJoin;

  METHOD c_TestBelongAndRemove( IN ind,tran : Integer ) : Boolean;
  BEGIN
    IF c_TestTransitionsOut.BelongAndRemove(tran) = false
    THEN RETURN false; END;
    IF ind=s_Tests2 AND (
       c_TestTransitionsOut.CurrentState(t_Tests2t_Testt2)=false OR
       c_TestTransitionsOut.CurrentState(t_Tests2t_Testt3)=false)
    THEN c_TestStates.Minus(ind); END;
    RETURN true;
  END c_TestBelongAndRemove;

  METHOD c_TestUpdateServerStates;
  BEGIN
    IF c_TestStates.CurrentState(s_Tests3) = true THEN
      IF not(true) THEN
        c_TestStates.Minus(s_Tests3);
      END;
    END;
```

test.gui Page 4/4

```
  END c_TestUpdateServerStates;

  CONTROL
    EXCLUSIVE( c_TestPlusJoin , c_TestBelongAndRemove );

END b_monTestTest.

CLASS monTest SUBCLASS OF b_monTestTest IMPLEMENTS monTest IS

  ATTRIBUTE UNMOVABLE;

END monTest.
```

A.2.3 Classes auxiliaires pour la traduction CAOLAC vers GUIDE

Le résultat de la traduction d'un programme CAOLAC en GUIDE utilise deux classes génériques *c_VectStates* et *c_VectTransitions*. La première gère un vecteur de booléens pour l'activation et la désactivation d'états. La seconde gère un vecteur d'entiers pour les transitions.

cocar.gui Page 1/2

```
TYPE c_Vect IS
  METHOD Init;
  METHOD Plus( IN Integer ); SIGNALS OutOfLimit;
  METHOD Minus( IN Integer ); SIGNALS OutOfLimit;
  METHOD CurrentState( IN Integer ) : Boolean; SIGNALS OutOfLimit;
  METHOD BelongAndRemove( IN Integer ) : Boolean; SIGNALS OutOfLimit;
END c_Vect.

CLASS CONSTRUCTOR c_VectStates [size] IMPLEMENTS c_Vect IS

  ATTRIBUTE UNMOVABLE;

  Tab : Array [size] OF Boolean;

  METHOD Init;
    i : Integer;
  BEGIN
    FOR i := 0 TO size-1 DO
      Tab[i] := false;
    END;
  END Init;

  METHOD Plus( IN ind:Integer ); SIGNALS OutOfLimit;
  BEGIN
    IF ind<0 OR ind>=size THEN RAISE OutOfLimit; END;
    Tab[ind] := true;
  END Plus;

  METHOD Minus( IN ind:Integer ); SIGNALS OutOfLimit;
  BEGIN
    IF ind<0 OR ind>=size THEN RAISE OutOfLimit; END;
    Tab[ind] := false;
  END Minus;

  METHOD CurrentState( IN ind:Integer ) : Boolean; SIGNALS OutOfLimit;
  BEGIN
    IF ind<0 OR ind>=size THEN RAISE OutOfLimit; END;
    RETURN Tab[ind];
  END CurrentState;

  METHOD BelongAndRemove( IN ind:Integer ) : Boolean; SIGNALS OutOfLimit;
  BEGIN
    IF ind<0 OR ind>=size THEN RAISE OutOfLimit; END;
    IF Tab[ind] = false THEN RETURN false;
    ELSE Tab[ind] := false; RETURN true;
    END;
  END BelongAndRemove;

  CONTROL
    EXCLUSIVE( Init , Plus , Minus , CurrentState , BelongAndRemove );

END c_VectStates.

CLASS CONSTRUCTOR c_VectTransitions [size] IMPLEMENTS c_Vect IS

  ATTRIBUTE UNMOVABLE;

  Tab : Array [size] OF Integer;

  METHOD Init;
    i : Integer;
  BEGIN
    FOR i := 0 TO size-1 DO
      Tab[i] := 0;
    END;
  END Init;

  METHOD Plus( IN ind:Integer ); SIGNALS OutOfLimit;
  BEGIN
```

cocar.gui Page 2/2

```
    IF ind<0 OR ind>=size THEN RAISE OutOfLimit; END;
    Tab[ind] := Tab[ind]+1;
  END Plus;

  METHOD Minus( IN ind:Integer ); SIGNALS OutOfLimit;
  BEGIN
    IF ind<0 OR ind>=size THEN RAISE OutOfLimit; END;
    Tab[ind] := Tab[ind]-1;
  END Minus;

  METHOD CurrentState( IN ind:Integer ) : Boolean; SIGNALS OutOfLimit;
  BEGIN
    IF ind<0 OR ind>=size THEN RAISE OutOfLimit; END;
    RETURN (Tab[ind] # 0);
  END CurrentState;

  METHOD BelongAndRemove( IN ind:Integer ) : Boolean; SIGNALS OutOfLimit;
  BEGIN
    IF ind<0 OR ind>=size THEN RAISE OutOfLimit; END;
    IF Tab[ind] = 0 THEN RETURN false;
    ELSE Tab[ind] := Tab[ind]-1; RETURN true;
    END;
  END BelongAndRemove;

  CONTROL
    EXCLUSIVE( Init , Plus , Minus , CurrentState , BelongAndRemove );

END c_VectTransitions.
```

Bibliographie

Bibliographie

[Abr96] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[AEdC90] E. Audureau, P. Enjalbert, and L. Farinas del Cerro. *Logique Temporelle: Sémantique et Validation de Programmes Parallèles*. Masson, 1990.

[AG94a] R. Allen and D. Garlan. Formal connectors. Technical Report CMU-CS-94-115, Carnegie Mellon University, 1994.

[AG94b] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of ICSE'94*, 1994.

[AGMB91] C. Atkinson, S. Goldsack, A. Di Maio, and R. Bayan. Object-oriented concurrency and distribution in DRAGOON. *Journal of Object-Oriented Programming*, Mars 1991.

[AH86] G. Agha and C. Hewitt. *A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[AH89] J.F. Allen and P.J. Hayes. Moments and points in an interval-based temporal logic. *Computational Intelligence*, 5(4):225–238, 1989.

[AIS+77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977.

[Ame87] P. America. POOL-T: A parallel object-oriented language. In *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[Amy91] O. Amyay. *Méthodologie de Spécification d'Activités de Communication dans une Architecture Multi-Couches vers la Définition d'une Base de Connaissances*. PhD thesis, Laboratoire LAAS, Université de Toulouse, Décembre 1991.

[Ara92] C. Arapis. *Dynamic Evolution of Object Behaviour and Object Cooperation*. PhD thesis, Centre Universitaire d'Informatique, Université de Genève, Février 1992.

[Ara95] C. Arapis. A temporal perspective of composite objects. In *Object-Oriented Software Composition*, pages 123–152. Prentice Hall, 1995.

[Atk91] C. Atkinson. *Object-Oriented Reuse, Concurrency and Distribution*. Addison-Wesley, 1991.

[AvdL90] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (ECOOP/OOPSLA'90)*, volume 25 of *SIGPLAN Notices*, pages 161–168. ACM Press, Octobre 1990.

[Bar89] J. Barnes. *Programming in Ada*. Addison-Wesley, 1989.

[Bar95] J. Barnes. *Programming in Ada*. Addison-Wesley, 2nd edition, 1995.

[BBD+91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, P. Le Dot, M. Meysembourg, H. Nguyen, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, and G. Vandome. Architecture and implementation of GUIDE, an object-oriented distributed system. *Computing Systems*, 4(1):31–67, 1991.

[BCSL94] U. Bellur, G. Craig, K. Shank, and D. Lea. Clustering: Composition for active object systems. In *Proceedings of 27th Hawaii International Conference on System Sciences*, Janvier 1994.

[BDB+88] D.G. Bobrow, L.G. DeMichiel, R.P. Babriel, S. Keene, G. Kiczales, and D.A. Moon. Common lisp object system specification. *ACM SIGPLAN Notices*, 23, Septembre 1988.

[BDFS95] L. Bonnet, L. Duchien, G. Florin, and L. Seinturier. A method for specifying and proving distributed cooperative algorithms. In *Proceedings of the Distributed Intelligent and Multi-Agent Systems Workshop (DIMAS'95)*, pages 66–73, Novembre 1995.

[BDFS96] L. Bonnet, L. Duchien, G. Florin, and L. Seinturier. Spanning tree object-oriented distributed algorithm: Specification and proof. In *Proceedings of the Method Integration Workshop (MIW'96)*, Mars 1996.
`http://www.springer.co.uk/ewic/workshops/MI96/`.

[BDFS97] L. Bonnet, L. Duchien, G. Florin, and L. Seinturier. Some specification steps of a spanning tree algorithm with an object-oriented approach. In *Proceedings of the 1st IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS'96)*, pages 115–131. Chapman & Hall, 1997. Extended version of [BDFS95].

[BDMN73] G. Birtwistle, O. Dahl, K. Myhraug, and K. Nygaard. *Simula begin*. Petrocelli Charter, 1973.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BHJ+87] A.P. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Computer Systems*, 13(1):65–76, 1987.

[BLR94] R. Balter, S. Lacourte, and M. Riveill. The GUIDE language: Design and experience. *Computer Journal*, 37:519–530, 1994.

[BN84] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Février 1984.

[Bok96] B. Bokowski. Interaction protocol for composing concurrent objects. In *Workshop for Doctoral Students at the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, Juillet 1996.

[Bon94] L. Bonnet. Spécification et preuve d'algorithmes distribués en approche orientée objet. Mémoire d'ingénieur CNAM-CEDRIC, Décembre 1994.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Company, 2ème edition, 1994.

[BRJ97] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Method for Object-Oriented Development.* Rational Software Corporation, 1997. Version 1.0.

[BW95] A. Burns and A. Wellings. *Concurrency in Ada.* Cambridge University Press, 1995.

[Car94] D. Caromel. Programmation parallèle à objets : Eiffel//. *Journal des Calculateurs Parallèles*, 22:51–85, 1994.

[CDFS96] C. Coste, L. Duchien, G. Florin, and L. Seinturier. Différentes approches des relations d'ordre dans un environnement coopératif. In *Actes des Journées de Recherche sur le Contrôle Réparti dans les Applications Coopératives (CRAC'96)*, pages 17–22, Mai 1996.

[CH73] R.H. Campbell and A.N. Habermann. The specification of process synchronization by path expressions. volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer-Verlag, 1973.

[CHC90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 125–135. ACM Press, Janvier 1990.

[Che76] P.P. Chen. The entity-relationship model: Towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[Che80] B.F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, 1980.

[Chi95] S. Chiba. A metaobject protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'95)*, volume 30 of *SIGPLAN Notices*, pages 285–299. ACM Press, Octobre 1995.

[CI 97] CI Labs. *OpenDoc Documentation 1.2*, 1997.

[CK80] R.H. Campbell and R.B. Kolstad. An overview of Path Pascal's design and the Path Pascal user manual. *ACM SIGPLAN Notices*, 15(9):13–24, Septembre 1980.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Février 1985.

[Cor97] T. Cornilleau. *Mémoire Répartie Causale.* PhD thesis, CNAM-CEDRIC, Janvier 1997.

[Cos95] C. Coste. Appel de procédure à distance sur groupe en approche objet réparti. Mémoire d'ingénieur CNAM-CEDRIC, Décembre 1995.

[Cox86] B.J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1986.

[CT91] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC'91)*, pages 325–340, Août 1991.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Des94] P. Desfray. *Object Engineering.* Addison-Wesley, 1994.

[DFGS96] L. Duchien, G. Florin, R. Gomez, and L. Seinturier. The wave algorithms as a self-stabilizing control structure. Technical Report 9614, CNAM-CEDRIC, Septembre 1996.

[Dij65] E.W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, 1965. Reprinted in [Gen68].

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Août 1975.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DoD80] DoD. *Ada Reference Manual*, Juillet 1980.

[EH85] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1–24, 1985.

[EH86] E.A. Emerson and J.Y. Halpern. *Sometimes* and *not never* revisited. *Journal of the ACM*, 33(1):151–178, 1986.

[Eme90] E.A. Emerson. *Temporal and Modal Logic.* Elsevier Science Publishers, 1990.

[FGL93] G. Florin, R. Gomez, and I. Lavallée. Recursive distributed programming schemes. In *Proceedings of the International Symposium on Autonomous Decentralized Systems (ISADS'93)*, Avril 1993.

[FH94] P. Fauvel and F. Horn. Présentation d'ODP. Technical Report NT/PAA/TSA/TLR/3837, CNET, 1994.

[FHMV95] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. *Reasoning about Knowledge.* MIT Press, 1995.

[FHMV96] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. Commom knowledge revisited. In *Proceedings of the 6th Conference on Theoretical Aspects of Rationality and Knowledge (TARK'96)*, Mars 1996.

[Fid88] C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computing Conference*, Février 1988.

[Flo95] G. Florin. Algorithme d'election sur un anneau par vagues contra-rotatives. Communication Personnelle, 1995.

[Frø92] S. Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92)*, volume 615 of *Lecture Notes in Computer Science*, pages 185–196. Springer-Verlag, Juin 1992.

[Fro96] E. Fromentin. *Détection de Propriétés Instables dans les Exécutions Réparties.* PhD thesis, Université de Rennes, 1996.

[FZ90] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 134–141, Mai 1990.

[Gal90] A. Galton. *Logic for Information Technology.* John Wiley and Sons Ltd, 1990.

[Gen68] F. Genyus. *Programming Languages.* Academic Press, 1968.

[Ger77] A.J. Gerber. Synchronisation by counter variables. *Operating System Review*, 11(4):6–17, Octobre 1977.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GK97a] M. Golm and J. Kleinöder. MetaJava: A platform for adaptable operating-system mechanisms. Technical Report TR-I4-97-10, Computer Science Department, Friedrich Alexander University, Erlangen-Nürnberg, Germany, Avril 1997.

[GK97b] M. Golm and J. Kleinöder. MetaJava: A platform for adaptable operating-system mechanisms. In *Workshop on Object-Orientation and Operating Systems at the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Juin 1997.

[GM95] J. Gosling and H. McGilton. *The Java Language Environment: A White Paper.* Sun Microsystems Computer Company, Mai 1995.
`http://www.javasoft.com`.

[Gom95] R. Gomez. *Conception et Spécification d'Algorithmes Distribués : la Vague Récursive.* Thèse de doctorat, Université Paris VIII, Avril 1995.

[GR83] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its Implementation.* Addison-Wesley, 1983.

[Gra93] I. Graham. *Object-Oriented Methods.* Addison-Wesley, 2nd edition, 1993.

[Gue95] R. Guerraoui. Les langages concurrents à objets. *Techniques et Sciences Informatiques*, 14(8):945–972, Octobre 1995.

[GW91] W. Gerteis and W. Wirz. Synchronizing objects by conditional path expressions. In *Proceedings of the 6th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 6)*, pages 193–201, 1991.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, Mai 1988.

[HC68] G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic*. Methuen, 1968.

[HC84] G.E. Hughes and M.J. Cresswell. *A Companion to Modal Logic*. Methuen, 1984.

[HF89] J.Y. Halpern and R. Fagin. Modelling knowledge and action in distributed systems. *Distributed Computing*, 3(4):159–179, 1989.

[HG96] D. Harel and E. Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, pages 246–257. IEEE Computer Society Press, Mars 1996.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, Octobre 1974.

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

[ISO87a] ISO 8807. *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1987.

[ISO87b] ISO 9074. *Estelle, A Formal Description Technique Based on an Extended State Transition Model*, 1987.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[Jon86] C.B. Jones. *Systematic Software Devolpment Using VDM*. Prentice-Hall, 1986.

[JP96] J.M. Jezequel and J.L. Pacherie. Parallel operators. In *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 275–294. Springer-Verlag, Juillet 1996.

[KdRB91] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KL86] S. Kraus and D. Lehmann. Knowledge, belief and time. In *Proceedings of the 13th International Colloquim on Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[KL89a] D.G. Kafura and K.H. Lee. Inheritance in actor based concurrent object-oriented languages. In *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, pages 131–145. Cambridge University Press, Juillet 1989.

[KL89b] D.G. Kafura and K.H. Lee. Inheritance in actor based concurrent object-oriented languages. *The Computer Journal*, 32(4):297–304, 1989.

[KM94] G. Knight and J. Ma. Time representation: A taxonomy of temporal models. *Artificial Intelligence Review*, 7:401–419, 1994.

[Kri63] S. Kripke. Semantic analysis of modal logic. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[LA88] R.J. Leblanc and W.F. Appelbe. The Clouds distributed operationg system. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 2–9, Juin 1988.

[Lal91] W. Lalonde. Subclassing ≠ subtyping ≠ is-a. *Journal of Object-Oriented Programming*, 3(5):11–22, Mars 1991.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Juillet 1978.

[Lam90] L. Lamport. *Sometime* is sometime *not never*. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 174–185. ACM Press, Janvier 1990.

[Lam91] L. Lamport. The temporal logic of actions. Technical Report 79, Digital Systems Research Center, Décembre 1991.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, pages 872–923, Mai 1994.

[Lev84] H.J. Levesque. A logic of implicit and explicit belief. In *Proceedings of the National Conference on Artificial Intelligence*, pages 198–202, 1984.

[Lew12] C.I. Lewis. Implication and the algebra of logic. *Mind*, 21:522–523, 1912.

[Lie87] H. Lieberman. Concurrent object-oriented programming in Act/1. In *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[LJP93] R. Lea, C. Jacquemot, and E. Pillevesse. Cool: System support for distributed object-oriented programming. *Communications of the ACM*, 36(9), Septembre 1993.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proceedings of the Worshop on Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer-Verlag, 1985.

[LS76] B. Lampson and A. Sturgis. Crash recovery in a distributed data storage system. Technical report, Computer Science Laboratory, Xerox Palo Alto Research Center, 1976.

[LS79] B. Liskov and A. Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, 5(6), 1979.

[LS83] B. Liskov and R. Sheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3), 1983.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[Maf96] S. Maffeis. The object group design pattern. In *Proceedings of the USENIX Conference on Object-Oriented Technologies*, Juin 1996.

[Mat88] F. Mattern. Virtual time and global states in distributed systems. In *Proceedings of the International Conference on Parallel and Distributed Algorithms*, pages 215–226, 1988.

[McA95] J. McAffer. Meta-level programming with CodA. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 190–214. Springer-Verlag, Août 1995.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Mil80] R. Milner. A calculus of communicating systems. volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[MvdH95] J.J. Meyer and W. van der Hoek. *Epistemic Logic for AI and Computer Science*. Cambridge University Press, 1995.

[MvRT+90] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, Mai 1990.

[MWY91] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, pages 231–247. Springer-Verlag, Juillet 1991.

[MY90] S. Matsuoka and A. Yonezawa. Metalevel solution to concurrent object-oriented languages. In *Proceedings of the Workshop on Reflections and Metalevel Architectures in Object-Oriented Languages (OOPSLA'90)*. Springer-Verlag, Octobre 1990.

[MY93] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[Nie87] O. Nierstrasz. Active objects in Hybrid. In *Proceedings of the 2nd Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87)*, volume 22 of *SIGPLAN Notices*, pages 243–253. ACM Press, Décembre 1987.

[OHE96] R. Orfali, D. Harkey, and J. Edwards. *The Essentiel Distributed Object Survival Guide*. Wiley, 1996.

[OMG95] OMG. *Common Object Request Broker Architecture 2.0*, Juillet 1995. `http://www.omg.org`.

[PDFS95a] P. Placide, L. Duchien, G. Florin, and L. Seinturier. A consistent global state algorithm to debug distributed object-oriented applications. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, Mai 1995. Extended version of [PDFS95b].

[PDFS95b] P. Placide, L. Duchien, G. Florin, and L. Seinturier. Debugging of distributed object-oriented applications. In *Proceedings of the European Research Seminar on Advances in Distributed Systems (ERSADS'95)*, pages 200–207, Avril 1995.

[Pla94] P. Placide. Mise au point d'applications réparties, outil de visualisation. Mémoire d'ingénieur CNAM-CEDRIC, Septembre 1994.

[PS88] G.D. Parrington and S.K. Shrivastava. Implementing concurrency control in reliable distributed object-oriented systems. In *Proceedings of the 2nd European Conference on Object-Oriented Programming (ECOOP'88)*, volume 322 of *Lecture Notes in Computer Science*, pages 234–249. Springer-Verlag, 1988.

[PT92] P. Panangaden and S. Taylor. Concurrent common knowledge: Defining agreement for asynchronous systems. *Distributed Computing*, 6(2):73–94, 1992.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RV77] P. Robert and J.P. Verjus. Toward autonomous descriptions of synchronisation modules. In *Proceedings of the International Federation of Information Processing Congress (IFIP'77)*, pages 981–986, Août 1977.

[SB94] C. Sibertin-Blanc. Cooperative petri nets. In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*. Springer-Verlag, Juin 1994.

[SD96] L. Seinturier and L. Duchien. Group behavior patterns in an object-oriented methodology for distributed applications. In *Workshop on Methodologies for Distributed Objects at the 11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (MDO/OOPSLA'96)*, Octobre 1996. `http://albertina.inesc.pt/people/ars/mdo.html`.

[SDF97] L. Seinturier, L. Duchien, and G. Florin. A meta-object protocol for distributed OO applications. In *Proceedings of the 23rd Conference on Technologies of Object-Oriented Languages and Systems (TOOLS USA'97)*, Août 1997. Extended version of [Sei97b].

[Sei96] L. Seinturier. A language for distributed behavior design of cooperative applications. In *Workshop for Doctoral Students at the 10th European Conference on Object-Oriented Programming (DS/ECOOP'96)*, Juillet 1996. `http://www.ifs.uni-linz.ac.at/ecoop96/`.

[Sei97a] L. Seinturier. Manuel du langage CAOLAC. Technical Report 9710, CNAM-CEDRIC, Septembre 1997.
`http://cedric.cnam.fr/~seintur/tr9710.ps.gz`.

[Sei97b] L. Seinturier. Un langage de coordination pour les applications coopératives. In *Actes des 9ème Rencontres Francophones du Parallélisme (RenPar'9)*, pages 145–148, Mai 1997.

[SGH+89] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. SOS: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 4(2):287–338, Décembre 1989.

[Sie96] J. Siegel. *CORBA Fundamentals and Programming*. Wiley, 1996.

[Sin94] M.P. Singh. *Multiagent Systems*, volume 799 of *Lecture Notes in Computer Science*. Springer-Verlag, Septembre 1994.

[Ske82] D. Skeen. Non blocking commit protocols. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 133–147, Juin 1982.

[SM92] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. Technical Report SFB 124 - 15/92, University of Kaiserslautern, Décembre 1992.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'86)*, volume 21 of *SIGPLAN Notices*, pages 38–45. ACM Press, Novembre 1986.

[Sou86] B. Soustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Tel94] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[TS89] C. Tolimson and V. Singh. Inheritance and synchronization with enabled sets. In *Proceedings of the 4th Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *SIGPLAN Notices*. ACM Press, Octobre 1989.

[UIT95] UIT-T X.901, ISO 10746. *RM-ODP*, 1995.
`http://www.dstc.edu.au/AU/research_news/odp/ref_model/standards.html`.

[vdBL89] J. van den Bos and C. Laffra. PROCOL: A parallel object language with protocols. In *Proceedings of the 4th Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *SIGPLAN Notices*, pages 95–102. ACM Press, Octobre 1989.

[vW51] G.H. von Wright. *An Essay in Modal Logic*. North-Holland Publishing Company, 1951.

[vW80] G.H. von Wright. Problems and prospects of deontic logic: A survey. In *Modern Logic - A Survey: Historical, Philosophical and Mathematical Aspects of Modern Logic and its Applications*, pages 399–423. D. Reidel Publishing Company, 1980.

[Weg87] P. Wegner. Dimensions of object-based language design. In *Proceedings of the 2nd Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87)*, volume 22 of *SIGPLAN Notices*, pages 168–182. ACM Press, Décembre 1987.

[Wir82] N. Wirth. *Programming in Modula 2*. Springer-Verlag, 1982.

[WSMB95] Z. Wu, R.J. Stroud, K. Moody, and J. Bacon. The design and implementation of a distributed transaction system based on atomic data types. *Distributed System Engineering*, 2, Juillet 1995.

[WY88] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of the 3rd Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'88)*, volume 23 of *SIGPLAN Notices*. ACM Press, Septembre 1988.

[Yon90] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. Computer Systems Series. MIT Press, 1990.

[YSTH87] Y. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, 1987.

[YT87] Y. Yokote and M. Tokoro. Concurrent programming in concurrent SmallTalk. In *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, 1987.