Compilation et environnement d'exécution d'un langage à base d'objets

Hiep Nguyen Van

THESE

présentée par

NGUYEN VAN Hiep

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 23 novembre 1988)

Spécialité : **INFORMATIQUE**

COMPILATION et ENVIRONNEMENT D'EXECUTION D'UN LANGAGE A BASE D'OBJETS

Date de soutenance : 01 Fevrier 1991

Composition du jury :

Sacha KRAKOWIAK	Président
Daniel HERMAN	Rapporteur
Claude KAISER	Rapporteur
Jacques MOSSIERE	Directeur de thèse
Roland BALTER	Examinateur

Thèse préparée au sein du Laboratoire Bull-IMAG

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

46 avenue Felix Viallet
38031 GRENOBLE cedex

Tél. : 76.57.45.00

Année universitaire 1989

Président de l'Institut :
Monsieur Georges LESPINARD

Professeurs des Universités

BARIBAUD Michel	ENSERG	JAUSSAUD Pierre	ENSIEG
BARRAUD Alain	ENSIEG	JOST Rémy	ENSPG
BAUDELET Bernard	ENSPG	JOUBERT Jean-Claude	ENSPG
BEAUFILS Jean-Pierre	INPG	JOURDAIN Geneviève	ENSIEG
BLIMAN Samuel	ENSERG	LACOUME Jean-Louis	ENSIEG
BOIS Philippe	ENSHMG	LADET Pierre	ENSIEG
BONNETAIN Lucien	ENSEEG	LESIEUR Marcel	ENSHMG
BONNET Guy	ENSPG	LESPINARD Georges	ENSHMG
BRISSONNEAU Pierre	ENSIEG	LONGEQUEUE Jean-Pierre	ENSPG
BRUNET Yves	IUFA	LORET Benjamin	ENSHMG
CAILLERIE Denis	ENSHMG	LOUCHET François	ENSEEG
CAVAIGNAC Jean-François	ENSPG	LUCAZEAU Guy	ENSEEG
CHARTIER Germain	ENSPG	MASSE Philippe	ENSIEG
CHENEVIER Pierre	ENSERG	MASSELOT Christian	ENSIEG
CHERADAME Hervé	UFR PGP	MAZARE Guy	ENSIMAG
CHERUY Arlette	ENSIEG	MOHR Roger	ENSIMAG
CHOVET Alain	ENSERG	MOREAU René	ENSHMG
COHEN Joseph	ENSERG	MORET Roger	ENSIEG
COLINET Catherine	ENSEEG	MOSSIERE Jacques	ENSIMAG
CORNUT Bruno	ENSIEG	OBLED Charles	ENSHMG
COULOMB Jean-Louis	ENSIEG	OZIL Patrick	ENSEEG
COUMES André	ENSERG	PA ULEAU Yves	ENSEEG
CROWLEY James	ENSIMAG	PERRET Robert	ENSIEG
DARVE Félix	ENSHMG	PIAU Jean-Michel	ENSHMG
DELLA-DORA Jean	ENSIMAG	PIC Etienne	ENSERG
DEPEY Maurice	ENSERG	PLATEAU Brigitte	ENSIMAG
DEPORTES Jacques	ENSPG	POUPOT Christian	ENSERG
DEROO Daniel	ENSEEG	RAMEAU Jean-Jacques	ENSEEG
DESRE Pierre	ENSEEG	REINISCH Raymond	ENSPG
DOLMAZON Jean-Marc	ENSERG	RENAUD Maurice	UFR PGP
DURAND Francis	ENSEEG	ROBERT André	UFR PGP
DURAND Jean-Louis	ENSPG	ROBERT François	ENSIMAG
FAUTRELLE Yves	ENSHMG	SABONNADIERE Jean-Claude	ENSIEG
FOGGIA Albert	ENSIEG	SAUCIER Gabrièle	ENSIMAG
FONLUPTJean	ENSIMAG	SCHLENKER Claire	ENSPG
FOULARD Claude	ENSIEG	SCHLENKER Michel	ENSPG
GANDINI Alessandro	UFR PGP	SERMET Pierre	ENSERG
GAUBERT Claude	ENSPG	SILVY Jacques	UFR PGP
GENTIL Pierre	ENSERG	SIRIEYS Pierre	ENSHMG
GENTIL Sylviane	ENSIEG	SOHM Jean-Claude	ENSEEG
GREVEN Hélène	IUFA	SOLER Jean-Louis	ENSIMAG
GUEGUEN Claude	ENSIEG	SOUQUET Jean-Louis	ENSEEG
GUERIN Bernard	ENSERG	TROMPETTE Philippe	ENSHMG
GUYOT Pierre	ENSEEG	VINCENT Henri	ENSPG
IVANES Marcel	ENSIEG	ZADWORNY François	ENSERG

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
COURNIL M.
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LACHENAL D.
LADET Pierre
LATOMBE Claudine
LE HUY H.
LE GORREC Bernard
MADAR Roland
MEUNIER G.
MULLER Jean
NGUYEN TRONG Bernadette
NIEZ J.J.
PASTUREL Alain
PLA Fernand
ROGNON J.P.
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri
YAVARI A.R.

Chercheurs du C.N.R.S

DIRECTEURS DE RECHERCHE CLASSE 0

LANDEAU	Ioan
NAYROLLES	Bernard

Directeurs de recherche 1ère Classe

ANSARA Ibrahim
CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
KRAKOWIAK Sacha
LEPROVOST Christian
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ARMAND Michel
AUDIER Marc
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
CHATILLON Christiant
CLERMONT Jean-Robert
COURTOIS Bernard
DAVID René
DION Jean-Michel
DRIOLE Jean
DURAND Robert
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GARNIER Marcel
GUELIN Pierre
JOUD Jean-Charles
KAMARINOS Georges
KLEITZ Michel
KOFMAN Walter
LEJEUNE Gérard
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MEUNIER Jacques
PEUZIN Jean-Claude
PIAU Monique
RENOUARD Dominique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
VENNEREAU Pierre
WACK Bernard
YONNET Jean-Paul

Personnalités agrées à titre permanent à diriger des travaux de recherche
(décision du conseil scientifique)

E.N.S.E.E.G
HAMMOU Abdelkader
MARTIN-GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G
BOREL Joseph

E.N.S.I.E.G
DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.M.G
ROWE Alain

E.N.S.I.M.A.G
COURTIN Jacques

C.E.N.G
CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIEB Maurice
VINCENDON Marc

Laboratoires extérieurs :

C.N.E.T
DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Situation particulière

PROFESSEURS D'UNIVERSITE

DETACHEMENT

ENSIMAG	LATOMBE	J..Claude	Détachement	21/10/1989
ENSHMG	PIERRARD	J.Marie	Détachement	30/04/1989
ENSIMAG	VEILLON	Gérard	Détachement	30/09/1990
ENSIMAG	VERJUS	J.Pierre	Détachement	30/09/1989
ENSPG	BLOCH	Daniel	Recteur à c/	21/12/1988

SURNOMBRE

INPG	CHIAVERINA	Jean	30/09/1989
ENSHMG	BOUVARD	Maurice	30/09/1991
ENSEEG	PARIAUD	J.Charles	30/09/1991

Je tiens à remercier

Monsieur **Sacha Krakowiak**, Professeur à l'Université Joseph Fourier de Grenoble et responsable du projet Guide, pour la grande confiance qu'il m'a accordée tout au long des quatre années pendant lesquelles j'ai travaillé à ce projet, pour l'aide précieuse qu'il m'a apportée, et pour m'avoir donné les moyens de faire mon travail. Je le remercie également pour l'honneur qu'il me fait en acceptant la présidence du jury de cette thèse.

Monsieur **Jacques Mossière**, Professeur à l'Institut National Polytechnique de Grenoble et Directeur de l'ENSIMAG, pour l'honneur qu'il m'a fait en acceptant d'être mon directeur de thèse, pour l'aide précieuse qu'il m'a apportée tout au long des années pendant lesquelles j'ai vécu loin de ma famille, et notamment lors de la conception et de la rédaction de cette thèse.

Monsieur **Daniel Herman**, Professeur à l'Université de Rennes I et Monsieur **Claude Kaiser**, Professeur au CNAM, qui ont accepté d'être les rapporteurs de mon travail. Je les remercie particulièrement pour l'intérêt qu'ils lui ont porté et pour l'évaluation qu'ils en ont faite.

Monsieur **Roland Balter**, Directeur du Centre de Recherche Bull de Grenoble, qui a accepté de faire partie du jury de cette thèse. Je le remercie également pour l'aide qu'il m'a apportée pendant mon travail de thèse.

Monsieur **Raymond Bouttaz**, Directeur de recherche à l'ENSIMAG, pour m'avoir accueilli en France et pour l'aide qu'il m'a apportée.

Michel Riveill, Cécile Roisin, Xavier Rousset de Pina, Marie Meysembourg et **Miguel Santana** pour m'avoir aidé tout au long des années pendant lesquelles j'ai travaillé, et pour les corrections de fond et de syntaxe de ma thèse. Je les remercie également pour les discussions très intéressantes que nous avons eues.

Je tiens à remercier également l'ensemble des membres du projet Guide : le travail que je présente dans cette thèse est en effet le résultat d'une recherche collective.

TABLE DES MATIERES

INTRODUCTION 1

PREMIERE PARTIE : ARCHITECTURE DU SYSTEME GUIDE 5

CHAPITRE 1 : ARCHITECTURE GENERALE DU SYSTEME GUIDE 7

1.1 OBJECTIFS GENERAUX DU PROJET 7

1.2 ARCHITECTURE DU SYSTEME GUIDE 9

1.2.1 Modèle d'objets 9

Désignation des objets 10

Sous-typage, Héritage, Généricité et réutilisation du code de programmes 11

1.2.2 Modèle d'exécution 12

1.2.3 Modèle de conservation d'objets 14

1.3 SERVICES PRINCIPAUX 16

1.3.1 Archivage des objets 16

1.3.2 Construction de machines virtuelles et exécution répartie 17

1.3.3 Interface d'utilisation 17

1.3.4 Interpréteur interactif des commandes Shell 18

1.3.5 Gestionnaire des types et des classes 18

CHAPITRE 2 : LE LANGAGE GUIDE ET SON MODELE D'OBJETS 21

2.1 CONSTRUCTION D'OBJETS - TYPES ET CLASSES ELEMENTAIRES 21

2.2 TYPE 22

2.2.1 Définition 22

2.2.2 Conformité syntaxique entre types 24

2.2.3 Sous-types 25

2.2.4 Relation entre sous-typage et conformité syntaxique 26

2.3 CLASSE 28

2.3.1 Définition 28

2.3.2 Modèle d'état des objets 29

2.3.3 Programme des opérations 30

2.3.3.1 Affectation de variable 31

2.3.3.2 Appel de méthode 31

2.3.3.3 Instruction ASSERTYPE 31

2.3.3.4 Instruction TYPECASE 33

2.3.3.5 Bloc d'exécution parallèle CO_BEGIN ... CO_END 34
2.3.4 Conditions d'exécution des méthodes sur les objets 34
2.3.5 Héritage 37
2.4 OBJETS 38
2.4.1 Création d'objets 39
2.4.2 Partage d'objets et accès aux objets 39
2.4.3 Destruction d'objets 39
2.5 GENERICITE : LES CONSTRUCTEURS DE TYPES ET DE CLASSES 40
2.5.1 Généricité sur les paramètres de type 40
2.5.2 Généricité sur les paramètres de dimensionnement 42

CHAPITRE 3 : ARCHITECTURE DE LA MEMOIRE D'OBJETS GUIDE 45
3.1 ARCHITECTURE GENERALE DE LA MEMOIRE D'OBJETS GUIDE 45
3.2 DESCRIPTION DES OBJETS EN MVO 47
3.2.1 Images d'objets 47
3.2.2 Création et destruction d'images 49
3.3 DESCRIPTION DES OBJETS EN MPO 49
3.3.1 Systèmes d'objets 50
3.3.2 Objets stockés 51
3.3.3 Création d'objets stockés 52
3.4 CYCLE DE VIE DES OBJETS 52
3.5 REFERENCES ET LOCALISATION DES OBJETS 56
3.6 CONCLUSION 57

DEUXIEME PARTIE : REALISATION DU COMPILATEUR GUIDE 59
CHAPITRE 4 : REALISATION DU COMPILATEUR GUIDE 61
4.1 OBJECTIFS DU COMPILATEUR 61
4.2 ARCHITECTURE GENERALE DU COMPILATEUR GUIDE 62
4.2.1 Contraintes du développement 62
4.2.2 Schéma de traduction proposé 63
4.3 GESTIONNAIRE DES TYPES ET DES CLASSES 65
4.4 ANALYSE SEMANTIQUE 66
4.4.1 Vérification de types dans les déclarations de variables 67
4.4.2 Vérification de conformité 67
4.4.3 Traitement des erreurs sémantiques 70

4.5 GENERATION DE CODE C 71
4.5.1 Structure de mémoire d'une activité Guide 72
4.5.2 Chargement d'une classe dans la mémoire d'exécution d'une activité 74
4.5.3 Format d'une classe exécutable 74
4.5.4 Liaison dynamique et Table des Définitions Externes d'une classe 76
4.5.5 Format de la référence langage 78
4.5.6 Format du bloc de paramètres dans un GuideCall 80
4.5.7 Traduction d'un type 82
a. Traduction d'une variable d'état visible 83
b. Traduction d'une signature de méthode 83
4.5.8 Traduction d'une classe 85
a. Traduction d'une référence à un objet 85
b. Construction du modèle de données des objets 85
c. Construction du descripteur de la classe 86
d. Construction de la Table des Définitions Externes associée à une classe 87
e. Traduction d'un appel de méthode sur un objet 88
f. Principe de compilation d'une méthode 89
g. Génération de code pour l'instruction ASSERTYPE 90
h. Génération du code pour l'instruction TYPECASE 90
i. Création des objets 92
j. Génération du code pour l'instruction CO_BEGIN ... CO_END 92
4.6 CONCLUSION 95

TROISIEME PARTIE : ETUDE ET REALISATION DES RAMASSE-MIETTES POUR LE SYSTEME GUIDE 99

CHAPITRE 5 :
ETUDE SYNTHETIQUE DES ALGORITHMES DE RAMASSE-MIETTES EXISTANTS 101

5.1 RAMASSE-MIETTES CENTRALISE 101
5.1.1 Modèle de mémoire d'objets 101
5.1.2 Ramasse-miettes d'un tas en contexte centralisé 105
5.1.2.1 Techniques par compteurs de références 105
5.1.2.2 Techniques par parcours-marquage 107
1. Suspension du mutateur pendant l'exécution du collecteur 108
2. Exécution simultanée du mutateur et du collecteur 109

5.1.2.3 Algorithmes du type compactage 114
1. Algorithme de générations d'objets 116
2. Algorithme de compactage dans une mémoire virtuelle ayant la protection de pages 118
5.2 RAMASSE-MIETTES EN REPARTI 120
5.2.1 Modèle de mémoire d'objets répartie fréquemment utilisé 120
5.2.1.1 Les tables d'adressage 121
5.2.1.2 Les opérations sur les objets 122
5.2.2 Ramasse-miettes de type compteurs de références 124
5.2.2.1 Algorithme dérivé directement du centralisé 124
5.2.2.2 Distribution du compteur 125
5.2.2.3 Les références pondérées 126
5.2.3 Ramasse-miettes de type marquage global 128
5.2.3.1 Construction de l'arborescence globale 130
5.2.3.2 Références en transit lors de l'activation des collecteurs 134
5.2.3.3 Les mutateurs s'exécutent pendant le travail des collecteurs 135
5.2.4 Récupération locale des miettes 136
5.3 CONCLUSION 136

CHAPITRE 6 : ALGORITHMES DE RAMASSE-MIETTES POUR GUIDE 139
6.1 COMPTEURS DE REFERENCES POUR GUIDE MONO-SITE 139
6.2 COMPTEURS DE REFERENCES POUR GUIDE REPARTI 146
6.3 PARCOURS-MARQUAGE MONO-SITE AVEC ACTIVITES INTERROMPUES .. 148
6.3.1 Modèle de mémoire d'objets Guide pour cet algorithme 148
6.3.2 Principe de l'algorithme 149
6.4 PARCOURS-MARQUAGE MONO-SITE AVEC ACTIVITES 152
6.5 PARCOURS-MARQUAGE REPARTI 155
6.5.1 Algorithme de détection de terminaison répartie de MISRA 155
6.5.2 Modèle de mémoire d'objets Guide pour cet algorithme 157
6.5.3 Principe de l'algorithme de parcours-marquage 158
6.6 RAMASSE-MIETTES LOCAL SUR CHAQUE SITE 161
6.7 CONCLUSION 162

CHAPITRE 7 : CONCLUSION 163

ANNEXE 1 : CONVENTION DE PROGRAMMATION EN GUIDE..... 169
ANNEXE 2 : SYNTAXE DU LANGAGE GUIDE..... 173
ANNEXE 3 : ANALYSE LEXICALE ET SYNTAXIQUE 181
ANNEXE 4 : TYPES ELEMENTAIRES DE GUIDE..... 193
ANNEXE 5 : BIBLIOTHEQUE DES TYPES GUIDE..... 187
BIBLIOGRAPHIE 193

INTRODUCTION

1. OBJECTIFS PRINCIPAUX DE LA THESE

Depuis la mise en service des premiers ordinateurs, les programmeurs conçoivent et expérimentent des méthodes d'analyse et de production du logiciel dans le but de faciliter leur travail et d'améliorer la qualité de leurs produits. L'évolution constante des techniques de programmation vers une production de code d'une qualité toujours meilleure est une preuve vivante de cet effort. La programmation à objets apparaît comme le sommet de cette évolution après les techniques de programmation procédurale et modulaire.

La technologie de fabrication des composants informatiques a également évoluée avec une grande vitesse. L'apparition de postes de travail puissants avec leur écran graphique à haute résolution et leur souris a transformé la notion de langage de commandes des systèmes d'exploitation, et facilité grâce au multi-fenêtrage la gestion simultanée d'activités multiples indépendantes ou corrélées. Les fonctions de traitement et de stockage d'informations ne sont plus confiées à un organe central unique multiplexé entre un ensemble d'utilisateurs, mais chaque utilisateur dispose d'un poste de travail et accède, via un réseau à grand débit, aux ressources fournies par un ensemble de serveurs coopérants.

Plusieurs projets de conception et de réalisation d'un système d'exploitation qui intégrerait les progrès matériels et logiciels ont été définis. Un exemple de ces projets est le projet Guide dans lequel a été effectué mon travail de recherche.

GUIDE* (Environnement Distribué et Intégré des Universités de Grenoble) est un projet de recherche du Laboratoire Bull-IMAG à Grenoble. Son objectif est la conception et la réalisation d'un système d'exploitation expérimental réparti, sur un réseau local de postes de travail et de serveurs. Les applications pilotes prévues sont le génie logiciel et la gestion de documents structurés.

Le trait le plus important du système Guide est celui de l'intégration : l'intégration des composants physiques puisque chaque poste de travail est considéré comme un composant d'un système d'exploitation global réparti ; l'intégration des concepts et des techniques de génie logiciel. Cette intégration a été largement facilitée par le choix d'un modèle à objets comme modèle de structuration des applications et du système lui-même.

---

* Le projet Guide est soutenu par le Centre National de la Recherche Scientifique (CNRS) via le PRC C^3 (Coopération, Concurrence, Communication) et par le Centre National d'Etudes des Télécommunications (CNET). Une partie des travaux du projet Guide est soutenue par la Commission des Communautés Européennes dans le cadre du projet ESPRIT n° 834, Comandos (COnstruction and MANagement of Distributed Office Systems).

L'objet est l'entité de base du système et chaque composant du système n'est autre qu'un objet. Un objet est à la fois l'unité d'abstraction, d'exécution et de conservation ; il est autonome et n'est accédé par l'extérieur que via une interface. L'interface d'un objet est un ensemble d'opérations d'accès (ou méthodes) et plusieurs objets peuvent avoir la même interface. La notion de **type** de Guide (type abstrait) sert à exprimer cette interface. Ainsi, les objets ayant la même interface appartiennent à un même type.

Un objet se compose de données (ensemble de champs de types éventuellement différents) appelées *état* et du code des méthodes. L'état d'un objet évolue par l'exécution du code d'une des méthodes de l'objet ; cette exécution peut être initialisée à l'extérieur de l'objet en appelant la méthode sur l'objet.

La notion de **classe** de Guide (type concret) sert à définir le modèle de l'état de l'objet et le code des méthodes d'accès. Plusieurs objets peuvent avoir le même modèle d'état et le même code des méthodes, et dans ce cas, ils appartiennent à une même classe. Le langage Guide permet de définir la classe d'un objet, autorisant ensuite sa création, et le type de l'objet, qui en contrôle la manipulation. Après avoir spécifié les types et les classes, il faut disposer d'un compilateur pour traduire ces spécifications écrites dans le format source (texte) en **objets exécutables** dont les caractéristiques sont connues par le système. Un des deux objectifs principaux de mes études concerne ce problème de compilation : à partir du langage Guide et des primitives du noyau Guide, définir et réaliser un schéma de traduction de ce langage en code exécutable.

Comme nous l'avons déjà dit, l'objet dans le système Guide est à la fois l'unité d'exécution et de conservation des données ; il est en plus **permanent** : il doit être présent avant qu'une méthode puisse s'exécuter sur lui, et il doit rester pour qu'une autre méthode puisse s'exécuter un moment plus tard. Les objets sont tous stockés dans la mémoire d'objets du système. Chaque objet possède un **identificateur** unique et grâce à cet identificateur, on peut identifier et localiser l'objet. Pour désigner un objet dans le système entier, on utilise une **référence** qui contient, entre autres choses, l'identificateur unique de l'objet. Lorsqu'il possède la référence à un objet, l'utilisateur peut accéder à ce dernier et faire évoluer son état en appelant une de ses méthodes. L'état de l'objet est un ensemble de champs de données dont certains peuvent être des références à d'autres objets ; dans ce cas, on dit que l'objet est composé.

Une application dans le système Guide consiste en un ensemble d'objets ; on peut lancer son exécution en appelant une méthode sur un objet de départ. Au cours de l'exécution d'une application, de nouveaux objets peuvent être créés dynamiquement au fur et à mesure des besoins ; ils sont liés les uns aux autres via les références contenues dans l'état des objets. Plusieurs applications peuvent être exécutées en parallèle et peuvent partager des objets (par exemple un catalogue commun).

Quand le système évolue, le nombre d'objets devient de plus en plus grand et la mémoire d'objets est saturée après un certain temps si l'on n'a aucun moyen de détruire les objets inutiles.

Un objet devient indestructible quand un utilisateur lui associe un nom symbolique. Les objets indestructibles sont dits **racines** du système et tous les objets qui sont référencés soit directement soit indirectement par eux sont dits accessibles ou utiles. Les autres objets dans le système, s'il existent, sont des objets inutiles ou inaccessibles, ou encore des **miettes**. Dans un système où les objets sont créés dynamiquement, un mécanisme de détection et de ramassage des miettes est nécessaire pour garantir que la mémoire d'objets n'est pas saturée tant que la zone occupée par les objets utiles du système ne dépasse pas la capacité de stockage. L'étude et la réalisation des mécanismes de ramasse-miettes pour le système Guide est le deuxième objectif principal de mes études.

2. PRESENTATION DU PLAN DE LA THESE

Les deux grandes parties de ma thèse correspondent à ces deux objectifs principaux. Elles sont précédées d'une présentation de l'architecture du système Guide.

Première partie : Architecture du système Guide.

Chapitre 1 : Architecture générale du système Guide

L'objectif de ce chapitre est de donner une vue générale de l'organisation du système Guide avec ses trois principaux éléments : le modèle d'objets, le modèle d'exécution et le modèle de mémoire d'objets. Les services principaux du système ainsi que les mécanismes de mise en œuvre sont aussi présentés.

Chapitre 2 : Le langage Guide et son modèle d'objets

Il est naturel de présenter le langage lui-même avant les mécanismes de traduction. Un langage est défini d'une part par sa syntaxe et d'autre part par sa sémantique. Malheureusement, la présentation de la syntaxe d'un langage est toujours sèche même dans le cas où on donne la sémantique de chaque élément syntaxique. Ainsi, nous résumons la syntaxe du langage Guide dans une annexe (annexe 2) et nous ne présentons dans ce chapitre que les éléments principaux du modèle d'objets de Guide et leur expression (informelle) en langage Guide. L'objectif de ce chapitre est de donner les informations principales, nécessaires d'une part à la programmation des applications en langage Guide et d'autre part à la présentation du compilateur qui constitue le chapitre 4.

Chapitre 3 : Architecture de la mémoire d'objets Guide

Nous présentons dans ce chapitre l'architecture de la mémoire d'objets Guide. Cette mémoire contient tous les objets créés par les utilisateurs, et dont certains sont à ramasser ; son organisation va fortement influencer les algorithmes de ramasse-miettes. Nous précisons aussi pour quelle partie de la mémoire d'objets les ramasse-miettes travaillent.

Deuxième partie : Définition et réalisation d'un schéma de traduction du langage Guide.

Chapitre 4 : Réalisation du compilateur Guide

La conception d'un schéma de traduction et la réalisation d'un compilateur pour Guide est le premier travail effectué au cours de la thèse. Ce travail est résumé dans ce chapitre où les points essentiels du processus de compilation des types et des classes Guide sont présentés.

Troisième partie : Etude et réalisation des ramasse-miettes pour le système Guide

Chapitre 5 : Etude synthétique des familles d'algorithmes de ramasses-miettes existants

Le problème du ramassage de miettes est classique mais chaque système a ses propres caractéristiques et, selon ces caractéristiques, un algorithme de ramasse-miettes peut être plus efficace que d'autres. Ainsi, dans ce chapitre, nous présentons notre étude synthétique des familles d'algorithmes de ramasse-miettes existants avec les avantages et les inconvénients de chacune.

Chapitre 6 : Proposition et réalisation des algorithmes de ramasse-miettes pour Guide

Avec les caractéristiques de la mémoire d'objets de Guide et les avantages et les inconvénients des algorithmes de ramasse-miettes existants présentés dans les deux chapitres 3 et 5, nous présentons dans ce chapitre les ramasse-miettes choisis pour notre système ainsi que des adaptations que nous avons introduites afin de les rendre encore plus satisfaisants et plus efficaces.

Conclusion

Nous effectuons dans ce dernier chapitre un bilan du travail réalisé (avec certaines mesures de performance) et définissons quelques prolongements qu'il serait souhaitable d'y apporter.

Première partie

ARCHITECTURE DU SYSTEME GUIDE

CHAPITRE 1 : ARCHITECTURE GENERALE DU SYSTEME GUIDE

1.1 OBJECTIFS GENERAUX DU PROJET

Comme nous l'avons présenté dans l'introduction, le thème général du projet Guide est le développement d'un système d'exploitation pour des postes de travail individuels et des serveurs interconnectés par un réseau local, et utilisés pour des applications générales. Pourtant il ne s'agit pas, au moins dans un premier temps, de construire un système pleinement opérationnel destiné à remplacer les systèmes existants, mais de réaliser un système expérimental, qui doit servir d'outil d'investigation et de banc d'essai pour des recherches sur les systèmes répartis : validation de nouvelles méthodes de conception, structure des systèmes d'exploitation répartis, outils pour l'expression et le développement d'applications réparties. Les classes d'applications prévues, qui sont parmi les plus courantes, sont le génie logiciel (production, mise au point, maintenance de programmes), et la gestion de documents (édition, archivage, courrier électronique).

L'objectif est que l'utilisateur du système Guide ait à sa disposition, de façon facile, efficace et permanente, un ensemble de services fournis par le système : stockage d'information avec des garanties suffisantes de fiabilité et de confidentialité, accès à des informations partagées, communication, utilisation d'outils pour le développement et la mise au point d'applications. En termes généraux, ces spécifications sont voisines de celles qui ont servi de base, en 1965, à la conception du système Multics, d'où sont issus les principaux concepts des systèmes en temps partagé. Néanmoins, trois aspects ont considérablement changé depuis cette époque :

- Les modalités de prestation de services et de partage des ressources sont profondément différentes. Les fonctions de traitement et de stockage ne sont plus confiées à un organe central unique multiplexé entre un ensemble d'utilisateurs, mais chaque utilisateur dispose d'un poste de travail puissant et accède en outre, via un réseau à grand débit, aux ressources fournies par un ensemble de serveurs coopérants.
- Les interfaces présentées par le système aux utilisateurs ont également changé : les possibilités de visualisation synthétique (les écrans graphiques à haute résolution) et d'interaction rapide (les souris) ont transformé la notion de langage de commandes. Le multi-fenêtrage facilite la gestion simultanée d'activités multiples, indépendantes ou corrélées.
- Des concepts nouveaux pour la construction du logiciel (modules, types abstraits, objets, généricité) ont été élaborés. Ils permettent notamment de dissocier la description des interfaces de celle des réalisations et de définir des classes

d'objets ayant un même comportement ; ils facilitent ainsi la réutilisation du logiciel et permettent de dissimuler aux utilisateurs l'hétérogénéité du matériel ou des systèmes.

Des systèmes répartis interconnectant des postes de travail et des serveurs ont été réalisés et utilisés. On peut schématiquement les classer en trois catégories, selon le degré d'intégration de leurs composants :

1. Systèmes maintenant une séparation complète des fonctions du système d'exploitation local et de la gestion des ressources distantes. De tels systèmes sont des extensions des systèmes d'exploitation locaux, qui permettent l'accès à des ressources distantes (généralement des fichiers), par demande explicite de transfert.

2. Systèmes fournissant un espace unique de désignation et permettant l'intégration des machines individuelles dans cet espace. L'interface commune est généralement celle définie par les primitives d'un système d'exploitation (cas, par exemple, des diverses extensions réparties d'Unix comme la "Newcastle Connection" [Brownbridge 82], ou le NFS de Sun [Sandberg 86]), mais peut aussi être la machine virtuelle définie par un langage de programmation, comme dans le projet Pilot/Cedar [Donahue 85].

3. Systèmes à haut degré d'intégration, où chaque poste de travail est considéré comme un composant d'un système d'exploitation global réparti. Au contraire des systèmes précédents, les fonctions liées à la répartition ne sont pas surajoutées à un système déjà existant, mais prises en compte dès l'origine. Les travaux dans ce domaine sont moins avancés que dans les deux précédents, mais plusieurs systèmes de ce type ont été réalisés. Par exemple, dans le système Apollo/Domain [Leach 83], l'intégration est réalisée par une "mémoire d'objets" globale répartie ; dans le système Locus [Popek 85] elle est réalisée par la répartition des mécanismes de base d'Unix (fichier et processus), obtenue par une réécriture complète du noyau. Il subsiste néanmoins de nombreux problèmes ouverts, notamment les problèmes de fiabilité, ceux de la gestion globale des ressources, ceux de la coopération des composants d'une application répartie.

Un des objectifs du projet Guide est d'atteindre un plus grand degré d'intégration des composants du système réparti. Avec le système Guide, on souhaite décharger l'utilisateur de la nécessité de connaître la localisation des ressources qu'il demande ou le détail des protocoles de communication nécessaires pour y accéder.

Les objectifs du projet, qui viennent d'être présentés, tendent à privilégier une démarche de conception descendante et une vue intégrée des divers composants du système. On vise donc :

- à transposer, dans l'organisation d'un système d'exploitation réparti, les concepts introduits dans les langages de programmation pour faciliter l'abstraction et la réutilisation de composants,

- à intégrer en un ensemble cohérent les aspects relatifs à l'exécution et ceux touchant à la conservation permanente de l'information.

Ces considérations ont amené les concepteurs du projet Guide à adopter un des modèles issus des travaux sur la programmation par objets, en y intégrant les aspects liés à la conservation permanente des objets, qui sont traditionnellement traités séparément sous la rubrique des systèmes de gestion de fichiers ou des bases de données. Par ailleurs, on voudrait chercher à spécifier un modèle abstrait d'exécution et de conservation aussi indépendant que possible de la réalisation physique et notamment de la répartition. On espère ainsi pouvoir faciliter l'expérimentation et la comparaison de divers mécanismes de mise en œuvre du modèle sur un système réparti, et la prise en compte de l'hétérogénéité des composants physiques du système.

Les objectifs du projet sont très larges, il est important de pouvoir spécifier clairement quels sont les domaines d'exploration et ceux où on utilise les solutions plus ou moins standard (ou ceux qu'on laisse totalement de côté au moins dans un premier temps). Les domaines d'exploration du projet Guide sont en priorité :

- l'étude de la structuration et du langage d'expression d'applications réparties et des mécanismes correspondants au niveau du système,
- le développement d'un modèle pour la conservation permanente des objets, qui permet de fournir facilement des services évolués tels que la gestion de versions et la gestion d'objets composés,
- la prise en compte d'un certain degré d'hétérogénéité.
- l'allocation de ressources, au sens large (construction de machines virtuelles multi-sites, répartition de charge, etc).
- les interfaces homme-machine.

On utilise les solutions existantes et on les intègre au système pour ce qui concerne la gestion de transactions, la protection et la sécurité.

1.2 ARCHITECTURE DU SYSTEME GUIDE

Un modèle à objets étant choisi comme structure de base du système, on cherche un schéma pour la conservation des objets et pour l'exécution des programmes sur ces objets. En conséquence, le système Guide se compose de trois éléments principaux : le modèle d'objets, le modèle d'exécution et le modèle de gestion d'objets. Dans cette section, nous présentons de façon générale ces trois modèles et les liens qui existent entre eux.

1.2.1 MODELE D'OBJETS

Guide est un système à objets uniforme au sens où tous les éléments manipulés par l'utilisateur sont des objets. Le modèle d'objets Guide sera présenté plus en détail dans le chapitre 2. Cependant, une introduction générale du modèle est donnée dans les paragraphes qui suivent.

Un objet n'est accessible qu'à travers un ensemble d'opérations d'accès (ou méthodes) qui sont les seules pouvant accéder directement aux données de l'objet. Tous les objets ayant le même ensemble de méthodes appartiennent à un type, qui définit les signatures des

méthodes applicables à ces objets ainsi que la spécification de leur effet. La signature d'une méthode spécifie son nom symbolique, le nombre de ses paramètres ainsi que leur statut et leur type. Tout ce qu'on doit connaître pour pouvoir utiliser "correctement" un objet est son type. Les ressources générales du système telles que les domaines, les activités, les périphériques, etc... (cf. 1.2.2) sont traitées comme des objets et leur type est prédéfini.

Un objet est composé de deux parties : des données constituant son état et le programme des méthodes. Tous les objets qui ont une même structure de données et le même code des méthodes font partie d'une même classe qui est une réalisation particulière de leur type. Du point de vue de la programmation, les classes servent à définir la structure de données ainsi que le programme des méthodes de leurs objets ; pour le système, les classes servent à stocker les informations communes qui sont partagées par tous leurs objets (appelés les instances ou exemplaires).

La distinction entre type et classe dans Guide est analogue à la séparation entre interface et réalisation dans les langages modulaires (Modula-2 [Wirth 85]), mais il y a deux différences importantes :

- la génération dynamique des instances d'une classe,
- la réalisation d'un même type par plusieurs classes différentes dont les instances peuvent être créées et utilisées dans une même application.

Remarque : Dans de nombreux langages à objets (Smalltalk [Goldberg 85], C++ [Stroustrup 86], Eiffel [Meyer 88]), la séparation entre type et classe n'apparaît pas (le type est implicitement extrait de la définition de la classe), ce qui ne permet pas de réaliser un même type par plusieurs classes.

Désignation des objets

Les objets dans Guide sont tous permanents au sens où leur durée de vie est indépendante de celle des programmes qui les créent et les manipulent. Dans les systèmes traditionnels, ces objets permanents sont habituellement conservés explicitement par le programmeur sous forme de fichiers. Dans Guide, les objets sont tous persistants et les utilisateurs sont déchargés de la tâche de stockage et de récupération explicite. De plus, on désigne les objets de manière uniforme et indépendamment de leur utilisation. Les utilisateurs peuvent utiliser un des deux types de désignation d'objets : la désignation par références (noms internes interprétables par le système) et la désignation symbolique (celui utilisé pour désigner des fichiers dans les systèmes traditionnels tels qu'Unix, Multics).

- `Désignation par références` : Une référence est une structure de données contenant des informations qui permettent de désigner un objet indépendamment de tout mécanisme de gestion de mémoire d'objets à l'exécution. Une référence comporte un identificateur unique interne (ou `Oid`) attribué à l'objet lors de sa création, et des informations de service destinées notamment à permettre la localisation plus rapide de l'objet (cf. 3.5). Plusieurs exemplaires d'une même référence peuvent exister dans le système à un moment donné, cette propriété permet le partage de l'objet identifié par cette référence.

- `Désignation symbolique` : Les noms symboliques sont utilisés pour la désignation des fichiers dans les systèmes traditionnels et cette possibilité est aussi offerte dans Guide pour la désignation des objets. Le service de désignation symbolique, fourni par le système, permet aux utilisateurs d'associer un nom symbolique à un objet et d'obtenir une référence à un objet dont ils connaissent le nom symbolique. Plusieurs réalisations de ce service ont été réalisées, une première qui s'inspire du schéma de désignation de la Newcastle connexion et une seconde qui est une réalisation de la norme ISO X500.

Les références sont utilisées par les utilisateurs :

- pour appeler des méthodes sur l'objet référencé. En fait, pour appeler une méthode `op` d'un objet `o`, on écrit `x.op(<paramètres>)`, où `x` est une variable typée contenant une référence à `o` ;
- pour passer en paramètre un objet construit : on passe une référence de cet objet ;
- pour fabriquer un objet composé (cf. 2.1) : les liens entre cet objet et les objets qui le composent sont matérialisés par des références contenues dans l'état de l'objet composé.

Par ailleurs, les noms symboliques sont utilisés par les utilisateurs :

- pour nommer et désigner de façon "confortable" certains objets intéressants (par exemple, les types et les classes) ;
- pour retrouver un objet d'une application à l'autre, surtout dans le cas où les deux applications sont indépendantes.

Sous-typage, héritage, généricité et réutilisation du code de programmes

Dans un système à objets, on rencontre fréquemment des objets qui ont des comportements similaires. Plus précisément, les objets d'un type ont une interface qui englobe celle des objets d'un autre type. Rien n'empêche les utilisateurs de définir les deux types indépendamment, mais ce travail est rendu plus confortable grâce à la relation de sous-typage fournie par Guide (cf. 2.3). La relation de sous-typage permet aux utilisateurs de définir de nouveaux types en utilisant des types existants et en ajoutant de nouveaux éléments ; elle facilite donc la réutilisation de programme (plus précisément d'interface).

Pour la réutilisation du modèle de données et de programmes des objets, Guide fournit la notion d'héritage (notion fondamentale des modèles à objets) au niveau des classes (cf. 2.3.5) : on peut définir une nouvelle classe en héritant d'une classe existante, puis en ajoutant de nouveaux éléments. L'héritage dans Guide est simple, c'est-à-dire qu'une classe ne peut avoir qu'une seule super-classe.

Puisque chaque classe définit le modèle de données et le programme des méthodes de tous les objets qui lui appartiennent et qu'elle est utilisée par le système pour la création des objets, nous disons que les classes sont des constructeurs d'objets. La généricité permet, quant à elle, de définir des constructeurs de types ou de classes (cf.2.4). Un constructeur de types (classes) est un modèle pour la définition d'un ensemble de types (classes) paramètrées. La

possibilité de paramètrer des types et des classes est un autre moyen, en plus de celui de sous-typage et d'héritage, pour permettre la réutilisation du code des programmes et pour éviter des définitions redondantes.

1.2.2 MODELE D'EXECUTION

En ce qui concerne le schéma d'exécution, Guide fournit aux utilisateurs des "**machines virtuelles**" où, par défaut, la distribution est transparente et le parallélisme apparent, car on pense que la gestion d'activités multiples simultanées est une façon d'exploiter aussi bien la grande puissance des postes de travail modernes (éventuellement multi-processeurs) que la répartition. La machine virtuelle sera donc multi-processeur. Cela peut se réaliser de diverses façons : rendre les objets eux-mêmes multi-processus (Amoeba [Tanenbaum 86], Chorus [Rozier 88]) ; permettre à un ensemble de processus d'accéder aux objets considérés comme passifs [Allchin 83] ; combiner les deux, certains objets étant actifs et d'autres passifs (Emerald [Black 86a]). Le choix de Guide consiste à laisser les objets passifs et à définir des environnements d'exécution (regroupement d'un ensemble d'objets) multi-processus. Ce choix est justifié par le souci de garder une structure légère tant pour les objets que pour les processus, et donc de dissocier les deux. Ces considérations conduisent les concepteurs de Guide à définir une unité d'exécution constituée d'un ensemble d'objets et d'un ensemble de processus opérant sur ces objets, la composition de ces deux ensembles pouvant évoluer dynamiquement. On appelle cette unité un **domaine**, et on donne aux processus le nom d'**activités**. Le terme de domaine est emprunté aux systèmes à capacités [Organick 72], où il désigne, de manière analogue, un environnement d'exécution composé d'objets auxquels sont associés des droits d'accès. Le terme d'activité a été choisi à la place de celui de processus pour réserver ce dernier aux processus du système Unix qui est, au moins dans un premier temps, l'outil de mise en œuvre du système Guide.

Un objectif du système étant de dissimuler la répartition à ses utilisateurs, il n'y a pas de lien a priori entre domaines et sites : un domaine peut s'étendre dynamiquement sur plusieurs sites, un site peut être le siège de plusieurs domaines. La notion de domaine et sa réalisation sont développées de façon plus détaillée dans [Guide-R3], [Decouchant 88b].

Les activités et les domaines sont deux éléments dans le modèle d'exécution et nous détaillons maintenant leur construction. Notons d'abord que les activités et les domaines eux-mêmes sont rendus visibles aux applications sous forme d'objets (ce qui permet d'approcher un modèle uniforme). Néanmoins, ces objets ont un statut spécial : leur classe est prédéfinie, réalisée par le système, et soumise à des restrictions d'accès. Dans la suite de ce paragraphe, le terme d'objet désigne, sauf mention contraire, un objet "ordinaire", qui n'est ni une activité ni un domaine.

L'ensemble des objets qui composent un domaine à un instant donné est appelé son contexte. La composition du contexte peut changer au cours du temps, grâce à des opérations de liaison et de détachement dynamiques des objets. Dans un domaine, une structure de données gérée par le système, le descriptif du domaine, permet de retrouver les objets de son contexte. Un objet peut appartenir à plusieurs domaines (il possède alors une entrée dans le descriptif de chacun d'eux).

Par exemple, la figure 1.1 ci-après représente deux domaines `D1` et `D2`, dont les descriptifs sont `desc1` et `desc2`. Ces domaines partagent, à l'instant considéré, l'objet `d` ; chacun d'eux a en outre des objets qui lui sont propres.

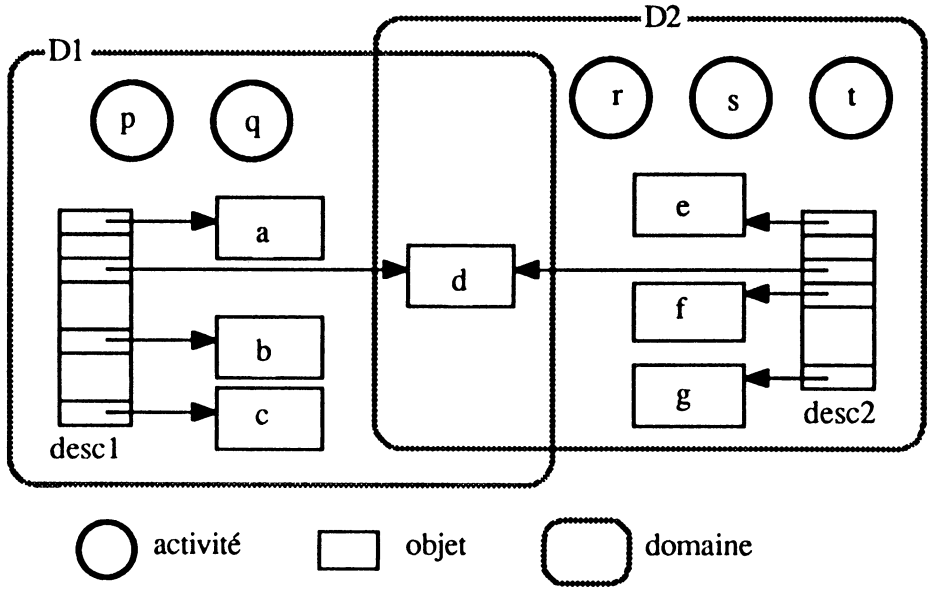


Figure 1.1 : Domaines, activités et objets

L'exécution d'une activité dans un domaine consiste en appels successifs aux méthodes des objets du domaine. La communication entre activités se fait donc à travers des objets, protégés si nécessaire par des mécanismes de synchronisation associés aux objets partagés (cf. 2.2.4 pour plus de détails).

Par souci de garder le modèle d'exécution simple et clair (sans perdre la souplesse ni la puissance), on a choisi un modèle dans lequel :

- un domaine ne peut pas être inclus dans un autre domaine ;
- une activité est toujours propre à un domaine et ne peut pas être partagée entre plusieurs domaines.

Examinons maintenant brièvement les aspects dynamiques du modèle d'exécution : création d'un domaine, création d'une activité, création d'un objet et liaison dynamique.

`Création d'un domaine` : Un domaine est créé par un autre domaine (plus précisément par une activité de ce dernier). L'opération de création spécifie un contexte initial (ensemble d'objets, non vide) et un point d'entrée, qui est une opération de l'un de ces objets. Le domaine contient initialement une activité (dite activité principale), créée dans l'état actif. Cette activité principale exécute le point d'entrée spécifié ; elle engendre s'il y a lieu les autres activités du domaine. Une fois créé, un domaine est autonome, mais le domaine qui l'a créé peut le contrôler (les opérations que l'on peut effectuer sur un domaine sont définies dans le type `DOMAIN` présenté dans l'annexe 4) : suspendre et reprendre son exécution, consulter son statut, le détruire. Les domaines communiquent entre eux par partage d'objets. En particulier,

le système fournit des objets de communication prédéfinis, les canaux, qui réalisent un flot unidirectionnel d'objets de même type.

`Création d'une activité` : Une activité act_2 est créée par une autre activité act_1, à l'intérieur d'un domaine D ; act_2 fait également partie de D, et peut accéder à tous les objets du contexte de D. Les activités communiquent entre elles par partage d'objets. Contrôler l'accès à des objets est donc un des moyens de synchroniser les activités. De plus, pour les activités qui existent dans un même domaine, on fournit également une autre possibilité de synchronisation en fin d'exécution d'une instruction de parallélisme (voir l'instruction `CO_BEGIN ... CO_END` en 2.3.3.5).

`Création d'un objet` : Un objet est créé par une activité donnée et à l'intérieur d'un domaine. La création se fait par appel de la méthode `New` sur la classe à laquelle appartient l'objet.

`Liaison dynamique` : Lorsqu'une activité donnée accède à un objet (par appel d'une de ses méthodes) qui n'existe pas encore dans le contexte du domaine de l'activité, le système doit chercher l'objet dans la mémoire d'objets et l'amener dans son contexte. Cette action est dite liaison dynamique, elle est nécessaire pour que les objets et leur classe soient présents et adressables dans le contexte du domaine. Par contre, lorsqu'un objet n'est plus utilisé dans un domaine (aucune de ses méthodes n'est en cours d'exécution), il peut être détaché du contexte du domaine pour libérer de la place occupée. S'il est détaché, il devra à nouveau être lié au prochain appel de l'une de ses méthodes.

1.2.3 MODELE DE CONSERVATION D'OBJETS

Il nous reste à présenter le mode de conservation des objets et le lien entre le support de conservation et les domaines. Le modèle de conservation d'objets est présenté plus en détail dans le chapitre 3, nous en décrivons ici les points généraux.

Un modèle général (dont une réalisation centralisée est fournie par Multics et une réalisation répartie par Appolo/Domain [Leach 83] ou Hydra [Cohen 74]) est celui d'un espace universel contenant tous les objets, auquel les unités d'exécution accèdent par couplage (rappelons que ce terme désigne la mise en correspondance d'un objet ou d'une partie d'objet avec un espace d'adressage, par exemple une mémoire virtuelle). C'est ce schéma que l'on a adopté pour Guide : pour être accessible dans un domaine, un objet doit y avoir été lié ; un mécanisme de chargement à la demande assure la présence effective de l'objet au moment de l'accès. En ce qui concerne le mode de réalisation de l'espace universel d'objets, on choisit une organisation à deux niveaux : le niveau supérieur, ou **`Mémoire Virtuelle d'Objets (MVO)`** et le niveau inférieur, ou **`Mémoire Permanente d'Objets (MPO)`**.

La Mémoire Virtuelle d'Objets est le support des objets liés dans au moins un domaine. Dans la première version, la MVO est réalisée par l'ensemble des mémoires principales des sites, ou plus précisément, par l'ensemble des mémoires virtuelles fournies par le système sous-jacent (UNIX) local à chaque site.

La Mémoire Permanente d'Objets est le support permanent des objets de Guide. Le mouvement des objets entre MVO et MPO n'est pas commandé explicitement par les utilisateurs dans les domaines, mais automatiquement depuis des primitives du système. Par exemple, le chargement d'un objet est provoqué chaque fois que l'on veut exécuter une méthode de cet objet et qu'il est absent ; le déchargement a lieu uniquement si l'objet n'est lié (il n'est pas présent) dans aucune pile d'aucune activité et s'il est nécessaire de récupérer de la place.

La création dynamique des objets et le fait que leur durée de vie ne soit pas liée à celle d'un domaine ou d'une activité impliquent qu'il y a des objets inutilisables, des miettes, dans la mémoire d'objets. Le nombre de miettes augmente au cours de la vie du système et il est nécessaire de prévoir un mécanisme de ramassage des miettes (cf. partie 3).

L'architecture de la MVO et MPO sera présentée plus en détail dans le chapitre 3.

Le choix d'un modèle à deux niveaux permet d'isoler la partie du système chargée de la conservation permanente des objets, qui joue ainsi le rôle d'un service réparti réalisé par l'ensemble des sites pourvus d'une mémoire secondaire réservée au stockage. Il est donc possible d'avoir des sites Guide qui n'ont pas de MPO locale.

En résumé, l'architecture globale du système Guide est schématisée sur la figure 1.2. L'exécution d'une application par un utilisateur provoque la création d'un domaine, qui regroupe les objets nécessaires. Dans le cas d'une application simple pour laquelle tous les objets sont disponibles sur le site, le domaine se réduit à un espace virtuel où les objets nécessaires sont liés, puis exécutés par une activité unique (ou plusieurs s'il y a parallélisme). Dans le cas d'une application qui met en jeu un ou plusieurs sites distants (par exemple pour utiliser un serveur, ou pour accéder à des objets partagés), le domaine est multi-sites et multi-activités.

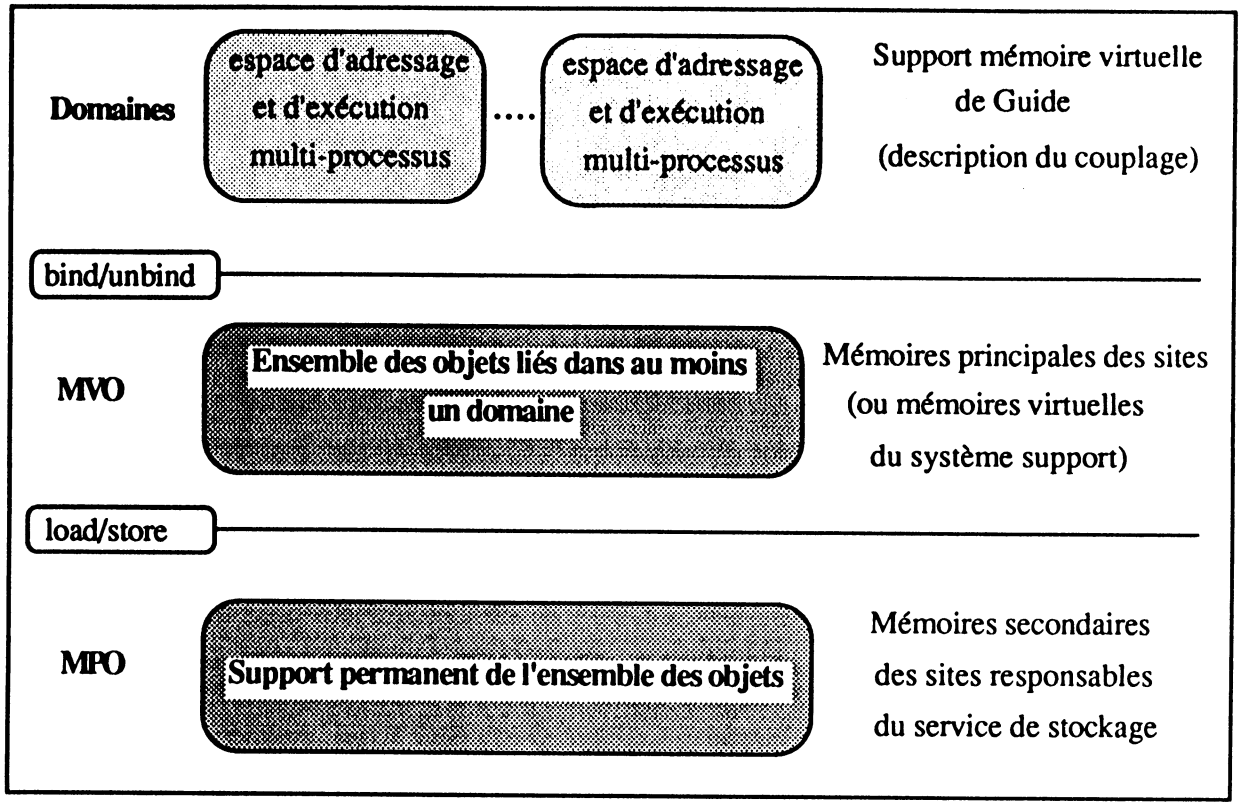


Figure 1.2 : Structure globale du système Guide

1.3 SERVICES PRINCIPAUX

Nous présentons maintenant les principaux services fournis aux utilisateurs de Guide et le schéma de la réalisation de chacun, au moyen des mécanismes qui viennent d'être décrits.

1.3.1 ARCHIVAGE DES OBJETS

Le service d'archivage d'objets de Guide est un service de base du système qui est assuré par la MPO en fournissant des fonctions d'un niveau d'abstraction supérieur à celles habituellement disponibles sur les serveurs de fichiers classiques. Avec ce service de base du système, on souhaite notamment :

- gérer des structures complexes dont les composants sont des objets. Un choix important de Guide est d'introduire une entité unique, l'objet, à la fois comme unité de structuration des applications et comme unité de conservation permanente. Il n'y a donc pas de "système de gestion de fichiers" au sens habituel. La conservation permanente des objets est un service de base du système fourni par la MPO.
- permettre aux applications de désigner les objets de façon souple, notamment par le biais de noms symboliques. Le lien entre les objets et les applications qui les manipulent, via des langages de programmation ou de commande, est assuré par un service de désignation symbolique, qui est une application particulière utilisant les mécanismes standards du système. Ce service permet d'associer aux objets

des noms symboliques globaux ; plus généralement, il peut être utilisé par toute application qui a besoin d'associer des noms symboliques à des entités réparties sur le réseau (ressources logiques, utilisateurs, etc).

1.3.2 CONSTRUCTION DE MACHINES VIRTUELLES ET EXECUTION REPARTIE

Le domaine est l'outil de base pour la construction de machines virtuelles. Rappelons que la machine définie par un domaine est potentiellement multi-processus et multi-sites ; elle peut s'étendre dynamiquement sur plusieurs sites, depuis son site de création. Les communications internes à un domaine utilisent le partage d'objets ; le mécanisme UNIX sous-jacent est, pour les communications locales à un site, le partage de la mémoire virtuelle et pour les communications inter-sites l'appel de procédure à distance. La possibilité d'extension des domaines sera notamment utilisée pour mettre en œuvre des politiques de répartition de charge, par exemple par utilisation de serveurs banalisés.

Le lancement d'une application particulière provoque en principe la création d'un nouveau domaine dans lequel sont liés les objets nécessaires à l'exécution de l'application. Un utilisateur peut contrôler simultanément plusieurs domaines, suivre leur évolution et interagir avec eux, par exemple à travers des fenêtres distinctes sur son écran.

Un service commun (par exemple le service de désignation symbolique) peut être réalisé par un domaine implanté sur l'ensemble des machines qui coopèrent à la réalisation du service, ou par un objet partagé qui utilise éventuellement d'autres objets. Dans le premier schéma, les domaines clients communiquent avec le service par des flots d'objets sur des canaux ; dans le second par appels de méthode. L'étude des critères de choix entre ces deux modes de réalisation d'un service est un des thèmes de recherche du projet.

1.3.3 INTERFACE D'UTILISATION

Le principe directeur de la conception de l'interface homme-machine (IHM) de Guide est de dissocier la conception de l'interface de celle des applications, en fournissant aux concepteurs de ces dernières un outil d'usage général. Pour cela, on introduit des objets interactifs, définis à un niveau intermédiaire entre l'utilisateur et l'application. Ces objets sont des abstractions des entités et des opérations usuelles mises en jeu dans une application interactive : chaînes de caractères, menus, figures, etc. Une bibliothèque d'objets courants est fournie par le système, mais le concepteur d'une application interactive peut lui-même définir ses propres objets.

L'IHM peut alors être décomposée en deux niveaux :

1. Un niveau "haut", ou gestion du dialogue, qui traduit le dialogue abstrait, défini par le concepteur d'une application interactive, en un dialogue concret, utilisant des objets interactifs et leurs fonctions d'accès.
2. Un niveau "bas", ou affichage, qui réalise la représentation physique des objets interactifs et de leurs fonctions d'accès au moyen des outils matériels et logiciels disponibles (écrans, gestionnaires de fenêtres, etc), en tenant compte de contraintes d'affichage spécifiées.

Un exemple très simple permet d'illustrer ces notions : une séquence de dialogue abstrait est le choix par un utilisateur d'une action parmi plusieurs. Ce choix peut être mis en œuvre au moyen d'un objet interactif de type menu. Le contrôleur de dialogue traduit alors l'opération abstraite de choix au moyen des fonctions d'accès au menu (montrer, dérouler, pointer, etc). Le contrôleur d'affichage assure la représentation visuelle du menu et de ses fonctions d'accès, en utilisant l'écran, la souris, le clavier, etc. Le concepteur de l'application n'a à connaître que l'opération abstraite de choix. La définition des objets interactifs, et leur réalisation en fonction des outils disponibles, sont indépendantes de l'application et restent à la charge du concepteur de l'IHM.

1.3.4 INTERPRETEUR INTERACTIF DES COMMANDES SHELL

Dans les systèmes classiques, on possède toujours au moins un interpréteur de commandes interactif (Shell d'Unix par exemple). Dans Guide, on fournit aussi un tel outil pour permettre aux utilisateurs de :

- manipuler les objets ayant un nom symbolique. On trouve les fonctions fréquemment utilisées telles que la visualisation des noms des objets dans un répertoire, la visualisation des caractéristiques des objets, le changement du répertoire de travail, la création d'un nouveau répertoire, la suppression d'un répertoire ou d'un objet, etc.
- appeler une des méthodes sur un objet ayant un nom symbolique donné. C'est le moyen principal de lancement des applications en Guide.

Bien que l'on puisse réaliser n'importe quel service en utilisant un domaine implanté éventuellement sur l'ensemble des machines qui coopèrent à la réalisation du service, on réalise actuellement cet interpréteur comme un objet partagé qui utilise principalement le service de noms symboliques, ce dernier est aussi réalisé comme un objet partagé.

1.3.5 GESTIONNAIRE DES TYPES ET DES CLASSES

Dans Guide, les objets qui sont créés par appel de la méthode New sur leur classe sont dits les objets normaux. L'évolution de l'état de ces objets est faite par les affectations qui sont contrôlées par la relation de conformité (cf. 2.2.2). Les types et les classes exécutables sont aussi des objets, mais leur création et éventuellement l'évolution de leur état est faite par le compilateur (automatiquement ou à la demande explicite de l'utilisateur après la création ou la mise à jour du source). La création d'un nouveau type ou d'une nouvelle classe exécutable ne pose pas de problème. Par contre, à cause des relations entre les types, entre les classes, entre les types et les classes, et entre les objets et leur classe, l'évolution de l'état d'un type ou d'une classe existants doit être contrôlée afin d'assurer la cohérence entre l'ensemble des types, des classes et de leurs objets dans le système :

- lorsqu'un type est modifié au niveau du source, est-il possible de modifier sa version exécutable ou bien doit-on garder son ancienne version et la considérer comme un type différent ? Ce problème se pose dans le cas où il existe déjà des sous-types ou des objets du type en question.

- comme dans le cas des types, lorqu'une classe est modifiée au niveau du source, quelle décision doit-on prendre en ce qui concerne sa version exécutable ? Remplace-t-on l'ancienne version par la nouvelle ou bien doit-on garder l'ancienne version et la considérer comme une classe différente ? Ce problème se pose dans le cas où il existe déjà des sous-classes ou des objets de la classe en question.

Ce contrôle est un thème de recherche intéressant et un service pour gérer les relations entre types et classes est fortement nécessaire dans un système à objets. On appelle souvent ce service **le gestionnaire de types et de classes**. Un 'Browser' est un outil qui utilise le gestionnaire de types et de classes pour permettre aux utilisateurs de visualiser et d'examiner les types et les classes dans leur hiérarchie pour mieux faire évoluer leur état.

CHAPITRE 2 : LE LANGAGE GUIDE ET SON MODELE D'OBJETS

Dans le chapitre 1 nous avons présenté l'architecture générale du système Guide, un système basé sur un modèle à objets. Ce modèle est accessible aux programmeurs grâce au langage Guide, un langage lié étroitement au système Guide qui permet l'expression des applications réparties. Dans ce chapitre, nous allons présenter le modèle d'objets de Guide en décrivant plus précisément les aspects les plus caractéristiques de Guide : contrôle statique et dynamique de types, structures liées au modèle d'exécution. Ces différents aspects sont en effet ceux qui sont les plus intéressants et originaux sur le plan de leur mise en œuvre dans le compilateur qui sera présenté dans le chapitre 4. Le modèle et le langage Guide sont décrits précisément dans les rapports de Guide [Guide-R2], [Krakowiak 90], [Roisin 90].

Les entités essentielles du modèle d'objets de Guide sont les **objets**, les **types** et les **classes** ; ces trois entités sont également les principales entités des autres modèles d'objets (Smalltalk [Golberg 85], Trellis/Owl [Schaffert 85], Emerald [Black 86b], Eiffel [Meyer 88]). On peut trouver une étude comparative du modèle d'objets de Guide et celui de Trellis/Owl, Emerald et Eiffel dans [Meysembourg 89]. La définition ainsi que les caractéristiques propres à Guide de ces trois entités sont présentées dans ce chapitre.

2.1 CONSTRUCTION D'OBJETS - TYPES ET CLASSES ELEMENTAIRES

Dans le modèle d'objets Guide, tout objet est défini par composition d'autres objets. En effet, la construction des objets est un processus récursif et il faut absolument avoir un moyen pour arrêter cette récursivité. La notion d'**objet élémentaire** est introduite pour résoudre ce problème. Les objets élémentaires sont des objets dont le modèle d'état ne contient pas d'autre objet, il contient seulement sa propre valeur. Comme tous les objets de Guide, chaque objet élémentaire appartient aussi à un type (cf. 2.2) et à une classe (cf. 2.3). Le type (ou la classe) des objets élémentaires est dit type (ou classe) élémentaire. Pour la simplicité, on a adopté les contraintes suivantes pour Guide :

- chaque type élémentaire est réalisé uniquement par une classe élémentaire et on utilise le même nom symbolique pour désigner les deux ;
- les types et les classes élémentaires n'ont aucune relation de sous-typage ni d'héritage (cf. 2.2.3 et 2.3.5). Ils ne peuvent pas être redéfinis par les utilisateurs.

Comme la plupart des langages de programmation, Guide fournit les types élémentaires **Boolean, Char, Integer**. La taille des objets élémentaires est très petite et le code des méthodes des objets élémentaires est souvent assez simple, il est donc raisonnable de mettre les objets élémentaires (leur propre valeur et le code des méthodes d'accès) dans l'objet englobant. Avec

ce choix, les objets élémentaires sont visibles seulement à l'intérieur de leur objet englobant et ils ne sont pas gérés directement par le système de gestion d'objets de Guide (cf. chapitre 3).

Les objets élémentaires des types **Boolean, Char et Integer** sont trop primitifs et comme la plupart des langages de programmation classiques, Guide fournit aussi les autres types élémentaires tels que **String[], Array[] OF et Record.** Ces derniers sont dits **types élémentaires structurés** et les éléments de ces types peuvent être soit des objets élémentaires (primitifs ou structurés) soit des références à d'autres objets composés (voir le paragraphe suivant).

Les autres objets non élémentaires sont dits **objets composés** (ou **objets construits**). Le type (ou la classe) d'un objet construit est dit type (ou classe) construit. Les objets construits ainsi que leur type et leur classe sont tous gérés par le système de gestion d'objets. Bien que les objets construits puissent contenir des objets élémentaires, leur état ne contient jamais l'état d'autres objets construits : si l'on veut qu'un objet fasse partie d'un autre, on met la référence du premier dans l'état du dernier. Ce choix a été pris dans Guide pour les raisons suivantes :

- la simplicité de la structure de l'état des objets (construits) ;
- la facilité du partage et de la gestion des objets ;
- la facilité de la modification des objets complexes.

Comme les objets élémentaires ne sont pas gérés par le système de gestion d'objets de Guide, nous utilisons dans le reste de cette thèse, sauf pour les cas particuliers, les termes **objet, type et classe** pour parler respectivement des objets construits, des types construits et des classes construites de Guide.

2.2 TYPE

2.2.1 DEFINITION

Un type sert uniquement à spécifier un comportement commun à un ensemble d'objets en termes d'**opérations** (ou **méthodes**) applicables à ces objets. Le comportement de chacune de ces méthodes est défini par une **signature** et une **spécification** :

- la signature spécifie le nom de la méthode, le nombre et le type des paramètres de la méthode ainsi que leur nature : paramètre d'entrée (valeur transmise par l'appelant), paramètre de sortie (valeur rendue par l'appelé), ou paramètre d'entrée/sortie,
- la spécification décrit l'effet de la méthode en fonction de l'état de l'objet et des valeurs des paramètres d'entrée. Bien que cette spécification fasse partie du type, elle n'est exprimée que de manière informelle et ne fait l'objet d'aucun traitement.

Les types utilisés dans la déclaration des paramètres des signatures d'un type sont dits les **types utilisés** de ce dernier. Dans certains cas pratiques, la relation "**types utilisés**" peut être croisée soit directement (un type `a` utilise le type `b` et le type `b` utilise `a`) soit indirectement (un type `a` utilise `b`, `b` utilise `c`, et `c` utilise `a`) et le langage Guide permet ces possibilités.

Pour manipuler un champ de donnée (désigné par une variable d'état) d'un objet à l'extérieur de ce dernier, on doit définir d'abord des méthodes de lecture et d'écriture sur cette

variable, et puis appeler ces méthodes. Pour des raisons de simplicité d'écriture tant au niveau de la définition de ces méthodes qu'au niveau des accès, on les exprime en utilisant la notion de **variables d'état visibles** dans les types. Une telle variable correspond à un champ de donnée de même type dans l'état de l'objet avec ses méthodes de lecture et d'écriture. Si une telle variable est précédée du mot-clé **CONST**, seule l'opération de lecture est autorisée.

En conséquence, dans la définition syntaxique d'un type, on décrit :

- le nom symbolique du type et de son super-type (cf. 2.2.3) ;
- pour chaque méthode, on décrit sa signature, c'est-à-dire son nom symbolique et sa liste de paramètres. La signature est un moyen de différencier des méthodes ; il est donc possible d'avoir des méthodes de même nom à condition qu'elles aient un nombre de paramètres différent : ce sont autant de méthodes différentes. Les mots-clés **IN, OUT** et **IN_OUT** permettent de distinguer l'attribut des paramètres dans les signatures des méthodes ;
- les variables d'état visibles.

Exemple : Nous donnons ci-dessous une définition particulière du type `BoîteAObjets` qui spécifie l'interface d'accès aux objets `BoîteAObjets`.

```
TYPE BoîteAObjets IS

        // synonymes
        SYNONYM Name = String [80] ;

        // variable d'état visible
        propriétaire : Name ;          // variable d'état visible correspondant à
                                       // deux méthodes Rd_propriétaire et
                                       // Wr_propriétaire

        // méthodes
        METHOD mettre (IN REF Objet) ; SIGNALS boîte_pleine ;
                       // ajout d'un élément dans la boîte
        METHOD prendre (OUT REF Objet) ; SIGNALS  boîte_vide ;
                       // prise d'un élément de la boîte
        METHOD nbObjets : Integer ;
                       // rend le nombre actuel d'objets de la boîte
        METHOD concat (IN REF BoîteAObjets) : REF BoîteAObjets ;
                       // Enchaînement de cette boîte avec la boîte désignée
                       // par le paramètre passé
END BoîteAObjets.
```

Remarque :

- La clause **SYNONYM** dans un type ou dans une classe permet d'en donner un équivalent.
- Les identificateurs qui suivent le mot-clé **SIGNALS** dans la signature d'une méthode désignent des exceptions qui peuvent être déclenchées au cours de l'exécution de cette méthode [Lacourte 90].

On peut trouver la syntaxe précise de définition d'un type dans l'annexe 2 de cette thèse ou encore dans [Roisin 90].

2.2.2 CONFORMITE SYNTAXIQUE ENTRE TYPES

Dans les programmes Guide, les objets sont désignés par des **variables références** typées. Soit une variable référence `refObj` de type `T` (dont la déclaration est `refObj : REF T`) ; `T` spécifie le comportement que doivent respecter les objets désignés par `refObj`. Dans les langages de programmation classiques, le contrôle statique de type est basé sur l'égalité des types. Dans les langages comme Smalltalk [Goldberg 85], par contre, il n'y a pas de contrôle statique, et c'est au moment de l'exécution de chaque méthode que la vérification de son existence aura lieu ; même si la méthode à exécuter existe, son exécution peut donner des erreurs inattendues car on ne vérifie pas la compatibilité entre le type des paramètres formels et celui des paramètres effectifs. Dans Guide, nous cherchons à réaliser un contrôle statique, mais nous voulons offrir une contrainte moins forte que l'égalité des types pour qu'une variable référence donnée puisse désigner les objets de différents types, surtout pour exploiter la relation "IS-A" définie par le sous-typage. C'est la raison d'être de **la relation de conformité syntaxique** définie ci-dessous.

La relation de conformité syntaxique spécifie les conditions que doit satisfaire le type de l'objet `O` par rapport au type de la variable référence `refObj`, pour que `refObj` puisse désigner `O`. Elle a été définie dans les langages Trellis/Owl [Schaffert 85], Eiffel [Meyer 88] et Emerald [Black 86b] ; elle spécifie un ensemble minimal de contraintes syntaxiques qui, si elles sont respectées, garantissent la validité de tout accès à `O` par le biais de la variable référence `refObj`.

En reprendant celle définie dans Emerald et en l'étendant aux aspects spécifiques de Guide (paramètres `IN_OUT`, signaux), l'expression formelle de la relation de conformité syntaxique dans Guide est la suivante :

Soient deux types T_r et T_o ; **T_o est conforme à T_r** , c'est-à-dire que tout contrôle syntaxique effectué sur T_r est valide pour T_o, **si et seulement si :**

- pour chaque signature de méthode définie dans T_r, il existe une et seulement une signature dans T_o qui lui est conforme ;

Soient deux signatures op_r et op_o ; **op_o est conforme à op_r,** c'est-à-dire que tout contrôle syntaxique effectué sur op_r est valide pour op_o, **si et seulement si :**

- op_o et op_r ont le même nom symbolique, le même nombre de paramètres d'entrée, le même nombre de paramètres de sortie et le même nombre de paramètres à la fois d'entrée/sortie,
- le type de chaque paramètre d'entrée de op_r est conforme au type du paramètre correspondant de op_o (sens inverse).
- le type de chaque paramètre de sortie de op_o est conforme au type du paramètre correspondant de op_r,
- le type de chaque paramètre à la fois d'entrée/sortie de op_r est égal au type du paramètre correspondant de op_o,
- pour chaque signal déclaré dans op_o, il existe un signal de même nom déclaré dans op_r [Lacourte 90] ;

Grâce à la relation de conformité entre types, une variable référence, même si elle est typée, peut désigner des objets de différents types tout en ayant l'assurance, si la compilation est bien terminée, qu'il n'y aura pas d'erreur à l'exécution concernant l'appel de méthode. Mais chaque fois que l'on affecte une référence d'un objet à une variable référence d'un autre type, on perd une partie de l'interface de l'objet désigné (cf. 2.3.3.3).

Le langage Guide fournit des moyens, les instructions ASSERTYPE (cf. 2.3.3.3) et TYPECASE (cf. 2.3.3.4), pour permettre aux programmeurs de reprendre la vraie interface de l'objet.

Avec sa définition, nous pouvons déduire que la relation de conformité entre types est transitive, c'est-à-dire que si le type c est conforme au type b et si le type b est conforme au type a, alors le type c est aussi conforme au type a.

Dans le cas où le type a est conforme au type b et réciproquement, nous disons que les deux types a et b sont égaux. Cette égalité est dite **égalité de comportement** (ou égalité d'interface puisque les noms de types peuvent être différents) et elle est plus souple que l'égalité symbolique dans les langages classiques.

Remarque :

- Dans cette présentation, nous ne prenons pas en compte la spécification des méthodes ; il est évident que dans la mesure où nous ne disposons actuellement d'aucun moyen pour exprimer et contrôler ces spécifications, leur prise en compte resterait tout à fait symbolique. Le contrôle de types mis en œuvre par le biais de la relation de conformité est donc uniquement syntaxique comme cela a été précisé ci-dessus. Ce contrôle est la tâche principale de la phase d'analyse sémantique du compilateur Guide que nous présentons au chapitre 4 (cf. 4.4.2).
- La relation de conformité entre les types élémentaires est la relation d'égalité (cf. annexe 4 pour le détail).

2.2.3 SOUS-TYPES

Une autre relation entre types dans Guide est la relation de sous-typage qui a été introduite au chapitre 1. Son objectif est de permettre d'exprimer la spécialisation de comportement des objets. La relation de sous-typage dans Guide est simple, c'est-à-dire qu'un type ne peut avoir qu'un seul super-type. Un sous-type hérite de toutes les définitions faites dans son super-type, a la possibilité de surcharger les signatures (modification des types des paramètres) et de compléter le type par de nouvelles définitions de synonymes, de variables d'état visibles ou de signatures de méthodes.

La construction d'un type par sous-typage se fait en indiquant le nom du super-type après le mot clé **SUBTYPE OF** sur la ligne de l'en-tête de sa déclaration (cf. annexe 2). Lorsqu'on ne donne aucune clause de sous-typage dans l'en-tête d'un type, cela signifie que le type est un sous-type du type prédéfini **Top** ; ce dernier est la racine de la hiérarchie des types et donne une interface d'accès minimale à n'importe quel objet.

La règle de surcharge des signatures des méthodes définie pour les sous-types sera donnée au paragraphe 2.2.4.

Exemple :

```
TYPE Personne IS
       âge : Integer ;
       nom : String[20] ;
END Personne.

TYPE Etudiant SUBTYPE OF Personne IS
       université : String[20] ;
END Etudiant.
```

Dans cet exemple, le sous-typage porte uniquement sur des variables d'état visibles. Le type '**Etudiant**' est défini par trois attributs : '**âge**' et '**nom**', hérités du type '**Personne**', et '**université**' qui est un nouvel attribut de ce type.

La notion de sous-typage dans Guide (comme dans les autres langages à objets) est une des possibilités qui permettent la réutilisation de code (il s'agit ici du code source de Guide qui spécifie l'interface d'objets).

2.2.4 RELATION ENTRE SOUS-TYPAGE ET CONFORMITE SYNTAXIQUE

Bien que le rôle du sous-typage et celui de conformité soient différents, l'approche que nous avons choisie est de considérer que, par définition, un sous-type est conforme à son super-type. Cela permet l'utilisation des objets d'un sous-type comme des objets de son super-type, c'est-à-dire qu'ils peuvent être accédés via l'interface d'accès définie par le super-type. La règle de surcharge des signatures dans les sous-types doit donc respecter les conditions fixées par la règle de conformité syntaxique :

- la signature d'une méthode M1 d'un type T1 peut être surchargée, dans un de ses sous-types T2 à condition que la nouvelle signature soit conforme à l'ancienne. En pratique, une signature n'est surchagée que dans le cas où on veut spécialiser le type de paramètres. Dans ce cas, après la règle de conformité, il n'est possible de redéfinir que des types des paramètres de sortie.
- les variables d'état visibles ne peuvent pas être redéfinies car leur redéfinition provoque pratiquement la violation de la règle de surcharge de signatures présentée ci-dessus. Rappelons qu'une variable d'état visible correspond à deux méthodes de lecture et d'écriture sur la même variable retenue dans la classe. En conséquence, redéfinir une variable d'état visible surcharge implicitement les deux méthodes qui lui sont associées. Cependant, la méthode d'écriture a un paramètre IN et on ne peut pas redéfinir son type.

Exemple

```
TYPE Document IS
    METHOD Editer (IN REF Chan);
            // Edition du document
    METHOD Imprimer (IN REF Chan);
            // Impression du document
    METHOD Concat (IN REF Document) : REF Document ;
            // Enchaînement de deux documents
    METHOD Init ;
            // Initialisation du document
END Document.

TYPE Rapport SUBTYPE OF Document IS
    METHOD Edsujet (IN REF Chan);
            // Edition du sujet, le champ spécifique aux objets 'Rapport'
    METHOD Concat (IN REF Document) : REF Rapport ;
            // Enchaînement de deux rapports
END Cercle.
```

Dans l'exemple ci-dessus le type **Rapport** est construit par sous-typage du type **Document**. Dans le type **Rapport** est ajoutée une nouvelle méthode '**Edsujet**' pour l'édition d'un champ spécifique à ses objets. D'autre part, la méthode '**Concat**' est surchargée dans le type **Rapport** pour l'enchaînement de deux rapports. Afin de respecter le fait que la signature surchargée doive être conforme à la signature héritée, on ne peut pas changer le type de paramètre IN de **Document** en **Rapport**.

Le fait de ne pas pouvoir redéfinir le type des paramètres IN est un inconvénient majeur de notre approche concernant la relation entre sous-typage et conformité. Dans Eiffel, par exemple, il n'y a aucune contrainte de surcharge mais on suppose que le sous-type est conforme à ses super-types. Cela provoque surement des erreurs à l'exécution des méthodes.

En résumé, dans Guide un sous-type est toujours conforme à ses sous-types mais le fait qu'un type est conforme à un autre n'implique pas toujours que le premier est un sous-type du dernier. Cela est nécessaire (la conformité doit donc être rendue explicite) pour plusieurs cas :

- *besoin de restriction de l'interface* : un utilisateur a déjà des objets d'un certain type, il veut les rendre accessibles au public en souhaitant que l'accès à ces objets par le public soit limité à certaines méthodes. Dans ce cas, il définit un nouveau type par restriction de l'interface de l'ancien et le donne au public.
- *besoin d'héritage multiple* : dans certains cas, on veut définir un nouveau type en héritant de plusieurs types existants. Guide ne fournit pas cette possibilité et l'utilisateur doit définir le nouveau type en recopiant les méthodes des types existants. Dans ce cas, le nouveau type n'est pas un sous-type des autres mais il est conforme aux autres et ses objets peuvent être utilisés comme des objets des autres types.

2.3 CLASSE

2.3.1 DEFINITION

Une classe décrit une réalisation particulière d'un type. Elle définit un modèle d'objets dont le comportement est celui spécifié par le type qu'elle réalise. Une classe spécifie :

- la structure de données (modèle) de l'état des objets,
- le programme des opérations d'accès (méthodes) à ces objets,
- les conditions d'exécution de ces méthodes.

Toute classe est elle-même un objet de type "**Class**" ; elle est donc manipulée comme telle par l'appel des opérations d'accès spécifiques. En particulier, la méthode **New** permet la génération des objets conformes au modèle défini par la classe ; les classes sont donc dites les constructeurs d'objets du système Guide.

Il est important de rappeler qu'un type donné peut être réalisé par des classes différentes et les instances de ces dernières peuvent cohabiter dans une même application. C'est la motivation de la séparation entre les types et les classes Guide.

Bien que chaque objet ait ses propres valeurs de données, le modèle de ces données et les programmes des méthodes d'accès à ces données sont identiques pour des objets appartenant à une classe donnée. En conséquence, il est naturel de laisser les objets de même classe partager les informations communes. Un des rôles des classes est de garder ces informations communes.

Dans l'exemple ci-dessous, les deux classes '**PetiteBoite**' et '**GrandeBoite**' sont deux réalisations du même type '**BoiteAObjets**' présenté dans le paragraphe 2.2.1.

Exemple.

```
CLASS PetiteBoite
IMPLEMENTS BoiteAObjets IS
// boite ne pouvant pas contenir plus de 5 éléments

        // Attributs des objets
        ATTRIBUTE UNMOVABLE ;

        // variables d'état
        CONST taille : Integer = 5 ;
        nb_elt, dernier, premier : Integer = 0 ;
        contenu : Array [taille] OF REF Objet ;

        // méthodes
        METHOD mettre (IN elt : REF Objet) ; SIGNALS boite_pleine ;
        BEGIN
           IF (nb_elt = taille) THEN RAISE boite_pleine ;
           ELSE
             buffer[dernier]:= elt;
             dernier:= (dernier + 1) MOD taille;
             nb_elt:= nb_elt + 1;
           END
        END mettre ;

        METHOD prendre (OUT elt : REF Objet) ; SIGNALS  boite_vide ;
```

```
        BEGIN
           IF (nb_elt = 0) THEN RAISE boite_vide ;
           ELSE
             elt:= buffer[premier];
             premier:= (premier + 1) MOD taille;
             nb_elt:= nb_elt - 1;
           END
        END prendre ;

        METHOD nbObjets : Integer ;
        BEGIN
              RETURN nb_elt ;
        END nbObjets ;

        METHOD concat (IN boite: REF BoiteAObjets) : REF BoiteAObjets ;
              SIGNALS boite_trop_petite ;
        BEGIN
              ...
        END concat ;

        // clause de contrôle
        CONTROL
              concat, mettre, prendre, nbObjets : EXCLUSIVE;

END PetiteBoite.

CLASS GrandeBoite
IMPLEMENTS BoiteAObjets IS
// boite ne pouvant pas contenir plus de 100 éléments

        // variables d'état
        CONST taille : Integer = 100 ;
        ...                    //suite similaire à la classe PetiteBoite

END GrandeBoite.
```

Dans la déclaration d'une classe, on peut indiquer les caractéristiques communes à tous ses objets en utilisant la clause ATTRIBUTE. On peut actuellement définir l'attribut UNMOVABLE. Un objet UNMOVABLE est un objet fixé à son site de création et qui ne peut pas être déplacé sur un autre site.

2.3.2 MODELE D'ETAT DES OBJETS

Dans la définition d'une classe, la description du modèle d'état des instances est exprimée en termes d'objets. D'après le paragraphe 2.1, l'état d'un objet composé de Guide peut contenir :

- des objets élémentaires (leur propre valeur) désignés par des variables de type élémentaire ;
- des références à d'autres objets composés désignées par des variables appelées **variables références**. Pour les distinguer des variables de type élémentaire et pour la visibilité des programmes, le mot-clé REF est utilisé dans la déclaration des variables références.

La partie de définition du modèle d'état des objets d'une classe contient automatiquement les variables d'état visibles qui ont été déclarées dans le type réalisé par cette classe.

Les variables d'état peuvent être déclarées de trois façons : soit comme des constantes (mot-clé **CONST**), soit comme des variables initialisées, soit comme des variables non initialisées.

Exemple : Dans l'exemple de définition de la classe `PetiteBoîte` présenté en 2.3.1, on a défini cinq variables d'état :

```
// variables d'état
CONST taille : Integer = 5 ;
nb_elt, dernier, premier : Integer = 0 ;
contenu : Array [taille] OF REF Objet ;
```

- une constante qui s'appelle `taille`,
- trois variables initialisées qui s'appellent respectivement `nb_elt`, `dernier`, `premier`,
- et une variable non-initialisée qui s'appelle `contenu`.

Bien que les cinq variables soient de type élémentaire, les objets de la classe `PetiteBoîte` contiennent aussi des références à d'autres objets (car la variable `contenu` est un tableau de références).

2.3.3 PROGRAMME DES OPERATIONS

Le programme de chaque opération d'accès aux instances d'une classe doit être soit hérité de sa super-classe (cf. 2.3.5), soit défini dans le corps de la classe, préfixé par le mot-clé **METHOD**. On peut aussi définir des procédures internes (préfixées par le mot-clé **PROCEDURE**) dans une classe pour la factorisation du code à l'intérieur de la classe : une procédure peut être utilisée (appelée) plusieurs fois par une ou plusieurs méthodes (ou procédures) définies dans la même classe. Les procédures sont définies au même niveau que les méthodes. Dans un souci de simplicité, Guide ne fournit pas la possibilité d'imbrication des procédures dans les méthodes (ni dans les procédures) .

Les instructions dans le corps des méthodes et des procédures sont soit des instructions classiques :

- affectation,
- appel de procédure,
- structure de contrôle : **IF**, **CASE**, **FOR**, **WHILE**, **REPEAT**, etc.
- instruction de retour.

soit des instructions spécifiques au modèle :

- appel d'une opération sur un objet,
- contrôle dynamique des types par les instructions **ASSERTYPE** et **TYPECASE**,
- bloc d'exécution parallèle **CO_BEGIN** ... **CO_END**,
- déclenchement et traitement des exceptions (**RAISE**, **EXCEPT**).

Dans les paragraphes suivants nous allons présenter les instructions spécifiques dont le processus de compilation est présenté au chapitre 4.

2.3.3.1 Affectation de variables

L'affectation de variable est l'instruction la plus utilisée dans les programmes d'application. Dans Guide on peut distinguer deux cas d'affectation différents selon que la variable affectée est de type élémentaire ou de type construit.

a. Variable de type élémentaire : Comme la variable de type élémentaire désigne la valeur de l'objet élémentaire, l'affectation des variables de ce genre consiste à copier la valeur de l'objet.

b. Variable référence : Comme la variable référence désigne la référence de l'objet et non pas l'objet lui-même, l'affectation des variables de ce genre consiste à copier la valeur de la référence.

2.3.3.2 Appel de méthode

L'appel de méthode sur un objet est l'élément clé d'un langage à objets tel que Guide. C'est une forme d'envoi de message utilisée dans Smalltalk [Goldberg 85]. Pour faire un appel de méthode, on doit préciser la référence à l'objet sur lequel la méthode s'exécute, le nom de la méthode et les paramètres à passer :

```
<appel de méthode>  =  <variable_référence>'.'<ident_méthode>
                       '('<liste_paramètres_effectifs> ')'
```

Au niveau de la sémantique, les paramètres sont tous passés par valeur : valeur de l'objet élémentaire ou valeur de la référence à un objet construit. Ce choix est naturel car les objets dans Guide sont indépendants et le site d'exécution de la méthode appelée peut être différent de celui de l'appelant (par exemple dans le cas où l'objet sur lequel la méthode appelée va s'exécuter se situe sur un autre site).

Le contrôle de la validité de l'appel est fait statiquement à la compilation (cf.4.3) : on vérifie si la méthode appelée est bien définie dans le type de la variable référence et si la liste des paramètres effectifs est compatible ou non avec celle définie dans la méthode appelée.

L'appel de méthode est considéré comme une expression. Si la méthode possède une valeur de retour, l'appel de méthode peut être combiné avec d'autres expressions pour la construction de nouvelles expressions plus complexes. Sinon l'appel de méthode est utilisé tout seul et devient une instruction Guide.

2.3.3.3 Instruction ASSERTYPE

Grâce à la relation de conformité entre types, on peut affecter des références d'objets de types différents à une variable référence donnée à condition que les types des objets soient tous conformes au type de la variable référence. Chaque fois qu'une affectation de ce genre a lieu, l'interface d'accès à cet objet via la variable référence diminue. Pour pouvoir augmenter l'interface d'accès (ou reprendre l'interface d'origine ou réelle de l'objet), on utilise l'instruction ASSERTYPE qui s'écrit :

```
<variable_référence> ':=' ASSERTYPE '(' <expression> ')'
```

où <expression> est une expression dont la valeur est une référence, le type effectif de cette référence n'est pas connu à la compilation et on doit contrôler dynamiquement la conformité entre lui et le type de la variable référence à l'exécution.

Exemple : Avec les types Document et Rapport définis en 2.2.4, on peut écrire les codes suivants :

```
refTop : REF Top ;
refDocument : REF Document ;
refRapport : REF Rapport ;
...
refRapport := Rapport.New ;                    // (1)
...
refDocument := refRapport ;                    // (2)
...
refTop := refDocument ;                        // (3)
...
refDocument := ASSERTYPE (refTop) ;            // (4)
...
refRapport := ASSERTYPE (refDocument) ;        // (5)
```

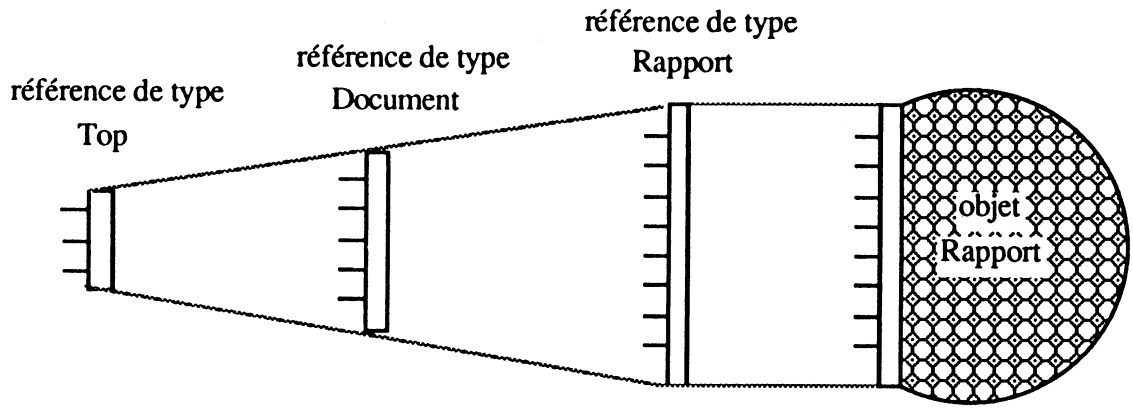
Après l'exécution de (1), un objet Rapport est créé et sa référence est affectée à la variable refRapport. Comme refRapport est aussi de type Rapport, on peut accéder à toute méthode de l'objet Rapport via refRapport.

Après l'exécution de (2), la variable refDocument contient une référence à l'objet Rapport, mais via cette variable, on ne peut accéder qu'à des méthodes de l'objet Rapport définies dans le type Document. De même, après l'exécution de (3), la variable refTop contient une référence à l'objet Rapport et via cette variable, on ne peut accéder qu'à des méthodes de l'objet Rapport définies dans le type Top.

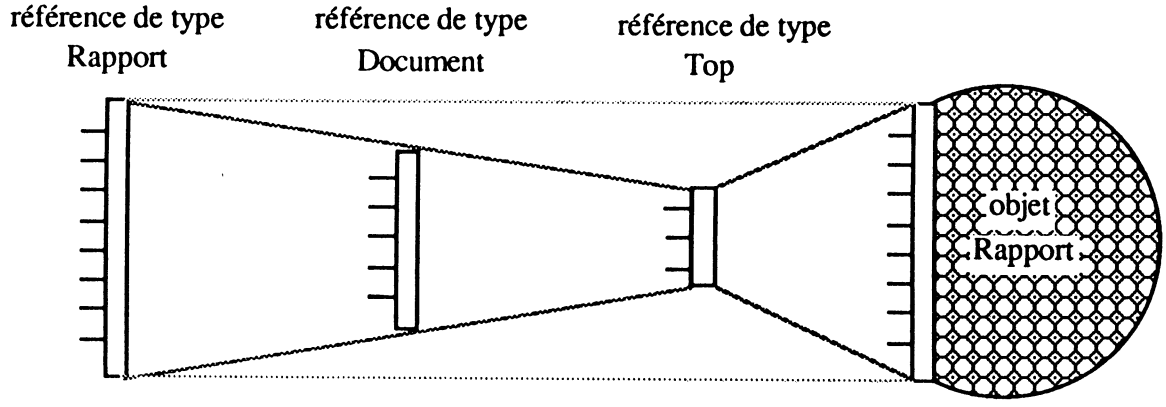
Supposons maintenant qu'une seule référence à l'objet Rapport existe, celle contenue dans la variable refTop. Comment peut on accéder à des méthodes de l'objet Rapport qui ne sont pas définies dans le type de refTop (type Top) ? Avec le contrôle statique de type basé sur la relation de conformité, on ne peut rien faire. C'est le rôle des instructions ASSERTYPE et TYPECASE (cf. 2.3.3.4) de forcer un contrôle dynamique afin de permettre de retrouver l'interface voulue en garantissant son utilisation correcte.

Dans l'exemple ci-dessus, le compilateur ne peut pas savoir si les affectations (4) et (5) sont correctes ou non, il génère donc le code pour forcer un contrôle dynamique à l'exécution.

Le schéma 2.1 nous permet de mieux visualiser l'effet du contrôle statique et dynamique des types dans Guide. Il faut remarquer que le contrôle dynamique doit être demandé explicitement par le programmeur. A l'exécution, lorsque le contrôle n'est pas correct, une exception système est déclenchée (le signal **CONFORMANCE_ERROR**).



L'effet du contrôle statique



L'effet du contrôle dynamique

Figure 2.1 : L'effet des contrôles statique et dynamique

2.3.3.4 Instruction TYPECASE

Pour demander un contrôle dynamique plus souple que celui de l'instruction ASSERTYPE, l'instruction TYPECASE spécifie la sélection et l'exécution d'une suite d'instructions en fonction du type réel de l'objet désigné (au moment de l'exécution) par la variable référence donnée comme sélecteur. Cette instruction s'écrit :

```
TYPECASE <variable_référence> OF
      <ident_type_1> ':' <liste_instructions> END ';'
      <ident_type_2> ':' <liste_instructions> END ';'
      ...
      <ident_type_n> ':' <liste_instructions> END ';'
      OTHERS ':' <liste_instructions>
END
```

Le principe de sélection et d'exécution de cette instruction est :

- vérification séquentielle de l'égalité entre le type réel de l'objet référencé par la variable référence et un des types donnés ;

- exécution de la liste d'instructions associée au premier type qui satisfait la vérification ;
- si aucun type ne satisfait la vérification et si la clause OTHERS est présente, la liste d'instructions associée est exécutée.
- dans chaque branche d'instructions associée à un type, le sélecteur (variable référence) est considéré comme étant du type en question.

Exemple : Si on continue l'exemple en 2.3.3.3, on peut écrire les codes suivants :

```
// refTop est une variable référence de type Top
// mais elle peut désigner des objets de type Document ou Rapport
...
TYPECASE refTop OF
      Rapport : /* traite refTop comme étant de type Rapport */
                END ;
      Document : /* traite refTop comme étant de type Document */
                END ;
      OTHERS : /* traite refTop comme étant de type Top */
END
...
```

2.3.3.5 Bloc d'exécution parallèle CO_BEGIN ... CO_END

L'instruction CO_BEGIN ... CO_END permet de lancer plusieurs activités en parallèle. L'activité qui exécute l'instruction CO_BEGIN ... CO_END est dite l'activité mère et elle est suspendue pendant l'exécution de cette instruction. Chaque activité fille créée va exécuter une méthode sur un objet donné. Le bloc d'exécution parallèle CO_BEGIN ... CO_END s'écrit :

```
CO_BEGIN
     <ident_activité_1> ':' <appel_de_méthode> ';'
     <ident_activité_2> ':' <appel_de_méthode> ';'
     ...
     <ident_activité_n> ':' <appel_de_méthode> ';'
CO_END [<expression_booléenne>]
```

où l'expression <expression_booléenne>, si elle existe, est une expression booléenne qui spécifie sous quelles conditions l'instruction se termine. Cette expression se constitue des identificateurs des activités créées et d'opérateurs logiques. Chaque identificateur d'une activité est considéré comme une valeur booléenne qui exprime l'état de l'activité correspondante (TRUE si l'activité est terminée et FALSE sinon).

2.3.4 CONDITION D'EXECUTION DES METHODES SUR LES OBJETS

Un objet est potentiellement partageable par un nombre quelconque d'activités qui exécutent ses méthodes, ces dernières sont les seules qui peuvent manipuler directement l'état de l'objet. En conséquence, le contrôle de l'exécution des méthodes est un moyen évident pour synchroniser l'accès aux objets depuis des activités, et pour maintenir la cohérence des données à l'intérieur d'un objet. Pour des raisons de simplicité, Guide fournit seulement ce moyen de synchronisation entre des activités.

A chaque méthode on associe une **condition d'activation** ; cette dernière permet d'exprimer sous quelles conditions la méthode définie sur un objet peut être exécutée [Decouchant 88a]. Elle est exprimée sous la forme d'une expression booléenne fonction des variables d'état, des

paramètres d'entrée de la méthode et de compteurs associés à la méthode [Robert 77]. Les compteurs définis pour chaque objet et chaque méthode sont les suivants :

- invoked(m) : nombre total d'appels de la méthode depuis le début de la vie de l'objet,
- started(m) : nombre d'appels de la méthode qui ont été acceptés (non bloqués),
- completed(m) : nombre d'exécutions terminées,
- pending(m) : nombre d'activités qui sont en attente d'exécution de la méthode,
- current(m) : nombre d'appels en cours d'exécution.

Les trois premiers compteurs (**invoked, started, completed**) font partie de l'état de l'objet ; ils sont créés et initialisés à 0 à la création de l'objet. Leur valeur ne peut que croître au cours de la vie de l'objet et le problème de débordement est résolu en remettant à zéro les trois compteurs quand ils sont tous égaux (à une valeur donnée). Les deux autres compteurs (**pending** et **current**) sont calculés à partir des premiers avec les relations :

```
pending(m) = invoked(m) - started(m),
current(m) = started(m) - completed (m).
```

La condition d'activation d'une méthode pourrait être mise dans la méthode en question, mais dans Guide nous préférons regrouper toutes les conditions d'activation des méthodes d'une même classe en une seule entité (une clause de contrôle) et nous la mettons à la fin de la classe en question.

Pour les conditions utilisées les plus fréquemment, un certain nombre de mot-clés les représentant sont offerts aux programmeurs :

- **EXCLUSIVE** pour désigner la condition d'exclusion mutuelle, c'est-à-dire : $\sum \text{current}(m_i) = 0$. Une méthode possédant cette condition d'activation ne peut s'exécuter sur un objet que lorsqu'aucune méthode (y compris elle) ne s'exécute sur l'objet en question. En conséquence, dans le code d'une méthode **EXCLUSIVE**, on ne peut pas appeler une autre méthode **EXCLUSIVE** directement ou indirectement sur l'objet en question. Sinon, un interblocage se produit. Actuellement, il n'y a aucun mécanisme de détection des appels de ce genre. Par conséquent, le programmeur doit faire très attention pour éviter ce problème d'interblocage.
- **READER** pour désigner que la méthode associée sera utilisée avec une politique de lecture/écriture sur l'ensemble de toutes les variables d'état de l'objet. Une méthode ayant la caractéristique READER ne s'exécute sur un objet que quand aucune méthode ayant la caractéristique WRITER ne s'exécute ;
- **WRITER** pour désigner que la méthode associée ne s'exécute sur un objet que quand aucune méthode WRITER (y compris elle) et READER ne s'exécute. Comme les méthodes EXCLUSIVE, une méthode WRITER ne peut pas appeler une méthode WRITER/READER sur le même objet.

Il est important de remarquer que les méthodes contrôlées par READER/WRITER sont des méthodes respectant la politique de lecture/écriture sur l'ensemble de toutes les variables d'état de l'objet. Pour réaliser le contrôle sur les méthodes qui respectent la politique de lecture/écriture seulement sur certaines variables d'état, on doit utiliser des expressions booléennes explicites.

Lorsqu'il n'y a pas de clause de contrôle définie pour une méthode donnée, il n'y a aucune contrainte de synchronisation pour l'accès à cette méthode.

Remarque : Avec le mécanisme de synchronisation présenté précédemment, on constate certaines limitations suivantes :

- il n'y a pas de condition d'activation pour les procédures internes. Cela provient du fait que les procédures internes sont appelées directement par les méthodes de même classe sans passer par le noyau. Si l'on veut associer une condition d'activation à une procédure (cela peut se produire en pratique), on doit prendre une des solutions suivantes :
  + soit on redéfinit la procédure comme une méthode ;
  + soit on ajoute la condition d'activation de la procédure à celle des toutes les méthodes qui l'appellent.
- toutes les méthodes de même nom d'une même classe sont contrôlées par une seule condition d'activation.

Exemple : Dans l'exemple de la définition de la classe PetiteBoîte présenté en 2.3.1, on a utilisé les signaux pour indiquer le fait que la boîte était pleine ou vide afin de laisser l'appelant traiter ces cas. Nous donnons ci-dessous une autre définition de la classe PetiteBoîte en utilisant les clauses d'activation. Avec ces clauses de contrôle, si la boîte est pleine lorsque la méthode mettre est appelée, cette dernière doit être bloquée (par le noyau) en attendant l'extraction d'un objet de la boîte. De même, dans le cas où la boîte n'est pas pleine, la méthode mettre n'est pas exécutée tant que la méthode prendre ou une autre session de mettre est en cours d'exécution. Pour la méthode prendre, on a aussi une situation similaire.

```
CLASS PetiteBoîte
IMPLEMENTS BoîteAObjets IS
// boîte ne pouvant pas contenir plus de 5 éléments

    // Attributs des objets
    ATTRIBUTE UNMOVABLE ;

    // variables d'état
    CONST taille : Integer = 5 ;
    nb_elt, dernier, premier : Integer = 0 ;
    contenu : Array [taille] OF REF Objet ;

    // méthodes
    METHOD mettre (IN elt : REF Objets) ;
    BEGIN
       buffer[dernier]:= elt;
       dernier:= (dernier + 1) MOD taille;
       nb_elt:= nb_elt + 1;
```

```
   END mettre ;

   METHOD prendre (OUT elt : REF Objet) ;
   BEGIN
      elt:= buffer[premier];
      premier:= (premier + 1) MOD taille;
      nb_elt:= nb_elt - 1;
   END prendre ;

   METHOD nbObjets : Integer ;
   BEGIN
         RETURN nb_elt ;
   END nbObjets ;

   METHOD concat (IN boite: REF BoiteAObjets) : REF BoiteAObjets ;
   BEGIN
         ...
   END concat ;

   // clause de contrôle
   CONTROL
    mettre: (nb_elt < taille) AND current (mettre) = 0 AND current (prendre) = 0;
    prendre: (nb_elt > 0) AND current (prendre) = 0 AND current (prendre) = 0;

END PetiteBoite.
```

2.3.5 HERITAGE

Parallèlement à la relation de sous-typage, Guide fournit une relation hiérarchique entre les classes par l'héritage. Une classe C2 peut être construite par héritage d'une classe C1 si C2 réalise un type qui est un sous-type du type réalisé par C1(fig. 2.2). La possibilité de construction des classes par héritage permet d'exprimer le partage de propriétés physiques entre des objets qui partagent déjà un même comportement. Dans la version actuelle du langage, le mécanisme d'héritage mis en œuvre est simple, c'est-à-dire qu'une classe a une et une seule super-classe, à l'exception de la classe **Top** qui est la racine de la hiérarchie des classes et qui n'a pas de super-classe. La règle de construction de classes par héritage est la suivante :

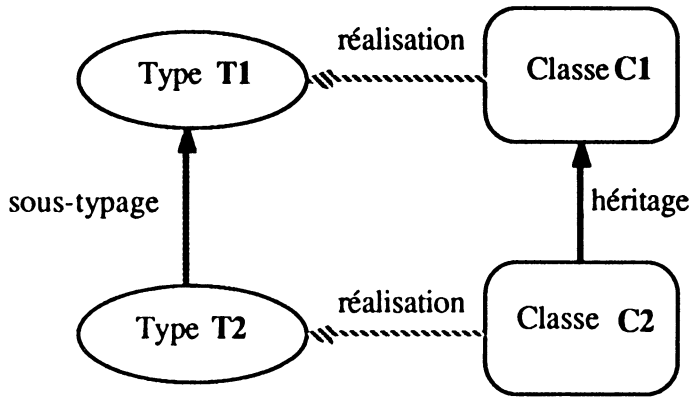


Figure 2.2 : Sous-typage et héritage

Soit deux types T1 et T2 tels que T2 est un sous-type de T1 ;

Soit C1 une classe qui réalise T1 ;

Une classe C2 qui réalise T2 est construite par héritage de C1 de la façon suivante :

- C2 hérite du modèle d'état des instances de C1,
- C2 a la possibilité de compléter ce modèle d'état hérité par la définition de nouvelles données propres à ses instances,
- C2 hérite de toutes les méthodes (mais non pas des procédures internes) définies dans C1, ainsi que de leur clauses de synchronisation,
- C2 a la possibilité de redéfinir une méthode héritée en la surchargeant (la surcharge d'une méthode `m1` consiste à la remplacer par une méthode `m'1` qui lui est conforme),
- C2 a la possibilité de redéfinir une clause de synchronisation héritée en la remplaçant par une nouvelle,
- C2 a la possibilité de définir de nouvelles méthodes propres à ses instances ; ces méthodes sont celles qui sont spécifiées dans `T2` et qui n'apparaissent pas dans `T1`.

Les objets de classe C2 partagent avec ceux de classe C1 une partie du modèle de leur état (modèle hérité) et toutes les méthodes héritées sans surcharge par C2. De plus, il est toujours possible de demander l'exécution d'une méthode de la classe mère (à partir des programmes des méthodes de la sous-classe) : on utilise pour cela le mot-clé SUPER comme en Smalltalk-80.

Comme le sous-typage entre types permet la réutilisation des spécifications, l'héritage des classes est une des possibilités qui permet la réutilisation du modèle de données et de code des méthodes.

2.4 OBJETS

Un objet est un exemplaire d'une classe. Bien que les caractéristiques des objets aient été présentées au paragraphe 2.1, nous résumons ici les différences entre les catégories d'objets :

- **un objet composé** est une entité autonome, il est permanent au sens où sa durée de vie est indépendante de tout mécanisme d'exécution et est potentiellement illimitée. Dans les programmes, les objets composés sont gérés par un service particulier du système Guide appelé **Système de Gestion d'Objets** ou SGO [Guide-R4]. Un objet composé est créé explicitement par appel de la méthode `New` définie sur sa classe. Il ne peut être désigné que par des **variables références** (typées par un type construit) qui contiennent les informations nécessaires au système pour connaître la localisation de l'objet ;
- **un objet élémentaire** (primitif ou structuré) est toujours interne à l'état d'un autre objet composé ; il n'est pas connu du SGO et il n'est visible et accessible que dans le corps de l'objet qui le contient ; un tel objet a une durée de vie égale à celle de son objet englobant ; il est créé implicitement à la création de l'objet englobant et est désigné par une variable de type élémentaire ;
- **les données de travail** d'une méthode ou d'une procédure sont définies dans le corps de la méthode ou de la procédure. Elles sont réservées implicitement dans la pile

d'exécution chaque fois que la méthode ou la procédure est appelée et détruites à la fin de l'exécution de cette dernière. Les données de travail peuvent contenir des objets élémentaires et des références à des objets. La durée de vie des données de travail est égale au temps d'exécution de la méthode ou de la procédure qui les définit. Les données de travail d'une méthode ou d'une procédure appartiennent à l'état de l'activité dans laquelle s'exécute la méthode ou la procédure.

2.4.1 Création d'objets

Tout objet doit être créé avant d'être utilisé. Le seul moyen de création des objets est l'appel à la méthode `New` sur leur classe. En fait, la création d'un objet est faite dans une activité et l'objet créé est normalement utilisé dans cette activité de création. Pour rendre l'objet accessible aux autres activités, l'activité de création doit :

- soit associer un nom symbolique à l'objet (en appelant le Serveur de noms symboliques) ;
- soit affecter la référence de l'objet à une variable d'état d'un autre objet ; ce dernier est alors accessible par les autres activités ;
- soit passer en paramètre la référence de l'objet.

2.4.2 Partage d'objets et Accès aux objets

Un objet peut être partagé par plusieurs programmes d'application, ou plus précisément par plusieurs activités qui exécutent les programmes. La référence à des objets est le seul moyen de désigner les objets et d'y accéder (cf. 3.5). En possédant la référence à un objet, on peut accéder à ce dernier en appelant une des méthodes définies sur lui. Avant qu'une méthode puisse s'exécuter dans une activité, les conditions d'activation définies sur elle doivent être satisfaites. Ces conditions sont le seul moyen de synchronisation entre plusieurs activités qui partagent les objets.

2.4.3 Destruction d'objets

Chaque objet possède une méthode spéciale, la méthode `Destroy` définie dans le type `Top`, qui demande au système de libérer la place dans la mémoire d'objets occupée par l'objet. Si l'utilisateur utilise cette méthode sans aucune précaution, un objet peut en désigner un autre qui a été détruit. Il est donc déconseillé d'utiliser cette méthode, surtout sur les objets partagés par plusieurs programmes d'applications différents. Seul le ramasse-miettes, un service du système (cf. chapitre 6), peut utiliser la méthode `Destroy` pour ramasser l'espace occupé par les objets inutiles sans faire passer la mémoire d'objets dans un état incohérent.

2.5 GENERICITE : LES CONSTRUCTEURS DE TYPES ET DE CLASSES

L'introduction de la généricité dans un langage offre au programmeur un moyen de factoriser des traitements en les paramétrant par les types des données sur lesquels ils s'appliquent. Dans le langage Guide, la généricité permet de définir des constructeurs de types et de classes paramétrés qui seront utilisés pour définir des types et des classes Guide.

Le langage Guide offre un mécanisme de généricité non-contrainte (le paramètre formel de généricité n'est pas typé) et un mécanisme de généricité contrainte (le paramètre formel est typé de façon à imposer et contrôler statiquement que le paramètre effectif est conforme au type du paramètre formel). L'intérêt de la généricité contrainte est de permettre d'accéder à l'interface des objets désignés par les variables de type générique (les variables déclarées dans un type ou une classe générique et dont le type est un des paramètres de généricité).

Dans les constructeurs de classes, on permet en outre l'utilisation des paramètres de dimensionnement de type `Integer` initialisés lors de l'instanciation des instances (classes) de cette classe générique.

Pour des raisons de simplicité, le super-type (la super-classe) de tout type (toute classe) générique est toujours le type (la classe) `Top`, il n'y a pas de hiérarchie de types et de classes génériques.

2.5.1 GENERICITE SUR LES PARAMETRES DE TYPE

Généricité non-contrainte

Dans ce type de généricité, le type du paramètre formel de généricité n'est pas indiqué et le type du paramètre effectif de généricité est un des types offerts par le langage.

Dans cette forme de généricité, on ne peut pas appliquer d'appel de méthode sur les variables de type générique (l'appel de méthode n'est pas défini sur le type `Integer` par exemple, et l'on ne connaît pas a priori le type du paramètre effectif), mais on peut appliquer sur ces variables les opérations définies sur tous les types : affectation, égalité, différence, comparaison (:=, =, #).

Lors de la déclaration d'une variable référence typée par un type instancié d'un type générique, on doit préciser les paramètres effectifs de généricité.

Exemple :

```
TYPE CONSTRUCTOR List OF [ T ] IS
  // T est l'identificateur de substitution
  METHOD Insert (IN o : T);
  METHOD Delete : T;
  ...
END List.

CLASS CONSTRUCTOR List OF [ T ] IMPLEMENTS List IS
  // T est l'identificateur de substitution
```

```
  CONST max : Integer = 100;
  list : Array[max] OF T;
  dernier : Integer = 0;
  METHOD Insert (IN o : T);
  BEGIN
    list[dernier] := o;     // o est une variable de type générique
    dernier := dernier+1;
  END Insert;

  METHOD Delete : T;
  ...
END List.

/* exemple d'utilisation */
r : REF List OF Integer;
s : REF List OF REF Document;
r := List.New;
s := List.New;
```

Généricité contrainte

Dans ce type de généricité, le type du paramètre formel de généricité doit être indiqué et lors de l'instanciation d'un type générique contraint, le type effectif de généricité doit être conforme au type formel. Une des conséquences de ce choix est que l'on peut appliquer sur les variables de type générique les méthodes définies pour ce type. Comme il n'est pas possible de définir des sous-types pour les types élémentaires, cette forme de généricité ne s'applique que pour les types construits (les paramètres de généricité sont de la forme `REF aType`).

Exemple :

```
TYPE Document IS
nom : String[20];
END Document.

TYPE CONSTRUCTOR Collection OF [ T : REF Document ] IS
  // T est l'identificateur de substitution ;
  METHOD Search (IN titre : String) : T;
  METHOD Delete : T;
  ...
END Collection.

CLASS CONSTRUCTOR Collection OF [ T : REF Document ]
IMPLEMENTS Collection IS
  // T est l'identificateur de substitution
  CONST max : Integer = 100;
  contenu : Array[max] OF T;
  dernier : Integer = 0;
```

```
  METHOD Search (IN titre : String) : T;
  i : Integer;
  BEGIN
    i := 0;
    WHILE (titre # contenu[i].Titre AND i<dernier) DO
/* connaissant le type du paramètre formel de généricité, on peut
appliquer sur la variable une méthode implémentée sur ce type. Titre est
une méthode du type Document qui renvoie le titre du document. */
      i := i+1;
    END;
    IF i # dernier THEN
      RETURN contenu[i];
    END;
  END Search;

  METHOD Delete : T;
  ...
END Collection.

/* exemple d'utilisation */
r : REF Collection OF REF Rapport;
/* Collection est un type générique qui utilise la généricité contrainte par le
type Document. Le type du paramètre effectif peut être n'importe quel type qui
lui est conforme, comme ici Rapport. */

r := Collection.New; // création d'un objet collection de rapports.
```

Remarque :

- Il n'y a aucun rapport entre la généricité et la relation de conformité. Par exemple, si le type `Rapport` est conforme au type `Document`, on ne peut pas déduire de la relation de conformité que le type `List OF REF Rapport` est conforme au type `List OF REF Document`.
- La forme de généricité non-contrainte ou contrainte s'applique à chaque type formel de généricité. Un type générique peut avoir éventuellement plusieurs types formels de généricité, chacun pourrait être déclaré comme un paramètre non-contraint ou contraint.

2.5.2 GENERICITE SUR LES PARAMETRES DE DIMENSIONNEMENT

Le paramètre de dimensionnement apparaît uniquement dans la déclaration de la classe générique et il est de type `Integer`.

Une classe générique peut implémenter soit un type non générique, et dans ce cas, le seul paramètre de généricité est un paramètre de dimensionnement, soit un type générique (avec des paramètres de généricité portant sur les types).

La règle d'instanciation des classes, instances des classes génériques, est identique à celle des types.

Exemple :

```
TYPE Tableau OF [ T ] IS
  METHOD Put (IN i : Integer; o : T); SIGNALS dépassement_de_borne;
  METHOD Get (IN i : Integer) : T; SIGNALS dépassement_de_borne;
END Tableau.

CLASS CONSTRUCTOR Tableau [taille] OF [ T ] IMPLEMENTS Tableau IS
 // taille est un identificateur de substitution ;
  contenu : Array[taille] OF T;

  METHOD Put (IN i : Integer; o : T); SIGNALS dépassement_de_borne;
  BEGIN
    IF (i < 0 AND i >= taille) then RAISE dépassement_de_borne;
    ELSE contenu[i] := o ;
    END;
  END Put;

  METHOD Get (IN i : Integer) : T; SIGNALS dépassement_de_borne;
  BEGIN
    IF i < 0 AND i >= taille THEN RAISE dépassement_de_borne;
    ELSE RETURN contenu[i];
    END ;
  END Get;

END Tableau.

TYPE Chaine IS
  METHOD Put (IN i : Integer; o : Char); SIGNALS dépassement_de_borne;
  METHOD Get (IN i : Integer) : Char; SIGNALS dépassement_de_borne;
END Chaine.

CLASS CONSTRUCTOR Chaine [taille] IS
 // taille est un identificateur de substitution ;
  contenu : String[taille];

  METHOD Put (IN i : Integer; o : Char); SIGNALS dépassement_de_borne;
  BEGIN
    IF i < 0 AND i >= taille then RAISE dépassement_de_borne;
    ELSE contenu[i] := o;
    END;
  END Put;

  METHOD Get (IN i : Integer) : Char; SIGNALS dépassement_de_borne;
  BEGIN
    IF i < 0 AND i >= taille then RAISE dépassement_de_borne;
    ELSE RETURN contenu[i];
    END;
```

```
  END Get;

END Chaine.

/* exemple d'utilisation */
r : REF Tableau OF Integer;
r := Tableau[100].New;
s : REF Chaine;
s := Chaine[100].New;
```

Remarque : Dans Guide, on utilise les classes uniquement pour créer des objets. La création d'objets est souvent faite dans les affectations (voir l'exemple précédent) et dans ce cas, on peut supprimer les types effectifs de généricité dans l'instance (classe) de la classe générique, ceux de type générique de la variable référence du côté gauche de l'affectation sont utilisés.

CHAPITRE 3 : ARCHITECTURE DE LA MEMOIRE D'OBJETS GUIDE

On trouvera dans [Guide-R4], [Guide-R6], [Scioville 89] ainsi que dans [Freyssinet 90] une description détaillée du modèle et de la réalisation de la mémoire d'objets de Guide. Dans ce chapitre nous rappelons seulement les notions nécessaires à la présentation des algorithmes de ramasse-miettes décrits au chapitre 6.

3.1 ARCHITECTURE GENERALE DE LA MEMOIRE D'OBJETS GUIDE

Guide est un système à objets dans le sens où les objets sont en même temps les unités d'exécution et les unités de conservation de données. De ce fait résulte la double fonction que doit remplir la mémoire d'objets : supporter l'exécution des méthodes des objets et garantir la conservation du contenu des objets jusqu'à leur destruction qui peut être faite soit explicitement par les utilisateurs, soit automatiquement par le ramasse-miettes.

Le support d'exécution consiste, d'une part, à permettre et même faciliter aux entités du modèle d'exécution (activités et domaines, cf. 1.2) l'accès aux objets pour l'exécution de leurs méthodes et, d'autre part, à prendre en compte les conséquences de ces exécutions. Il se peut, en effet, que l'état d'un objet soit modifié ou même que sa taille change à la suite de l'exécution d'une de ses méthodes.

Le fait qu'un objet soit persistant n'implique pas qu'une fois créé, il soit simplement et indéfiniment conservé en mémoire d'objets. Celle-ci doit le tenir prêt à toute manipulation, en particulier à celles qu'entraîne l'éventuelle exécution d'une de ses méthodes. De même, la mémoire doit intégrer dans son état, de manière permanente, les modifications qui résultent de cette exécution.

Etant donné qu'une exécution de méthode ne peut avoir lieu que dans une mémoire principale et que la conservation ne peut être garantie que par une mémoire secondaire, la mémoire d'objets Guide est naturellement découpée en deux niveaux :

1. Le support des objets en mémoire principale, ou Mémoire Virtuelle d'Objets (MVO), qui permet l'exécution des méthodes.
2. Le support des objets en mémoire secondaire, ou Mémoire Permanente d'Objets (MPO), qui réalise la conservation.

Avec ces deux niveaux, le terme '**objet**' désigne désormais un concept abstrait, correspondant en réalité à deux entités différentes : la représentation de l'objet en MVO, qu'on appelle **image**, et la représentation de l'objet en MPO, appelé **objet stocké**. Sur une image s'exécutent effectivement les méthodes de l'objet. Etant rangé en mémoire secondaire, un objet stocké conserve l'état de l'objet correspondant.

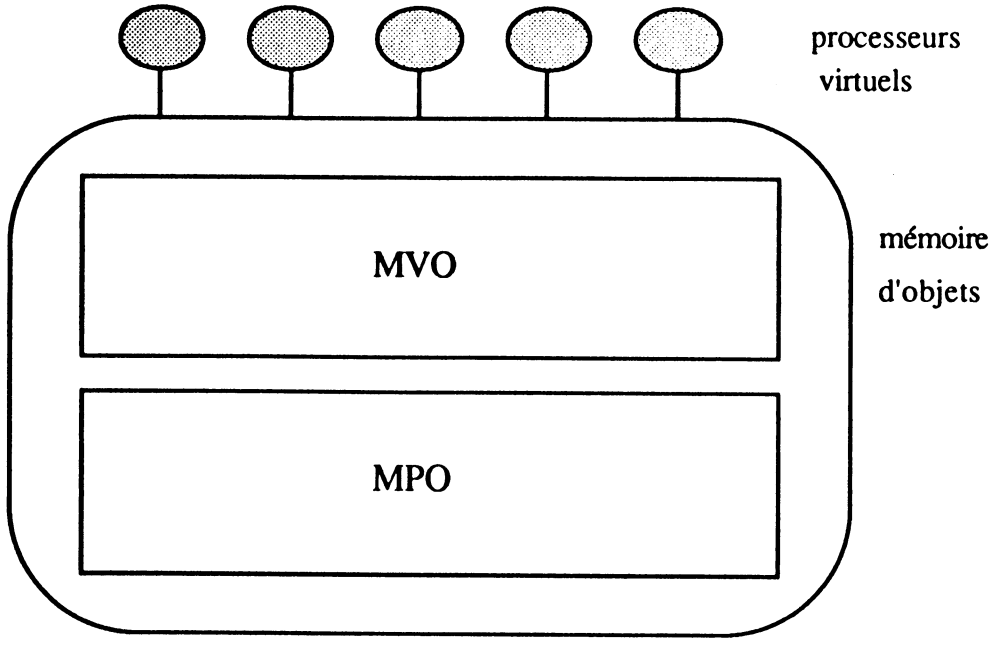


Figure 3.1 : Les deux niveaux de la mémoire d'objets Guide

Afin de permettre d'identifier un objet à l'intérieur du système, on lui associe un identificateur universel et unique nommé `identificateur d'objet` (ou `oid`). Il est attribué lors de la création de tout objet.

Lorsqu'une activité appelle une des méthodes définies sur un objet, une image de ce dernier est créée en chargeant en MVO l'objet stocké correspondant. L'image reste en MVO pendant son exécution ou jusqu'à ce que, pour des raisons qu'on analysera plus tard (cf. 3.4), on décide de la détruire. Auparavant, si l'image a été modifiée à la suite de l'exécution de la méthode, elle doit être rangée en MPO de manière à remplacer l'ancien contenu de l'objet stocké. Celui-ci n'est pas détruit, ce qui garantit que l'objet correspondant est conservé.

La figure 3.2 montre les représentations d'un objet et les échanges qui ont lieu entre la MVO et la MPO pour leur mise en place.

Le chargement d'un objet stocké et le rangement d'une image constituent les principales interactions entre la MVO et la MPO. Cette interface est interne à la gestion de la mémoire utilisée par le noyau. Elle n'est donc pas visible aux autres niveaux de la machine à objets.

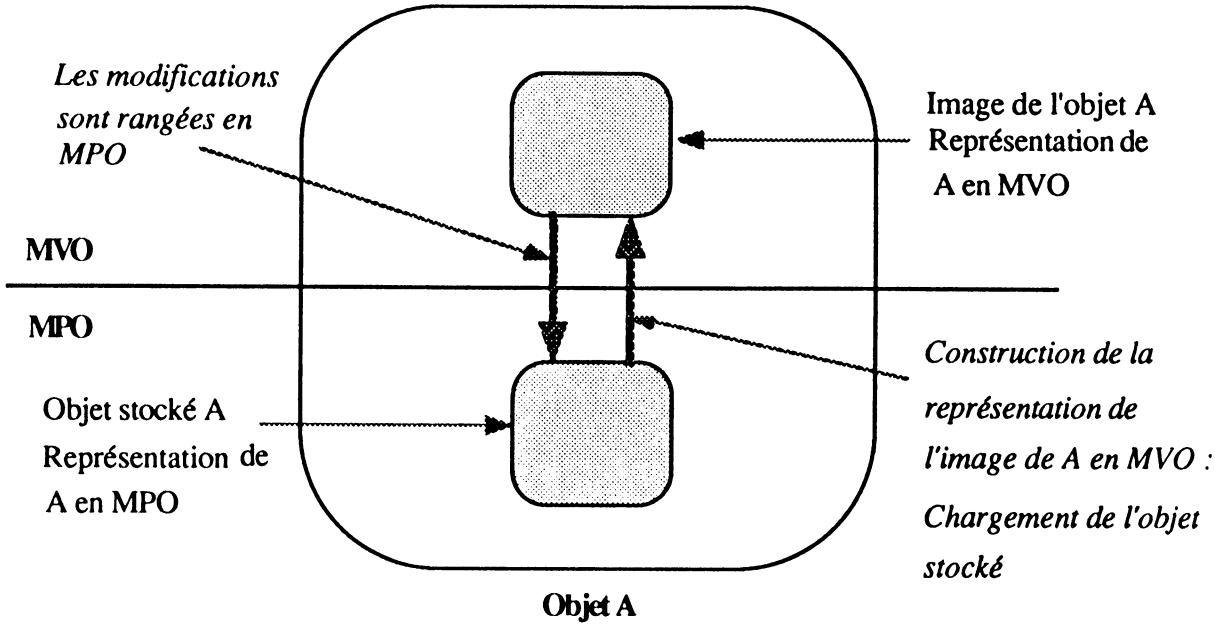


Figure 3.2 : Les échanges entre la MVO et la MPO

3.2 DESCRIPTION DES OBJETS EN MVO

La machine Guide est une machine multi-processeurs qui se compose d'un ensemble de sites reliés entre eux par un réseau local (Ethernet) qui est le seul moyen de communication inter-sites. Sur chaque site, on a une mémoire virtuelle du site qui est partagée par plusieurs processeurs virtuels qui se chargent d'exécuter les programmes d'application. Chaque processeur virtuel dispose d'une mémoire virtuelle propre appelée **domaine** et qui définit les objets que le processeur peut adresser pour l'exécution des méthodes. Pour qu'un objet puisse être adressable par un processeur virtuel, son image doit être présente dans son domaine, c'est-à-dire dans la mémoire virtuelle du site. En conséquence, la composition de cette mémoire virtuelle évolue dynamiquement en fonction des appels de méthodes sur les objets et des demandes de création d'objets.

L'union des mémoires virtuelles des sites (c'est-à-dire des domaines résidant sur tous les sites) constitue la MVO du système Guide. En conséquence, elle est le support d'exécution des objets. La figure 3.3 montre les composants de la MVO et les situe par rapport aux autres éléments de la mémoire d'objets.

3.2.1 IMAGES D'OBJETS

Chaque objet est représenté dans la MVO par son image. L'image d'un objet est matérialisée par le descripteur de l'objet en MVO et par un segment de mémoire virtuelle contenant l'état de l'objet. On appelle ce segment le `segment de l'image`. Un descripteur en MVO comporte les informations suivantes :

- L'`oid` de l'objet ;
- La taille de l'objet ;

- Des informations sur les domaines ayant lié l'objet ;
- L'identificateur du **segment de l'image**.

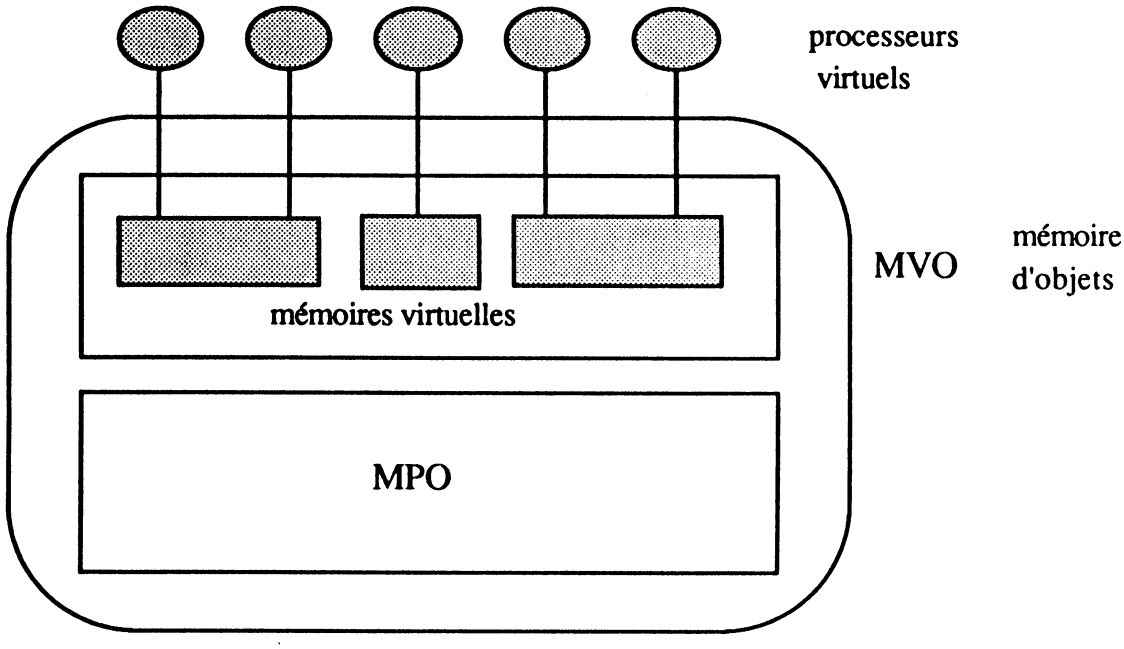


Figure 3.3 : Composition de la MVO

Un objet est verrouillé dans la mémoire virtuelle d'un site lorsqu'une image de l'objet ne peut être créée dans aucune autre des mémoires virtuelles des sites qui composent la MVO. Un objet peut être verrouillé dans une mémoire virtuelle avant que son image ne soit créée sur cette mémoire virtuelle. Après son déverrouillage explicite, un objet peut à nouveau être verrouillé.

Par contre, l'image d'un objet verrouillé doit être partagée par plusieurs domaines ayant lié l'objet car le verrouillage empêche la création d'autres images. Le mécanisme de verrouillage n'est cependant pas appliqué aux **objets exécutables des classes** dont l'image peut être créée sur une mémoire virtuelle quelconque. Dans la suite de ce chapitre nous utilisons le mot **classe** pour parler de **l'objet exécutable** de la classe. La figure 3.4 illustre la raison pour laquelle une classe n'est pas verrouillée : l'objet `C1` est une classe, il comporte le code des méthodes de ses instances. Si l'image de `C1` ne pouvait pas être créée sur la mémoire virtuelle 2, l'image de `O2` (instance de `C1`) serait forcément créée sur la mémoire virtuelle 1.

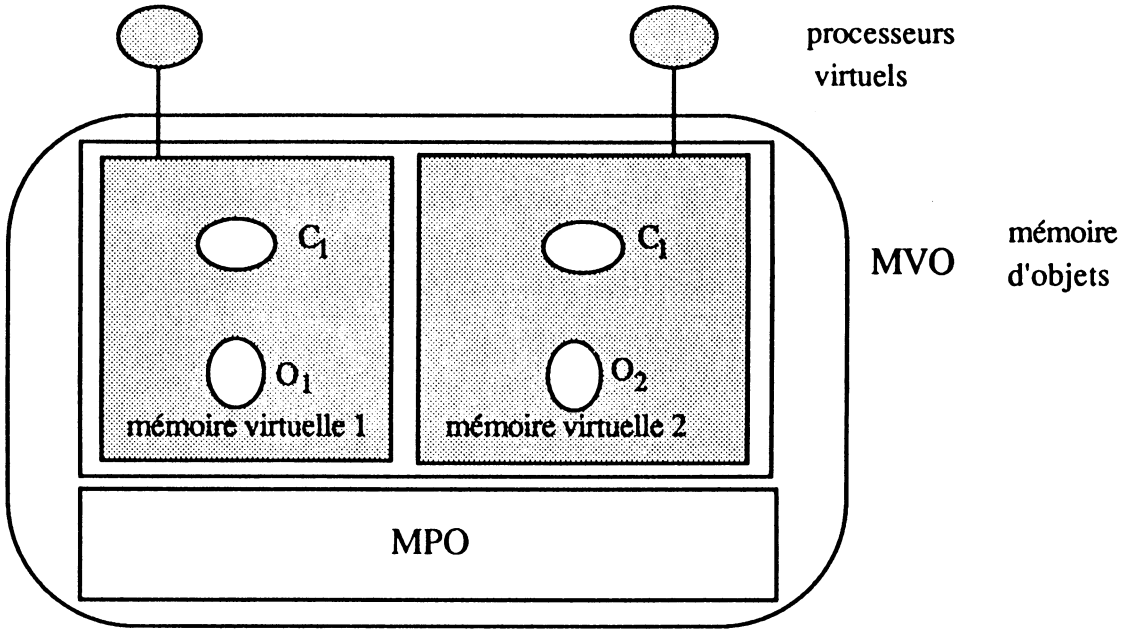


Figure 3.4

3.2.2 CREATION ET DESTRUCTION D'IMAGES

La création d'une nouvelle image, provenant de l'appel de la méthode **New** sur une classe, s'effectue en trois étapes :

1. Création du descripteur de l'objet en MVO. Ce descripteur est initialisé à partir des informations du descripteur en MPO (cf. 3.3.2). De ce fait, chaque fois qu'un objet est créé, son descripteur en MPO est créé avant celui en MVO. Pour la simplicité, l'objet stocké est créé avant son image (cf. 3.3.2) ;
2. Allocation du segment de l'image ;
3. Initialisation de ce segment. Elle se réalise normalement en chargeant l'objet stocké correspondant à partir de la MPO. Si l'objet lui-même n'a pas encore été initialisé, le segment de l'image ne peut pas l'être. Il le sera plus tard, par l'exécution de la méthode `Init`, si celle-ci est définie sur l'objet.

Une image ne peut être créée dans une mémoire virtuelle que si l'objet est verrouillé dans cette mémoire virtuelle. Il s'agit normalement de la mémoire virtuelle du site où le processeur virtuel exécute l'appel d'une des méthodes de l'objets.

La destruction d'une image consiste en la destruction du descripteur en MVO et en la libération de son segment de l'image. Une image ne peut être détruite que si l'objet correspondant est déverrouillé.

3.3 DESCRIPTION DES OBJETS EN MPO

Afin de faciliter la fonction de localisation des objets stockés et aussi pour des raisons d'administration de ressources, on a choisi de ne pas banaliser complètement l'espace de stockage en mémoire secondaire. L'ensemble des objets stockés est donc décomposé en un

certain nombre de sous-ensembles appelés **systèmes d'objets**. Chaque site du système peut supporter un ou plusieurs systèmes d'objets.

A un instant donné, un objet stocké n'est rangé que dans un seul système d'objets. Les objets stockés repérés par les références existant dans un objet O peuvent cependant être rangés dans des systèmes d'objets différents de celui de l'objet stocké correspondant à O. La composition d'un système d'objets évolue dynamiquement en fonction de la création des objets.

L'union de tous les systèmes d'objets constitue la MPO du système Guide. En conséquence, elle est le support de conservation des objets. La figure 3.5 montre les composants de la MPO et les situe par rapport aux autres éléments de la mémoire d'objets.

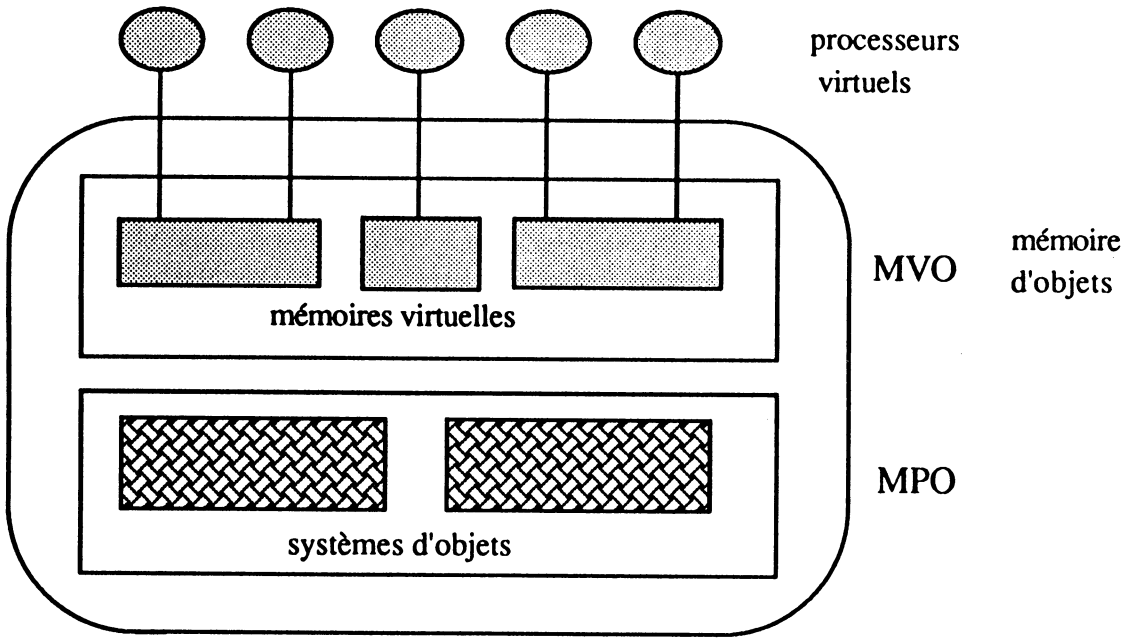


Figure 3.5 : Les composants de la MPO

3.3.1 SYSTEMES D'OBJETS

Les raisons d'administration évoquées ci-dessus pour la décomposition de l'espace de stockage permanent (MPO) en systèmes d'objets appartiennent à deux catégories. Il y a d'abord les raisons de gestion de cet espace. Par exemple, un système d'objets ayant été alloué à une application, celle-ci est assurée de disposer de l'espace correspondant sans être perturbée par les applications dont les objets stockés sont rangés dans d'autres systèmes d'objets.

Les autres raisons sont celles du service d'administration des objets. Parmi les critères d'administration qui sont à l'origine de la définition d'un système d'objets, les critères suivants sont les plus marquants :

- Critère d'organisation. Tous les objets stockés créés par les membres d'une organisation (service, projet,...) sont rangés dans le même système d'objets.

- Critère d'application. Tous les objets stockés utilisés par une application sont rangés dans le même système d'objets.
- Critère de nature des objets. Par exemple, tous les objets stockés de même classe sont rangés dans le même système d'objets.
- Critère de performance. Le rangement des objets stockés est réparti entre plusieurs systèmes d'objets en fonction de l'utilisation et de la charge de chacun d'eux.

Ces différents critères peuvent bien sûr être combinés afin de permettre une gestion plus fine de l'espace de stockage.

3.3.2 OBJETS STOCKES

Un objet stocké est matérialisé par le descripteur de l'objet en MPO et par un ensemble de blocs du système d'objets contenant l'état de l'objet. On appelle ces blocs **les blocs de l'objet stocké**. Un descripteur en MPO comporte les informations suivantes :

- L'**oid** de l'objet ;
- La taille de l'objet ;
- Des informations permettant de trouver le descripteur en MPO de la classe de l'objet ;
- Les dates de création, de dernière lecture et de dernière modification.
- Des informations permettant de trouver **les blocs de l'objet stocké**.

Un objet stocké peut être rangé dans un seul bloc de système d'objets. Cependant, dans Guide, pour des raisons justifiées par la suite, chaque objet est rangé sous forme d'un arbre de blocs non contigus. Le descripteur en MPO ne comporte ainsi que l'adresse d'un bloc de système d'objets : celle du bloc qui contient soit tout l'état de l'objet, soit la racine de l'arbre de blocs.

Cette arborescence, inspirée des mécanismes mis en œuvre dans les systèmes de gestion de fichiers des systèmes classiques, permet la réalisation des fonctions suivantes :

- L'adaptation du rangement des objets stockés à la fois aux objets de grande taille et à ceux de petite taille.
- L'extension dynamique de la taille des objets et l'insertion ou la suppression de données dans un bloc sans que les autres blocs de l'objet stocké soient modifiés.
- La validation (commit) d'une opération atomique de modification de l'état d'un objet par la seule écriture en mémoire stable du bloc racine de l'arbre. Ceci concerne la réalisation des transactions.
- La libération et la création dynamique des objets de tailles différentes sans avoir besoin d'un mécanisme de retassement de blocs de la mémoire.

3.3.3 CREATION D'OBJETS STOCKES

La création d'un objet stocké, provenant normalement de l'appel de méthode `New` sur sa classe, s'effectue en trois étapes :

- Création et initialisation du descripteur en MPO de l'objet ;
- Allocation des blocs de l'objet stocké ;
- Initialisation de ces blocs. Cette étape n'a lieu que lors du rangement du segment de la première image de l'objet.

Le système d'objets dans lequel un objet stocké est créé est déterminé soit automatiquement en fonction d'un ensemble de critères d'administration, soit explicitement par l'utilisateur. Par défaut, un objet stocké est rangé dans un système d'objets local et commun à tous les utilisateurs du système.

Un objet stocké ne disparaît que lorsque l'objet correspondant est libéré. La libération des objets stockés peut être faite explicitement par l'utilisateur, mais cela est d'une part très dangereux et d'autre part charge l'utilisateur. Ainsi, des ramasse-miettes doivent être installés pour ramasser de façon automatique et cohérente des miettes dans le système entier. La destruction d'un objet stocké consiste en la destruction du descripteur en MPO et en la libération de ses blocs.

3.4 CYCLE DE VIE DES OBJETS

La création d'un objet peut maintenant être décrite en termes de création de l'objet stocké et de la première de ses images. D'après le modèle de données, lorsqu'une activité exécute sur une classe la méthode `New`, une instance de cette classe est générée. Dans le modèle de mémoire, cette génération se compose, d'une part, de la création et de l'initialisation du descripteur en MPO du nouvel objet et, de l'autre, de l'allocation des blocs de l'objet stocké.

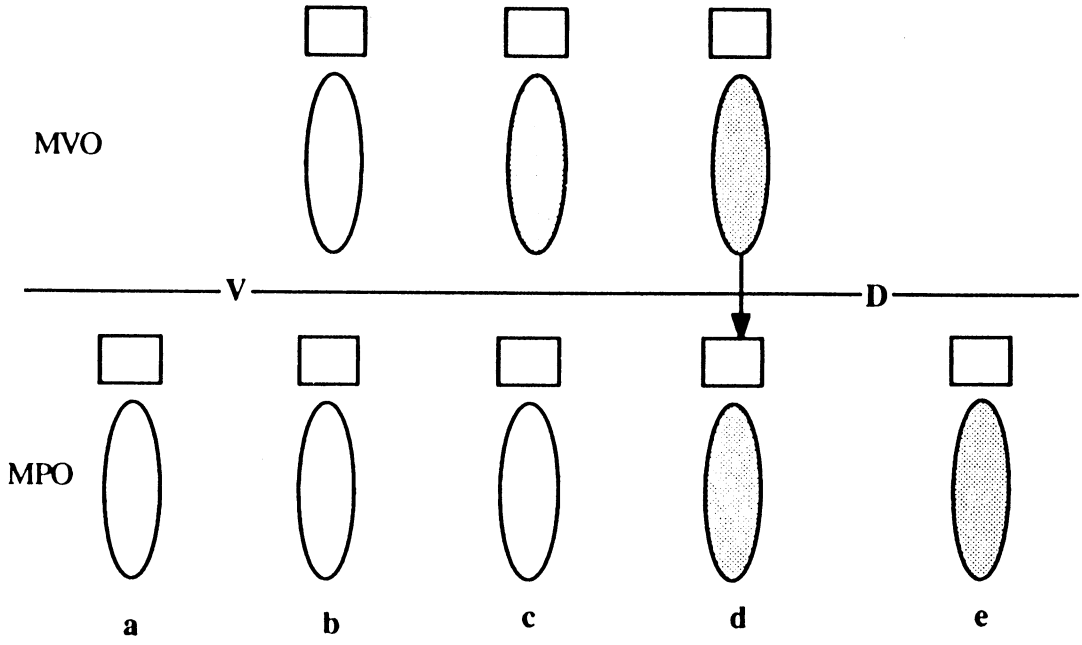
L'activité verrouille ensuite l'objet dans le domaine de son processeur d'exécution. La première image de l'objet est alors créée dans cette MVO.

En général, la méthode `Init`, si elle existe dans l'objet, est la première méthode qu'une activité exécute sur un objet et le segment de l'image est initialisé. Puis, à un moment décidé par le noyau, l'image de l'objet est rangée pour la première fois en MPO et les blocs de l'objet stocké sont initialisés. Lorsque plus aucune méthode ne s'exécute sur l'objet, ce dernier est délié du domaine et éventuellement déverrouillé. La figure 3.6 illustre la création d'un objet.

Après le premier rangement de l'objet stocké, l'objet correspondant mène une vie `'normale'` : le segment de son image est initialisé en chargeant les blocs de l'objet stocké ; de même, les blocs de l'objet stocké sont modifiés lors du rangement de l'image. Le cycle de vie d'un objet normal est détaillé par la suite.

Un objet est couplé dans un domaine lorsqu'une activité de ce domaine appelle une méthode sur cet objet et que celui-ci ne fait pas encore partie du domaine.

L'objet ne peut cependant être lié que s'il est verrouillé dans la mémoire virtuelle du site et s'il comporte une image sur cette mémoire virtuelle.



□ : descripteur de l'objet

0 : l'objet lui même

V : Verrouillage

D : Déverrouillage

```
a    : Création et initialisation du descripteur en MPO,
         les blocs de l'objet stocké ne sont pas initialisés
b    : Création et initialisation du descripteur en MVO
         le segment de l'image n'est pas initialisé
c    : Initialisation du segment de l'image
d    : Rangement en MPO de l'image
         (initialisation des blocs de l'objet stocké)
e    : Destruction éventuelle de l'image
```

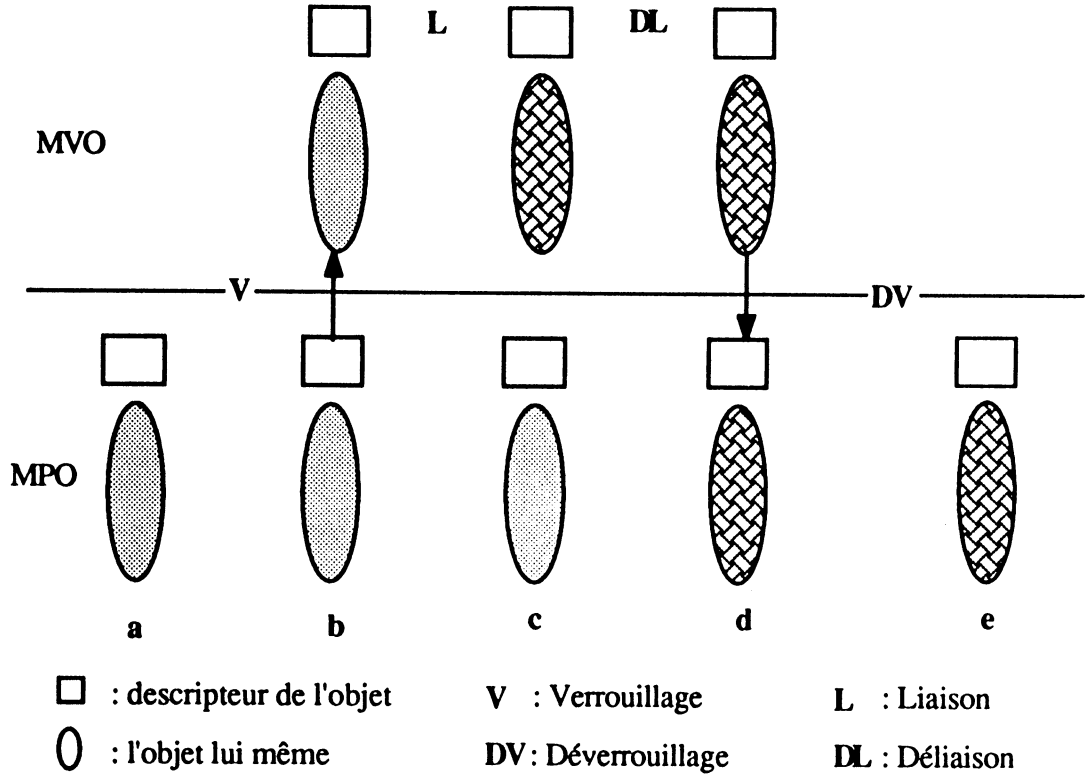
Figure 3.6 Les étapes de la création d'un objet

En conséquence, tout objet non verrouillé doit être verrouillé sur un site avant qu'un domaine (ou sa représentant sur le site) puisse le lier. S'il n'a pas d'image sur cette mémoire virtuelle, une image doit être créée. On rappelle qu'une image ne peut être créée que si l'objet est verrouillé.

L'objet est délié lorsque plus aucune activité du domaine n'exécute une de ses méthodes. Dès qu'il n'est plus lié dans aucun domaine, il est alors déverrouillé. Cependant, il peut rester verrouillé dans une mémoire virtuelle du site. Si une de ses méthodes est appelée, il doit être lié à nouveau.

Le segment de l'image est rangé en MPO seulement si l'objet est verrouillé. Le rangement du segment de l'image et le déverrouillage de l'objet sont deux fonctions indépendantes. Ceci rend possibles les rangements intermédiaires. Avant le déverrouillage de l'objet, le segment de son image doit cependant être rangé en MPO s'il a été modifié depuis son dernier rangement.

La figure 3.7 illustre les mécanismes de gestion de la mémoire d'objets mis en œuvre pour lier et délier un objet d'un domaine.



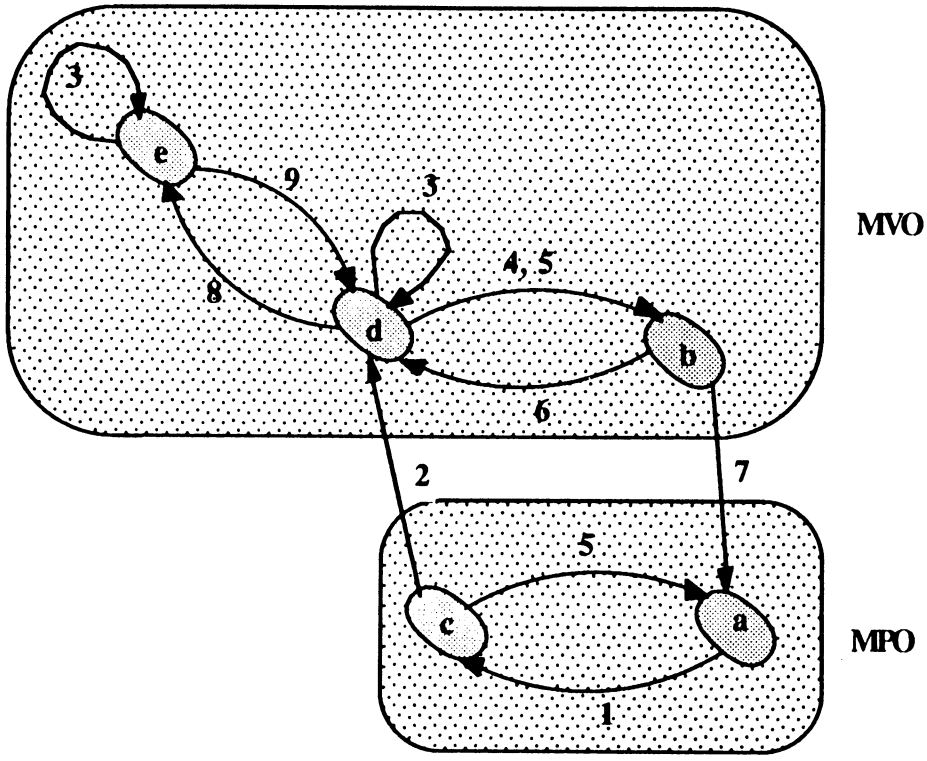
a : L'objet ne possède pas d'image
b : Après verrouillage, l'image est créée en chargeant en MVO l'objet stocké
c : Les modifications se réalisent sur l'image lors de l'exécution des méthodes de l'objet
d : L'image est rangée en MPO
e : L'image est éventuellement détruite

Figure 3.7

On rappelle qu'une image ne peut être détruite que si l'objet est déverrouillé. Cependant, il est possible de déverrouiller un objet sans détruire son image. Ceci évite la création d'une nouvelle image lors d'une utilisation ultérieure. *La décision de détruire ou de garder l'image des objets dans la MVO est totalement prise par le noyau et on n'a plus besoin d'avoir un ramasse-miettes pour ramasser les images d'objets inutiles.*

Finalement, le diagramme de transitions dans la figure 3.8 explique le cycle de vie d'un objet en termes de verrouillage, de chargement en MVO et de liaison dans un domaine.

Il est intéressant de noter qu'au cours de sa vie, plusieurs images d'un objet peuvent être créées et qu'elles peuvent même exister simultanément, une seule étant cependant valide. L'idée de la figure 3.8 complète celle de la figure 3.2 : le terme objet correspond à plusieurs images mais seulement à un objet stocké.



Etat de l'objet

a : non lié, non verrouillé, non chargé
b : non lié, non verrouillé, chargé
c : non lié, verrouillé, non chargé
d : non lié, verrouillé, chargé
e : lié, verrouillé, chargé

Transitions

1 : Verrouillage
2 : Chargement
3 : Rangement
4 : Rangement + Déverrouillage
5 : Déverrouillage
6 : Validation acceptée
7 : Destruction d'image
8 : Liaison
9 : Déliaison

Figure 3.8 : Cycle de vie d'un objet

Les fonctions de la gestion de la mémoire qui supportent le cycle de vie des objets sont les suivantes :

- Création d'un objet stocké ;
- Verrouillage d'un objet ;
- Création d'une image ;

- Chargement en MVO des blocs d'un objet stocké (appelé aussi 'chargement d'un objet stocké' ;
- Rangement en MPO du segment de l'image (appelé aussi 'rangement d'une image') ;
- Déverrouillage d'un objet ;
- Validation d'une image ;
- Destruction d'une image ;
- Destruction d'un objet stocké.

Certaines fonctions ne concernent que la MVO ou que la MPO. D'autres constituent de véritables interactions entre les deux niveaux de la mémoire d'objets comme par exemple les fonctions de chargement et de rangement. La plupart de ces fonctions sont mises en œuvre par les primitives de l'interface entre la MVO et la MPO.

3.5 REFERENCES ET LOCALISATION DES OBJETS

Dans Guide on utilise la notion de '**référence**' pour accéder aux objets et les références sont le moyen unique d'accès aux objets. Une référence permet d'identifier de manière unique l'objet repéré dans le système entier. Bien que la notion de référence soit simple, les références sont utilisées et interprétées dans Guide à différents niveaux du système [Guide-R2] :

- Les **références système** sont interprétées par la MPO et contiennent, en plus de l'`oid` de l'objet, une indication de localisation en MPO : système d'objets de résidence et index de cet objet dans ce système d'objets (fig. 3.9). L'`oid` de l'objet se compose de l'identificateur du système d'objets de création et d'une estampille relative à ce système d'objets (numéro d'ordre de création de l'objet). La partie de localisation se compose de l'identificateur du système d'objets de résidence et de l'index de l'objet dans ce système. Le terme index est utilisé pour désigner un mécanisme d'accès direct (adresse disque) ou indirect (index dans une table) à l'objet dans le système d'objets de résidence.

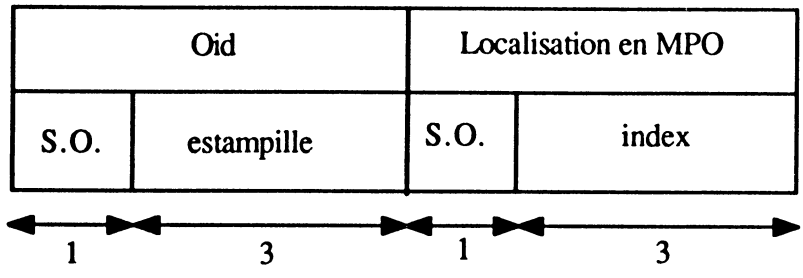


Figure 3.9 : Structure d'une référence système (8 octets)

- Les **références d'exécution** sont interprétées par la MVO et contiennent, en plus de la référence système à l'objet, une indication de localisation de l'objet

en MVO. Cette indication est l'index de l'entrée de l'objet dans la table descriptive TMVO du site si l'objet est lié sur le site, sinon (l'objet est lié sur un autre site) l'indication est le numéro du site où il est lié (fig. 3.10).

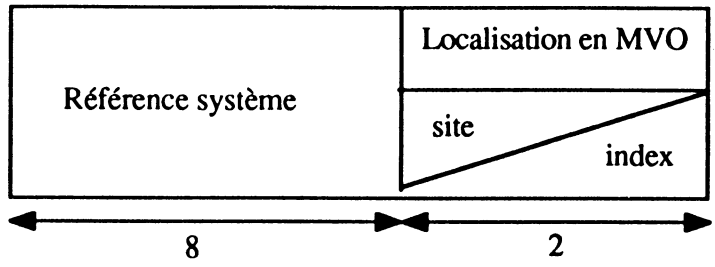


Figure 3.10 : Structure d'une référence d'exécution (10 octets)

Remarque : on n'utilise pas encore les références d'exécution dans le prototype actuel. La MVO utilise seulement les références système.

- Les **références langage** sont interprétées par le langage Guide et ne sont manipulées que par le code généré par le compilateur. Chaque référence langage contient, en plus de la référence système sur l'objet, l'information de localisation de l'objet et de la classe de l'objet dans la MVO.

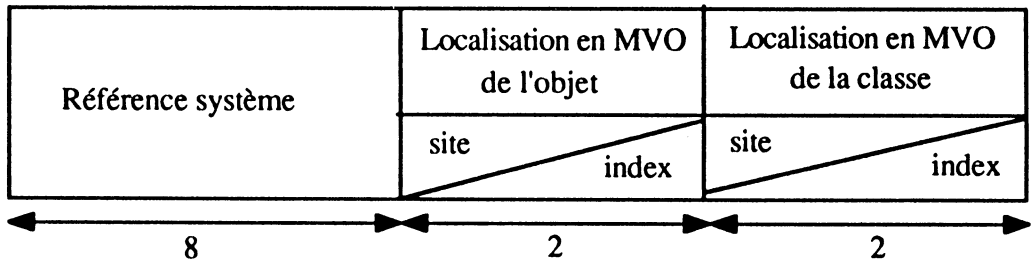


Figure 3.11 : Structure d'une référence langage (12 octets)

3.6 CONCLUSION

Afin de pouvoir supporter l'exécution des méthodes sur les objets et la conservation permanente des objets, la mémoire d'objets Guide est divisée en deux niveaux : la MVO et la MPO. Pour cacher cette division aux utilisateurs, l'interface entre les deux niveaux de la mémoire est rendue transparente aux applications. Dans la MVO, l'image d'un objet est détruite automatiquement par un mécanisme du noyau lorsque l'image n'est plus utilisée, donc on n'a pas besoin d'un ramasse-miettes travaillant sur ce niveau. Par contre, les objets stockés dans la MPO restent toujours là même quand ils ne sont plus utilisables, donc on doit fournir au moins un ramasse-miettes pour ramasser ces objets inutilisables.

Deuxième partie

REALISATION DU COMPILATEUR GUIDE

CHAPITRE 4 : REALISATION DU COMPILATEUR GUIDE

4.1 OBJECTIFS DU COMPILATEUR

Comme nous l'avons présenté, toutes les entités dans le système Guide sont des objets qui sont créés et manipulés grâce aux informations de leur classe et qui sont utilisés via une interface spécifiée par leur type.

Un programme d'application est exprimé dans Guide par un ensemble de types et de classes. Ecrire un programme en Guide revient à définir les types et les classes qui sont nécessaires mais qui n'existent pas encore dans le système (on réutilise les types et les classes existants).

La syntaxe du langage Guide (cf. annexe 2) permet aux programmeurs de définir les types et les classes sous forme de texte. L'objet qui contient le source d'un type ou d'une classe est appelé par la suite **l'objet source**. Comme les objets sources sont écrits manuellement par les programmeurs, il peuvent éventuellement contenir des erreurs à différents niveaux (lexical, syntaxique, sémantique,...). Par conséquent, un des objectifs du compilateur Guide est de vérifier les sources, de signaler correctement et clairement des erreurs, et éventuellement les corriger.

Les classes sources contiennent des méthodes qui vont s'exécuter sur les objets. Pour que les instructions symboliques du corps des méthodes soient exécutées directement, elles doivent être traduites en instructions machine. C'est le deuxième objectif du compilateur Guide.

Les objets sont manipulés par le système grâce à des informations mises dans leurs classes, il faut que ces informations soient directement exploitables par le système. Le troisième objectif du compilateur Guide est de construire, pour chaque classe compilée, les informations nécessaires à l'exécution, sous le format connu par le système. Avec le choix de regroupement en un objet unique de toutes les informations relatives à une classe (cf. 4.5), le code exécutable des méthodes d'une classe ainsi que les informations nécessaires à la manipulation des objets sont regroupés dans une seule entité qui s'appelle **l'objet exécutable** de la classe. En conséquence, après la traduction d'une classe source, on obtient un objet exécutable de la classe correspondante. Au niveau du source Guide, le lien entre les types et les classes est purement symbolique (via le Service de Noms symboliques), ce lien est traduit en lien de référence au niveau du code exécutable.

Le langage Guide est un langage typé et le compilateur vérifie statiquement la relation entre types. Pourtant, il y a des cas où on a besoin des vérifications dynamiques des types à l'exécution (cf. TYPECASE et ASSERTYPE en 2.3.3). Pour rendre plus efficace le processus de

vérification dynamique des types, on transforme les informations symboliques des types en un format interne structuré et on l'enregistre dans un objet que nous appelons **l'objet description** du type. C'est le quatrième objectif du compilateur Guide et il est indispensable dans le cas où le lien entre les objets sources et les objets exécutables n'existe plus après la compilation (cf. 4.3).

Au cours de la compilation d'un type ou d'une classe, le compilateur rencontre d'autres types et d'autres classes, par exemple les types des paramètres et des variables. Pour accomplir sa tâche de traduction, le compilateur a donc besoin des informations relatives à d'autres types et classes. Le stockage et la gestion des informations associées à chaque type et chaque classe sont effectués par un service que nous appelons le **gestionnaire de types et de classes** (cf. 4.3). Ainsi, le compilateur interagit avec ce gestionnaire pour accomplir sa tâche de traduction.

Le schéma 4.1 fait la synthèse de la compilation des types et des classes Guide.

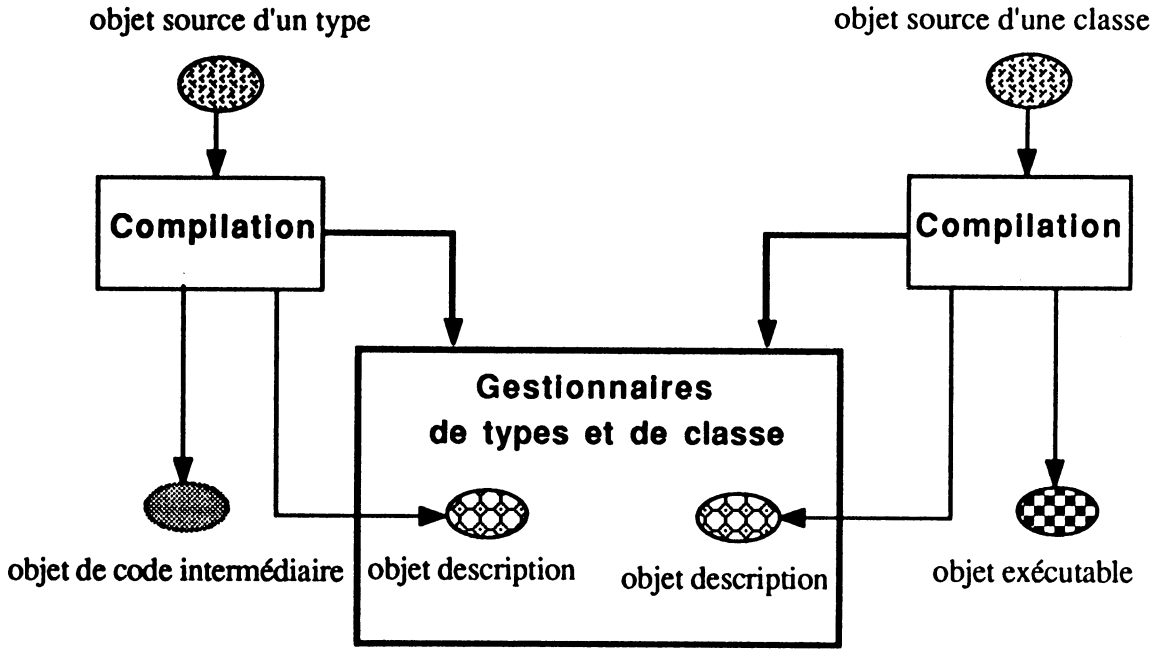


Figure 4.1 Schéma de synthèse de la compilation des types/classes

4.2 ARCHITECTURE GENERALE DU COMPILATEUR GUIDE

4.2.1 CONTRAINTES DU DEVELOPPEMENT

Le système Unix* est choisi comme support de développement d'un premier prototype du système Guide. Ce choix est motivé par les raisons suivantes :

- développer un système sur machine nue demanderait un investissement très important, aussi on a préféré faire un premier prototype au-dessus d'un système existant,

* Unix est une marque déposée par ATT

- Unix tend à devenir un standard sur les postes de travail et fournit une importante gamme d'outils,
- on souhaite arriver rapidement à un prototype qui fonctionne de façon fiable. Grâce aux mécanismes d'Unix et aux outils fournis sous Unix, notre objectif a été atteint et nous avons obtenu un prototype (après un an) qui réalise presque tous les objectifs prévus.

De façon similaire au développement du système, on a dû se servir des utilitaires existant dans le système UNIX sous-jacent pour réaliser le compilateur.

Le modèle d'objets Guide est en cours d'évolution et des modifications de la syntaxe du langage sont fréquentes. Par conséquent, le compilateur doit être codé de façon à être mis facilement à jour pour suivre les changements syntaxiques. D'autre part, l'interface du noyau, elle aussi, peut évoluer au cours du temps, d'où une deuxième contrainte lors de la réalisation du compilateur : on doit pouvoir mettre à jour le compilateur le plus facilement possible en fonction des changements de l'interface du noyau. Pour satisfaire aux besoins énoncés ci-dessus, nous avons choisi de décomposer le compilateur en deux parties : la partie **'front-end'** indépendante de l'interface du noyau et la partie **'back-end'** indépendante des changements syntaxiques. L'arbre sémantique joue le rôle d'interface entre les deux parties.

4.2.2 SCHEMA DE TRADUCTION PROPOSE

Avec l'objectif et les contraintes présentés ci-dessus, le compilateur est décomposé en deux parties (**'front-end'** et **'back-end'**) et il produit en cinq passes (fig. 4.2) du code exécutable chargé dans la MPO :

a. Partie **'front-end'** :

- La première passe consiste en une analyse lexicale et syntaxique de l'objet de texte source et produit l'arbre syntaxique correspondant (cf. annexe 3) ;
- La seconde consiste en l'analyse sémantique qui corrige et complète l'arbre syntaxique obtenu dans la première passe (cf. 4.4) ;

b. Partie **'back-end'** :

- La troisième passe produit du code C à partir de l'arbre syntaxique corrigé et complété ;
- La quatrième passe est la compilation du code C suivie de la résolution des références externes ;
- La cinquième est le chargement du code de la classe dans la MPO.

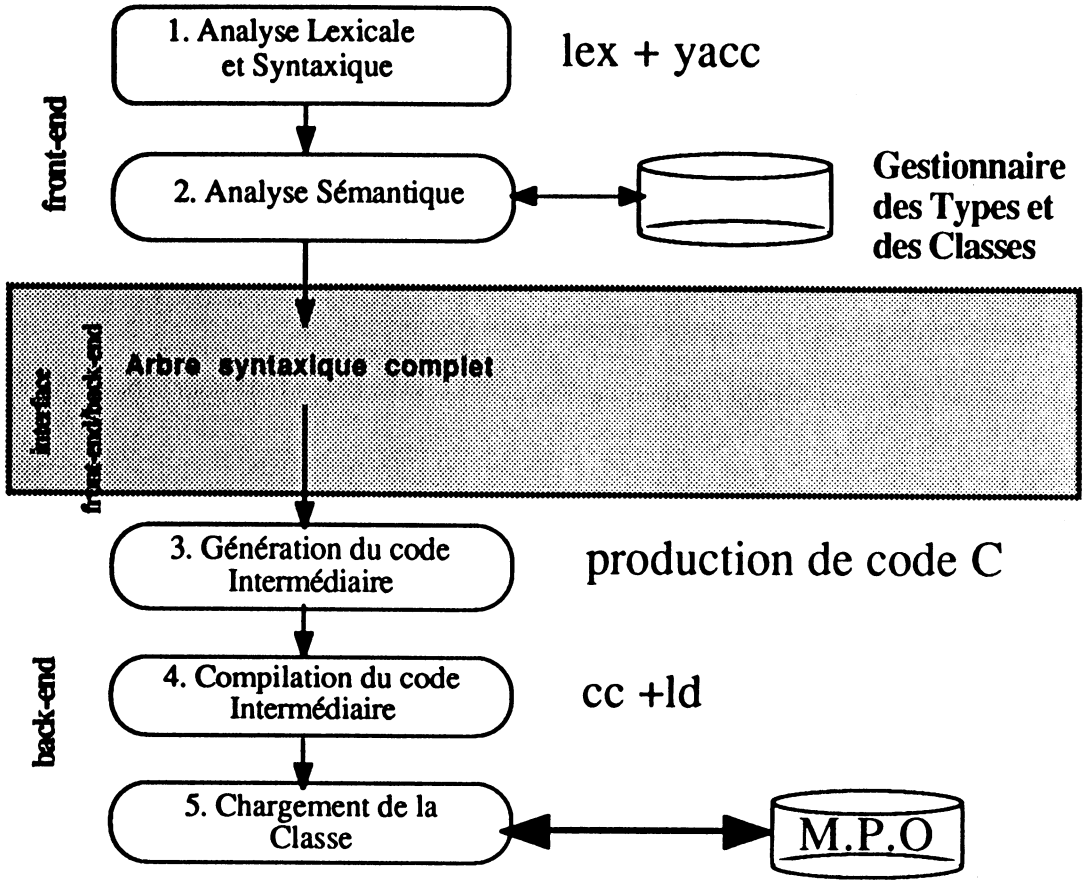


Figure 4.2 : Structure globale du compilateur Guide

Un objet source GUIDE peut éventuellement contenir des erreurs ; ces erreurs peuvent être situées à différents niveaux :

- lexical comme les fautes d'orthographe sur un mot-clé ou sur un opérateur ;
- syntaxique lorsque la chaîne des symboles produite par l'analyse lexicale n'obéit pas aux règles grammaticales définissant le langage ;
- sémantique quand on utilise des types ou des classes inconnus (qui ne sont pas encore compilés et gérés par le gestionnaire de types et de classes) ou non compatibles (dans l'affectation) ;
- logique quand l'algorithme voulu est mal traduit dans le programme source.

Les erreurs lexicales, syntaxiques sont de même nature que celles des autres langages, donc nous utilisons des méthodes classiques pour les traiter [Aho 76]. Les erreurs sémantiques associées à l'affectation de variables sont contrôlées par la relation de conformité entre les types (cf. 2.1.3). Dans la suite de ce chapitre, nous ne décrivons en détail que les phases spécifiques à notre langage, à savoir :

- la phase 2 : analyse sémantique,
- la phase 3 : génération de code intermédiaire C,
- la phase 5 : chargement statique du code dans la MPO.

Pour réaliser la phase 1, la phase d'analyse lexicale et syntaxique, nous utilisons les outils existants LEX et YACC. Dans l'annexe 2, nous présentons cette réalisation ainsi que le traitement des erreurs lexicales et syntaxiques. Quant à la phase 4, la phase de compilation de code C, il n'y a pas grande chose à en dire car nous utilisons le compilateur C du système Unix.

4.3 GESTIONNAIRE DES TYPES ET DES CLASSES

Comme nous l'avons présenté, les types et les classes sont des objets spéciaux dans notre système Guide, un système à objets dans lequel les objets sont manipulés grâce aux informations de leur type et leur classe. Les relations entre les types, entre les classes, et entre les types et les classes sont très délicates et très complexes à maintenir lorsque les types ou/et les classes évoluent. Le fait d'avoir un outil pour gérer ces relations est fortement nécessaire mais ce n'est pas un objectif de cette thèse. Cet outil est appelé **le gestionnaire de types et de classes** dans la plupart des systèmes à objets.

Dans ce paragraphe, nous ne présentons que les services offerts par le gestionnaire et nécessaires au compilateur et au système d'exécution pour la vérification statique et dynamique des types.

a. A la compilation :

Pour que le compilateur puisse détecter l'existence des types et vérifier statiquement la relation de conformité entre eux (cf. 4.4), le gestionnaire doit lui fournir les fonctions suivantes :

- `TypeWrite (type)` : `type` est le pointeur sur l'arbre syntaxique d'un type dont les informations vont être sauvegardées dans un objet description.
- `TypeExist (type)` : `type` est le nom symbolique d'un type dont on veut vérifier l'existence. Si le type existe (il a été compilé et son objet description existe), la fonction rend la valeur TRUE. Sinon, la valeur FALSE est rendue.
- `TypeRead (type)` : `type` est le nom symbolique d'un type qu'on veut récupérer. Les informations récupérées sont regroupées sous forme d'un arbre dont la structure est connue par le compilateur.
- `conform (type1, type2)` : `type1` et `type2` sont les noms symboliques de deux types à vérifier. Si le type désigné par `type1` est conforme au type désigné par `type2`, la fonction rend la valeur TRUE. Sinon, la valeur FALSE est rendue.
- `égalité (type1, type2)` : `type1` et `type2` sont les noms symboliques de deux types à vérifier. Si le type désigné par `type1` est égal (comportement) au type désigné par `type2`, la fonction rend la valeur TRUE. Sinon, la valeur FALSE est rendue.
- `ClassWrite (classe)` : `classe` est le pointeur sur l'arbre syntaxique d'une classe dont les informations vont être sauvegardées dans un objet description correspondant.

- `ClassExist (classe)` : `classe` est le nom symbolique d'une classe dont on veut vérifier l'existence. Si la classe existe (elle a été compilée et son objet description existe), la fonction rend la valeur TRUE. Sinon, la valeur FALSE est rendue.
- `ClassRead (classe)` : `classe` est le nom symbolique d'une classe qu'on veut récupérer. Les informations récupérées sont regroupées sous forme d'un arbre dont la structure est connue par le compilateur.

b. A l'exécution :

Pour vérifier dynamiquement la relation entre les types (cf. 4.5.8.g et h), le gestionnaire doit fournir les fonctions suivantes :

- `conform (type1, type2)` : `type1` et `type2` sont les références à deux types à vérifier. Si le type référencé par `type1` est conforme au type référencé par `type2`, la fonction rend la valeur TRUE. Sinon, la valeur FALSE est rendue.
- `égalité (type1, type2)` : `type1` et `type2` sont les références à deux types à vérifier. Si le type référencé par `type1` est égal (comportement) au type référencé par `type2`, la fonction rend la valeur TRUE. Sinon, la valeur FALSE est rendue.

Dans un premier temps, d'une part pour permettre au compilateur Guide de travailler sous le système sous-jacent Unix, et d'autre part pour accélérer le travail du compilateur, nous intégrons les fonctions du gestionnaire de types et de classes, nécessaires à la compilation, dans le compilateur.

4.4 ANALYSE SEMANTIQUE

Après la phase d'analyse lexicale et syntaxique d'un type ou d'une classe (cf. annexe 3), on obtient un arbre syntaxique. Cependant, cet arbre peut contenir des erreurs sémantiques ; il est insuffisamment documenté pour permettre la génération de code. La fonction essentielle de la phase d'analyse sémantique est de parcourir l'arbre syntaxique obtenu, de le compléter par les informations nécessaires à la génération du code C et de détecter les erreurs sémantiques. Le langage GUIDE étant un langage typé, le contrôle des types est fait statiquement par le compilateur GUIDE :

- la vérification de la validité de tout type utilisé dans les déclarations des variables et des variables références ;
- la vérification de la validité des sous-types et des sous-classes en utilisant la relation de conformité ;
- la vérification de la validité des affectations et des expressions en utilisant la relation de conformité.

Afin d'effectuer ces contrôles, le compilateur interagit avec le gestionnaire de types et de classes pour récupérer au cours de cette phase les "**objets description**" décrivant les types ou les classes utilisés dans l'objet source à compiler.

4.4.1 VERIFICATION DE TYPES DANS LES DECLARATIONS DE VARIABLES

A la fin de la compilation d'un type ou d'une classe, le compilateur demande au gestionnaire de types et de classes d'ajouter l'**objet description** associé au type ou à la classe compilé dans la base de données des types et des classes.

Au cours de la compilation d'un type ou d'une classe, lorsque le compilateur rencontre un type ou une classe utilisé dans les déclarations des variables, il appelle le gestionnaire pour lui demander si le type ou la classe utilisé existe ou non dans la base (sans récupération des informations). Dans le cas négatif, il s'agit d'une erreur sémantique et le compilateur doit la traiter (cf. 4.4.3).

Par conséquent, il existe normalement un ordre de compilation des types et des classes : tous les types et toutes les classes utilisés dans un type ou dans une classe doivent être compilés préalablement à la compilation du type ou de la classe en question.

Pour résoudre le problème de références croisées (un type utilise, de façon directe ou indirecte, un autre et réciproquement), c'est l'utilisateur qui doit indiquer explicitement ce problème en regroupant l'ensemble de types en une seule unité de compilation.

4.4.2 VERIFICATION DE CONFORMITE

Pour garantir l'utilisation correcte des données, la relation de conformité syntaxique (cf. 2.1.2) est vérifiée dans les cas suivants :

- Affectation : `variable := expression`, le type de l'expression doit être conforme au type de la variable,
- Appel de méthode ou de procédure : le type de chaque paramètre d'entrée doit être conforme au type du paramètre formel correspondant ; le type de chaque paramètre formel de sortie (y compris celui éventuellement délivré par la méthode ou la procédure) doit être conforme au type du paramètre effectif correspondant.
- Retour de méthode ou de procédure : `RETURN <expression>`, le type de l'expression `<expression>` doit être conforme au type formel annoncé lors de la déclaration de la méthode ou de la procédure,
- Compilation d'un sous-type : chaque fois que le compilateur compile un type, il doit vérifier si ce dernier est conforme ou non à son super-type.
- Compilation des classes : chaque fois que le compilateur compile une classe, il doit vérifier si les méthodes définies dans la classe sont conformes ou non à la signature correspondant dans le type implémenté.

Le type d'une expression est évalué à partir des types des variables et des constantes qui la constituent. Une structure de données (sous forme de sous-arbre) est associée à chaque variable, à chaque constante et à chaque expression pour indiquer son type.

Nous appelons **`conform (type1, type2)`**, la fonction qui rend la valeur booléenne TRUE si le type désigné par le nom symbolique **`type1`** est conforme au type désigné par **`type2`** et FALSE sinon.

Avec la définition de la relation de conformité syntaxique présentée dans 2.1.2, la vérification de cette relation entre deux types GUIDE revient à vérifier la conformité des signatures de méthodes de ces deux types, i.e. le type **type1** sera conforme au type **type2** si et seulement si :

- pour chaque signature dans le type **type2**, le type **type1** possède une signature qui lui est conforme.

La vérification de la conformité de la signature de deux méthodes revient à vérifier la conformité de chacun des paramètres. En conséquence, le processus de vérification de la conformité de deux types est généralement un processus récursif, ce qui pose les deux problèmes suivants :

- La vérification est coûteuse en temps. Ainsi, il faut absolument éviter de refaire ce travail pour deux types déjà vérifiés ;
- le processus de vérification peut boucler indéfiniment (pendant la vérification de la conformité de deux types, on a éventuellement besoin de ce résultat) et on doit absolument avoir une solution pour éviter ce genre de boucle.

Pour résoudre les deux problèmes ci-dessus, la solution que nous avons prise est d'associer à chaque type deux listes :

- une liste des types dont on sait déjà qu'ils sont conformes au type en question ;
- et une liste des types dont on sait déjà qu'ils ne sont pas conformes au type en question.

Nous donnons ci-dessous l'algorithme correspondant à ce contrôle de conformité :

Algorithme

```
conform (type1, type2) : booléen
/* vérifier si type1 est conforme ou non au type2 */
début
    si type1 = type2 alors      /* égalité symbolique */
                retour VRAI;
    finsi
    récupérer_objet_description(type1);
    récupérer_objet_description(type2);
    si type1 ∈ liste_type_conforme(type2) alors
                retour VRAI;
    finsi
    si type1 ∈ liste_type_non_conforme(type2) alors
                retour FAUX;
    finsi

    /* type1 et type2 n'ont jamais été comparés */
    /* et pour éviter la boucle éventuelle, on fait l'hypothèse suivante */
    liste_type_conforme(type2) = liste_type_conforme(type2) ∪ type1;
    pour chaque signature de méthode S2 de type2 faire
```

```
        s'il n'existe pas de signature S1 dans type1 ayant le même nom
        alors
              annul_hypothèse();
              retour FAUX;
        sinon
           pour chaque paramètre IN P2 de S2 faire
           /* type_P2 est le type de P2, */
           /* P1 est le paramètre de rang correspondant à celui de P2 dans S1 */
           /* et type_P1 est le type de P1 */
              si (P1 non existant) ou (conform(type_P2,type_P1)=FAUX) alors
                 annul_hypothèse();
                 retour FAUX;
              finsi
           finfaire
           pour chaque paramètre OUT P2 de S2 faire
                si (P1 non existant) ou (conform(type_P1,type_P2)=FAUX) alors
                 annul_hypothèse();
                 retour FAUX;
                finsi
           finfaire
           pour chaque paramètre IN_OUT P2 de S2 faire
                si (P1 non existant) ou (égalité(type_P1,type_P2)=FAUX) alors
                 annul_hypothèse();
                 retour FAUX;
                finsi
           finfaire
        finsi
    finfaire
    /* L'hypothèse est vraie */
    /* Ajout de type1 dans la liste_type_conforme de tous les super-types */
    /* de type2 */
    retour VRAI;
fin

/* Annulation de l'hypothèse prise au début de l'algorithme */
annul_hypothèse()
début
    liste_type_conforme(type2) = liste_type_conforme(type2) \ type1;
    liste_type_non_conforme(type2) = liste_type_non_conforme(type2) ∪ type1;
fin
```

La fonction `égalité (type1,type2)` sert à vérifier si les deux types sont égaux (comportement) ou non : deux types sont dits égaux s'ils ont la même interface. Une solution simple pour réaliser cette fonction est d'utiliser la fonction `conform` présentée ci-dessus :

```
égalité (type1, type2) : booléen
/* vérifier si type1 est égal ou non au type2 */
/* type1 et type2 sont les noms symboliques de ces deux types */
début
```

```
        si type1 = type2 alors  /* égalité symbolique */
                retour VRAI;
        finsi
        si conform (type1,type2) et conform (type2,type1) alors
                retour VRAI;
        sinon
                retour FAUX ;
        finsi
fin
```

Remarque : Dans l'algorithme de vérification de la conformité entre deux types ci-dessus (`type1, type2`), on adopte le résultat positif en ajoutant le type `type1` dans la liste "`types conformes`" du type `type2` avant la vérification proprement dite. Cette adoption nous permet d'éviter la boucle infinie éventuelle.

Les listes "`types conformes`" et "`types non conformes`" sont construites de la façon suivante :

- chaque fois que l'on compile un nouveau type, les deux listes sont mises à vides,
- chaque fois que l'on vérifie la conformité, on augmente l'une des deux listes avec le type concerné.

D'autre part, d'après la relation entre le sous-typage et la conformité, lorsqu'un type est conforme à un autre, le premier est conforme aussi à tous les super-types du dernier. Avec cette remarque, lorsqu'on ajoute un type dans la liste des types conformes d'un autre, on peut aussi ajouter le premier à la liste de tous les super-types du dernier. Cet ajout est fait dans deux cas :

- En considérant qu'un type est toujours conforme à ses super-types, lorsqu'on compile un sous-type, on ajoute son nom symbolique dans la liste "`types conformes`" de tous ses super-types ;
- lors de la vérification de la relation de conformité entre deux types, par exemple entre `a` et `b`. Si `a` est conforme à `b`, on ajoute `a` dans la liste des "`types conformes`" de `b` et de tous ses super-types (cf. l'algorithme).

Par contre, lorsqu'un type (par exemple `a`) n'est pas conforme à un autre (par exemple `b`), on ne peut pas généralement savoir si le premier est conforme ou non aux super-types du dernier et quels sont ces super-types.

4.4.3 TRAITEMENT DES ERREURS SEMANTIQUES

Sont considérées comme des erreurs sémantiques les erreurs suivantes :

- utilisation d'un type inconnu, ce dernier n'est pas compilé et son objet description n'existe pas encore dans la base des objets descriptions du gestionnaire des types et des classes. Nous prenons une solution simple pour traiter ce cas de façon à pouvoir continuer la compilation : remplacer le type inconnu par un type virtuel ANY, ce type est reconnu par le compilateur et toutes les erreurs associées à ce type ne sont plus signalées.

- utilisation d'une variable non déclarée. On affecte à cette variable le type virtuel ANY de façon que toute erreur concernant cette variable ne soit plus signalée.
- violation de la relation de conformité lors d'une affectation (passage de paramètres est un cas particulier de l'affectation). Le compilateur va afficher l'erreur pour cette violation.

Le compilateur réalise toujours la phase d'analyse sémantique sur le type (ou la classe) entier, même dans le cas où des erreurs sémantiques sont détectées. Avec l'utilisation du type virtuel ANY, l'erreur associée à une entité (variable) n'est signalée qu'une seule fois.

4.5 GENERATION DE CODE C

Après la phase d'analyse sémantique, s'il n'y a aucune erreur détectée (lexicale, syntaxique ou sémantique), l'arbre syntaxique obtenu est correct et complet, il contient toutes les informations nécessaires pour effectuer le processus de génération de code. Comme le premier prototype du système Guide est développé sur Unix et que le code de ce prototype est écrit en C, nous choisissons le code C comme code généré par le compilateur Guide. Cela permet au code généré d'utiliser facilement l'interface du système Guide.

Le but de la traduction d'un objet classe source est de le traduire en un objet exécutable qui peut être géré par le système Guide. Cette phase de traduction a été mise en œuvre à partir des choix de base suivants :

- Les méthodes des classes ne sont pas gérées individuellement, leur code est contenu dans l'objet classe exécutable qui est l'unité de stockage et de chargement du système. Il est également l'unité de partage de code pour toutes les instances de la classe. Donc si on veut exécuter une méthode d'une classe sur un objet, toute la partie code de la classe doit être chargée et amenée en mémoire d'exécution ainsi que l'état des données de l'objet.
- Ce travail de chargement des classes est bien entendu fait par le système de façon dynamique. En fait, la liaison dynamique est le choix inévitable dans un système à objets qui veut gérer les objets comme les entités autonomes.

Une des conséquences du choix ci-dessus est que tout appel à une méthode sur un objet (effectué dans le corps de la méthode appelante) doit être fait avec l'intervention du système.

Remarque : Dans de nombreux langages à objets (C++, Eiffel, Trellis/Owl), les classes ne sont pas gérées séparément. En effet, le code de toutes les classes utilisées dans une application est regroupé statiquement dans une seule unité, le fichier exécutable du programme. Dans ce cas, on n'a plus besoin de liaison dynamique.

Nous résumons dans cette section la structure de mémoire des activités Guide ainsi que le mécanisme de chargement du code exécutable des classes dans cette mémoire. Puis nous spécifions le format d'une classe exécutable tel qu'il est défini par le noyau Guide et le mécanisme de liaison dynamique lors des appels de méthodes sur les objets. Ensuite, nous précisons la structure de données spécifique qui doit être produite pour permettre à l'appelant,

au noyau et à l'appelé de communiquer. Enfin, nous résumons les points essentiels de la traduction des types et des classes.

4.5.1 STRUCTURE DE MEMOIRE D'UNE ACTIVITE GUIDE

Dans le rapport [Guide-5], le choix d'implémentation de la structure d'exécution de Guide est présenté de façon détaillée et précise. Pour faciliter la présentation du format d'une classe exécutable (cf.4.5.3), nous résumons ici les points essentiels concernant la structure de mémoire d'une activité Guide.

Une activité Guide est un environnement d'exécution séquentielle d'une suite d'appels de méthodes sur des objets qui peut s'étendre sur plusieurs sites. La mémoire virtuelle d'un représentant sur un site (ou mémoire locale de l'activité) est représentée par la mémoire virtuelle d'un processus Unix sur ce site. Un domaine regroupe plusieurs activités différentes qui partagent les objets liés dans ce domaine (données et programmes) ainsi que les informations qui servent à la gestion du domaine. D'autre part, toutes les activités présentes sur un site partagent les informations qui servent à la gestion des objets du site, ainsi que le code exécutable du système Guide. Ce code exécutable est partagé au moyen du mécanisme standard de partage de programmes sous Unix. Le partage des autres informations utilise la mémoire partagée.

Avec le mécanisme de gestion de processus d'Unix, la mémoire virtuelle locale d'une activité Guide (processus sous Unix) est partitionnée en trois zones : une zone privée propre à l'activité, une zone partagée et une zone réservée à la pile (fig. 4.3). Comme Unix se réserve la zone située de l'adresse 0 à β - 1 de la mémoire virtuelle d'un processus, l'adresse de base du début de la mémoire gérée par Guide sera égale à β. Le découpage statique de ces zones est présenté ci-dessous (fig. 4.3) :

La zone privée

La zone privée est de taille fixe et s'étend de l'adresse virtuelle Unix β à l'adresse β+`M1`-1 (`M1` est la valeur définie par Unix). Elle est partitionnée en deux blocs eux-mêmes de taille fixe.

Le premier bloc correspond au segment `.text` d'Unix et contient le code exécutable du système Guide. Ce code est partagé par toutes les activités (processus Unix) présentes sur un site.

Le second bloc correspond au segment `.data` d'Unix. Celui-ci contient les données du système Guide (de taille fixe) nécessaires à l'exécution de l'activité, le code de la classe en cours d'exécution, ainsi que les données locales à l'activité créées dynamiquement (par exemple par la primitive "`malloc`" du langage C). Ce bloc n'est pas partagé et sa gestion est assurée complètement par le système Guide.

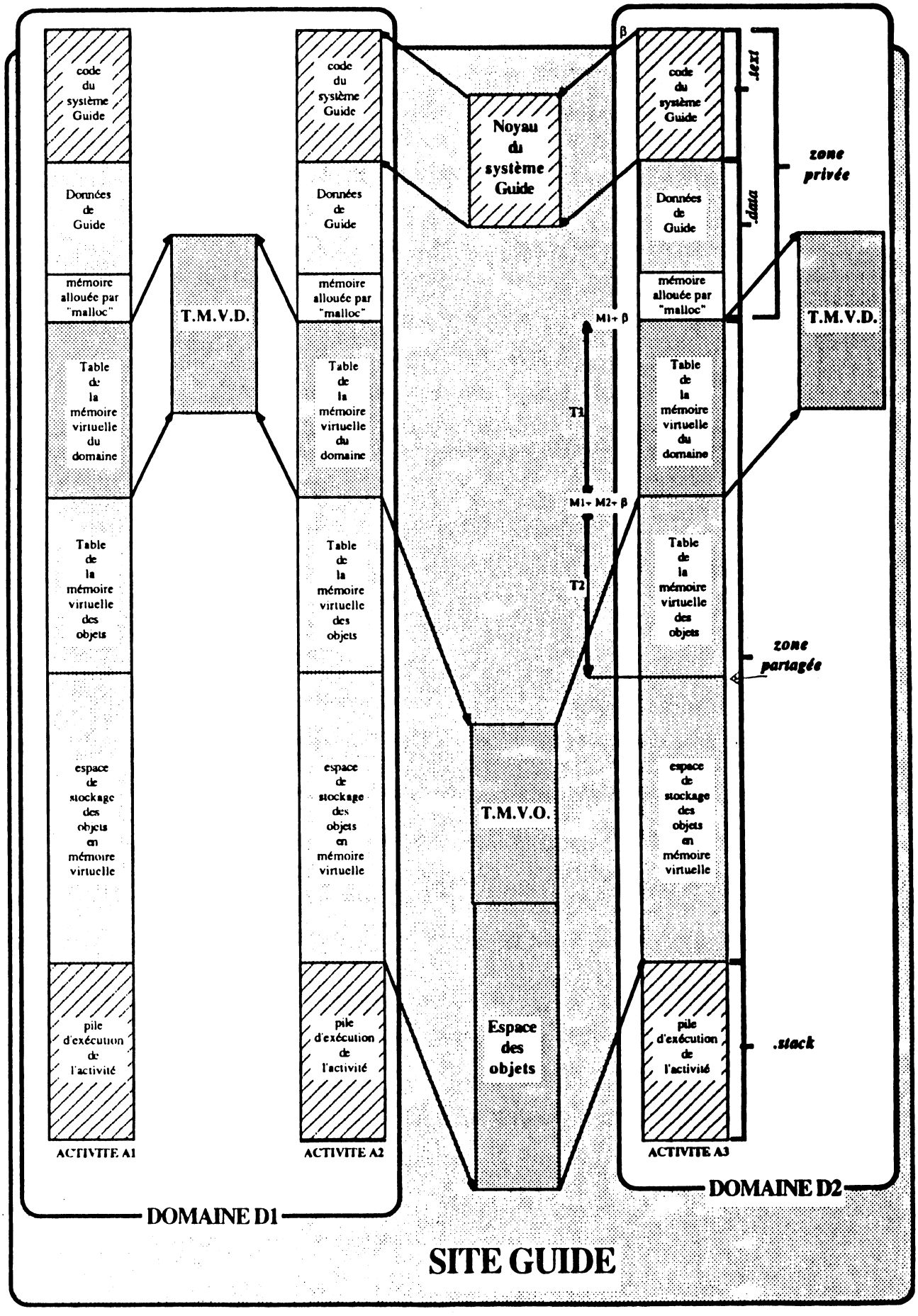


Figure 4.3 : Achitecture de la mémoire d'exécution d'un site Guide

La zone partagée

Cette zone est destinée à la représentation de la zone de la mémoire virtuelle d'objets d'un site du système Guide et des domaines. Elle est donc destinée à contenir l'ensemble des données du système Guide qui sont partagées par tous les domaines et par toutes les activités d'un même domaine ainsi que les objets (instances de classes ou classes). Cette zone est partitionnée en trois blocs de taille fixe :

- Le premier contient la Table de la Mémoire Virtuelle du Domaine (TMVD). C'est le descriptif qui décrit le contexte du domaine (ensemble des objets liés à un instant donné) auquel l'activité appartient.
- Le second contient la Table de la Mémoire Virtuelle d'Objets (TMVO) du site. C'est une table unique (du site) qui regroupe des informations de gestion sur l'ensemble des objets du site.
- Le troisième sert au stockage proprement dit des objets adressés par le site. On peut y trouver particulièrement des objets "classes exécutables", et pour cette raison que l'on doit obligatoirement coupler cette partie à une même adresse de toutes les activités sur un même site Guide.

4.5.2 CHARGEMENT D'UNE CLASSE DANS LA MEMOIRE D'EXECUTION D'UNE ACTIVITE

Lorsqu'une activité appelle une méthode sur un objet, le code de sa classe est dynamiquement chargé dans la mémoire d'exécution de l'activité (la zone partagée) à partir de la mémoire permamente MPO. La gestion dynamique du code des classes dans la mémoire d'exécution utilise la technique de **chargement dynamique**. Dans ce cas, une phase de translation des adresses du code de la classe doit généralement précéder toute exécution : une classe est chargée dynamiquement (en translatant des adresses) dans la mémoire d'exécution à partir de la MPO au premier appel d'une méthode de la classe par une activité et reste présente tant que la classe est utilisée par au moins d'une activité.

Pour faciliter le traitement des constantes, notamment des chaînes constantes, on les regroupe dans une zone contigue dans l'objet classe exécutable.

4.5.3 FORMAT D'UNE CLASSE EXECUTABLE

Le format d'une classe exécutable découle :

- de la structure de la mémoire d'une activité Guide sur laquelle le code des méthodes d'une classe s'exécute (cf. 4.5.1) ;
- du mécanisme que prend le système pour amener la classe exécutable dans la mémoire d'exécution de l'activité (cf. 4.5.2).

Dans sa version actuelle, le compilateur du langage Guide produit tout d'abord du code C qui est ensuite compilé par le compilateur C pour obtenir du binaire au format connu par le système UNIX sous-jacent (format différent suivant les machines). Ce binaire translatable constitue l'état d'un objet chargeable de la classe compilée. Dans la phase de chargement, le chargeur va transformer l'objet chargeable en objet exécutable en translatant des adresses.

La figure 4.4 représente une classe, résultat de la compilation. Chaque classe exécutable a trois parties :

- Le descripteur de la classe exécutable qui contient les informations nécessaires au système pour la manipulation de la classe exécutable et pour la création de ses instances. Les informations suivantes sont essentielles :
  - + **dataStart** et **dataSize** permettent de connaître le début et la taille de la zone de données (la zone qui contient les constantes chaînes de caractères utilisées dans les méthodes) de la classe exécutable ;
  - + **codeStart** et **codeSize** permettent de connaître le début et la taille de la zone de code des méthodes définies dans la classe ;
  - + **instanceSize** permet de connaître la taille des instances lors d'une création ;
  - + **méthodeNbr** permet de connaître le nombre total de méthodes applicables ;
  - + **typeRef** et **typeName** permettent de connaître le type implémenté. Ces informations sont nécessaires à la vérification dynamique de conformité (cf. 4.5.8.g et h) ;
  - + **listClassRef** et **listClassName** permettent de connaître la hiérarchie des super-classes ;
  - + **classIndex** permet de connaître le numéro d'ordre de la classe dans la hiérarchie des super-classes ;
  - + la table **TDE** permet la localisation des méthodes lors de la liaison dynamique (cf. 4.5.4).
- la zone qui contient les chaînes constantes de caractères utilisées dans les méthodes de l'objet classe exécutable ;
- la zone du code des méthodes de la classe (uniquement des méthodes dont le corps est défini explicitement dans la classe, et pas des méthodes héritées).

Remarques :

- Il n'y a pas de duplication de code des méthodes des super-classes héritées et non surchargées dans la sous-classe. Cela nous apporte :
  - + une optimisation importante de l'utilisation de la place tant au niveau de stockage (la MPO) qu'au niveau de mémoire d'exécution ;
  - + une facilité de partage du code. On n'a pas besoin de recompiler une sous-classe lorsque le code des méthodes d'une de ses super-classes est mis à jour.
- le chargement d'une sous-classe ne nécessite pas le chargement préalable de ses super-classes. Ces dernières sont chargées au fur et à mesure des besoins.

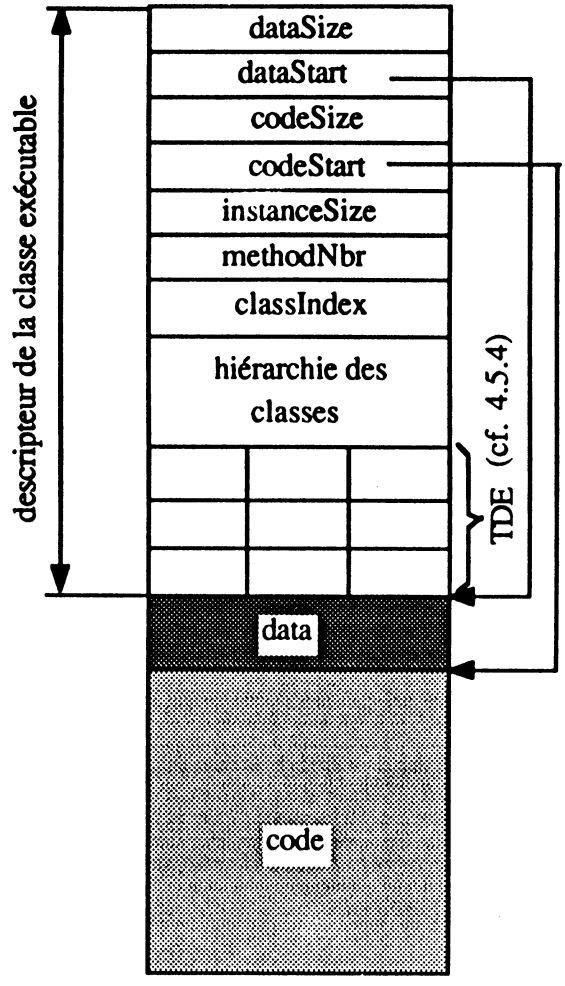


Figure 4.4 : Structure d'une classe exécutable

4.5.4 LIAISON DYNAMIQUE ET TABLE DES DEFINITIONS EXTERNES D'UNE CLASSE

Dans le langage Guide, le type associé à une variable référence détermine la liste des méthodes qui lui sont applicables quel que soit l'objet réellement désigné.

Toute variable référence définie dans le source Guide doit être traduite en une entité correspondante dans le langage objet (C dans notre cas). La structure de cette entité est fixée pour toute variable référence quel que soit le type associé. Nous appelons cette entité **la référence langage**. La liaison dynamique présentée précédemment implique deux contraintes :

- la référence langage doit permettre de retrouver la classe exécutable de l'objet qu'elle désigne,
- le nom de la méthode appelée doit permettre de retrouver l'adresse de cette méthode dans la classe.

Une technique classique qui prend en compte ces deux contraintes consiste à mettre dans la valeur de l'objet une désignation de sa classe et dans la référence langage la désignation de l'objet. De plus à chaque classe est associée une table appelée **Table des Définitions Externes**

(TDE) dont l'entrée i donne l'adresse relative de la i ème méthode. Avec cette hypothèse, un nom de méthode est remplacé, à la compilation, par le numéro de l'entrée de la TDE qui lui est associée (C++, Eiffel). Cette solution n'est pas applicable à Guide car :

- au moment de la compilation, on ne sait pas quelle est la classe de l'objet désigné. Rappelons qu'une variable référence peut désigner des objets de différents types (de différentes classes) à condition que ces derniers soient conformes au type de la variable ;
- cette solution imposerait à l'ensemble des classes "*conformes*" (dont des exemplaires peuvent être désignés par une même variable référence) d'avoir une même structure de TDE (à la ième entrée correspondrait toujours la même méthode). Mais cette dernière contrainte est impossible à réaliser (dans Guide, on peut avoir une classe qui est conforme à une autre mais la première n'est pas une sous-classe de la dernière).

On est donc amené à modifier le format de la TDE associée à chaque classe qui va être une table de couples (nom de point d'entrée, adresse relative dans la classe). La recherche dans la TDE se fait de manière associative à partir du champ nom.

A cause de l'héritage et avec la structure des classes exécutables choisie (cf. 4.5.3), le code associé à une méthode applicable à une classe peut ne pas être défini dans la classe courante mais dans l'une de ses super-classes. On doit donc mémoriser pour chaque point d'entrée la référence système de la classe contenant la définition du code du point d'entrée. Pour éviter la duplication des références système nous introduisons une table appelée **Hiérarchie** dont les entrées contiennent les références système des ancêtres de la classe courante, la dernière entrée contient la référence système de la classe courante.

Le format de la table TDE est donc (pour l'exemple de la classe `PetiteBoite`) :

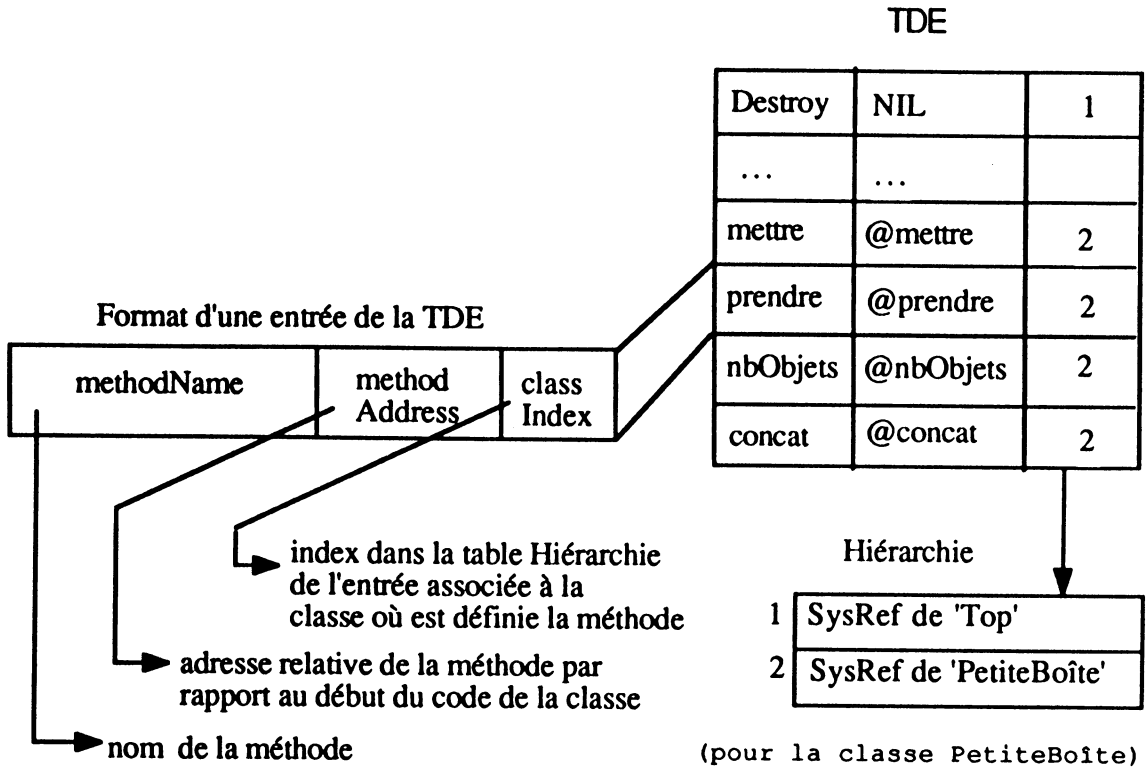


Figure 4.5 : Structure de la table TDE

4.5.5 FORMAT DE LA REFERENCE LANGAGE

Dans la recherche de la méthode à exécuter, deux modes de localisation de la méthode sont proposés :

- On passe la chaîne de caractères du nom de la méthode (ou plutôt un pointeur) au noyau. Celle-ci est utilisée pour rechercher dans la TDE l'adresse de chargement de la méthode correspondante. A chaque appel de cette méthode sur des objets de la même classe on devra refaire une recherche par nom dans la TDE. Dans la suite, nous appelerons ce mode de localisation **le mode normal.**
- Afin de minimiser le nombre de recherches par nom, et afin de tenir compte des informations connues à la compilation, on associe à chaque référence langage une **Table de Liaison** qui contient une entrée par méthode effectivement appelée sur cette référence. Cette entrée contient le nom de la méthode et l'index de la méthode dans la TDE ; ce dernier champ est initialisé lors du premier appel et modifié à chaque affectation de la référence langage. Ce mode de localisation a été proposé dans le compilateur Emerald [Black 86a] et dans la suite, nous l'appelerons **le mode accéléré.**

A chaque appel de méthode, le mode normal de localisation des méthodes prend plus de temps que le mode accéléré pour rechercher l'adresse de la méthode, mais on évide le problème de traitement de la table de liaison. De plus, dans les programmes où l'action

d'affectation de références est plus fréquente que l'action d'appel de méthodes, le mode accéléré perd son avantage à cause des recopies importantes des tables de liaison. Le choix entre ces deux modes doit être fait par le compilateur et être transmis au système lors du `GuideCall` (cf.4.5.6).

La structure de données d'une référence langage possède donc le format suivant :

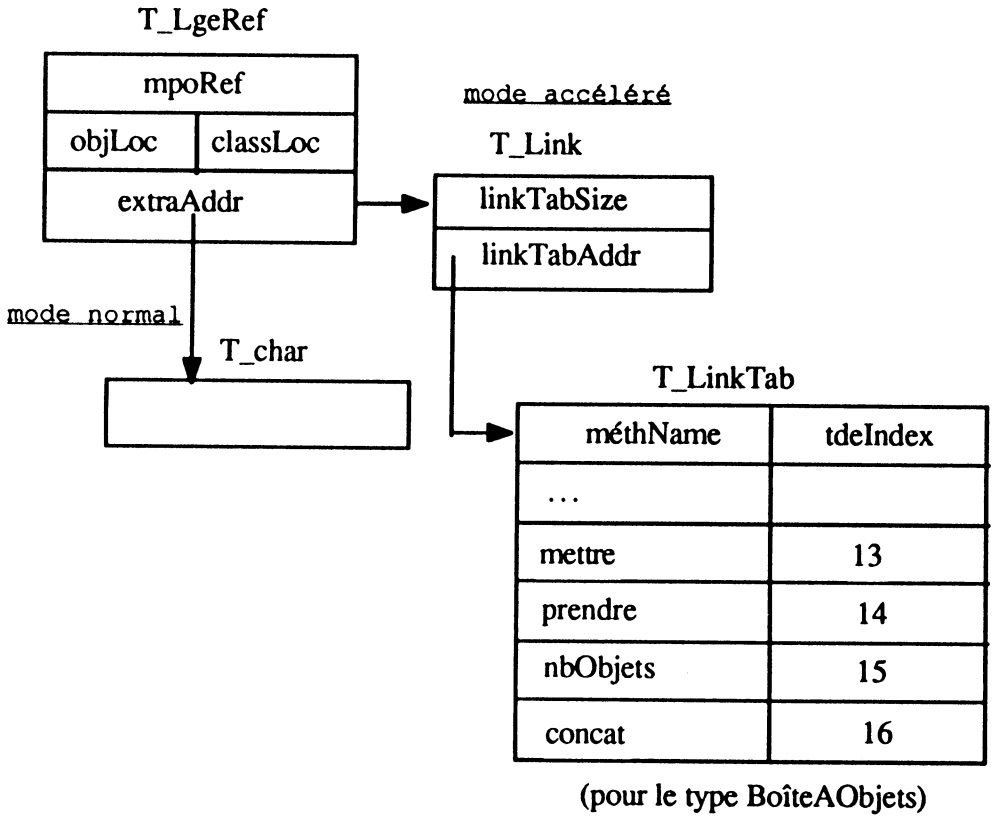


Figure 4.6 : Structure de la référence langage

où :

- **`mpoRef`** est la référence système de l'objet désigné (cf. 2.5) qui permet d'identifier et de localiser l'objet dans le système,
- **`objLoc`** est l'index de l'objet dans la TMVO après liaison de l'objet. TMVO est la table des descripteurs des objets qui existent déjà dans la MVO (cf. 4.5.1), chaque entrée de cette table contient les informations pour la localisation de l'objet. **`classLoc`** est l'index dans la TMVO de la classe de l'objet, après liaison de l'objet classe exécutable. Les deux champs **`objLoc`** et **`classLoc`** servent uniquement à l'accélération des opérations d'accès à l'objet et à sa classe,
- **`extraAddr`** est soit l'adresse d'une chaîne de caractères dont la valeur est le nom de la méthode dans le cas où on utilise le **mode normal** de localisation de méthodes, soit, dans le cas du **mode accéléré**, l'adresse d'un descripteur de la table de liaison `T_Link`.

Remarque : Dans un premier temps, le compilateur Guide utilise uniquement le mode normal de localisation et dans ce cas, la structure de données d'une référence langage est optimisée comme indiqué dans la figure 4.7. Le champ `extraAddr` est supprimé et on met l'adresse du nom symbolique de la méthode appelée dans le bloc de communication d'un `GuideCall` (cf. 4.5.6).

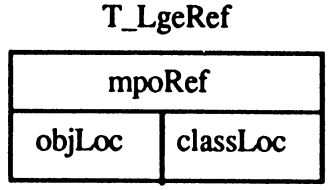


Figure 4.7 : Structure optimisée de la référence langage
(voir aussi fig. 3.11)

4.5.6 FORMAT DU BLOC DE PARAMETRES DANS UN GUIDECALL

Comme nous l'avons présenté précédemment, l'objet est une entité autonome d'exécution et de conservation ; pendant l'exécution d'une méthode sur un objet (faite par une activité), on peut appeler une autre méthode sur un autre objet dont on a une référence. Un appel de méthode est réalisé par l'intervention du système via une primitive prédéfinie appelée `GuideCall` qui provoque éventuellement la liaison dynamique de l'objet sur lequel la méthode appelée sera exécutée. Ainsi, le compilateur doit traduire un appel de méthode sur un objet en un appel de la primitive `GuideCall`. En conséquence, un appel de méthode fait intervenir une méthode appelante, le noyau et une méthode appelée.

Pour savoir exactement ce que le compilateur doit générer lors d'un appel de méthode, on doit tout d'abord savoir exactement quel est le comportement du noyau lors d'un `GuideCall` à travers la structure de données nécessaires à la communication entre l'appelant, le noyau et l'appelé. Dans l'interface du noyau Guide, la primitive `GuideCall` est définie par :

```
primitive GuideCall (x : Pt_T_CallBlock) ;
```

où `x` est l'adresse d'un bloc de paramètres de type `T_CallBlock` qui est préparé par l'appelant, complété (et éventuellement dupliqué) par le noyau et utilisé par l'appelé. Ce bloc de paramètres contient toutes les informations nécessaires pour :

- assurer la communication entre l'appelant, le noyau et l'appelé,
- assurer le passage de paramètres entre l'appelant et l'appelé.

Le bloc de paramètres contient les champs principaux suivants :

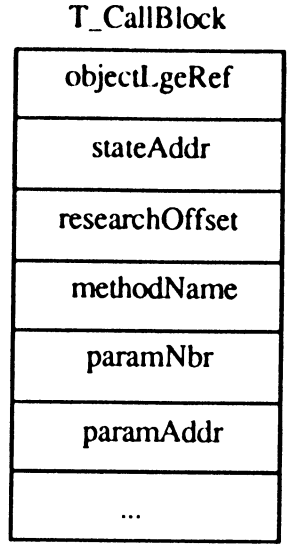


Figure 4.8 : Structure du bloc de paramètres passé au GuideCall

- **objectLgeRef** contient l'adresse de la référence langage de l'objet (cf. 4.5.5). C'est l'appelant qui doit remplir ce champ pour que le noyau puisse s'en servir.
- **researchOffset** permet d'indiquer la classe dans laquelle on commence à rechercher la méthode appelée sur l'objet. C'est l'appelant qui doit remplir ce champ. Il y a deux cas :
  + 0 : on commence la recherche à partir de la classe de l'objet sur lequel la méthode est appelé (pour les appels normaux par l'intermédiaire des références);
  + 1 : on commence la recherche à partir de la super-classe de la classe courante (pour les appels par l'intermédiaire de SUPER).
- **methodName** est l'information qui permet d'identifier la méthode appelée. C'est actuellement l'adresse de son nom symbolique.
- **paramNbr** est le nombre de paramètres de la méthode appelée sur l'objet. C'est l'appelant qui doit remplir ce champ.
- **paramAddr** est l'adresse du bloc de paramètres effectifs de la méthode appelée. Dans le bloc de paramètres effectifs de la méthode, chaque entrée comporte deux champs : un champ qui identifie la nature et le type du paramètre et un champ qui contient soit la valeur, soit l'adresse du paramètre. Cette représentation est rendue nécessaire par le fait que le noyau Guide peut être amené à recopier les paramètres d'un espace virtuel Unix dans un autre et ceci dans deux cas : soit parce que l'exécution doit s'effectuer sur un autre site, soit parce que l'exécution doit être faite par une autre activité (cf. l'instruction CO_BEGIN en 2.2.3).
- **stateAddr** contient l'adresse de l'état de l'objet ; après son chargement dans la mémoire virtuelle du domaine, le noyau remplit ce champ pour que la méthode appelée puisse accéder à l'état de l'objet.

La figure 4.9 suivante représente la structure complète des données associée à un `GuideCall`.

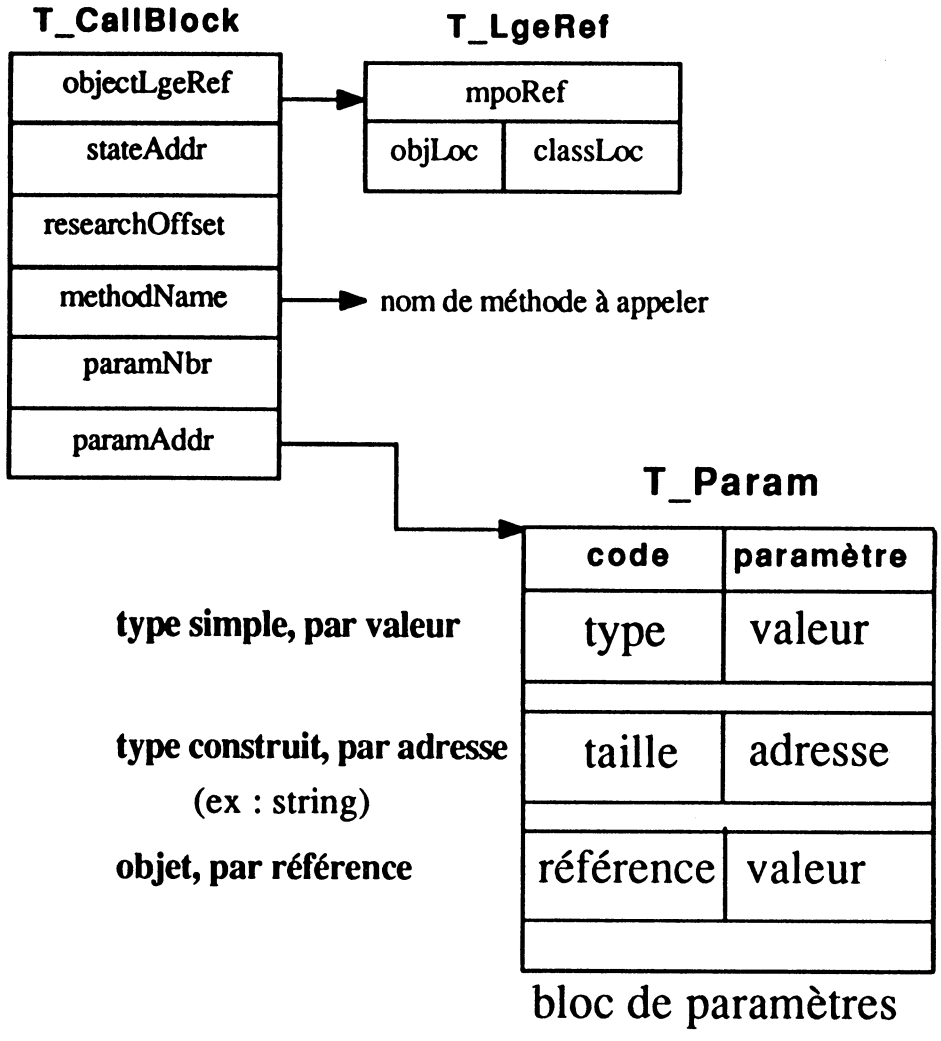


Figure 4.9 : Structure complète de données passée au GuideCall

A partir du principe de construction d'une classe exécutable et de la description du `GuideCall` nécessaire pour l'appel de méthode, nous décrivons dans les deux paragraphes suivants les points essentiels de la traduction des types et des classes.

4.5.7 TRADUCTION D'UN TYPE

Le type Guide joue le rôle d'interface d'utilisation des objets Guide, il ne contient pas d'information d'implémentation des objets. Il sert seulement à la vérification de conformité entre les types soit statiquement dans la phase d'analyse sémantique, soit dynamiquement. Pour cette vérification, le compilateur construit, pour chaque type compilé, un objet description à la fin de la phase d'analyse sémantique. Il contient les informations pour utiliser (appeler) une des méthodes sur un objet du type. En plus, lorsque l'on compile un appel de méthode sur un objet (référencé par une variable référence), on ne sait pas encore exactement

à quelle classe appartient l'objet, mais à ce moment là on doit générer le code pour construire le bloc de paramètres passé au noyau dans le `GuideCall`. En conséquence, lorsque l'on compile un type, on génère le code suivant :

- pour chaque méthode dans l'interface, les instructions de construction du bloc de paramètres pour appeler cette méthode. Chaque méthode a sa propre séquence d'instructions qui sont regroupées dans une macro-instruction. Cette macro-instruction sera utilisée chaque fois que l'on compile un appel de la méthode en question.
- les instructions de construction de la table de liaison `T_LinkTab`. Dans la version actuelle du compilateur Guide, nous n'utilisons que le mode normal de localisation des méthodes et dans ce cas, nous n'avons pas besoin de la table de liaison et nous ne générons donc pas les instructions pour la construire.

a. Traduction des variables d'état visibles

Lors de la compilation d'un type, une variable d'état visible est transformée en deux signatures de méthodes (cf. 2.1.1) dans la phase d'analyse sémantique. Ces signatures engendrées sont traitées de la même façon que les autres signatures dans la génération de code C que nous présentons à la suite.

b. Traduction d'une signature de méthode

La traduction d'une signature de méthode consiste à définir pour cette méthode comment est passé chacun des paramètres de la méthode. En effet, les instructions de construction du bloc de paramètres effectifs doivent :

- remplir le champ `paramNbr` du bloc `T_CallBlock` (cf. 4.5.6) par le nombre de paramètres de l'appel (0 si aucun paramètre) ;
- construire la table de paramètres effectifs (de type `T_Param`), elle contient la description et le chemin d'accès aux paramètres d'appel de la future méthode. Chaque entrée de cette table est constituée de deux champs :
  + un code qui indique la façon par laquelle la valeur du paramètre est accessible,
  + le moyen d'accéder à cette valeur (adresse virtuelle Unix ou valeur immédiate).
- remplir le champ `paramAddr` du bloc `T_CallBlock` par l'adresse de la table `T_Param` construite ci-dessus.

Afin de faciliter l'utilisation de ces instructions, chaque signature donne lieu à la définition d'une macro qui sera stockée dans un fichier de nom `Type.t` et qui aura pour nom `create_Type_Méthode`. Cette macro permet de réaliser de façon simple le remplissage du bloc de paramètres.

Exemple :

Nous reprenons le type `BolteAObjets` présenté dans le chapitre 2 :

```
TYPE BoîteAObjets IS

        // synonymes
        SYNONYM Name = String [80] ;

        // variable d'état visible
        propriétaire : Name ;        // variable d'état visible correspondant à
                                     // deux méthodes Rd_propriétaire et
                                     // Wr_propriétaire

        // méthodes
        METHOD mettre (IN REF Objet) ; SIGNALS boîte_pleine ;
                     // ajout d'un élément dans la boîte
        METHOD prendre (OUT REF Objet) ; SIGNALS  boîte_vide ;
                     // prise d'un élément de la boîte
        METHOD nbObjets : Integer ;
                     // rend le nombre d'objets mis dans la boîte
        METHOD concat (IN REF BoîteAObjets) : REF BoîteAObjets ;
END BoîteAObjets.
```

Ce type sera compilé dans le fichier `BoîteAObjets.t` qui contiendra :

```
#define create_BoîteAObjets_Rd_propriétaire01(blockAddr, ret) \
T_ParamItem paramBlock[1]; \
blockAddr.paramNbr = 1; \
blockaAddr.paramAddr = paramBlock; \
paramBlock[0].code = taille de l'état de la classe; \
paramBlock[0].paramItem.valAddr = (T_char *) ret;

#define create_BoîteAObjets_Wr_propriétaire10(blockAddr, arg1) \
T_ParamItem paramBlock[1]; \
blockAddr.paramNbr = 1; \
blockaAddr.paramAddr = paramBlock; \
paramBlock[0].code = taille de l'état de la classe; \
paramBlock[0].paramItem.valAddr = (T_char *) arg1;

#define create_BoîteAObjets_mettre10(blockAddr, arg1) \
T_ParamItem paramBlock[1]; \
blockAddr.paramNbr = 1; \
blockaAddr.paramAddr = paramBlock; \
paramBlock[0].code = REF_VAL ; \
paramBlock[0].paramItem.lgeRefAddr = &arg1;

#define create_BoîteAObjets_prendre01(blockAddr, res1) \
T_ParamItem paramBlock[1]; \
blockAddr.paramNbr = 1; \
blockaAddr.paramAddr = paramBlock; \
paramBlock[0].code = REF_ADDR ; \
paramBlock[0].paramItem.lgeRefAddr = &res1;

#define create_BoîteAObjets_nbObjets01(blockAddr, ret) \
T_ParamItem paramBlock[1]; \
blockAddr.paramNbr = 1; \
blockaAddr.paramAddr = paramBlock; \
paramBlock[0].code = INT_ADDR ; \
paramBlock[0].paramItem.longAddr = &ret;

#define create_BoîteAObjets_concat(blockAddr, arg1, ret) \
T_ParamItem paramBlock[2]; \
blockAddr.paramNbr = 2; \
```

```
blockaAddr.paramAddr = paramBlock; \
paramBlock[0].code = REF_VAL; \
paramBlock[0].paramItem.valAddr = &arg1; \
paramBlock[1].code = REF_ADDR; \
paramBlock[1].paramItem.lgeRefAddr = &ret;
```

4.5.8 TRADUCTION D'UNE CLASSE

Comme nous l'avons dit, la classe contient les programmes des méthodes et les définitions des variables d'état de ses instances. Les points essentiels de la traduction d'une classe sont :

- la traduction des références aux objets ;
- la construction du modèle de données d'objets appartenant à la classe ;
- la construction du descripteur de la classe qui permet au système de charger la classe exécutable.
- la construction de la table TDE pour la localisation des méthodes à l'exécution ;
- la traduction des appels de méthodes sur des objets ;
- la traduction des méthodes ;

a. Traduction d'une référence à un objet

La déclaration d'une variable référence s'écrit sous le format suivant dans le source Guide :

```
nom_référence  :  REF nom_type ;
```

Selon le principe de traduction présenté dans le paragraphe 4.5.5, cette déclaration est traduite en une déclaration C :

```
T_LgeRef  nom_référence ;
```

où `T_LgeRef` est une structure C qui est appelée `la référence langage` des objets et qui contient toutes les informations nécessaires pour identifier, localiser l'objet référencé et sa classe. Rappelons que le type `nom_type` sert uniquement à la vérification de conformité dans la phase d'analyse sémantique, et qu'il est ignoré dans la phase de traduction.

b. Construction du modèle de donnés des objets

Chaque objet a ses propres données et tous les objets d'une même classe ont le même modèle de données. Ce modèle de données doit être connu par toutes les méthodes et par toutes les procédures de la classe pour que ces dernières puissent accéder correctement aux données de l'objet sur lequel elles travaillent. Le modèle de données des objets à construire est une structure C qui regroupe toutes les variables d'état de la classe, y compris celles des super-classes. Lorsqu'une méthode est appelée sur un objet, le noyau va charger l'objet dans la mémoire d'exécution et puis donner à la méthode l'adresse du début de la zone de données de l'objet. Connaissant cette adresse et le modèle de données, la méthode peut accéder correctement à n'importe quel champ de données de l'objet.

Par exemple, le modèle de données des objets de la classe `PetiteBoîte` est construit par la structure C suivante :

```
struct T_PetiteBoîte {
        ( les champs de données de la super-classe (Top) )
        T_char propriétaire[80] ;      /* la variable d'état visible      */
        T_int nb_elt ;                 /* la variable d'état              */
        T_int premier ;          /* la variable d'état       */
        T_int dernier ;          /* la variable d'état       */
        T_LgeRef contenu[5] ;    /* la variable d'état       */
) ;
```

c. Construction du descripteur de la classe

Le descripteur de la classe (cf. 4.5.3) est réalisé par une structure C de type `T_Header` qui est réservée et initialisée par le compilateur de la façon suivante :

```
T_Header aClassHeader = (
    dataSize,          /* la taille de la zone des données de la classe
                          contenant toutes les chaînes de caractères
                          constantes utilisées dans la classe */
    NIL,               /* dataStart : adresse du début de la zone des
                          données de la classe */
    NIL,               /* textSize : la taille du code du programme de
                          de toutes les méthodes et les procédures qui sont
                          définies directement dans la classe */
    NIL,               /* textStart : adresse du début de la zone du
                          code de la classe */
    ième,              /* Numéro d'ordre de la classe dans la hiérarchie
                          : 1 si la classe n'a pas de super-classe (Top),
                            +1 par rapport à la super-classe sinon. */
    methodNbr,         /* Nombre total de méthodes applicables de cette
                          classe : nombre total de méthodes applicables
                          de la super-classe + nombre de méthodes définies
                          uniquement sur cette classe */
    instanceSize,      /* taille d'une instance d'un objet de cette
                          classe. C'est la taille de la structure
                          de données engendrée dans b). */
    typeRef,           /* la référence du type implémenté */
    typeName,          /* nom symbolique du type implémenté. Ce champ et
                          le champ typeRef servent à la vérification
                          dynamique de conformité */
    listClassRef,      /* table des références des super-classes,
                            cf. parag. d. suivant */
    listClassName,     /* table des noms des super-classes,
                            cf. parag. d. suivant */
    TDE                /* table des définitions externes */
)
```

Toutes les valeurs non initialisées par le compilateur (`NIL`) le seront par la primitive `guidelink` du système Guide lors de la phase 5 qui fait un chargement statique de la classe exécutable.

d. Construction de la Table des Définitions Externes associées à une classe (TDE)

Comme nous l'avons présenté en 4.5.4, chaque classe exécutable doit posséder sa propre table TDE qui permet de connaître, après chargement de son code exécutable, les adresses absolues de chacune de ses méthodes. Cette table est réservée à la compilation, et les adresses des méthodes sont mises à jour au moment du chargement.

Par exemple, pour la classe `PetiteBoîte`, le compilateur Guide engendre la table suivante :

```
T_EntryPoint      TDE[15+6] = { /* classe Top a 15 méthodes */

    /* méthodes héritées sans surcharge de la classe Top */
    {"Destroy00", 0, 0},
    ...

    /* méthodes surchargées de la classe Top */
    {"PrintState00",        (unsigned char *) PrintState00,        2},
    ...

    /* méthodes définies dans la classe PetiteBoîte */
    {"Rd_propriétaire01",   (unsigned char *) Rd_propriétaire01,   2},
    {"Wr_propriétaire10",   (unsigned char *) Wr_propriétaire10,   2},
    {"mettre10",            (unsigned char *) mettre10,            2},
    {"prendre01",           (unsigned char *) prendre01,           2},
    {"nbObjets01",          (unsigned char *) nbObjets01,          2},
    {"concat11",            (unsigned char *) concat11,            2},
} ;
```

Le champ niveau dans la hiérarchie des classes (troisième champ de chaque entrée de la table TDE) exprime le numéro d'ordre de la classe dans laquelle la méthode correspondante est définie. Ce numéro d'ordre correspond à la position de la classe considérée dans une table de hiérarchie engendrée par le compilateur pour chaque classe. Par exemple, pour la classe `PetiteBoîte`, le compilateur crée la table de hiérarchie suivante :

```
T_char* listClassName[] = {
        "Top",                      /* niveau d'ordre 1 */
        "PetiteBoîte"               /* niveau d'ordre 2 */
} ;
```

et :

```
T_SysRef listClassRef[] = {
        /* SysRef de la classe "Top", niveau d'ordre 1 */
        /* SysRef de la classe "PetiteBoîte", niveau d'ordre 2 */
} ;
```

Le compilateur met la valeur NIL dans les champs d'adresse et de niveau (deuxième et troisième champ de chaque entrée) associés à toutes les méthodes héritées et non surchargées. Le chargeur Guide, pour toutes ces méthodes héritées, met à jour :

- le champ niveau par le numéro d'ordre de la classe dans la hiérarchie dans laquelle la méthode correspondante est définie,

- les adresses relatives des méthodes par rapport au début de la classe où elles sont définies.

e. Traduction d'un appel de méthode sur un objet

Un appel de méthode sur un objet s'écrit en Guide :

```
aRéférence.aMéthode (argument, résultat) ;
```

et est traduit en un bloc d'instructions C :

```
{
   /* initialisation de la table des paramètres effectifs dans callBlock */
   /* par appel une macro générée lors de la compilation du type */
   creat_aType_aMéthode (callBlock, argument, résultat) ;

   /* initialiser d'autres informations dans callBlock */
   callBlock.objectRef = &C_aRéférence;
   callBlock.researchOffset = 0;
   callBlock.methodName = insertMethod("nom_méthode20");
   /* insertMethod est une fonction de codage de noms des méthodes*/
   ...

   /* appel au GuideCall du système */
   FCT_guideCall (&callBlock);
}
```

où **callBlock** est une structure C **T_CallBlock** (cf. 4.5.6) qui contient toutes les informations nécessaires et qui est passée au noyau via la primitive **GuideCall** ; **creat_aType_aMéthode** est la macro qui crée des instructions C pour construire la table des caractéristiques des paramètres effectifs (cf. 4.5.7.b) ; **FCT_guideCall** est aussi une macro qui provoque l'appel de la primitive **GuideCall** dans l'interface du noyau Guide. Cette primitive est chargée de localiser et de lier l'objet en question ainsi que sa classe dans l'espace du domaine de l'appelant ; elle remplit les champs dans le **callBlock** qui ne peuvent pas être remplis par l'appelant, puis le noyau donne le contrôle à l'appelé en lui passant le bloc **callBlock**.

De la même façon, l'appel de méthode via la référence **SELF** (cf. 2.2.3) :

```
i := SELF(n).aMéthode( argument, résultat)
```

sera compilé de la façon suivante :

```
{
   create_aType_aMethode21 ( callBlock, argument, résultat, i );
   callBlock.objectRef = self; /* cf. le paragraphe f. ci-dessous */
   callBlock.researchOffset = n;
   callBlock.methodName = insertMethod("aMethode21");
   /* insertMethod est une fonction de codage de noms des méthodes*/
   FCT_guideCall (&callBlock);
}
```

f. Principe de compilation d'une méthode

Lors de la réception de l'ordre d'exécution du noyau, la méthode appelée recoit le bloc de communication **`callBlock`**. Par ailleurs, comme l'appelée travaille avec ses variables locales, il est nécessaire de recopier les valeurs des paramètres à partir du bloc **`callBlock`** aux variables locales. En effet, le compilateur définit implicitement deux variables de travail **`state`** et **`self`** pour chaque méthode. Ces deux variables sont des pointeurs respectivement sur l'état de l'objet et sur la référence langage de l'objet sur lequel s'applique la méthode. Il produit également le code d'initialisation de ces deux variables locales à partir des champs du bloc de paramètres. De la même manière, le compilateur produit le code qui recopie dans des variables locales de la méthode la valeur des paramètres d'appel.

Par exemple, la méthode `mettre` définie dans la classe `PetiteBoîte` :

```
METHOD mettre (IN objet : REF Top) ;
// déclaration des variables de travail
BEGIN
// code de la méthode
END mettre;
```

sera compilée dans le code C suivant :

```
PetiteBoîte_mettre10( blockAddr )
T_CallBlock       *blockAddr;
{
   /* déclaration des pointeurs vers l'état de l'objet */
   T_PetiteBoîte *state;
   T_LgeRef      *self;

   /* déclaration des paramètres */
   T_LgeRef      objet;

   /* déclaration des variables de travail */
   ...
   T_CallBlock   callBlock;    /* bloc de paramètres pour des GuideCall dans cette
   méthode */

   /* prologue de la méthode */
   /* récupération de l'état de l'objet */
   self = blockAddr->objectLgeRef;
   state = (T_PetiteBoîte *) blockAddr->stateAddr;

   /* récupération des paramètres passés par l'appelant */
   objet = blockAddr->paramAddr[0].paramItem.refVal;

   { <code de la méthode> }

   /* épilogue de la méthode */
   { <copie les valeurs des paramètres IN_OUT et OUT dans callBlock> }
}
```

Remarque : Grâce au modèle d'objets et à l'héritage, il y a similitude parfaite entre les opérations systèmes et celles définies par l'utilisateur sur un objet tant au niveau de leur utilisation (appel) qu'au niveau de leur compilation.

g. Génération de code pour l'instruction ASSERTYPE

Comme nous l'avons présenté en 2.2.3, l'instruction **ASSERTYPE** est une des possibilités qui permettent aux programmeurs de retrouver l'interface réelle d'un objet référencé par une référence d'un autre type. Elle s'écrit en Guide :

```
var_référence := ASSERTYPE (expression) ;
```

où **expression** est une expression qui rend une référence (par exemple **eref**) en résultat de son évaluation. Le compilateur doit contrôler la validité de l'affectation de la référence **eref** à la variable **var_référence**, mais au moment de la compilation, on ne sait pas encore exactement à quel type appartient l'objet référencé par **eref**. C'est pour cette raison que le contrôle est fait en deux temps :

- A la compilation, le compilateur évalue le type de l'expression **expression**. Si le type évalué n'est pas une référence sur un objet, il affiche un message d'erreur. Sinon, il génère le code suivant pour le contrôle dynamique :

```
/* code généré pour l'instruction ASSERTYPE */
/*  var_référence := ASSERTYPE (expression) ; */
/* _var_référence est la référence langage correspondante à var_référence */
<code qui calcule l'expression expression>
/* _var_tmp est la référence langage qui contient le résultat */
<code d'appel de la méthode _var_tmp.ObjectTypeRef>
/* _type_obj_ref est la référence du type obtenue */
/* _type_var_réf est la référence du type de la variable var_référence
   et elle est déterminée à la compilation */
if (conform(_type_obj_ref, _type_var_réf)) {
      _var_référence = _var_tmp ;
} else {
      <code de traitement d'erreur>
}
<code de l'instruction qui suit l'instruction ASSERTYPE>
```

- A l'exécution, la primitive **conform** du gestionnaire de types et de classes (cf. 4.3) est appelée avec deux paramètres : le type de la variable et le type de l'objet référencé par **C_var_tmp**. La valeur de retour de la primitive **conform** permet au code généré par le compilateur soit de signaler une erreur, soit de continuer le programme après l'affectation de la référence.

h. Génération du code pour l'instruction TYPECASE

L'instruction **TYPECASE** (cf. 2.2.3) est la deuxième possibilité qui permet aux programmeurs de retrouver l'interface réelle d'un objet référencé par une référence d'un autre type. Elle s'écrit dans le source Guide :

```
TYPECASE <var_référence> OF
     <ident_type1> : <liste_d'instructions> END ;
     <ident_type2> : <liste_d'instructions> END ;
     ...
     OTHERS : <liste_d'instructions>
END
```

On a vu que la relation de conformité permet à une variable référence de désigner des objets, non seulement du type de la variable référence, mais aussi de tout type conforme. Ainsi, dans l'instruction **TYPECASE** ci-dessus, **var_référence** peut désigner à l'exécution des objets de différents types à condition que ces derniers soient conformes au type de **var_référence**. A la compilation, on ne peut pas savoir quel objet réel est désigné par la variable ni à quel type il appartient. C'est seulement à l'exécution que l'on peut déterminer le type réel de l'objet désigné par la variable et donc quelle est la branche à exécuter.

A la compilation, on effectue un contrôle minimum de validité de l'instruction et on génère le code permettant de déterminer dynamiquement le type réel de **var_référence** :

- Le compilateur évalue le type de **var_référence** pour savoir s'il s'agit d'un type élémentaire ou d'un type construit. S'il s'agit d'un type élémentaire, c'est une erreur et le compilateur affiche un message d'erreur. S'il s'agit d'un type construit, le compilateur continue d'analyser les branches : pour la liste d'instructions associée à chaque branche, **var_référence** est considérée comme étant du type associé à cette branche. Si aucune erreur n'est produite, le compilateur génère le code suivant pour l'instruction **TYPECASE** :

```
/* code généré pour l'instruction TYPECASE */
/* SYSREF_TYPE1, SYSREF_TYPE2, ... sont les références langages */
/* des types concernés, elles sont toutes déterminées à la compilation */
<code d'appel de la méthode var_référence.ObjectTypeRef>
/* sysRef_var est la référence du type obtenue de var_référence */
if  (égal_type(sysRef_var, SYSREF_TYPE1))  {
        <code généré pour la branche associée à <ident_type1> >
} else if (égal_type(sysRef_var, SYSREF_TYPE2))  {
        <code généré pour la branche associée à <ident_type2> >
} else if (...) {
} else {
        <code généré pour la branche OTHERS>
}
```

- A l'exécution, la primitive **égal_type** () est appelée séquentiellement avec deux paramètres : le type (la référence langage de ce type) de l'objet réellement désigné par la variable et le type de la branche à vérifier. La branche d'instructions correspondant au premier type qui donne le résultat positif (rendu par **égal_type**) est alors exécutée. Sinon la branche correspondant à **OTHERS** (si elle existe) est exécutée.

i. Création des objets

Pour créer un nouvel objet d'une classe, le programmeur fait appel à la méthode `New` sur la classe de l'objet :

```
var_référence := aClass.New ;
```

où `aClass` est le nom symbolique de la classe dont l'objet va être créé et `var_référence` est une variable référence qui va contenir la référence à l'objet créé.

Dans Guide, les classes exécutables sont aussi considérées comme des objets et ils sont tous de type `Class`, un type particulier. Ils appartiennent aussi à une classe, la classe `Class`. En conséquence, la création des objets devient l'appel de méthode (`New`) sur un objet (`aClass`) de la classe `Class`. Il y a une seule différence entre la création d'objet et l'appel de méthode : la classe est désignée par le nom symbolique dans l'instruction de création. Il faut chercher l'objet exécutable de la classe (une référence à lui) avant d'appeler la méthode `New` sur lui et cela peut être fait soit à la compilation, soit à l'exécution.

Si l'on cherche la classe exécutable statiquement à la compilation, on peut contrôler statiquement l'affectation de référence. Sinon un contrôle dynamique de conformité est nécessaire. Le choix entre deux méthodes de recherche est actuellement fait par le compilateur et il prend toujours la recherche dynamique, donc le code C généré pour une création d'objet est :

```
  <code pour chercher la référence de la classe>
/* _tmp0_ref contient cette référence */
  <code d'appel de la méthode New sur _tmp0_ref>
/* _ref_obj contient la référence à l'objet créé */
<code d'appel de la méthode _tmp1_ref.ObjectTypeRef>
/* _type_obj_ref est la référence du type obtenue */
/* _type_var_réf est la référence du type de la variable var_référence
      et elle est déterminée à la compilation */
if (conform(_type_obj_ref, _type_var_réf)) {
      var_référence = _ref_obj ;
} else {
      <code de traitement d'erreur>
}
<code de l'instruction qui suit l'instruction de création>
```

j. Génération du code pour l'instruction `CO_BEGIN ... CO_END`

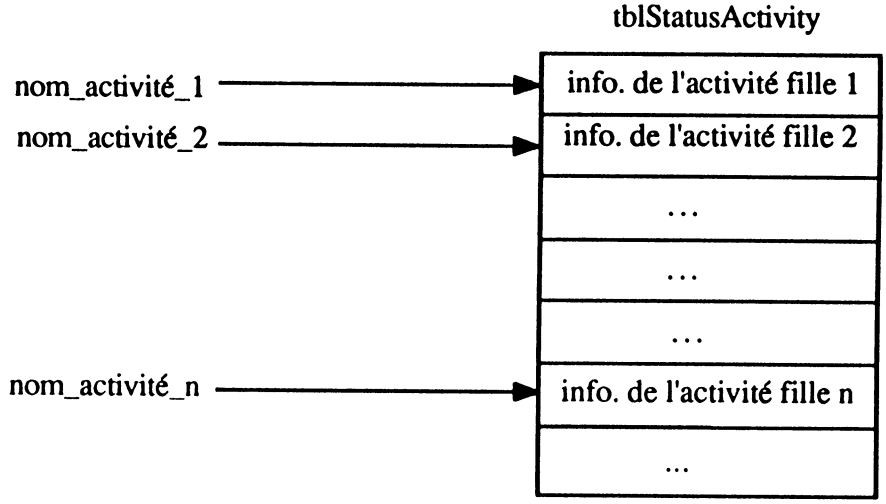
Comme nous l'avons présenté en 2.3.3, l'instruction `CO_BEGIN ... CO_END` est la seule possibilité en Guide de lancer explicitement des activités en parallèle. Elle s'écrit :

```
CO_BEGIN
      <nom_activité_1> : <appel de méthode sur un objet>
      <nom_activité_2> : <appel de méthode sur un objet>
      ...
      <nom_activité_n> : <appel de méthode sur un objet>
```

```
CO_END <expression_de_terminaison>
```

où **<expression_de_terminaison>** est une expression booléenne de terminaison de l'instruction qui est composée à partir des identificateurs des activités filles correspondant à chaque branche. Chaque identificateur porte une valeur booléenne à l'exécution pour indiquer si l'activité fille correspondante est terminée ou non.

Pour chaque activité, le système réserve une table d'état de ses activités filles dont chaque entrée contient les informations concernant l'activité fille correspondante.



En ce qui concerne la gestion des activités Guide, nous considérons les primitives suivantes dans l'interface du système Guide :

- `guideNewActivity (x : Pt_T_CallBlock)` est une primitive qui permet à une activité (mère) de créer une nouvelle activité (fille) en lui demandant d'exécuter une méthode sur un objet. Le paramètre **x** est l'adresse d'un bloc de communication de type `T_CallBlock` (cf. 4.5.6) qui est préparé par l'appelant (l'activité mère) et qui contient toutes les informations nécessaires. Le noyau crée une nouvelle activité en la considérant comme une activité fille de celle appelante. Puis l'état de l'objet ainsi que sa classe est chargé et lié dans l'espace d'exécution de la nouvelle activité. Enfin, la méthode appelée est exécutée sur l'objet chargé pour le compte de l'activité fille. Dès que la méthode se termine, le noyau enregistre cet événement dans un champ d'état correspondant dans l'activité mère et il appelle une fonction d'évaluation de la condition de réveil (voir le paragraphe suivant) pour décider de réveiller l'activité mère ou non.

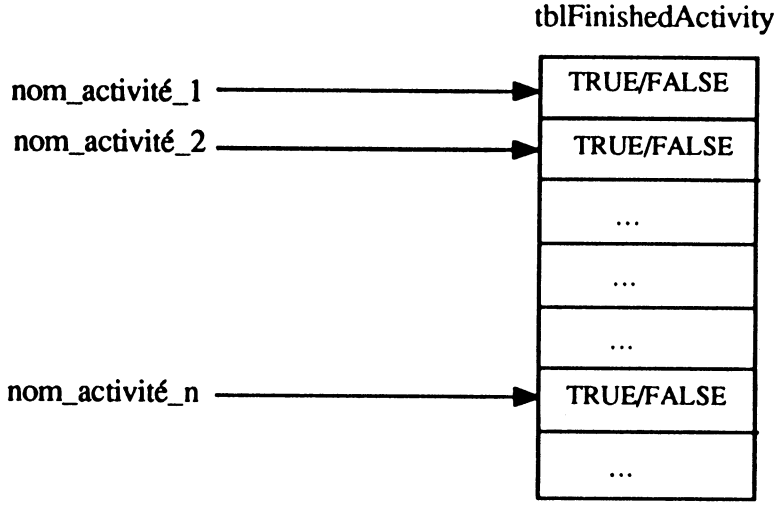
- `guideJoin (functionAdr)` est une primitive qui permet à une activité (mère) de se mettre dans l'état passif en attendant une condition de réveil. Dès la réception, le noyau met l'activité appelante en l'état passif. `functionAdr` est l'adresse de la fonction d'évaluation de la condition de réveil. Cette fonction utilise une table d'état des activités filles, créée et passée par le noyau, pour évaluer la condition. Cette fonction est appelée par le noyau chaque fois qu'une activité fille se termine,

et si la condition est vérifiée, le noyau réveille l'activité mère pour qu'elle puisse continuer son exécution.

Avec les deux primitives fournies par le système, présentées précédemment, la traduction de l'instruction CO_BEGIN est simple :

- On génère une fonction qui calcule l'expression de terminaison. Cette fonction utilise une table d'état des activités dont chaque entrée est un champ booléen qui indique si l'activité fille correspondante est terminée ou pas. La valeur de l'expression est aussi de type booléen. La traduction de l'expression de terminaison est simple en considérant que chaque identificateur symbolique de l'activité fille est traduit en un champ correspondant dans la table d'état :

nom_activité_i ====> tblFinishedActivity [i]



- A chaque branche de création d'activité dans l'instruction CO_BEGIN, on génère le code qui construit un bloc de communication T_CallBlock (cf. 4.5.6) et qui appelle la primitive guideNewActivity en lui passant le bloc de communication.
- Pour mettre l'activité mère en état passif et pour contrôler la terminaison de l'instruction CO_BEGIN, on génère le code qui appelle la primitive guideJoin en lui passant l'adresse de la fonction qui calcule l'expression de terminaison.

Par exemple, avec le code source suivant :

```
...
ref1, ref2 : REF Top ;
...
ref1 := Top.New ;
ref2 := Top.New ;
...
CO_BEGIN
        activity1 : ref1.Destroy ;
        activity2 : ref2.PrintStatus ;
CO_END activity2 ;
...
```

le compilateur génère le code suivant :

```
/* fonction de détection de la condition de terminaison */
_toto_stop0 (tblFinishedActivity)
T_Bool *tblFinishedActivity ;
{
        return (tblFinishedActivity [1]) ;
}

...
/* code généré pour l'instruction CO_BEGIN */
/* code généré pour la première branche */
<code pour construire le bloc de communication callBlock pour appeller Destroy>
FCT_guideNewActivity (&callBlock) ;
/* code généré pour la deuxième branche */
<code pour construire le bloc de communication callBlock pour appeller PrintStatus>
FCT_guideNewActivity (&callBlock) ;
/* code généré pour détecter la terminaison de l'instruction */
FCT_guideJoin (_toto_stop0) ;
/* code généré pour l'instruction suivante */
...
```

Après la troisième phase (la phase de génération de code C), on obtient le code C engendré et on appelle le compilateur `cc` pour produire le code objet d'Unix, puis on appelle l'éditeur de liens `ld` pour résoudre les références externes. Le résultat de la quatrième phase est un objet classe exécutable sans référence externe au sens Unix. Pour qu'il devienne un objet permanent et exécutable au sens Guide (cf. 4.5.2), on effectue certain traitement et on range le résultat dans la mémoire permanente d'objets (la MPO) grâce à la primitive `guidelink` du système Guide. Quand l'utilisateur appelle une méthode sur un objet ou quand il demande de créer un objet de cette classe, le noyau Guide charge et lie cette classe exécutable dans la mémoire d'exécution de l'activité appelante (cf. 4.5.2).

4.6 CONCLUSION

Le compilateur Guide a évolué en plusieurs versions. Actuellement, la version V1.5 supporte toutes les caractéristiques du modèle d'objets Guide présenté au chapitre 2 (voir aussi [Roisin 90]). Ce compilateur fonctionne bien entendu sous le prototype du système Guide et permet aux utilisateurs de programmer et d'exécuter leurs applications. Plusieurs programmes de démonstration écrits en Guide nous permettent d'évaluer notre modèle d'objets (cf.7.2.2) ainsi que la performance du compilateur. En ce qui concerne la performance du compilateur Guide, nous donnons ci-après quelques mesures effectuées sur le compilateur Guide et sur celui d'Eiffel (version 2.1), qui a une stratégie de compilation proche de celle de Guide.

Nous prenons au départ un jeux de programmes Eiffel et nous les traduisons en Guide (en ajoutant la synchronisation d'accès aux méthodes), puis nous compilons les programmes sur une même machine et nous faisons les mesures.

Mesures sur Eiffel

	100 méthodes	200 méthodes	300 méthodes	400 méthodes
temps passe 1	0.5s	1s	1s	1s
temps passe 2	1s	1.5s	2s	3s
temps passe 3	1s	1.5s	3s	4s
temps passe 4	49s	97s	150s	210s
temps passe 5	7.5s	8s	9s	10s
temps total de compilation	**69s**	**109s**	**165s**	**228s**
taille source	3034	6034	9034	12034
taille code C généré	45218	90121	134821	179521
taille du binaire .o	67642	134438	201142	267838
taille des informations pour le gestionnaire de Classe	4798	9998	15098	20198

Après la table ci-dessus, on a constaté que :

- 80 - 90% du temps est dans la compilation du code C produit.
- les autres phases de compilation sont à peu prêt constantes.
- le binaire est de 20-30 fois plus gros que le source.

Mesures sur Guide

	100 méthodes	200 méthodes	300 méthodes	400 méthodes
temps passe 1	3s	3s	4s	6s
temps passe 2	1s	4s	11s	20s
temps passe 3	5s	12s	19s	27s
temps passe 4	42s	60s	122s	170s
temps passe 5	14s	14s	22s	22s
temps total de compilation	**65s**	**93s**	**178s**	**245s**
taille source	4399	8699	12999	17299
taille code C généré	129789	251487	373503	495443
taille du binaire .o	37088	72212	107584	143036
taille des informations pour le gestionnaire de Type	1732	3431	5131	6831
taille des informations pour le gestionnaire de Classe	1783	3483	5183	6883

Après la table ci-dessus, on a constaté que :

- 70 - 80% du temps est dans la compilation du code C produit.
- 8 - 15% du temps pour insérer la classe dans la MPO.
- le binaire est de 9-10 fois plus gros que le source.

Dans les tables de mesures précédentes, nous constatons que :

- le langage Guide est plus bavard qu'Eiffel (mais selon les utilisateurs, son utilisation est agréable).
- la performance de la compilation du compilateur Guide est voisine de celle du compilateur Eiffel, à l'exception de la taille du binaire final. En Guide, on partage du code binaire entre la sous-classe et leur super-classe, ce qui rend la taille des classes exécutables la plus petite possible.

En résumé, les expériences suivantes concernant le compilateur sont, parmi d'autres, acquises :

- Le temps de compilation des types est très court parce qu'il n'y a pas de code généré pour les types.
- Le temps de compilation des classes est long car on doit générer un code beaucoup plus gros que le source (notamment le code C intermédiaire).
- Malgré la réalisation de certains mécanismes de compilation intelligente, le problème du maintien de la cohérence entre le source des classes et leur version exécutable n'est pas résolu ; c'est l'utilisateur qui doit assurer cette cohérence. Il faut décharger les utilisateurs de cette tâche en réalisant un gestionnaire de l'évolution des types et des classes (ce gestionnaire peut faire partie du gestionnaire de types et de classes).

Troisième partie

ETUDE et REALISATION DES RAMASSE-MIETTES pour GUIDE

CHAPITRE 5 : ETUDE SYNTHETIQUE DES ALGORITHMES DE RAMASSE-MIETTES EXISTANTS

Dans tous les systèmes qui créent dynamiquement les objets, un moyen de détruire automatiquement les objets inaccessibles, ***les miettes***, est indispensable. C'est le rôle d'un ***ramasse-miettes*** et de nombreux travaux sont consacrés à ce problème.

Les objets dans Guide sont persistants, partagés et répartis sur les différents sites du système. Cela implique que la tâche de détection et de ramassage des miettes dans Guide est une tâche compliquée, lourde, mais qui doit être résolue pour assurer une longue vie au système, surtout dans le cas où on l'utilise de façon massive. Afin de réaliser un ramasse-miettes adapté à Guide, il faut avoir une bonne connaissance des travaux concernant les algorithmes d'une part, et de la structure interne de la mémoire d'objets Guide d'autre part. Ainsi dans ce chapitre nous présentons les principaux algorithmes de ramasse-miettes pour des modèles différents : du modèle centralisé d'un processus au modèle réparti où plusieurs processus situés sur des sites différents partagent le même espace d'objets virtuel.

5.1 RAMASSE-MIETTES CENTRALISE

Dans ce paragraphe nous présentons les algorithmes de ramasse-miettes pour le modèle de mémoire le plus simple : le tas d'un processus d'application. Ces algorithmes servent de base pour les modèles plus compliqués et permettent de comprendre facilement les algorithmes de ramasse-miettes pour le modèle de mémoire répartie que nous présentons au paragraphe suivant.

5.1.1 MODELE DE MEMOIRE D'OBJETS

Le processus d'un programme d'application, communément appelé ***mutateur*** [Dijkstra 78], accède à un espace de mémoire géré dynamiquement. Cet espace de mémoire s'appelle ***le tas*** du processus. Chaque élément du tas est appelé objet et est accédé via des ***références*** (pointeurs) dont le contenu est l'adresse de l'objet. Alors que les références possèdent une taille fixe et connue, l'objet repéré est de taille quelconque et peut contenir éventuellement des références à d'autres objets. Pour accéder à son tas, le processus dispose principalement de primitives d'acquisition (***allouer***), de libération explicite (***libérer***) ou de modification d'accès (***affectation de références***) des objets dans le tas. Un objet du tas est accessible par le mutateur si on peut l'atteindre, directement ou indirectement, depuis un objet de la pile associée à l'application (Figure 5.1) :

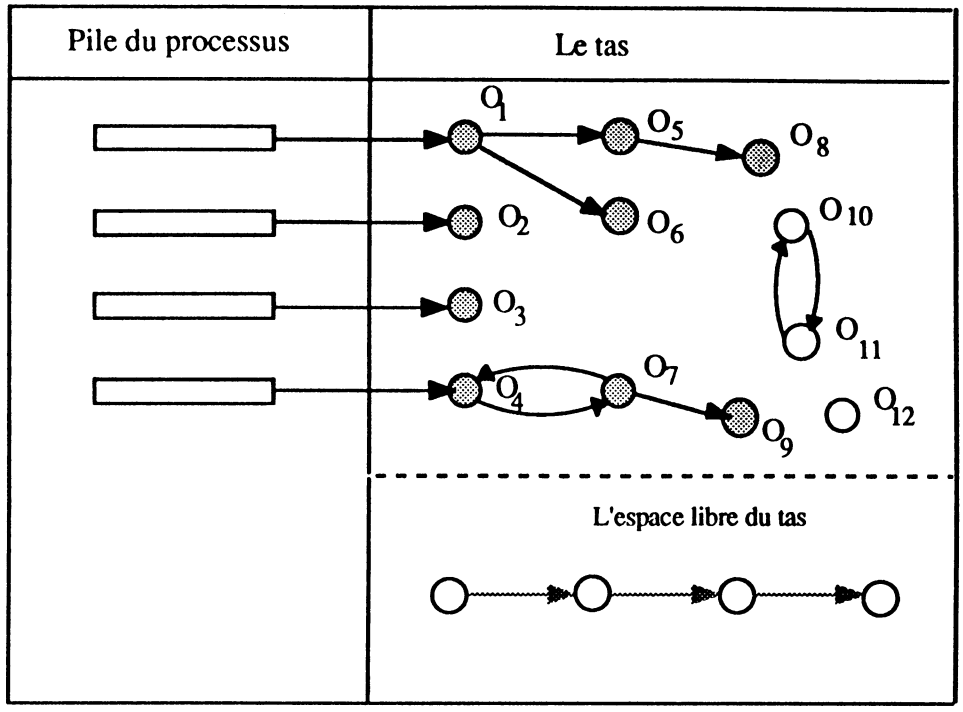


Figure 5.1 Modèle d'un tas

Nous faisons, par la suite, abstraction de la pile en la remplaçant par un objet unique et indestructible appelé la racine R_A. La figure 5.1 devient alors :

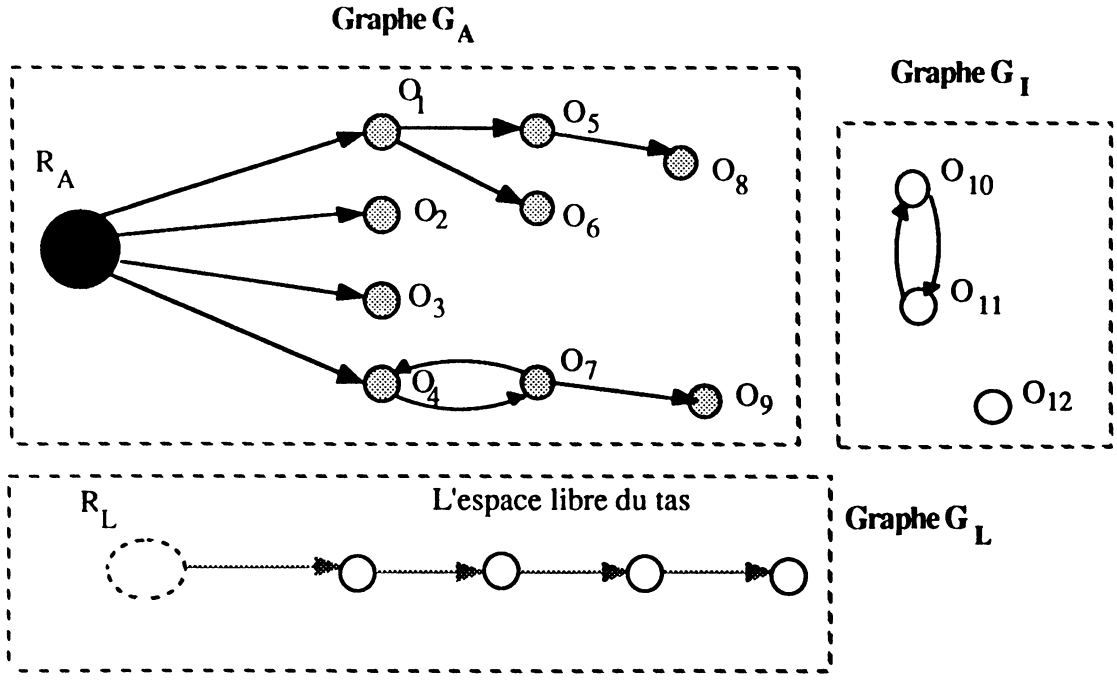


Figure 5.2 Abstraction d'un tas

Nous appelons ***successeur*** d'un objet O tout objet du tas repéré par l'une des références contenues dans O ; la notion inverse est celle de ***prédécesseur*** d'un objet. La fermeture transitive de ces relations (i.e. l'existence d'un chemin) nous amène à parler de ***descendants*** ou d'***ascendants***.

A chaque instant le tas G est divisé en trois parties disjointes :

- les objets accessibles (directement ou indirectement) depuis R_A, ce sont les descendants de R_A ;
- les objets libres, ce sont les objets qu'on peut prendre dans l'action d'allocation ;
- et les objets non libres et inaccessibles.

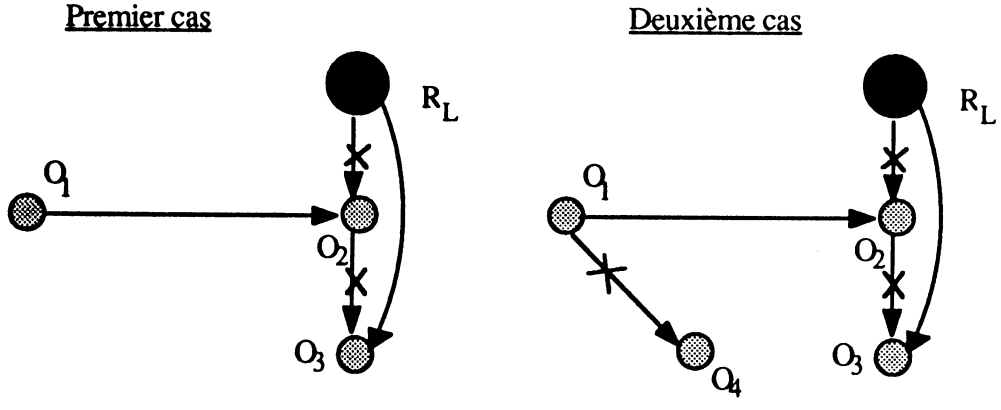
Supposons que les objets libres soient liés dans une liste qui possède une racine indestructible R_L. Dans ce cas, le tas G est partitionné en trois graphes :

- G_A, graphe des objets accessibles depuis R_A qui est son unique racine ;
- G_L, graphe des objets libres et R_L est son unique racine ;
- G_I, graphe des objets non libres et inaccessibles.

A l'aide des opérations allouer, libérer des objets et affectation de références (notée par la suite :=) exprimées en termes d'ajouts ou de retraits d'arcs dans G, le mutateur intervient sur les objets de $G_A \cup G_L$ de la façon suivante :

1• allouer un objet libre (O_2) et mettre sa référence dans un composant, un champ qui peut contenir une référence, de l'objet O_1.

O_1.une_référence := allouer() ;



où $O_1 \in G_A$, $O_2 \in G_L$ => retrait(O_1,O_4) ; {si O_1 pointe vers O_4}

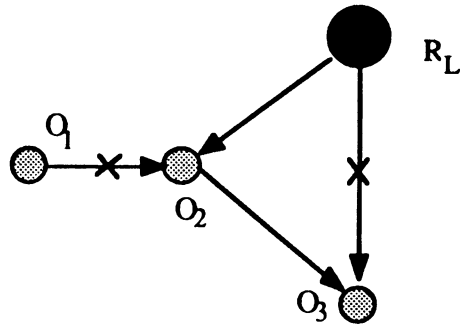
ajout(O_1,O_2) ; retrait(R_L,O_2) ;

ajout (R_L,O_3) ; retrait (O_2,O_3) ;

après avoir été alloué, l'objet O_2 appartient à G_A.

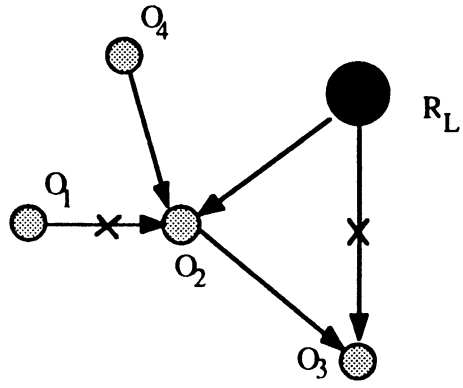
2• libérer un objet (O_2) référencé par une référence contenue dans l'objet O_1.

libérer(O_1.une_référence) ;



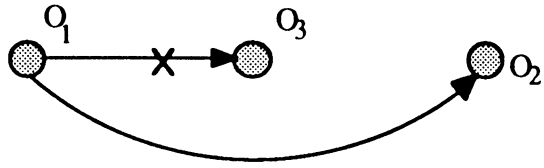
où $O_1, O_2 \in G_A$ => ajout(O_2,O_3) ; retrait(R_L,O_3) ;
ajout(R_L,O_2) ; retrait (O_1,O_2).

après avoir été libéré, l'objet $O_2 \in G_L$. Par ailleurs, si O_2 est accessible depuis un autre objet accessible (i.e. O_4 dans l'exemple ci-dessous), le problème d'incohérence se pose. C'est pour cette raison que le mutateur confie généralement l'action libérer à un processus bien particulier, le ***ramasse-miettes***, qui assure la libération sans incohérence.



3• affectation d'une référence (l'adresse) d'un objet (O_2) à un composant d'un autre objet (O_1).

O_1.une_référence := O_2 ;



où $O_1, O_2, O_3 \in G_A$ => retrait(O_1,O_3) ; ajout(O_1,O_2).

après cette action l'objet $O_3 \in G_A \cup G_I$, c'est-à-dire O_3 peut devenir miette.

Ainsi, l'état d'un objet peut évoluer de la façon suivante :

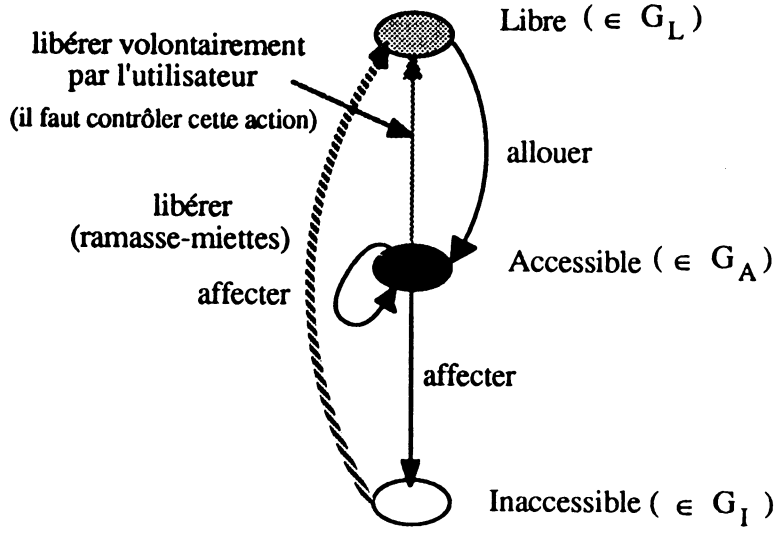


Figure 5.3 : Schéma d'évolution d'un objet

Le problème restant à résoudre est de faire passer les objets de G_I vers G_L, c'est le rôle d'un algorithme de ramasse-miettes, appelé ***collecteur***, afin d'assurer que toute demande d'allocation d'espace est acceptée tant que l'espace non accessible depuis R_A est suffisant pour satisfaire cette demande.

Lorsqu'une demande d'allocation nécessite une taille de zone de mémoire non disponible dans le tas, un tassement (ou ***compactage***) du tas peut s'avérer indispensable. Ce compactage est fait généralement par le ramasse-miettes aux moments convenables.

5.1.2 RAMASSE-MIETTES D'UN TAS EN CONTEXTE CENTRALISE

Dans la plupart des systèmes, le tas est partagé par plusieurs processus qui peuvent y accéder depuis leur pile d'exécution. Dans ce cas, le modèle de mémoire d'objets reste le même que celui du paragraphe 5.1.1 sauf que l'on a maintenant plusieurs objets racines R_A, chacun représentant une pile d'exécution d'un processus (mutateur). Les processus doivent respecter un protocole commun de synchronisation pour accéder aux objets dans le tas partagé.

Dans ce paragraphe, nous présentons les algorithmes de ramasse-miettes du tas d'un processus. Les algorithmes de ramasse-miettes du tas partagé par plusieurs mutateurs sont les mêmes que ceux du tas d'un processus sauf les cas cités clairement.

Le problème de ramasse-miettes dans un tas a été abondamment traité et les algorithmes proposés appartiennent à deux grandes familles :

- les compteurs de références,
- le marquage et la récupération des objets non marqués.

5.1.2.1 Techniques par compteurs de références

Dans les algorithmes basés sur ce principe il n'y a pas de véritable distinction entre le collecteur et le mutateur ; c'est en fait ce dernier qui prend totalement en charge la gestion des informations permettant de déterminer si l'objet est accessible ou non. Le rôle du collecteur est alors limité à la mise en file des objets libérés et au retassement éventuel du tas ; ces travaux sont souvent faits par des procédures appelées directement depuis le mutateur.

La méthode consiste à associer à chaque objet un entier de contrôle CR, comptabilisant le nombre de ses prédécesseurs ou, plus précisément, le nombre de références à l'objet qui existent dans ses prédécesseurs (une référence à un même d'objet peut exister en plusieurs exemplaires dans un même objet). Lorsqu'un objet est alloué et que sa référence est mise dans un autre objet, on met la valeur 1 à son champ CR. Pour un objet accessible, les opérations du mutateur concernant sa référence doivent mettre à jour son champ CR et lorsque sa valeur devient égale à 0, cela signifie que l'objet devient libre et qu'on peut le ramasser. Les opérations du mutateur sont alors :

- O_1.une_référence := allouer() ;

  => prend un objet libre O_i dans G_L et effectue O_i .CR := 1

- affectation O_1.une_référence := O_2 ;

  où $O_1, O_2, O_3 \in G_A$:

  + retrait (O_1,O_3) => O_3.CR := O_3.CR - 1

  + ajout (O_1,O_2) => O_2.CR := O_2.CR + 1

  Dans l'action retrait (O_1,O_3), si CR (O_3) passe à 0 on met O_3 dans G_L et on décrémente de 1 les compteurs des successeurs de O_3 qui pourraient devenir miettes (dans certaines mises en œuvre ce travail est fait dans l'action allouer).

Remarque : dans le contexte où le tas est partagé par plusieurs mutateurs, l'incrémentation ou la décrémentation d'un compteur par un mutateur doivent être indivisibles.

L'emploi de compteurs de références entraîne une bonne répartition dans le temps du travail de l'application et de la détection des objets inaccessibles ; par ailleurs, la gestion des compteurs est simple et ne nécessite pas d'interruption du programme utilisateur. La récupération des miettes est immédiate, lorsque le compteur d'un objet O_i passe à 0 l'objet est ajouté à G_L ; pour la descendance de O_i on peut :

- choisir de la traiter tout de suite ; on décrémente de 1 les compteurs des successeurs de O_i qui, à leur tour, peuvent éventuellement être ajoutés à G_L ;
- différer son traitement ; lorsque O_i est à nouveau réattribué à l'application, on décrémente les compteurs des successeurs de O_i qui peuvent, alors, éventuellement devenir libres.

Les inconvénients majeurs de l'approche par compteurs de références résident dans :

- l'obligation imposée à l'application de gérer les compteurs bien que ceci puisse être rendu transparent à l'utilisateur (par un compilateur par exemple) ;

- l'encombrement inhérent aux compteurs ; les champs compteurs de références sont souvent utilisés pour chaîner les objets libres, leur taille peut donc être celle d'une référence ;
- l'inadéquation de la méthode à détecter les structures circulaires inaccessibles ;
- la pénalité infligée au mutateur lors des opérations d'ajouts ou de retraits d'arcs dans G_A, surtout dans le cas où le tas est partagé par plusieurs mutateurs car dans ce cas, les deux actions d'incrémentation et de décrémentation associées à une affectation de référence doivent être rendues indivisibles. [Appel 87] pense que le coût de plus en plus faible des mémoires permet d'envisager un vaste espace d'adressage : il n'est plus alors nécessaire d'imposer au mutateur du travail destiné à la récupération ; lorsqu'une saturation du tas intervient, on active un algorithme de type parcours-marquage que nous présentons maintenant.

5.1.2.2 Techniques par parcours-marquage

Pour éviter les inconvénients de la technique par compteurs de références, on applique la technique de parcours-marquage. Les algorithmes basés sur cette technique réalisent les deux opérations suivantes :

- la construction du graphe G_A (à partir de la racine virtuelle R_A) recouvrant tous les objets accessibles afin d'en déduire le graphe des objets inaccessibles (le graphe des miettes) :

  $G_I = G - G_L - G_A$

- la mise à jour du graphe G_L des objets libres en y ajoutant les miettes détectées.

On n'a pas besoin d'avoir le graphe G_A avec toute la relation hiérarchique entre les nœuds dans le graphe pour déduire le graphe des objets inaccessibles. En effet, pour un objet donné, on a besoin de savoir uniquement s'il appartient au graphe G_A ou non. La construction du graphe G_A, ou phase de ***parcours-marquage***, peut être réalisée par un parcours en largeur (utilisation d'une file) ou en profondeur (utilisation d'une pile).

La mise à jour de G_L est appelée :

- phase de balayage lorsqu'elle est exécutée à la fin de la phase de parcours-marquage. Dans cette phase, on parcourt l'espace entier des objets et lorsque l'on rencontre un objet non marqué, on l'ajoute dans le graphe des objets libres G_L.
- phase de recopie (***compactage***) lorsqu'elle est exécutée au cours de la phase de parcours-marquage : chaque fois que l'on marque un objet (accessible), on le recopie aussi dans un autre espace. De cette façon, tous les objets accessibles sont recopiés dans le nouvel espace après la phase de parcours-marquage et il ne reste qu'une chose à faire : libérer tout l'espace parcouru.

La réalisation des phases parcours-marquage/balayage ou parcours-marquage/compactage est confiée à un processus appelé ***collecteur***. Pour faciliter la présentation, nous utilisons le terme '***parcours-marquage***' pour parler des algorithmes du type parcours-marquage/balayage, et le terme '***compactage***' pour les algorithmes du type parcours-

marquage/compactage. Dans ce paragraphe, nous ne présentons que des algorithmes du type parcours-marquage et les algorithmes du type compactage seront présentés au paragraphe suivant.

L'existence de deux processus mutateur et collecteur nous amène à considérer les deux approches suivantes :

- le mutateur est suspendu pendant l'exécution du collecteur ;
- le mutateur et le collecteur s'exécutent de manière simultanée.

1. Suspension du mutateur pendant l'exécution du collecteur

Le mutateur, lorsqu'il ne dispose plus d'un graphe des objets libres suffisant, active le collecteur et reste suspendu durant l'exécution de ce dernier. Ainsi, au cours de la phase de marquage du collecteur, les graphes G_A, G_I et G_L sont statiques et il est possible de construire exactement le graphe G_A. En conséquence, on peut déduire exactement le graphe G_I , c'est-à-dire qu'on peut ramasser totalement et seulement des miettes.

Pour éviter de parcourir un objet plusieurs fois, on associe à chaque objet O un champ ***couleur*** tel que, au cours du parcours-marquage, ***O.couleur*** est à blanc lorsque l'objet O n'a pas été visité et ***O.couleur*** est à noir lorsque O a été visité. Au début de la phase de parcours-marquage, tous les objets sont remis à blanc. La couleur des objets permet, d'autre part, de différencier les objets accessibles des objets inaccessibles à la fin de la phase de parcours-marquage. Ici, nous donnons l'algorithme qui parcourt le tas en profondeur en utilisant une pile p. Un objet est ***visité*** lorsqu'on a trouvé pour la première fois une référence à cet objet. A ce moment, on marque à noir l'objet et on empile sa référence. Le parcours d'un objet consiste à examiner toutes les références contenues dans cet objet. Lorsqu'on a terminé cet examen, l'objet est dit ***déjà parcouru***. Lorsqu'on rencontre une référence à un objet déjà visité ou parcouru, on n'a rien à faire. Avec cette définition des objets visités et des objets parcourus, les objets accessibles sont d'abord visités et puis parcourus dans la phase de parcours-marquage.

```
{initialisation du cycle de ramasse-miettes}
{p est la pile des objets à marquer}
{pour tout objet O, O.couleur = BLANC}
empiler(p, RA) ; RA.couleur = NOIR ;
{phase de parcours-marquage}
tantque non pile_vide(p) faire
       dépiler(p,O) ;
       pourtout Oi successeur de O
         faire
               si Oi.couleur = BLANC alors
                     empiler(p,Oi) ;
                     Oi.couleur = NOIR
               fsi
         fait
fait
```

```
{pour tout objet O :        O.couleur = BLANC ⇔ O inaccessible,
                            O.couleur = NOIR ⇔ O accessible ou libre}
{début de la phase de balayage}
pourtout objet O de la mémoire
faire
        si O.couleur = BLANC alors GL ⇐ GL ∪ {O} ;
        sinon O.couleur = BLANC ;
        fsi
fait
{pour tout objet O, O.couleur = BLANC}
```

L'approche présentée ci-dessus permet de récupérer tous les objets inaccessibles sans demander de travail supplémentaire au mutateur. Les inconvénients résident dans :

- l'inactivité imposée au mutateur lors de l'exécution du collecteur,
- l'occupation mémoire nécessaire à l'algorithme de parcours-marquage ; on trouvera dans [Cohen 81] de nombreuses références à des algorithmes permettant de réduire, voire de supprimer, la place mémoire utile au marquage.

Dans [Dijkstra 78], Dijkstra a proposé un algorithme de parcours-marquage en utilisant trois couleurs différentes. En effet, au lieu d'utiliser une pile de références, l'algorithme de Dijkstra utilise la troisième couleur, la couleur gris, pour dénoter les objets déjà visités mais pas encore parcourus. La signification des trois couleurs est alors :

- les objets blancs sont des objets non encore visités ; s'ils restent blancs à la fin de la phase de parcours-marquage, ce sont vraiment des miettes ;
- les objets gris sont les objets déjà visités mais non encore parcourus ; on doit les parcourir pour visiter leurs successeurs ;
- les objets noirs sont les objets déjà visités et parcourus.

Avec cet algorithme, on n'a pas besoin de place pour la pile de références mais on perd plus de temps pour chercher un objet visité (gris).

2. Exécution simultanée du mutateur et du collecteur

Dans le cas où les deux processus, mutateur et collecteur, s'exécutent de manière simultanée, les graphes G_A, G_I et G_L sont dynamiques au cours de la phase de parcours-marquage. En effet, les opérations de retrait d'arcs effectuées par le mutateur peuvent faire diminuer le graphe des objets accessibles G_A et augmenter le graphe des miettes G_I ; de même sur un ajout d'arc, G_A peut augmenter et G_L diminuer. Nous examinons deux cas différents :

a. Suppression d'arcs du mutateur lors du marquage

Considérons la situation du schéma 5.4 où $t_0 < t_1 < t_2 < t_3$:

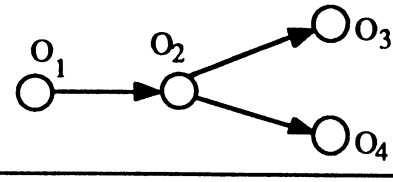
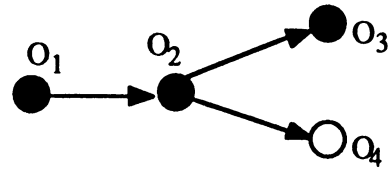
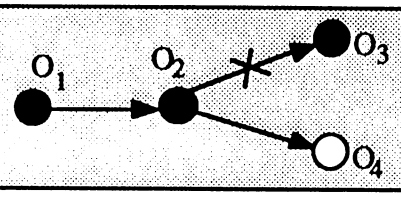
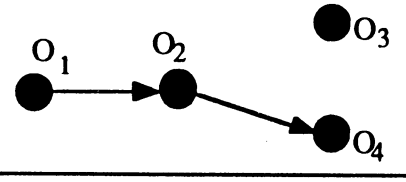
T	Mutateur	Collecteur	L'état de marquage
t_0	Inactif	Début de la phase de parcours-marquage => les objets sont blancs	
t_1	Inactif	Les objets O1, O2, O3 sont parcourus, ils sont noirs	
t_2	retire l'arc (O2,O3)	Inactif	
t_3	Inactif	Marque noir O4 et terminer la phase de parcours-marquage. O3 n'est pas libéré	

Schéma 5.4 : Des miettes récentes non pas ramassées

- à l'instant t_0, tous les objets sont blancs (pas marqués) ;
- à l'instant t_1, le collecteur a parcouru les objets O_1, O_2, O_3 ; ces objets sont marqués à noir ;
- à l'instant t_2, le mutateur retire l'arc (O_2,O_3) du graphe G_A ;
- à l'instant t_3, l'objet O_3, devenu inaccessible, est considéré accessible par le collecteur.

Ainsi, le collecteur récupère un graphe $G_I' \subset G_I$ et les objets inaccessibles de $G_I - G_I'$ sont les miettes engendrées récemment qui seront prises en compte lors de la prochaine activation du ramasse-miettes.

b. Ajout d'arcs du mutateur lors du marquage

Considérons la situation du schéma 5.5 où $t_0 < t_1 < t_2 < t_3 < t_4$:

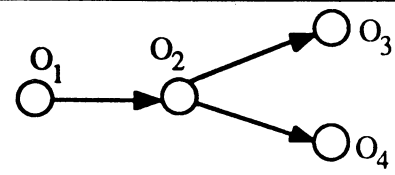
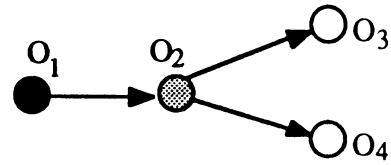
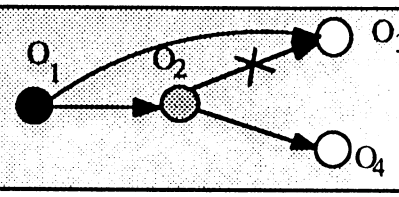
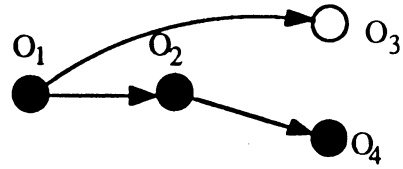
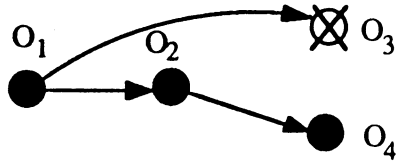
T	Mutateur	Collecteur	L'état de marquage
t_0	Inactif	Début de la phase de parcours-marquage => les objets sont blancs	
t_1	Inactif	L'objet O1 est parcouru O2 est visité et est mis en pile	
t_2	ajoute l'arc (O1,O3) retire l'arc (O2,O3)	Inactif	
t_3	Inactif	Parcourt O2 mais ne voit plus O3	
t_4	Inactif	Termine la phase de parcours-marquage Ramasse incorrectement l'objet O3	

Schéma 5.5 : Ramassage incorrect des objets accessibles (situation 1)

- à l'instant t_0, tous les objets sont blancs (pas marqués) ;
- à l'instant t_1, le collecteur parcourt O_1 et met en pile O_2 (O_2 n'est que visité) ;
- à l'instant t_2, le mutateur ajoute l'arc (O_1,O_3) et retire l'arc (O_2,O_3) du graphe G_A ;
- à l'instant t_3, le collecteur parcourt l'objet O_2 mais ne voit plus l'objet O_3 ;
- à l'instant t_4, l'objet accessible O_3 est ramassé incorrectement par le collecteur.

Remarque : nous utilisons la couleur grise dans les schémas d'illustration pour indiquer que l'objet est seulement visité et que ses successeurs ne sont pas encore traités.

Pour éviter la récupération incorrecte des objets accessibles, le mutateur doit participer au marquage lors de l'ajout d'arc. Ainsi, lorsqu'il ajoute l'arc (O_1,O_3) dans le schéma 5.5 ci-

dessus, il doit marquer à noir et mettre en pile l'objet O_3. L'action d'ajout du mutateur devient :

```
si O3 est blanc alors
       < le marquer noir et le mettre en pile ;
         ajout (O1,O3) proprement dit. >
sinon
         ajout (O1,O3) proprement dit.
fsi
```

Il est important de remarquer que les deux actions de marquer à noir l'objet repéré et d'affecter sa référence sont faites de façon indivisible vue du collecteur (nous utilisons ici les deux symboles <> pour le signifier), sinon le collecteur peut ramasser incorrectement les objets accessibles comme le schéma 5.6 le montre ; dans ce schéma, le mutateur est interrompu après avoir fait l'action de marquage de l'objet O_3. Pendant cette interruption, le collecteur termine son cycle de collection en cours et il fait un nouveau cycle ; il marque à blanc tous les objets, y compris l'objet O_3, donc l'action de marquage à gris du mutateur est défaite totalement avant que ce dernier puisse faire l'affectation :

- à l'instant t_0, tous les objets sont blancs (pas marqués) ;
- à l'instant t_1, le collecteur parcourt O_1 et met en pile O_2 (O_2 n'est que visité) ;
- à l'instant t_2, le mutateur veut affecter la référence de l'objet O_3 à l'objet O_1. Pour éviter le fait que l'objet O_3 soit ramassé accidentellement par le collecteur, le mutateur marque à noir et met en pile l'objet O_3 , puis il est interrompu avant de pouvoir affecter la référence de O_3 à O_1 ;
- à l'instant t_3, le collecteur parcourt les objets O_2,O_3,O_4 et termine le cycle sans récupération ;
- à l'instant t_4, le collecteur refait à nouveau un cycle de ramasse-miettes et tous les objets sont blancs (pas marqués) ;
- à l'instant t_5, le collecteur parcourt O_1 et met en pile O_2 ;
- à l'instant t_6, le mutateur ajoute l'arc (O_1,O_3) et puis retire l'arc (O_2,O_3) du graphe G_A ;
- à l'instant t_7, le collecteur parcourt l'objet O_2 mais ne voit plus O_3. Pour cette raison, l'objet accessible O_3 est ramassé incorrectement par le collecteur.

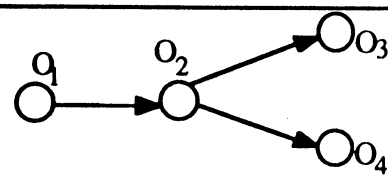
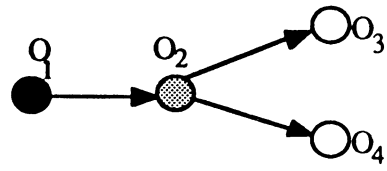
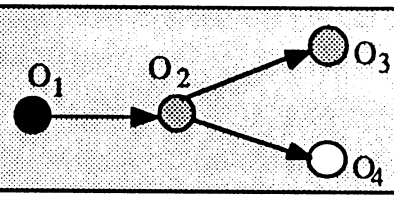
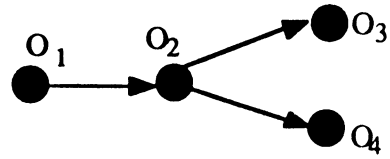
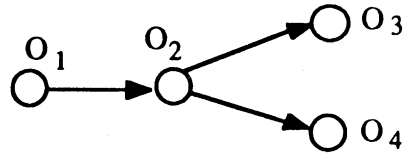
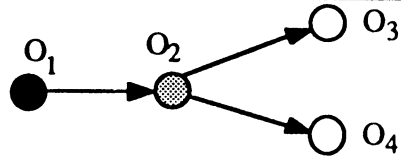
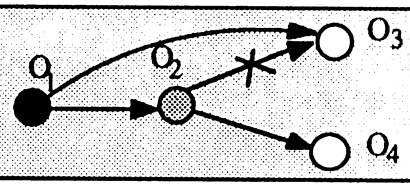
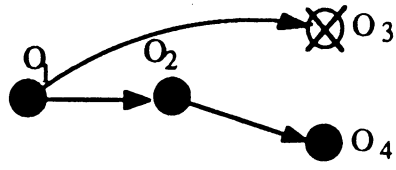
T	Mutateur	Collecteur	L'état de marquage
t_0	Inactif	Début de la phase de parcours-marquage => les objets sont blancs	
t_1	Inactif	L'objet O1 est parcouru O2 est visité et est mis en pile	
t_2	marque à gris O et le met en pile	Inactif	
t_3	Inactif	parcourt O2, O3, O4 Termine la phase de parcours-marquage	
t_4	Inactif	Début de la phase de parcours-marquage => les objets sont blancs	
t_5	Inactif	L'objet O1 est parcouru O2 est visité et est mis en pile	
t_6	ajoute l'arc (O1,O3) retire l'arc (O2,O3)	Inactif	
t_7	Inactif	parcourt O2 mais ne voit pas O3 ramasse O3	

Schéma 5.6 : Ramassage incorrect des objets accessibles (situation 2)

En conclusion, dans les algorithmes de type parcours-marquage, le collecteur s'exécute généralement lorsque le mutateur est interrompu. Si l'on veut laisser le collecteur marcher en parallèle avec le mutateur, ce dernier doit participer au marquage des objets chaque fois qu'il affecte la référence d'un objet à un autre : marquer à noir l'objet référencé avant d'affecter sa référence et ces deux actions doivent être faites de façon indivisible pour le collecteur. Cette

contrainte est un inconvénient majeur et on trouvera dans le chapitre 6 une solution pour l'éviter. Le collecteur, quant à lui, reste inchangé.

5.1.2.3 Algorithmes du type compactage

Dans les algorithmes du type parcours-marquage présentés au paragraphe précédent, on peut constater que le collecteur doit parcourir l'espace d'objets dans la phase de ramassage pour ramasser les miettes. Le temps de ce parcours est proportionnel au nombre d'objets de l'espace. Cette perte de temps devient plus sévère dans le cas où il y a beaucoup plus de miettes que d'objets accessibles. C'est pour résoudre ce problème que les algorithmes du type compactage ont été proposés. L'idée principale de ces algorithmes [Baker 78] est de recopier les objets accessibles dans un autre espace pendant la phase de parcours-marquage et de libérer totalement l'espace de départ.

A cet effet, les objets sont stockés dans un espace à deux parties : '***PrésentEspace***' qui contient tous les objets et '***FuturEspace***' qui va contenir les objets vivants lors du prochain cycle du collecteur. La partie '***FuturEspace***' est divisée en 3 régions différentes comme le montre la figure 5.7 :

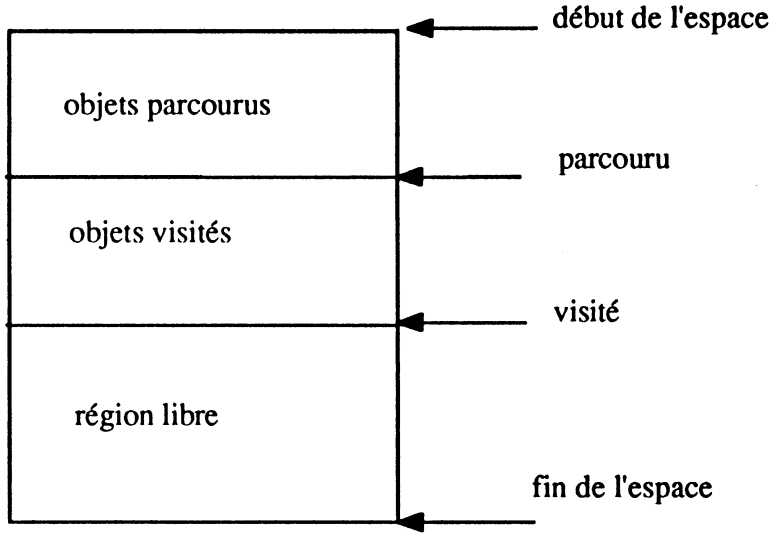


Figure 5.7 : Les régions de 'FuturEspace' et leur pointeur

- ***région des objets parcourus*** contient les objets déjà parcourus. Cette région est située entre le début de l'espace et le ***pointeur parcouru***, ce dernier est initialisé au début de l'espace au début de l'algorithme,
- ***région des objets visités*** contient les objets déjà visités mais pas encore parcourus. Cette région est située entre le ***pointeur parcouru*** et le ***pointeur visité***, ce dernier est initialisé au début de l'espace au début de l'algorithme,
- ***région libre*** est située entre le ***pointeur visité*** et la fin de l'espace.

Le mutateur ne voit que des objets dans '***PrésentEspace***'. Lorsque l'on a besoin de l'activation du collecteur, le mutateur est interrompu. Comme dans les algorithmes du type

parcours-marquage, on construit le graphe G_A des objets accessibles en les visitant et en les parcourant à partir de la racine R_A. L'action de visite et l'action de parcours d'un objet dans cet algorithme sont précisées :

Visite d'un objet :

- copie de l'objet dans '***FuturEspace***', dans la ***région d'objets visités*** et modification du ***pointeur visité*** de cette région ;
- marquage du champ de couleur de l'ancien exemplaire de l'objet qui reste dans '***PrésentEspace***'. On appelle souvent le champ de couleur '***est_copié***' car ce champ sert dans cet algorithme à déterminer si l'objet a été copié ou non pour toutes les prochaines consultations ;
- sauvegarde de la référence (l'adresse) du nouvel exemplaire de l'objet dans un champ particulier de l'ancien exemplaire que l'on appelle '***nouvelle_référence***'. Ce champ sert à la mise à jour des références vers cet objet.

Parcours d'un objet :

- balayage de l'état de l'objet pour mettre à jour les références qui y existent. Ces références repèrent toujours des exemplaires dans '***PrésentEspace***'. Il y a deux cas :
  - + référence à un objet déjà visité, c'est-à-dire déjà copié dans '***FuturEspace***'. Dans ce cas on remplace la référence par sa nouvelle version qui a été mise dans le champ '***nouvelle_référence***' de l'ancien exemplaire de l'objet référencé ;
  - + référence à un objet pas encore visité. Dans ce cas on doit visiter l'objet référencé avant de modifier la référence.

L'algorithme commence par le parcours de la racine R_A, la pile d'exécution du mutateur. Puis on parcourt de façon séquentielle les objets déjà visités mais pas encore parcourus (mis dans la région des objets visités). Lorsqu'il n'y a plus d'objet à parcourir (***pointeur parcouru = pointeur visité***), on exécute l'épilogue de l'algorithme :

- on libère l'espace '***PrésentEspace***',
- on échange le rôle de '***FuturEspace***' et '***PrésentEspace***',
- on termine le cycle de ramasse-miettes,
- on réveille le mutateur.

Le temps d'exécution d'un cycle de ramasse-miettes ne dépend maintenant que du nombre des objets vivants et de leur taille. C'est pour cette raison que les algorithmes de type compactage sont mieux adaptés aux systèmes où l'espace d'objets est grand mais où les objets accessibles sont beaucoup moins nombreux que les miettes. Mais lorsque le nombre d'objets vivants augmente et devient important, le temps nécessaire pour accomplir un cycle devient

inacceptable pour les applications interactives. Pour résoudre ce problème, il y a deux grandes voies de traitement :

- soit on contrôle le nombre et la taille des objets vivants pour assurer que le temps nécessaire à un cycle de ramasse-miettes reste inférieur à un seuil acceptable.
- soit on laisse le collecteur s'exécuter en parallèle avec le mutateur en s'assurant que le mutateur n'accède qu'à des objets valides.

L'algorithme de générations d'objets proposé pour le système Smalltalk-80 par D.Ungar [Ungar 84] que nous résumons dans le paragraphe suivant représente la première voie.

Algorithme de générations d'objets

Après une longue étude sur le comportement des objets dans le système Smalltalk-80, D. Ungar a constaté que la plupart des jeunes objets meurent très rapidement et que les vieux objets ont tendance à rester permanents. Les jeunes objets sont des objets utilisés de façon interne dans une procédure, une méthode ou une activité. Les vieux objets sont des objets qui sont indépendants des activités et qui sont partagés par ces dernières. A partir de cette remarque, Ungar a défini un ramasse-miettes portant uniquement sur les jeunes objets en utilisant un algorithme basé sur la génération des objets. Le but de cet algorithme est de contrôler le nombre et la taille des objets vivants pour garantir que le temps nécessaire à un cycle de ramasse-miettes reste inférieur à un seuil acceptable. Ce seuil est le temps maximum d'interruption des applications interactives (mutateurs).

Dans cet algorithme, les objets sont classés en plusieurs générations mais pour simplifier la présentation, nous nous limitons à deux, les vieux et les jeunes. Les vieux objets sont considérés comme permanents et ne sont pas sujets au ramasse-miettes. Pour savoir si un objet est jeune ou non, on utilise un champ '*âge*' associé à chaque objet. Si l'âge d'un objet est supérieur à un seuil (fixe ou modifié par le système de temps en temps), on dit qu'il devient vieux.

Les vieux objets sont stockés dans un espace qui s'appelle '*VieuxEspace*'. Lorsqu'un vieil objet référence un jeune objet, on mémorise la référence au premier dans une '*liste d'objets mémorisés*'. Ce traitement est fait à chaque fois qu'on affecte une référence à un jeune objet dans un vieil objet. Par contre, quand un vieil objet ne référence plus aucun jeune objet, on peut supprimer sa référence de la *liste d'objets mémorisés*. Cette dernière est considérée comme une racine dans cet algorithme.

Les jeunes objets sont stockés dans un espace à 2 parties : la première est réservée pour de nouveaux objets et des objets vivant depuis le dernier cycle de ramasse-miettes, elle est appelée '*PrésentEspace*'. La deuxième va contenir des objets vivant après la prochaine copie, elle est appelée '*FuturEspace*' et elle se compose de 3 régions (fig. 5.7). Cette partie est vide pendant l'exécution des mutateurs et n'est pas accessible depuis ces derniers.

On doit interrompre les mutateurs de temps en temps pour exécuter le ramasse-miettes. Cette interruption est décidée par le système. On construit le graphe G_A des objets accessibles en les visitant et en les parcourant à partir de la racine fictive R_A qui se compose de :

- la ***liste des objets mémorisés*** ;
- des piles d'exécution des mutateurs.

L'action de visite et l'action de parcours d'un objet dans cet algorithme sont précisées :

Visite d'un objet :

- on ne fait rien lorsque l'objet est vieux ;
- augmentation de l'âge de l'objet. Si son âge dépasse le seuil évalué par le système, on le copie dans '***VieuxEspace***' et il devient vieux et n'est plus objet de ramassage pour les cycles suivants. Sinon, on le copie dans '***FuturEspace***', dans la région d'objets visités et on modifie le ***pointeur visité*** de cette région ;
- marquage du champ '***est_recopié***' de l'ancien exemplaire de l'objet qui reste dans '***PrésentEspace***'. Ce champ sert à déterminer si l'objet a été copié ou non pour toutes les prochaines consultations ;
- sauvegarde de la référence (l'adresse) du nouvel exemplaire de l'objet dans le champ '***nouvelle_référence***' de l'ancien exemplaire. Ce champ sert à la mise à jour des références à cet objet qui existent dans d'autres objets lors de leur parcours que nous présentons à la suite.

Parcours d'un objet :

- balayage de l'état de l'objet pour mettre à jour les références qui y existent. Ces références repèrent soit des exemplaires dans '***PrésentEspace***', soit des vieux objets. Il y a trois cas :
  + référence à un vieil objet, on ne fait rien ;
  + référence à un objet déjà visité, c'est-à-dire déjà copié dans '***FuturEspace***' ou '***VieuxEspace***'. Dans ce cas on remplace la référence par sa nouvelle version qui a été mise dans le champ '***nouvelle_référence***' de l'ancien exemplaire de l'objet référencé ;
  + référence à un objet pas encore visité. Dans ce cas on doit visiter l'objet référencé avant de modifier la référence.

Remarques :

- les jeunes objets qui deviennent vieux au cours d'un cycle de ramasse-miettes sont considérés comme des jeunes objets dans ce cycle ;
- la région des objets visités se compose de deux parties disjointes : une dans '***FuturEspace***' et l'autre dans '***VieuxEspace***'.

On doit contrôler que le temps d'exécution d'un cycle de ramasse-miettes (temps interrompu des mutateurs) est inférieur à un seuil (<100ms dans SmallTalk-80) pour que les applications interactives puissent fonctionner.

Comme cet algorithme ne traite pas les vieux objets, un algorithme de type parcours-marquage global est nécessaire dans ce système pour ramasser les vieux objets inaccessibles.

Algorithme de compactage dans une mémoire virtuelle ayant la protection des pages

Dans le cas où la mémoire d'objets est une mémoire virtuelle paginée ayant la protection des pages, [Ellis 88] propose un algorithme du type compactage qui fonctionne en parallèle avec les mutateurs. Pour la simplicité de la présentation, nous nous limitons au cas où la taille de chaque objet est inférieure ou égale à une page de mémoire pour qu'il y soit mis entièrement.

Comme la plupart des algorithmes de type compactage, [Ellis 88] divise l'espace de mémoire d'objets en deux parties. La première partie s'appelle '***PrésentEspace***', c'est l'espace de travail des mutateurs et ces derniers n'accèdent qu'à cette partie. Lorsqu'un compactage est nécessaire, les mutateurs sont interrompus (mais seulement pour un temps très court) et le collecteur est lancé. Dans un premier temps, le collecteur parcourt toutes les piles d'exécution des mutateurs, recopie tous les objets référencés directement dans une autre partie et met à jour les références en question. La partie destinataire devient '***PrésentEspace***' et l'ancien '***PrésentEspace***' devient '***VieuxEspace***' qui contient les objets à copier et qui est accédé uniquement par le collecteur. Chaque fois que le collecteur termine le parcours d'une pile d'un mutateur, ce dernier est réveillé. Pendant le travail du collecteur, '***PrésentEspace***' est divisé en 4 régions différentes comme le montre la figure 5.8 :

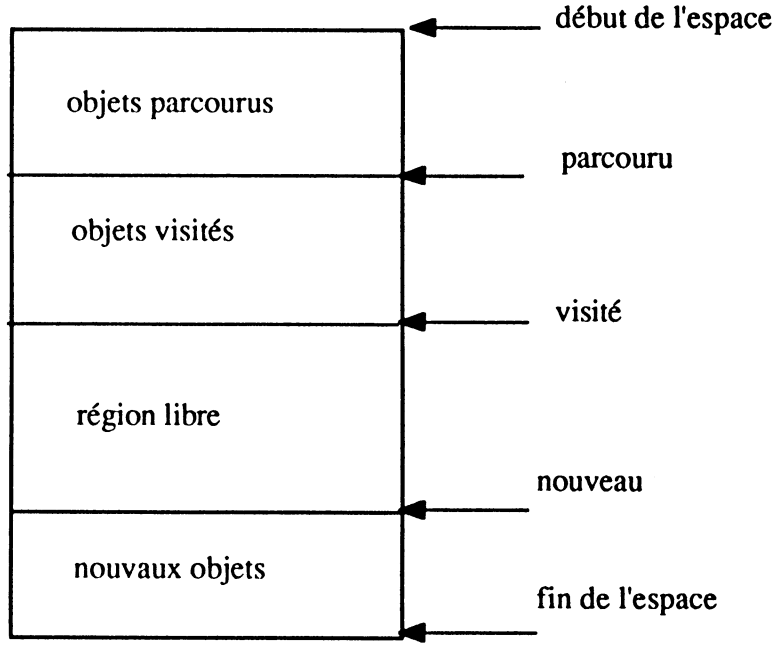


Figure 5.8 : Les régions de 'PrésentEspace'

La signification des régions reste inchangée par rapport à celle présentée dans l'algorithme de base (cf. § 3 ci-dessus) sauf la ***région des nouveaux objets*** qui est utilisée pour contenir les objets créés par les mutateurs pendant le cycle de ramasse-miettes du collecteur.

Comme nous l'avons dit, après le parcours des piles d'exécution des mutateurs, ces derniers sont réveillés pour continuer à travailler. Quant au collecteur, il continue, lui aussi, sa tâche : parcours des objets visités mais pas encore parcourus. Pour que les mutateurs accèdent toujours à des objets valides pendant l'exécution du collecteur, l'algorithme du collecteur doit maintenir les invariants suivants :

- les mutateurs ne voient que des références à des objets dans '***PrésentEspace***'. Ils ne peuvent donc jamais accéder aux objets dans '***VieuxEspace***' ;
- les objets dans la '***région de nouveaux objets***' ne contiennent que des références à des objets dans '***PrésentEspace***' ;
- les objets dans la '***région d'objets parcourus***' ne contiennent que des références à des objets dans '***PrésentEspace***' ;
- les objets dans la '***région d'objets visités***' contiennent des références à des objets soit dans '***PrésentEspace***' soit dans '***VieuxEspace***'.

Une solution évidente pour maintenir les invariants présentés ci-dessus est d'interdire aux mutateurs d'accéder à la ***région d'objets visités***. Pour cela, on laisse les mutateurs travailler en '***mode protégé***' pour qu'ils ne puissent pas accéder à la ***région d'objets visités*** dont les pages sont verrouillées en mode protégé par le collecteur. Quant à lui, il fonctionne en '***mode non-protégé***' et il accède librement soit à '***PrésentEspace***' soit à '***VieuxEspace***'.

Les mutateurs ne voient que des références à des objets dans '***PrésentEspace***' et ils n'accèdent qu'à cet espace. Il y a deux cas :

- lorsqu'un mutateur accède aux régions sauf la ***région d'objets visités***, il peut y accéder librement et il ne voit que des références dans '***PrésentEspace***' ;
- lorsqu'un mutateur accède à un objet O dans la ***région d'objets visités***, la protection est violée et une interruption est déclenchée. C'est le collecteur qui traite cette interruption ; il parcourt l'objet en question, c'est-à-dire qu'il copie les successeurs de l'objet dans la '***région d'objets visités***' de la partie '***PrésentEspace***', puis il modifie le mode de protection de l'objet en question car il appartient maintenant à la ***région d'objets parcourus***. Lorsque le mutateur reprend le contrôle, il accède à l'objet normalement.

Le collecteur fonctionne en tâche de fond ; il balaye la ***région d'objets visités*** et copie les objets accessibles qui restent encore dans '***VieuxEspace***'. Lorsque la ***région d'objets visités*** devient vide, les invariants ci-dessus confirment que les objets restant dans '***VieuxEspace***' sont inaccessibles et qu'on peut libérer cette partie. Le cycle de ramasse-miettes est terminé et le collecteur s'arrête. Les mutateurs continuent à s'exécuter et lorsqu'un nouveau cycle de compactage est nécessaire, le collecteur est relancé pour faire un autre cycle.

Le temps maximum d'interruption des mutateurs est le temps nécessaire pour parcourir toutes les piles d'exécution des mutateurs.

5.2 RAMASSE-MIETTES EN REPARTI

5.2.1 MODELE DE MEMOIRE D'OBJETS REPARTIE FREQUEMMENT UTILISE

Nous considérons un système distribué composé de plusieurs sites ayant des identités distinctes ; sur chacun des sites résident deux processus particuliers appelés mutateur et collecteur. Le premier est chargé de l'exécution d'un programme utilisateur alors que le second est chargé de la récupération des objets rendus inaccessibles par les opérations du programme utilisateur. La seule possibilité de communication entre sites est l'échange de messages à travers un réseau de communication que nous supposons fiable, c'est-à-dire que, sauf indications contraires, les hypothèses suivantes sont vérifiées :

- il n'y a ni perte ni altération de messages,
- les communications sont asynchrones,
- le déséquencement de messages est possible,
- le délai de transmission d'un message est quelconque mais fini.

Chaque programme utilisateur dispose d'un espace d'adressage virtuel lui permettant d'accéder à des objets sur son site de résidence et à des objets sur d'autres sites. Les communications inter-sites peuvent être soit des valeurs d'objets soit des références à des objets. Dans ce paragraphe, nous nous limitons au cas où les programmes ne se communiquent que des références à des objets.

Nous notons G le graphe, distribué sur l'ensemble des sites, de tous les objets du système (accessibles, libres et inaccessibles) composé des graphes G^i de chaque site i. Un objet quelconque d'un site i est accessible lorsqu'il appartient à $\bigcup_i G^i_A$ et libre lorsqu'il appartient à $\bigcup_i G^i_L$. Dans les autres cas, il est dit inaccessible.

A un instant donné et sur un site, un objet accessible est dit :

- ***privé*** ou ***accessible localement*** s'il n'est référencé directement par aucun objet distant,
- ***global*** ou ***accessible globalement*** s'il est référencé directement par un ensemble non vide d'objets distants et éventuellement par des objets locaux sur le même site.

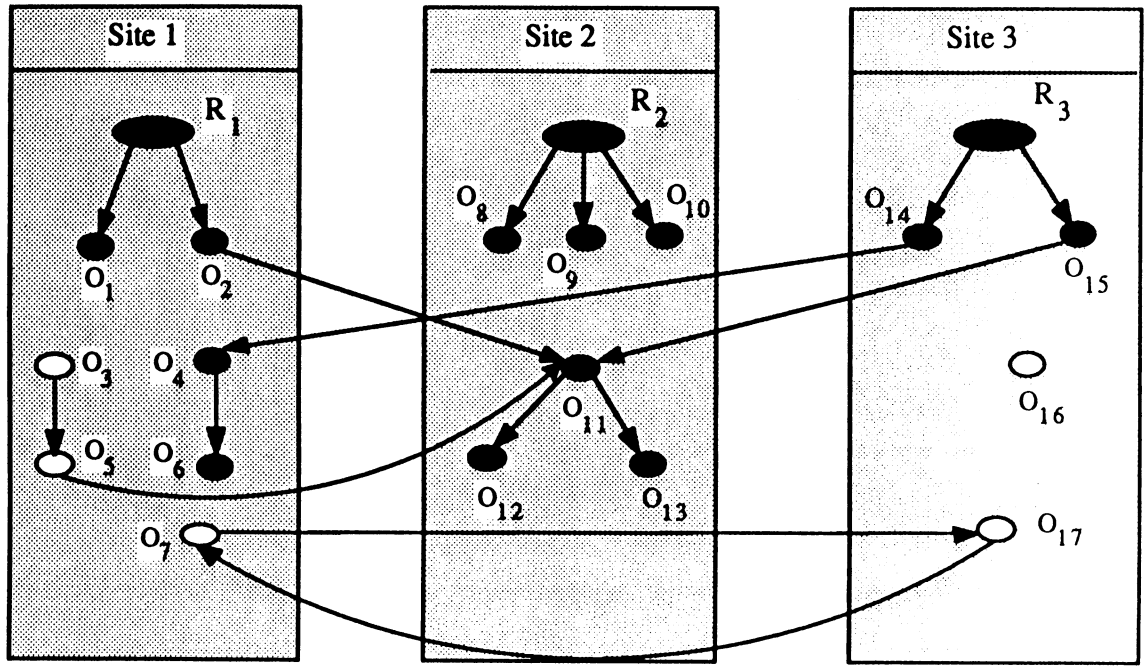


Figure 5.9 Exemple de la mémoire répartie

Pour l'exemple de la figure 5.9, les objets du site 1 vérifient :

- O_1, O_2, O_3, O_5, O_6 sont privés,
- O_4, O_7 sont accessibles globalement.

5.2.1.1 Les tables d'adressage

Les techniques proposées pour la mise en œuvre des objets accessibles globalement (référencés directement à distance) utilisent généralement :

- d'une part deux tables pour chaque site i :
  - + ***TabDef*$_i$** : contient l'ensemble des références à des objets accessibles globalement situés sur le site i ;
  - + ***TabRef*$_i$** : contient l'ensemble des références à des objets accessibles globalement référencés par les objets du site i mais situés sur d'autres sites.
- et d'autre part un mécanisme de '***triple indirection***' entre la référence à un objet et l'objet lui-même, comme le décrit la figure 5.11.

Par exemple, la référence distance O_i--O_j entre les sites i et j (figure 5.10) :

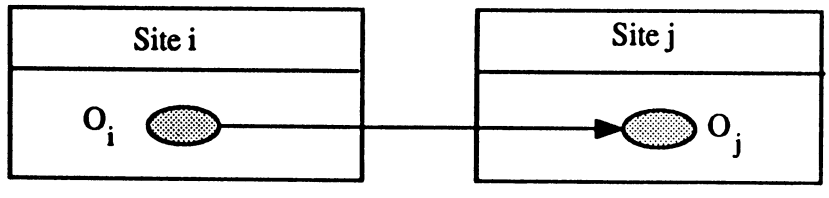


Figure 5.10

est mise en œuvre par la triple indirection (figure 5.11) :

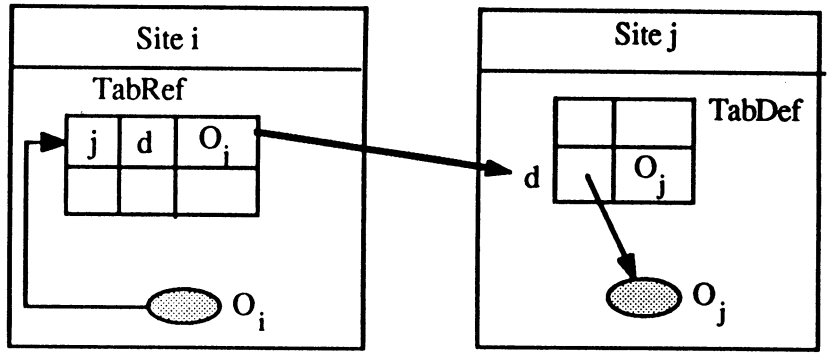


Figure 5.11

Le couple `p = (j,d)`, où `j` est le numéro du site propriétaire de l'objet O_j et `d` l'entrée associée à O_j dans $TabDef_j$, est appelé ***référence distante*** de l'objet O_j. Ce mécanisme de triple indirection donne la possibilité au site propriétaire de reloger un objet accessible globalement (en cas de retassement de la mémoire par exemple) sans avertir les sites référençant cet objet. Avec l'exemple de la figure 5.9, on obtient la mise en œuvre suivante :

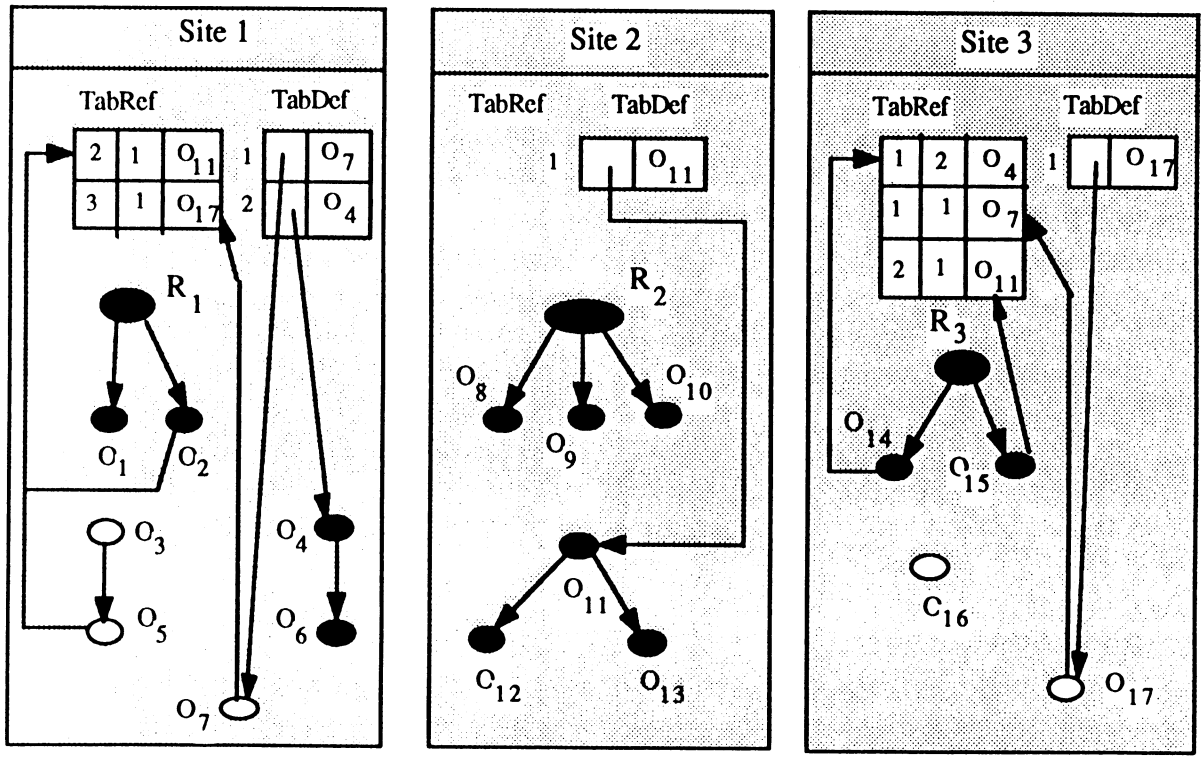


Figure 5.12

5.2.1.2 Les opérations sur les objets

Dans cette section, nous allons voir comment les actions réparties (allouer, libérer et affectation de référence) du mutateur s'expriment en fonction des deux tables $TabDef_i$ et $TabRef_i$, leur homologue réparti. A partir des choix de [Ali 85], [Hudak 82], [Hughes 85] et [Rudalics 86], on définit trois catégories de primitives pour la création de références distantes.

a. Création de références par diffusion

Chaque mutateur i peut diffuser des références distantes des objets de son site aux autres mutateurs, pour cela, il dispose des fonctions :

- ***Implanter(O)*** où O est un objet local au site i. Cette fonction consiste à créer l'objet O s'il n'existe pas encore.
- ***CréerDef(O,p)*** où O est un objet local au site i. Cette primitive permet de créer une référence distante p (une entrée dans la table TabDef) en vue de son transfert vers un autre site.
- ***Diffuser(O,p)*** permet au site i d'envoyer vers d'autres sites la désignation de l'objet O et la référence distante associée p.

Lors de la récupération d'un message ***Diffuser(O,p)*** venant du site i, le site k crée une entrée dans sa table ***TabRef*** pour enregistrer cette référence distante de l'objet O.

b. Création de références par demande au site propriétaire

Chaque mutateur i peut demander des références distantes aux autres mutateurs, il utilise alors des primitives :

- ***LierRef(O_i,O_j,site j)*** pour affecter la référence de l'objet distant O_j à O_i

```
cas     Oj ∈ TabRefi (entrée r) => Oi repère r
        Oj ∉ TabRefi => envoyer DemandeRef(Oi,Oj) à site j
fcas
```

- Lors de la réception de ***DemandeRef(O_i,O_j)*** de site i, le site j doit faire

```
cas     Oj ∉ sitej => envoyer RefusRef(Oi,Oj) à site i
        Oj ∈ sitej =>
              si Oj ∉ TabDefj alors
                    si Oj non implanté alors
                          Implanter (Oj)
                    fsi ;
                    CréerDef(Oj,p) ;
              sinon (entrée d)
                    p := (j,d) ;
              fsi ;
              envoyer RetourRef(Oi,Oj,p) à site i
fcas
```

- Lors de la réception de ***RetourRef(Oi,Oj,p)*** de site j, le site i fait :

```
faire
  création d'une entrée r dans TabRefi
  pour conserver la référence distante p ;
  faire Oi repère r ;
fait
```

c. Création de références par demande à un site non propriétaire

Chaque mutateur i peut aussi demander une référence distante à un site non propriétaire de l'objet mais disposant de sa référence. La primitive ***DemandeRef*** est alors remplacée par ***PasserRef()*** et

- Lors de la réception de ***PasserRef(Oi,Oj,p)*** venant du site i, le site k fait :

```
cas     Oj ∈ TabRefk (entrée r qui contient (p,Oj))
              => envoyer RetourRef(Oi,Oj,p) à site i

        Oj ∉ TabRefk => suivant la stratégie du mutateur :
              message de refus ou
              diffusion de la demande vers un autre site
fcas
```

5.2.2 RAMASSE-MIETTES DE TYPE COMPTEURS DE REFERENCES

Comme nous l'avons vu en 5.1.2.1, le travail du collecteur est, en grande partie, effectué par le mutateur ; pour adapter la technique par compteurs de références dans un contexte distribué autorisant le partage d'objets entre sites, il est nécessaire d'aménager les opérations du mutateur décrites en 5.1.2.1.

5.2.2.1 Algorithme dérivé directement du centralisé

Avec cet algorithme, chaque objet possède un champ compteur CR qui comptabilise le nombre de références existant dans ses prédécesseurs (locaux et distants). Pour affecter une référence distante, on utilise la fonction ***LierRef*** et afin de maintenir à jour le champ $CR(O_i)$, l'opération ***LierRef*** doit transmettre le message ***DemandeRef*** même si le site demandeur possède déjà la référence souhaitée (cette dernière existe déjà dans la table ***TabRef***). Enfin, il faut prévoir des opérations correspondant à la suppression de références distantes : la destruction de la liaison O_i--O_j entre sites i et j provoque l'envoi au site j d'un message indiquant qu'il faut décrémenter $CR(O_j)$ de 1. Les inconvénients de cette solution sont :

a. le surcoût puisque les affectations locales (***LierRef*** avec référence distante déjà dans ***TabRef***) nécessitent un envoi de message au site propriétaire.

b. l'inadéquation à résoudre le problème de demande d'une référence déjà présente sur le site. Si le mutateur du site 1 exécute consécutivement les deux actions suivantes :

+ ***LierRef*** $(O_1, O_2, 2)$: envoi d'un message au site 2 pour incrémenter le compteur de références de O_2,

+ Suppression de la liaison O_1--O_2 : envoi d'un message au site 2 pour décrémenter le compteur de O_2,

alors un déséquencement des deux messages provoque une libération prématurée de O_2.

c. l'inadéquation à résoudre le problème de passage de références entre sites. Si l'on a les opérations consécutives suivantes :

+ C3 : envoyer ***PasserRef*** (O_3, O_2) au site 1 par le site 3,

+ C1 : envoyer ***RetourRef*** (O_3, O_2, p) au site 3 par le site 1

et puis les opérations :

+ P1 : suppression de la liaison O_1--O_2 par le site 1, c'est-à-dire envoi d'un message pour décrémenter le compteur de O_2 au site 2 ;

+ P3 : lors de la réception de ***RetourRef*** (O_3, O_2, p) du site 1 par le site 3, envoyer un message pour incrémenter le compteur de O_2 au site 2.

Les opérations se passent correctement si le site 2 reçoit le message d'incrémentation du site 3 avant le message de décrémentation du site 1. Par contre, si l'ordre de réception est inversé, le site 2 peut libérer prématurément l'objet O_2.

5.2.2.2 Distribution du compteur

Pour éviter les inconvénients a) et b) de la solution présentée dans 5.2.2.1 ci-dessus, [Ellis 88] propose de distribuer les compteurs. Nous rappelons ici que chaque objet accessible globalement possède une entrée dans la table ***TabDef*** de son site et que chaque objet distant référencé par un ou plusieurs objets locaux possède une entrée dans la table ***TabRef***. Toutes les liaisons entre sites doivent être faites via les deux tables.

Dans cet algorithme, chaque entrée de la table ***TabDef*** et ***TabRef*** est considérée comme un objet. Avec le mécanisme de "***triple indirection***" (cf. 5.2.1.1), on peut distribuer le compteur d'un objet O du site i, accessible globalement, de manière suivante :

- CR(O) est le nombre de références existant dans les prédécesseurs locaux de O, y compris son entrée dans la table ***TabDef*** ;
- pour son entrée $TabDef_i[d]$, sa référence distante $p = (i,d)$, le champ CR comptabilise le nombre de sites possédant la référence distante p de O, c'est également le nombre d'entrées de toutes les tables ***TabRef*** qui repèrent la référence distante p de O,
- pour une entrée $TabRef_j[r]$ du site j possédant la référence distante p de O, le champ CR est le nombre de références à O (distant) existant dans les objets locaux du site j.

Par exemple, on obtient les résultats suivants avec le schéma 5.13 :

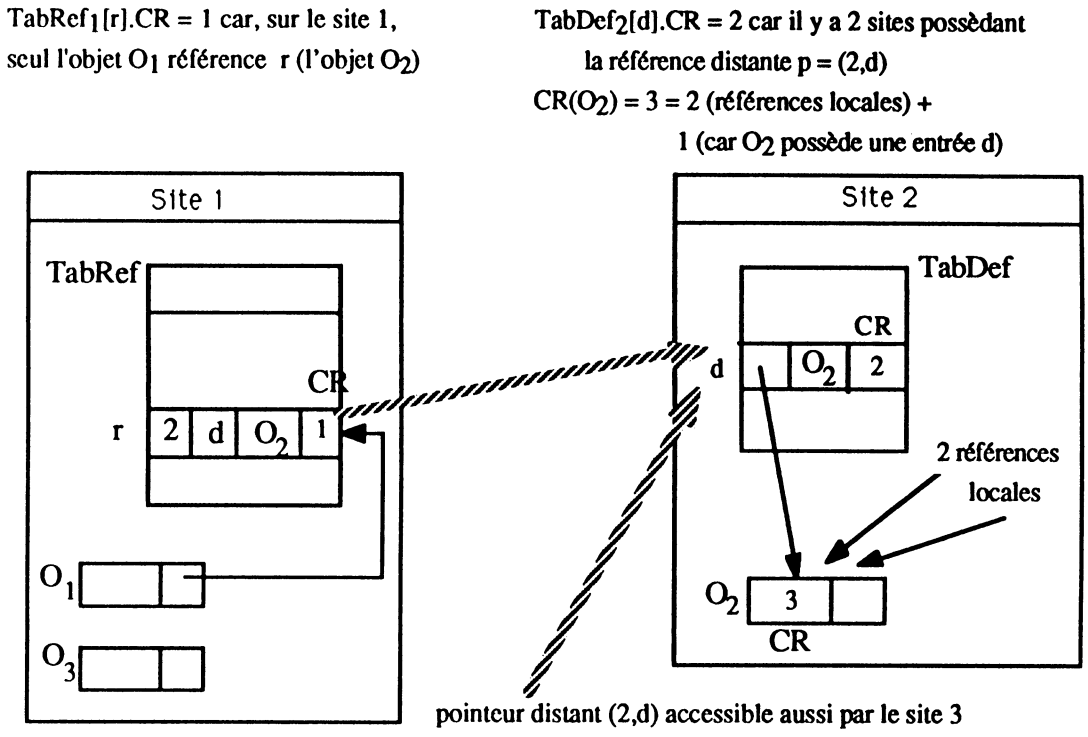


Schéma 5.13 : Exemple de la distribution du compteur

Cette mise en œuvre permet de traiter localement l'établissement de liaisons distantes vers des objets dont la référence réside déjà dans `TabRef`. Par exemple, avec le schéma 5.13, l'action `LierRef(O3,O2,2)` nécessite seulement d'augmenter le champ $TabRef_1[r].CR$ de 1. De la même manière, les suppressions de références sont traitées localement tant que $TabRef_1[r].CR$ reste positif. Lorsque $TabRef_1[r].CR$ devient nulle, il appartient au site 1 de déterminer si la référence est devenue inutile ou non ; dans l'affirmative, un message est transmis vers le site 2 (possesseur de l'objet O_2) afin qu'il décrémente $TabDef_2[d].CR$ de 1 ; si ce compteur passe à 0, l'entrée d est libérée et disponible, $CR(O_2)$ est décrémenté de 1 et O_2 devient privé puis libre lorsque $CR(O_2)$ atteint la valeur 0.

Bien que les algorithmes utilisant la distribution des compteurs évitent les deux inconvénients a) et b) dans 5.2.2.1, le troisième (c) est toujours là.

5.2.2.3 Les références pondérées

Le mécanisme des références pondérées proposé par [Beckerle 86][Bevan 87] permet d'éviter les problèmes des deux algorithmes précédents. L'idée principale est d'associer à chaque objet accessible globalement un crédit maximum donné et à chaque référence distante un crédit alloué. Lorsqu'une référence distante à un objet accessible globalement est transmise à un autre site, on lui alloue un crédit et on le transmet avec la référence afin d'assurer l'équation suivante :

crédit total d'un objet accessible globalement = crédit restant + Σ crédits alloués pour ses références distantes

a. Transfert de la référence distante par le site propriétaire

Le crédit maximum donné à chaque objet accessible globalement est mis dans un champ de son entrée dans la table ***TabDef***. Chaque fois qu'on alloue un crédit pour une référence distante, le crédit restant doit être diminué ; on utilise donc un champ pour sauvegarder ce crédit restant. En conséquence, chaque entrée de la table ***TabDef*** d'un objet accessible globalement possède 2 champs de crédits :

- l'un est le ***crédit total CT*** associé à l'objet,
- et l'autre le ***crédit restant CA***.

Lorsqu'un site i reçoit un message ***DemandeRef(Oj, Oi)*** pour un objet O_i qui lui appartient et qui ne figure pas dans sa table $TabDef_i$:

- O_i est implanté dans $TabDef_i$[d] (on obtient une référence distante p = (j,d)) avec un crédit total CT que l'on supposera égal à une puissance de 2 (justifié dans [Beckerle 86][Bevan 87]),
- on initialise aussi le champ $TabDef_i$[d].CA avec la valeur $TabDef_i$[d].CT ; celui-ci représente la disponibilité du propriétaire de l'objet correspondant, c'est-à-dire le nombre de références distantes à cet objet potentiellement attribuables à d'autres sites,
- enfin, on transmet la référence distante p, accompagnée d'un crédit alloué de valeur $TabDef_i$[d].CT div 2. A ce moment, la valeur du champ $TabDef_i$[d].CA doit être modifiée et ne garde que la valeur $TabDef_i$[d].CT div 2.

Lors de la réception de ***RetourRef(Oj, Oi, p, ca)*** , le site demandeur j crée, dans sa table **TabDef**, une entrée r, qui contient, entre autres, deux champs :

- l'un est le crédit distribué CA associé à la référence reçue. Ce champ enregistre le crédit ca reçu ;
- l'autre est le compteur CR qui est utilisé comme dans l'algorithme de distribution de compteurs pour conserver le nombre des références locales.

b. Transfert de la référence distante par le site non propriétaire

Lors de la réception de **PasserRef(O_k, O_i)** d'un autre site k qui demande la référence distante de l'objet O, le site j envoie **RetourRef(O_k, O_i, p, ca')** au site k où **ca'** est la moitié de la valeur **r.CA** enregistrée à ce moment là, la moitié restante est gardée dans **r.CA** comme la capacité restante.

c. Suppression de la référence distante

Lors de la suppression de l'entrée r de la table **TabDef**, on envoie au site propriétaire de l'objet un message pour cet événement. Ce message contient le crédit **r.CA** restant du site j. Le site i, après avoir reçu ce message, met à jour le crédit restant en additionnant le crédit reçu

au champ `d.CA` de l'entrée d. Si la somme devient égale au crédit total `d.CT`, l'objet repéré par cette entrée n'est plus référencé à distance et redevient un objet privé. Dans ce cas, on libère cette entrée de la table `TabDef` et on décrémente de 1 le compteur de l'objet O_i. Si ce compteur devient nul, l'objet est alors ramassé.

d. Conclusion

L'approche par références pondérées est a priori séduisante pour :

- sa facilité de mise en œuvre (à peine plus complexe que la solution de distribution de compteurs) ;
- son adéquation à toute topologie de réseau ;
- l'absence de synchronisation.

Par contre, le problème de la détection des circuits inaccessibles, problème commun aux algorithmes fondés sur les compteurs de références, n'est toujours pas résolu. Seules les techniques du type parcours-marquage peuvent résoudre ce problème.

5.2.3 RAMASSE-MIETTES DE TYPE MARQUAGE GLOBAL

L'objectif de ces algorithmes est de construire une arborescence globale $\bigcup_i G_A^i$ (cf. 5.2.1) recouvrant tous les objets accessibles. Une solution évidente est d'utiliser les collecteurs locaux pour faire le travail sur leur site et d'utiliser un algorithme de détection de terminaison globale pour savoir quand la tâche globale est terminée. La plupart des algorithmes proposés actuellement construisent l'arborescence répartie de contrôle et utilisent l'algorithme de calcul diffusant de Dijkstra [Dijkstra 80] pour la détection de la terminaison globale. Nous résumons ici l'algorithme proposé par [Couvert 89].

Dans cet algorithme, un processus particulier, appelé synchroniseur, joue le rôle de racine virtuelle pour l'arborescence de contrôle et aussi pour le graphe global $\bigcup_i G_A^i$ des objets. Ainsi pour l'exemple donné à la figure 5.9, une arborescence globale des objets accessibles peut être :

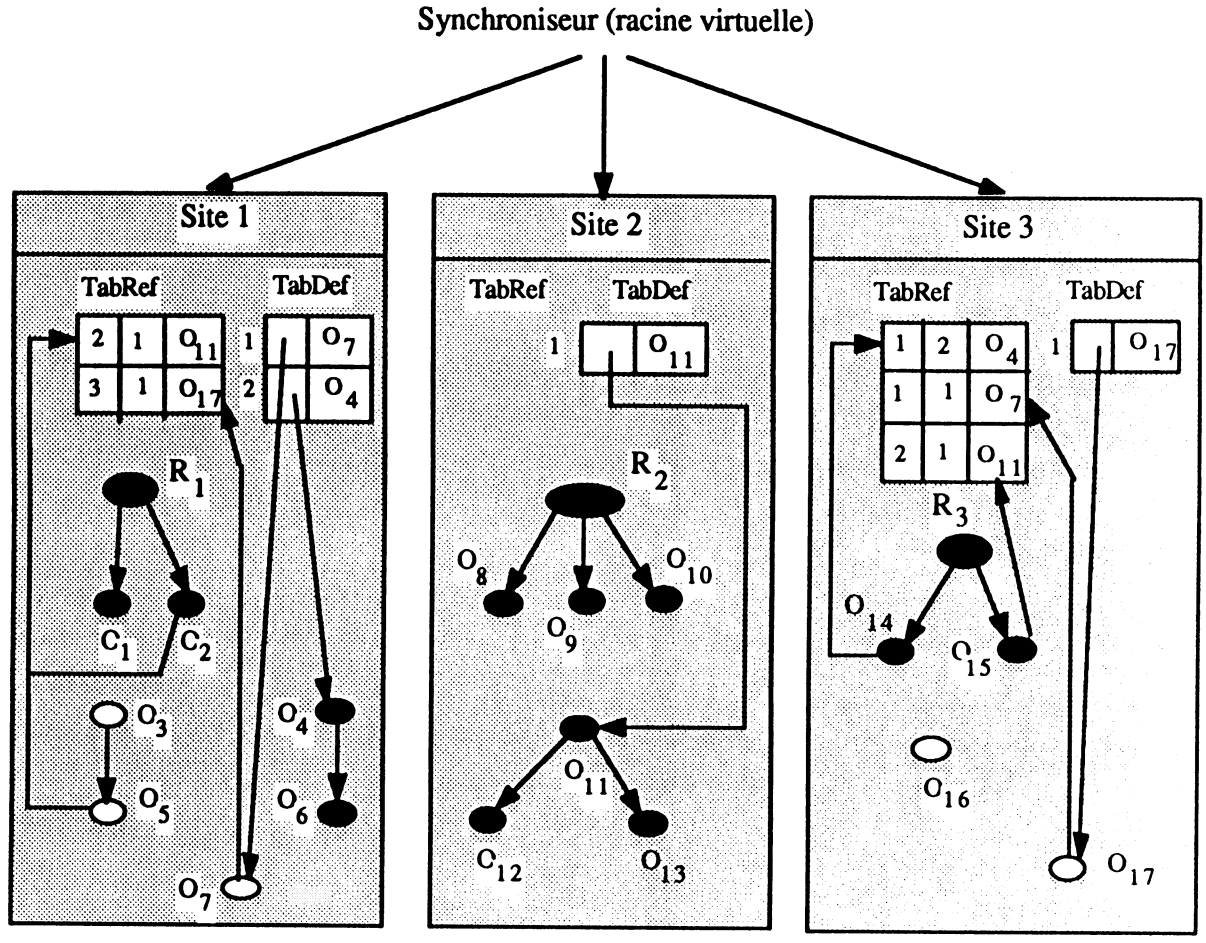


Figure 5.14

Lors d'une demande de ramasse-miettes, la tâche du synchroniseur, pour des collecteurs supposés de type parcours-marquage est de :

1. lancer les phases de parcours-marquage de tous les collecteurs C_i (pour tout site i) ; en effet, il apparaît clairement sur l'exemple 5.14 que le lancement des seuls collecteurs C_1 et C_2, provoque la récupération incorrecte, par le site 1, de l'objet O_7 référencé par le site 3,
2. détecter la terminaison de toutes les phases de parcours-marquage des collecteurs C_i (l'arborescence globale est alors construite),
3. lancer les phases de balayage de tous les collecteurs C_i,
4. détecter la terminaison des phases de balayage.

Dans sa phase de parcours-marquage, chaque collecteur local visite et parcourt les objets locaux accessibles (cf.5.1.2.2) et, pour les objets distants accessibles, envoie un message de demande de marquage au site propriétaire. Afin d'appréhender, de manière simple, le travail de marquage des collecteurs, nous supposons dans un premier temps :

- l'arrêt du mutateur lors de l'activation du collecteur placé sur le même site (ainsi chaque graphe local est statique lors du marquage et le graphe distribué G est statique lorsque tous les collecteurs sont en cours de marquage),
- l'absence de message en transit entre mutateurs lors du lancement des collecteurs.

Ensuite, nous prendrons en considération les références en transit et la coopération entre les mutateurs et les collecteurs.

5.2.3.1 Construction de l'arborescence globale

Le synchroniseur travaille sur un site particulier (ici nous supposons le site 0) et l'algorithme du collecteur est exécuté pour chacun des sites. Pour contrôler la construction de l'arborescence globale, le synchroniseur et les collecteurs doivent participer à la construction du graphe de contrôle dont le synchroniseur est la racine virtuelle. Chaque collecteur est dans l'un des deux états :

- `actif` : lorsque le collecteur est en cours de marquage ou en attente d'acquittement des messages (de demande de marquage) émis vers d'autres collecteurs,
- `passif` : lorsque le collecteur a terminé son marquage et a reçu les acquittements de tous les messages émis vers d'autres collecteurs.

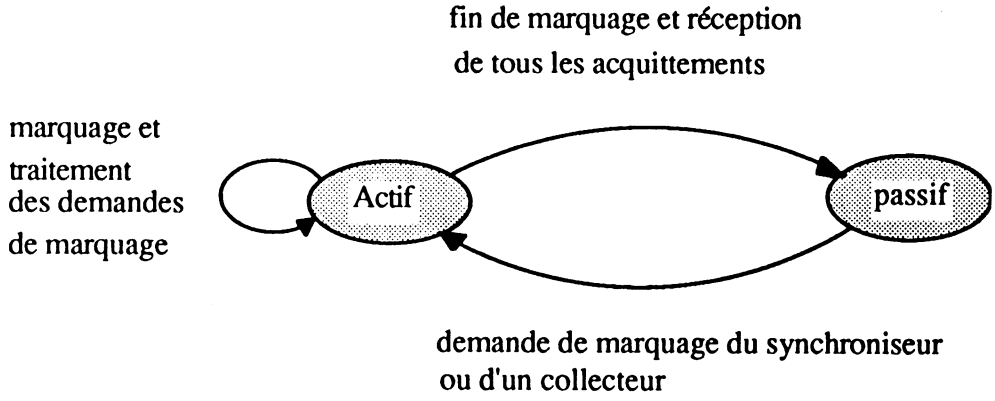


Figure 5.15 : Evolution de l'état d'un collecteur local

Pour assurer la terminaison de toutes les phases de marquage, on construit une arborescence de contrôle distribuée contenant à chaque instant les collecteurs actifs. Dans cette arborescence :

- le synchroniseur est représenté par la racine ;
- chaque collecteur actif correspond à un nœud dont le père est le processus l'ayant activé (par message de demande de marquage).

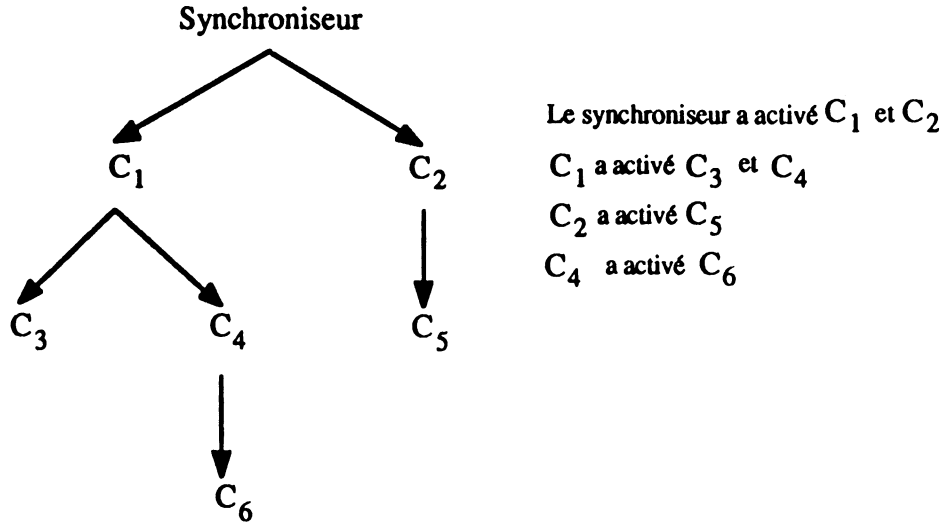


Figure 5.16 : Exemple d'une arborescence de contrôle

Lorsqu'il devient passif, un collecteur, représenté par une feuille de l'arborescence, envoie un message d'acquittement à son père puis disparaît de l'arborescence. La technique utilisée assure que chaque objet est marqué au plus une fois ; comme il y a un nombre fini d'objets, toutes les phases de marquage se terminent et l'arborescence est alors réduite à la racine.

a. Travail du synchroniseur

Le synchroniseur connaît le nombre total des sites du système et procède par échanges de messages avec les différents sites. Il dispose, en outre, de la variable `nbrep` donnant à tout moment le nombre de réponses attendues.

```
faire
       nbrep := 0 ;
       ∀i (i=1,...,n) : envoyer DebMarquage au site i
fait
Lors de la réception de AcqMarquage de site i
faire
       nbrep := nbrep + 1 ;
       si nbrep = n alors
         nbrep := 0 ;
           ∀i (i=1,...,n) : envoyer DebBalayage au site i
       fsi
fait
Lors de la réception de AcqBalayage de site i
faire
       nbrep := nbrep + 1 ;
       si nbrep = n alors termine fsi
fait.
```

b. Travail des collecteurs

Tous les objets du site i, y compris les entrées de **TabDef$_i$** et **TabRef$_i$** possèdent un champ couleur (cf. 5.1.2.2).

Chaque collecteur C_i dispose des variables suivantes :

- **reçu$_i$** : indique si C_i a déjà reçu la première demande de marquage, soit du synchroniseur (**DebMarquage**), soit d'un collecteur (**Marquage**) (initialement **reçu$_i$** est à faux),
- **nbacq$_i$** : désigne le nombre de messages non acquittés de demandes de marquage émis vers d'autres collecteurs (initialement **nbacq$_i$** = 0) ,
- **père$_i$** : permet de repérer le processus ayant activé C_i dans l'arborescence de contrôle ;
- **p$_i$** : est la pile contenant les références à des objets locaux à marquer.

Le travail de chaque collecteur local est exprimé par l'algorithme ci-dessous :

```
{ initialement le collecteur est passif
  et ses objets locaux sont tous blancs }
```

- Lors de la réception de **DebMarquage** du synchroniseur {du site 0}

```
faire
      si reçu_i = FAUX alors {état_i = passif}
        reçu_i := VRAI ;
        empiler (p_i, R^i_A) ; R^i_A.couleur := NOIR ;
        père_i := site 0 ;
        { état_i = actif}
        parcours-marquage-local;
        attente (nbacq_i = 0) ;
        envoyer AcqMarquage à père_i {état = passif}
      sinon envoyer AcqMarquage à site 0
      fsi
fait
```

- Lors de la réception de **Marquage** [entrée] de site j

```
faire
      si TabDef_i[entrée].couleur = NOIR alors
        envoyer AcqMarquage à site j
      sinon
        empiler (p_i, TabDef_i[entrée]) ;
        TabDef_i[entrée].couleur := NOIR ;
        si nbacq_i = 0 alors {état_i = passif}
              si reçu_i = FAUX alors
                    reçu_i := VRAI ;
                    empiler (p_i, R^i_A) ; R^i_A.couleur := NOIR ;
              fsi
              père_i := site j ;
              { état_i = actif}
```

```
            parcours-marquage-local;{cet appel peut modifier nbacq_i}
            attente (nbacq_i = 0) ;
            envoyer AcqMarquage à père_i {état = passif}
        sinon {état_i = actif}
            parcours-marquage-local ;
            attente (nbacq_i = 0) ;
            envoyer AcqMarquage à père_i {état = passif}
        fsi
        envoyer AcqMarquage à père_i
      fsi
fait
```

- Lors de la réception de **AcqMarquage** de site j

```
faire {état_i = actif}
      nbacq_i := nbacq_i - 1 ;
fait
```

- Lors de la réception de **DemandeBalayage** de site 0 (du synchroniseur)

```
faire
      balayage ;
      envoyer AcqBalayage à site 0
fait
```

- Procédure **parcours-marquage-local** ; {sur le site i}

```
début
      tantque non pile_vide(p_i)
      faire
        dépiler(p_i,O) ;
        pourtout O_k successeur de O faire
            si O_k.couleur = blanc alors
                cas   O_k ∉ TabRef_i => {objet local}
                            empiler(p_i,O_k) ;
                            O_k.couleur := noir
                      O_k ∈ TabRef_i => {objet distant}
                            envoyer Marquage(O_k.entrée)
                                    à site O_k.site ;
                            nbacq_i := nbacq_i + 1 ;
                fcas ;
            fsi
        fait
      fait
fin
```

La synchronisation des phases de parcours-marquage et de balayage, effectuée dans l'algorithme par le synchroniseur, peut être réalisée par un collecteur quelconque.

5.2.3.2 Références en transit lors de l'activation des collecteurs

On conserve l'hypothèse selon laquelle chaque mutateur est suspendu lors de l'activation du collecteur placé sur le même site.

Le partage d'objets entre sites et les délais de communication nous amènent à considérer les traitements associés aux références en transit lors de l'activation des collecteurs.

Considérons la situation suivante où M désigne le mutateur et C le collecteur :

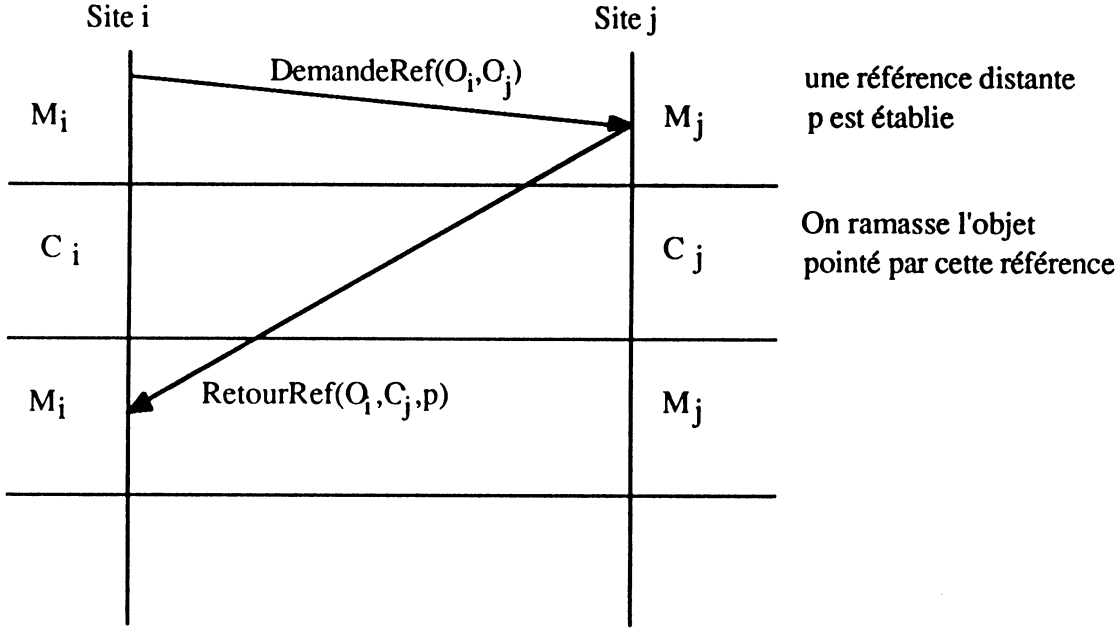


Figure 5.17 : Problème des références en transit lors de l'activation des collecteurs

Lors de l'exécution du ramasse-miettes, l'objet désigné par O_j, et attribué par M_j, n'est pas connu du site i ; il n'y a donc pas de message de demande de marquage de C_i à C_j pour cet objet, et C_j peut récupérer 'à tort' l'emplacement occupé par l'objet O_j et éventuellement sa descendance.

Le problème se pose également lorsque la réception du message **`RetourRef`** est consécutive à une demande **`PasserRef`** à un site non propriétaire.

La propriété à maintenir est de considérer comme accessible tout objet O_j figurant dans un message **`RetourRef`**. Pour éviter la récupération de tels objets (attribués et non référencés), plusieurs solutions sont envisagées selon que le mutateur participe ou non au marquage des objets.

a. Les mutateurs ne participent pas au marquage des objets

Pour vérifier la propriété précédente, nous présentons une solution qui nécessite une hypothèse supplémentaire pour le réseau de communication :

- la connaissance du délai maximum (Δ_{max}) de transmission d'un message entre tout couple de sites, (le temps de traitement du message est considéré comme non

significatif ou compris dans Δ_{max}). Ce délai maximum Δ_{max} est connu par tous les sites.

Lorsqu'un message de demande de référence (**DemandeRef** ou **PasserRef**) est émis par M_i à l'instant t, il est traité par M_j au plus tard à l'instant $t + \Delta_{max}$ et le site i reçoit l'acquittement (**RetourRef**) au plus tard à l'instant $t + 2\Delta_{max}$.

Pour recevoir les messages **RetourRef** on introduit une attente de $2\Delta_{max}$ avant le début du parcours-marquage. Ainsi, lors de la réception de la première demande de marquage, provenant du synchroniseur (**DebMarquage**) ou d'un collecteur (**Marquage**), le collecteur C_i attend un temps $2\Delta_{max}$ pendant lequel il prend en compte les messages RetourRef :

```
si reçui = FAUX alors
pendant 2Δmax faire
       Lors de la réception de RetourRef(Oi,Oj,p) de site j
       faire
         envoyer Marquage(p.entrée) à p.site ;
         {p.site = site j pour l'acquittement de DemandeRef}
         {p.site ≠ site j pour l'acquittement de PasserRef}
         nbacqi := nbacqi + 1 ;
         {le message RetourRef sera traité par le mutateur
               lors de sa réactivation}
       fait ;
       reçui := vrai ;
fait;
{algorithme donné ci-dessus}
fsi
```

On peut diminuer le temps d'exécution de la phase de parcours-marquage d'un collecteur en prenant en compte les messages RetourRef avant la fin du ramassage. Pour cela, chaque phase de marquage doit durer au moins $2\Delta_{max}$ (dans le cas contraire une attente est nécessaire) et accepter les messages **RetourRef(Oi,Oj,p)**.

b. Les mutateurs participent au marquage des objets

Lorsque le mutateur M_j envoie un message **RetourRef(Oi,Oj,p)** à autre site (i), il doit positionner un champ particulier ('**gelé**') de l'entrée associée à la référence **p** dans la table **TabDef** pour que le collecteur C_j puisse considérer l'objet référencé par **p** comme un objet accessible jusqu'à sa première utilisation par M_i. A ce moment là, le mutateur M_i envoie un message au mutateur M_j pour lui demander d'annuler le champ '**gelé**'.

5.2.3.3 Les mutateurs s'exécutent pendant le travail des collecteurs

Sur chaque site i, le mutateur M_i et le collecteur C_i s'exécutent simultanément et dans ce cas, chaque graphe local est dynamique. Comme on l'a présenté en 5.1.2.2, les mutateurs doivent participer au marquage des objets lors de l'ajout d'arcs, c'est-à-dire lors de l'action d'affectation de références :

```
ajout d'arc (O1,O3) =>
si O3 est blanc alors
       < le marquer noir et le mettre en pile ;
         ajout (O1,O3) proprement dit. >
sinon
         ajout (O1,O3) proprement dit.
fsi
```

Dans le cas où O_3 est un objet à distance, on le marque à noir et on met en pile l'entrée correspondante dans la table **TabRef** pour que le collecteur puisse envoyer à un autre site la demande **Marquage(entrée)**.

5.2.4 RECUPERATION LOCALE DES MIETTES

Les algorithmes du type parcours-marquage global présentés dans le paragraphe 5.2.3 ne distinguent pas les objets privés des objets accessibles globalement. Il nécessite que tous les sites soient présents et participent au ramassage. De plus, le temps nécessaire pour accomplir un cycle de ramassage est très important. En conséquence, il est nécessaire d'avoir un algorithme local qui ramasse les miettes résidant sur le site sans avoir besoin de communiquer avec d'autres sites.

Grâce aux tables **TabDef**, on peut réaliser un algorithme local pour chaque site en considérant que les objets référencés par les entrées de ces tables sont des racines. Ceci permet de récupérer immédiatement les objets privés inaccessibles. Comme les seuls objets référencés à distance sont ceux qui ont une entrée dans **TabDef** de son site, les objets qui ne sont pas marqués (blanc après la phase de parcours-marquage locale) sont vraiment les miettes et on peut les récupérer immédiatement sans avoir besoin des informations supplémentaires d'autres sites.

5.3 CONCLUSION

Avec la tendance très forte à la baisse du prix des mémoires, il nous semble qu'il n'y a plus de raison de laisser aux mutateurs la charge du ramasse-miettes ; ainsi les algorithmes du type compteurs de références ne sont plus intéressants dans la plupart des systèmes actuels. Bien que l'algorithme du type parcours-marquage global soit le seul qui puisse ramasser toutes les miettes dans un système distribué, le temps nécessaire pour faire un cycle devient très important. On a toujours un algorithme supplémentaire dans chaque site pour ramasser les miettes locales.

D'autre part, dans les algorithmes du type parcours-marquage où les collecteurs fonctionnent en parallèle avec des mutateurs, il faut que ces derniers participent au marquage lorsqu'ils affectent une référence. De plus, l'action de marquage et d'affectation de référence doivent être faites de façon indivisible. Est-ce qu'il existe une solution à ce problème ? Nous y répondrons au chapitre 6.

Quel que soit le type d'algorithme de ramasse-miettes utilisé, le problème de traitement de la pile d'exécution des mutateurs est un problème très difficile à résoudre et il n'existe pas

actuellement de solution intéressante. Est-ce qu'il existe une solution pour supprimer le parcours des piles d'exécution ? Nous répondrons aussi à cette question au chapitre 6.

CHAPITRE 6 : ALGORITHMES DE RAMASSE-MIETTES POUR GUIDE

Une étude synthétique de toutes les familles d'algorithmes de ramasse-miettes a été présentée au chapitre 5 et le modèle de mémoire d'objets de Guide au chapitre 3. Dans ce chapitre, nous présentons les algorithmes de ramasse-miettes proposés et réalisés pour Guide. Bien que les algorithmes de type compactage soient mieux adaptés aux systèmes dont la mémoire d'objets est la mémoire centrale et dans laquelle les miettes sont beaucoup plus nombreuses que les objets accessibles (cf. 5.1.2), ces algorithmes sont exclus de nos premières réalisations pour les raisons suivantes :

- Le ramasse-miettes de type compactage s'exécute en cycle et pendant son exécution, il a généralement besoin de l'interruption totale des activités des utilisateurs. Cette interruption est trop longue dans Guide où le ramasse-miettes doit travailler sur les objets stockés dans la MPO (la mémoire secondaire, cf. 3.6). Par conséquent, cette interruption n'est pas acceptable pour un système interactif comme le système Guide.
- Dans la MPO, les objets Guide sont stockés dans des blocs de mémoire (disque) non contigus. Dans ce contexte, le compactage de l'espace disque n'est plus indispensable. Le seul intérêt que l'on peut obtenir est d'augmenter l'efficacité d'accès aux objets (l'accès aux blocs disque contigus ou voisins est plus rapide que l'accès aux blocs non-contigus).

Il ne nous reste que deux familles d'algorithmes de ramasse-miettes : compteurs de références et parcours-marquage. Dans ce chapitre, nous présentons d'abord les algorithmes de type compteurs de références pour Guide mono-site ainsi que pour Guide réparti. En ce qui concerne les algorithmes de type parcours-marquage, nous présentons quatre ramasse-miettes dans un ordre de complexité croissante : le ramasse-miettes pour le système Guide mono-site avec activités interrompues, le ramasse-miettes pour le système Guide mono-site fonctionnant en parallèle avec les activités, le ramasse-miettes global pour le système Guide réparti et le ramasse-miettes local pour le système Guide réparti.

6.1 COMPTEURS DE REFERENCES POUR GUIDE MONO-SITE

Dans le cas du système Guide mono-site, on peut appliquer directement l'algorithme présenté au paragraphe 5.1.2 : on réserve un champ de contrôle (CR) dans chaque objet et on y enregistre le nombre total de références à l'objet qui existent dans ses prédécesseurs ; la mise à jour des compteurs de références est faite par les activités utilisateurs lorsqu'elles exécutent les actions de création d'objets et d'affectation de références :

- lorsqu'un nouvel objet est créé, on met 0 à son champ CR à l'exception des objets racines dans lesquels on met la valeur 1 au champ CR ;

- lorsqu'une affectation de référence a lieu, on doit décrémenter de 1 le compteur de l'objet repéré par l'ancienne référence puis on incrémente de 1 le compteur de l'objet repéré par la nouvelle référence. Comme, dans Guide, plusieurs activités peuvent partager les mêmes objets, les opérations de décrémentation et d'incrémentation de la valeur du champ CR doivent être faites de façon indivisible.
- après une action de décrémentation de compteur, si le compteur de l'objet devient nul, on ramasse cet objet et on décrémente de 1 les compteurs de ses successeurs qui peuvent à leur tour devenir des miettes.

Le code de décrémentation et d'incrémentation de compteur associé aux affectations de références est généré automatiquement par le compilateur Guide.

Bien que cet algorithme de compteur de références soit un peu coûteux en temps, il est le seul qui puisse ramasser de façon immédiate des miettes ; il est donc utilisable dans Guide pour ramasser immédiatement un nombre très important d'objets qui sont créés et utilisés seulement dans une application et qui deviennent miettes dès que le programme d'application se termine.

PROBLEME DE PARCOURS DES PILES D'EXECUTION DES ACTIVITES

Comme nous l'avons présenté au chapitre 1, les activités d'application sont des objets Guide parmi d'autres objets dans le système. Une partie de l'état de ces objets qui doit être prise en compte dans la tâche de ramasse-miettes est leur pile d'exécution dans laquelle peuvent être contenues des références à des objets. La pile d'une activité est modifiée dans trois cas :

- Affectation d'une référence à une variable de travail propre à la méthode qui est en cours d'exécution. Les paramètres sont considérés comme des variables de travail de la méthode.
- Extension de la pile de l'activité lorsque l'on appelle une méthode sur un objet. La zone d'extension va contenir des paramètres passés ainsi que des variables de travail de la méthode appelée.
- Libération de la partie de la pile réservée pour la méthode lorsque cette dernière termine son programme et fait un retour à l'appelant. Toutes les références existant dans cette partie de la pile sont supprimées.

La manipulation de compteurs de références associée à l'affectation de référence a été présentée, il nous reste le problème de manipulation de compteurs associée à l'extension et à la libération de la pile (lorsqu'on appelle une méthode et lorsqu'une méthode se termine).

Nous reprenons les points essentiels concernant l'appel de méthode sur un objet (cf. 4.3) :

- avant d'appeler la primitive `GuideCall` du noyau, le code généré par le compilateur prépare un bloc de communication (une structure de type `T_CallBlock`) qui contient toutes les informations nécessaires. Il copie dans ce bloc les valeurs des paramètres IN et INOUT ;
- le noyau remplit ce bloc avant de le passer à l'appelé ;

- l'appelé copie les valeurs des paramètres IN et INOUT dans son environnement d'exécution avant d'exécuter son propre code ;
- avant de se terminer, l'appelé copie les valeurs des paramètres OUT et INOUT dans le bloc de communication T_CallBlock et le passe au noyau ;
- le noyau (service du GuideCall) passe le bloc de communication T_CallBlock à l'appelant ;
- l'appelant copie les valeurs des paramètres OUT et INOUT dans son environnement d'exécution.

Le principe de base de notre solution (au problème de manipulation de compteurs associée à l'extension et à la libération de la pile) est de confier la tâche de manipulation des compteurs de références aux applications, par du code généré par le compilateur. En effet, le noyau ne manipule des références que de manière temporaire et n'a pas à participer à la gestion des compteurs. De plus, si l'on ne s'intéresse qu'aux actions de l'appelant et de l'appelé, et qu'on ignore les actions concernant le bloc de communication, on constate deux choses :

- les valeurs des paramètres IN et INOUT sont recopiées du contexte de l'appelant au contexte de l'appelé (sa pile d'exécution) et s'il s'agit de références, on doit incrémenter le compteur de l'objet référencé ;
- les paramètres OUT et INOUT sont recopiés du contexte de l'appelé au contexte de l'appelant et s'il s'agit de références, on doit incrémenter le compteur de l'objet référencé ;

Pour réaliser notre solution, on définit sur chaque objet (de n'importe quel type) les méthodes suivantes :

- **Destroy** qui demande au système de ramasser l'objet ;
- **Incrémente_compteur** qui incrémente de 1 le compteur de l'objet ;
- **Décrémente_compteur_nonrécursive** qui décrémente de 1 le compteur de l'objet sans détecter si l'objet devient miette ou non ;
- **Décrémente_compteur_récursive** qui décrémente de 1 le compteur de l'objet et ramasse l'objet si le compteur passe à zéro (par appel de la méthode **Destroy**) puis appelle **Décrémente_compteur_récursive** sur les successeurs de l'objet ramassé.

Chacune des méthodes ci-dessus s'exécute en exclusion mutuelle (en utilisant la clause **EXCLUSIVE**, cf. 2.1.2).

Travail de l'appelé avant son exécution proprement dite

Comme nous l'avons dit précédemment, c'est l'appelé qui copie des paramètres **IN** et **INOUT** dans sa pile d'exécution. Avec cette remarque, pour maintenir la validité des compteurs des objets référencés par ces paramètres, le code de l'appelé doit incrémenter de 1 le compteur de l'objet référencé par chaque paramètre **IN** et **INOUT** qui contient une référence (appelle **Incrémente_compteur** sur l'objet).

Travail de l'appelé avant de retour

Lorsqu'une méthode se termine, sa pile d'exécution est libérée. Pour maintenir la validité des compteurs des objets référencés dans la partie de la pile libérée, le code de l'appelé doit décrémenter de 1 le compteur de l'objet référencé par chaque variable de travail, y compris les paramètres, qui contient une référence. Cependant, cette décrémentation de compteurs doit être faite avec une précaution :

a. Cas des paramètres IN et des variables de travail propres à l'appelé :

L'appelé appelle la méthode **Décrémente_compteur_récursive** pour décrémenter les compteurs des objets référencés par ces variables.

b. Cas des paramètres OUT et INOUT :

Après le retour de l'appelé, l'appelant doit copier les valeurs des paramètres INOUT et OUT du bloc de communication dans son contexte. S'il s'agit de références et si ces références n'existent que dans les variables OUT et INOUT de l'appelé, les objets référencés devraient être ramassés si l'appelé appelle **Décrémente_compteur_récursive** pour décrémenter les compteurs. En conséquence, pour que les objets ne soient pas ramassés accidentellement, l'appelé doit appeler **Décrémente_compteur_nonrécursive** pour décrémenter seulement les compteurs des objets référencés par ses paramètres OUT et INOUT.

c. Cas de la valeur de retour associée à la méthode :

Pour les méthodes qui rendent une référence à l'appelant, l'objet référencé peut être ramassé accidentellement si sa référence existe uniquement dans la partie de la pile associée à l'appelé. A titre d'exemple, dans une méthode, on crée un objet O et on met sa référence dans une variable de travail (son compteur est égal à 1) ; lorsque la méthode termine son programme, elle décrémente de 1 les compteurs de tous les objets référencés dans sa pile et l'objet O est ramassé avant que sa référence ne soit rendue à l'appelant. Afin d'éviter ce problème, on utilise la méthode **Décrémente_compteur_nonrécursive** pour décrémenter de 1 le compteur de l'objet dont la référence va être rendue à l'appelant.

Travail de l'appelant après le retour de l'appelé

Lorsque l'appelant reprend le contrôle, il copie les valeurs des paramètres OUT, INOUT, ainsi que la valeur de retour associée à la méthode à partir du bloc de communication dans son contexte. S'il s'agit de références, la copie doit être considérée comme une affectation de référence, c'est-à-dire que la tâche de manipulation de compteurs doit être faite pour chaque copie de référence.

Par exemple, avec la méthode m1 déclarée ci-dessous :

```
METHOD m1 ;
a1, a2, a3 : REF Top ;
référence : REF toto ;
résultat : REF Top ;
BEGIN
    ...
```

```
      a1 := Top.New ;
      a2 := Top.New ;
      référence := toto.New ;
      ...
      résultat := référence.m2 (a1, a2, a3) ;
      ....
END m1 ;
```

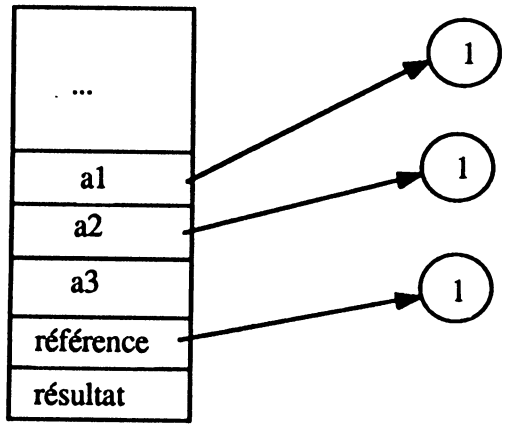
et la méthode m2 déclarée dans la classe `toto` (qui implémente le type `toto`) :

```
METHOD m2 (IN var1 : REF Top ; INOUT var2 : REF Top ;
         OUT var3 : REF Top) : REF Top ;
tmp : REF Top ;
BEGIN
      ...
      tmp := Top.New ;
      var3 := Top.New ;
      ...
      return tmp ;
END m2 ;
```

Les figures suivantes montrent l'évolution de l'état des piles d'exécution et de la valeur des compteurs des objets.

a. Lors de l'appel de la méthode `m2` sur `référence` :

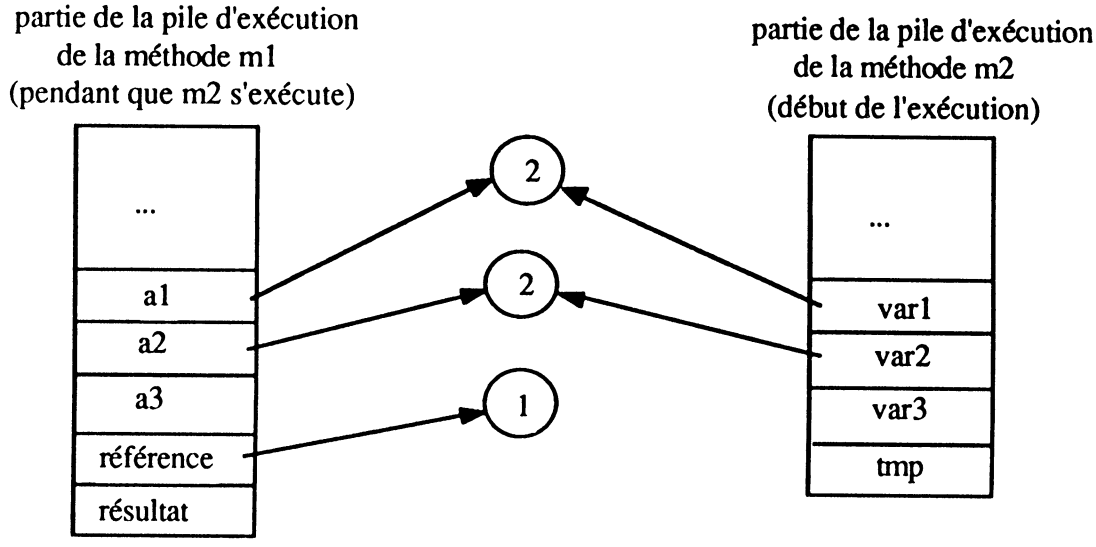
partie de la pile d'exécution
de la méthode m1
(avant d'appel de m2)



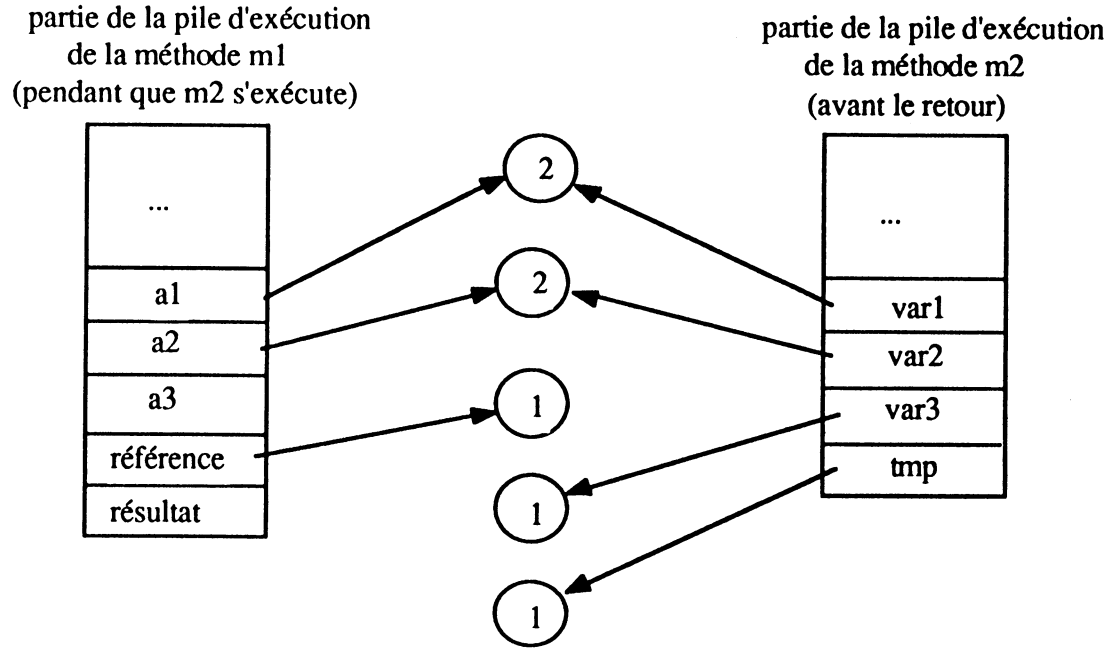
(x) : objet

x : compteur de références à l'objet

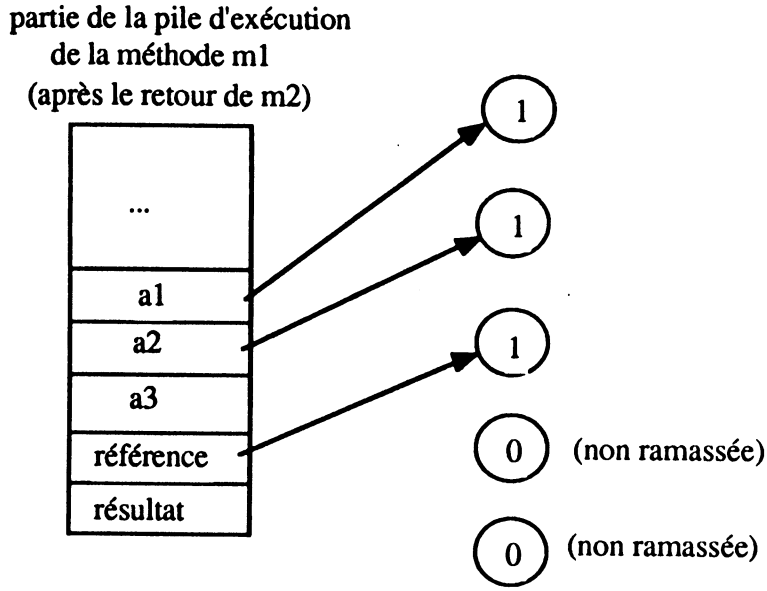
b. Après la copie des paramètres faite par l'appelé :



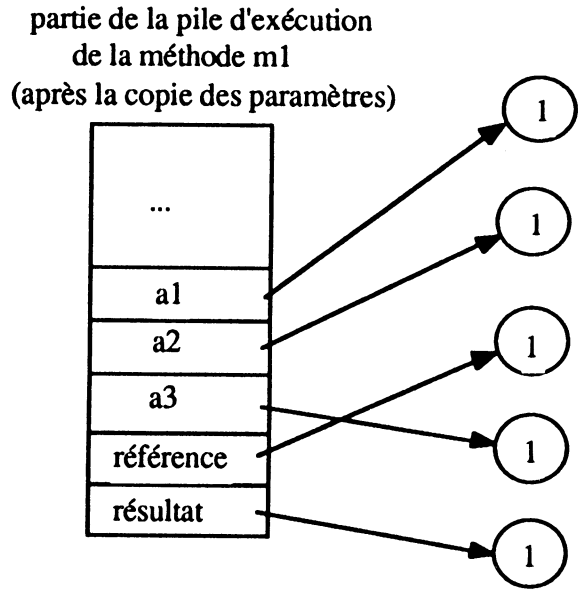
c. Avant le retour de l'appelé :



d. Après le retour de l'appelé :



e. Après la copie des paramètres faite par l'appelant :



Remarque : La manipulation des compteurs faite par l'appelé avant sa propre exécution peut également être faite par l'appelant avant d'appel. Cela évite l'incohérence des compteurs produite lorsque l'activité appelée ne s'exécute pas encore mais que son activité mère est déjà détruite

En résumé, dans notre solution, nous n'avons pas besoin de modifier le code du système ; tout travail concernant les compteurs de références est fait par le code généré par le compilateur. D'autre part, on évite la perte de temps de parcours de la pile (une partie pour chaque appel et chaque retour d'une méthode).

6.2 COMPTEURS DE REFERENCES POUR GUIDE REPARTI

Il n'y a pas de différences au niveau des applications entre l'accès aux objets distants et l'accès aux objets locaux puisque l'on utilise dans Guide des références uniques et valides pour le système entier. En conséquence, l'algorithme du type compteurs de références présenté en 6.1 fonctionne aussi bien pour Guide mono-site que pour Guide réparti. Les problèmes d'incohérence présentés en 5.2.2.1 ne se posent plus grâce à la transparence de la localisation, mais la performance de l'exécution des applications se dégrade à cause du travail de décrémentation et d'incrémentation des compteurs d'objets à distance. Pour chaque affectation d'une référence à un objet distant, on doit envoyer éventuellement deux messages à d'autres sites : l'un pour décrémenter le compteur de l'ancien objet (éventuellement à distance) et l'autre pour incrémenter celui du nouvel objet. Nous cherchons donc des solutions pour remédier au problème d'efficacité. Une solution inspirée de la technique présentée en 5.2.2.3 est de contrôler toutes les émissions et réceptions de références et d'utiliser les références pondérées pour toutes les références envoyées (cf. 5.2.2.3).

A chaque site qui possède au moins un système d'objets local, on associe les deux tables suivantes :

- `TabDef` contient une entrée pour chaque objet du site qui est éventuellement référencé par des sites distants. Chaque entrée de cette table possède 3 champs :
  - + le champ de référence qui contient la référence proprement dite,
  - + le champ de crédit total `CT` qui est alloué pour la référence,
  - + et le champ de crédit restant `CA`.

  Quand un objet devient accessible globalement, c'est-à-dire quand on envoie sa référence à un autre site pour la première fois, on réserve pour cet objet une entrée dans la table `TabDef`, on copie la référence dans le champ référence de l'entrée, on initialise les deux champs `CT` et `CA` avec une valeur de crédit maximale, puis en envoyant la référence, on envoie aussi la valeur `CA DIV 2` (`DIV` est la division entière). Après chaque envoi de référence, le champ `CA` est diminué de la valeur d'envoi `CA DIV 2` et il ne garde que `CA - CA DIV 2` pour les envois ultérieurs.
- `TabRef` contient une entrée pour chaque objet distant référencé par le site. Chaque entrée de cette table possède 3 champs :
  - + le champ de référence qui contient la référence proprement dite,
  - + le champ de crédit attribué `CA`,
  - + le champ de compteur `CR` qui compte le nombre des prédécesseurs locaux de l'objet distant.

  Quand une référence à un objet d'un autre site est reçue pour la première fois, on réserve pour cette référence une entrée dans la table `TabRef`, on copie la référence dans le champ référence de l'entrée, la valeur `CA` reçue dans le champ `CA` et on initialise le champ `CR` à zéro. En général cette réception est suivie par une

affectation à une variable référence qui incrémente le compteur CR à 1. Lorsque la référence à cet objet est envoyée à un autre site, on envoie aussi la valeur CA DIV 2 (en gardant la valeur restant CA - CA DIV 2 pour les envois ultérieurs). Lorsqu'une référence est reçue et que son entrée existe déjà dans la table TabRef, on additionne la valeur reçue CA au champ CA correspondant.

Lorsqu'un site envoie une référence et que son crédit restant n'est plus suffisant pour être divisé (égal à 1 dans notre cas), on doit réallouer le crédit selon deux cas différents :

- Cas 1 : le site d'envoi est le site propriétaire de l'objet. Dans ce cas, il réalloue un nouveau crédit et il l'additionne aux champs CA et CT de l'entrée correspondante ;
- Cas 2 : le site d'envoi n'est pas le site propriétaire de l'objet. Dans ce cas, il demande au site propriétaire de l'objet de lui envoyer une autre référence. Lors de la réception de cette nouvelle référence, il additionne la valeur CA reçue au champ CA de l'entrée correspondante.

Lorsqu'un site non-propriétaire d'un objet possède une entrée dans la table TabRef et que l'objet n'est plus référencé localement, ce dernier peut supprimer l'entrée en question et retourner la référence ainsi que la valeur CA restante au site propriétaire.

Lorsque le site propriétaire d'un objet reçoit une référence à cet objet retournée par un autre site, il additionne la valeur CA reçue au champ CA de l'entrée correspondante dans la table TabDef.

L'utilisation de deux tables TabDef et TabRef et les manipulations des références présentées ci-dessus assurent que l'équation suivante est vérifiée tout le temps :

crédit total d'un objet accessible globalement = crédit restant + Σ crédits alloués pour ses références distantes.

Sur le site propriétaire d'un objet, lorsque la valeur CA devient égale à la valeur CT, la référence en question n'existe plus sur aucun site distant (ni en transit) et on peut supprimer son entrée de la table TabDef. Cette suppression peut conduire au ramassage de l'objet si ce dernier n'est plus référencé localement.

Remarque : dans notre réalisation les valeurs des champs CA et CT sont tout simplement une valeur entière positive.

Conclusion

Dans Guide, les algorithmes du type compteurs de références sont moins intéressants que les algorithmes du type parcours-marquage pour les raisons suivantes :

- l'espace des objets Guide à ramasser est très grand et peut contenir des objets de différents programmes d'applications. Il n'y a pas raison de confier la tâche de ramasse-miettes à chaque opération d'accès des activités utilisateurs ;
- le partage des objets par plusieurs activités oblige à avoir des sections critiques pour les affectations de références et cela ralentit une fois encore la tâche principale des applications.

C'est pour les raisons ci-dessus que les algorithmes du type compteurs de références ne sont pas utilisés fréquemment dans les systèmes qui permettent le partage des objets.

6.3 PARCOURS-MARQUAGE MONO-SITE AVEC ACTIVITES INTERROMPUES

6.3.1 Modèle de mémoire d'objets Guide pour cet algorithme

- Le système Guide s'exécute sur un seul site dont l'espace d'objets persistants (la MPO) contient tous les objets du système, y compris les miettes. Cet espace est l'union des systèmes d'objets (containers) du site.
- Il y a un moyen d'identification de tous les containers ;
- Il y a un moyen d'identification et de localisation de tous les objets contenus dans chaque container ;
- Il y a un moyen d'identification et de localisation des références contenues dans un objet. Une méthode implémentée dans chaque classe d'objets permet de construire la liste des références contenues dans l'état d'un objet de cette classe.
- Aucune activité utilisateur ne fonctionne pendant l'exécution du ramasse-miettes (pour un cycle donné), donc l'état de la mémoire reste stable durant toute la durée du cycle. Chaque activité est un objet Guide qui contient, entre autres choses, sa pile d'exécution. La méthode qui construit la liste des références contenues dans l'état de l'objet `'activité'` prend en compte les références dans sa pile d'exécution. Lorsque l'activité est interrompue, cette liste de références est stable et on peut la construire exactement ;
- Les objets considérés comme permanents par le système (les racines) sont les objets des activités interrompues et l'objet `'Boot'` du site. Ce dernier contient, parmi d'autres objets, l'objet `'Serveur de noms symboliques'` qui, à son tour, contient directement ou indirectement toutes les références à des objets ayant au moins un nom symbolique (les objets exécutables des classes par exemple).

Le schéma ci-après résume ces hypothèses.

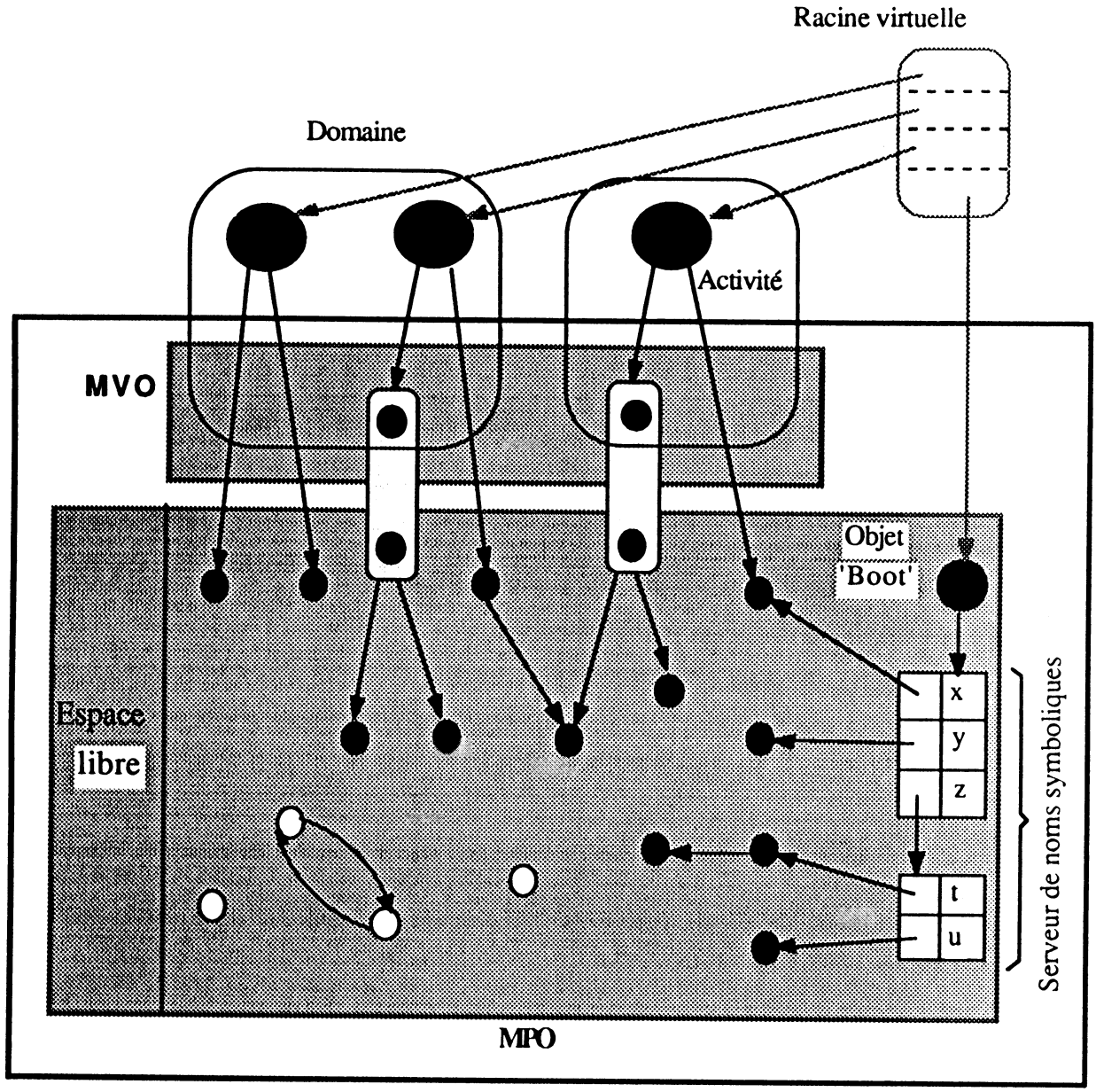


Figure 6.1 Modèle de la mémoire d'objets Guide mono-site

6.3.2 Principe de l'algorithme

Ce ramasse-miettes utilise la technique de parcours en profondeur en utilisant une pile de références (cf. 5.1.2). Mais le problème de l'espace réservé à la pile de références se pose dans Guide :

- La mémoire d'objets persistants de Guide est un très grand espace (mémoire disque) ; cela implique d'avoir une pile de références de très grande taille et on risque toujours d'avoir des débordements.

- Lorsqu'on appelle le ramasse-miettes, il est possible que l'espace mémoire soit très plein, et qu'il soit impossible d'augmenter dynamiquement l'espace alloué à la pile.

En conséquence, au lieu d'utiliser une pile et deux couleurs nous utilisons aussi une pile et trois couleurs [Dijkstra 78] (la troisième couleur est grise) :

- La pile de références est utilisée pour garder les références à des objets accessibles à parcourir. Dans le cas du système Guide, la taille de la pile doit être limitée à un nombre bien inférieur au nombre d'objets de l'espace. Le débordement de la pile de références doit donc être traité.
- La couleur grise des objets sert au traitement du débordement. Chaque fois qu'on visite un objet, on le marque à gris et on empile sa référence. Quand la pile atteint sa limite, on ne peut pas y mettre les références, et on positionne un drapeau indiquant cet état. Lorsque la pile devient vide, on vérifie ce drapeau pour savoir s'il y a des objets accessibles qui ne sont pas traités. La couleur grise permet au ramasse-miettes de retrouver ces objets dans l'espace entier. S'il n'y a pas de débordement de la pile, la couleur grise ne sert à rien.

Les significations des trois couleurs sont donc :

- les objets blancs sont les objets non visités ;
- les objets noirs sont des objets déjà visités et parcourus ;
- les objets gris sont des objets déjà visités mais pas encore parcourus. Ils sont tous dans la pile des références s'il n'y a pas de débordement de la pile. Dans le cas où la pile a débordé, certains objets gris peuvent ne pas être dans la pile et on doit les récupérer après. Pour la récupération des objets visités (gris), on parcourt l'espace d'objets entier et on les empile. La couleur grise nous permet de reconnaître les objets visités.

Le ramasse-miettes est exécuté par une seule activité Guide qui voit l'ensemble des différents systèmes d'objets.

Comme nous l'avons présenté, les algorithmes de type parcours-marquage fonctionnent en cycles. Pour faciliter la présentation des travaux effectués par le ramasse-miettes, nous séparons un cycle de l'algorithme en trois phases distinctes : la phase d'initialisation, la phase de parcours-marquage et la phase de ramassage.

- Dans la phase d'initialisation, en supposant que tous les objets sont remis à blanc dans la phase de ramassage du cycle précédent, on marque à gris les objets racines et on les met dans la pile de références.
- Dans la phase de parcours-marquage, on extrait une à la fois les références dans la pile et, pour l'objet référencé par la référence extraite :
  + on examine ses successeurs ; lorsque l'on trouve un successeur blanc, on le marque à gris et on le met dans la pile (visiter l'objet). Lorsque l'on veut

mettre une référence à un objet gris dans la pile et que la pile est pleine, on positionne le drapeau `pile_débordée` ;

+ on marque noir l'objet lui-même ;

+ lorsque l'on veut extraire un objet gris de la pile et que la pile est vide, on vérifie si la pile a débordé ou non. Dans l'affirmative, on parcourt l'espace d'objets pour récupérer les objets gris et les empiler.

+ la phase de parcours est terminée lorsque la pile est vide et que le drapeau `pile-débordée` n'est pas positionné. Tous les objets blancs sont alors ceux qui n'ont pas pu être atteints et sont donc des miettes.

• Dans la phase de ramassage, on parcourt encore une fois l'espace d'objets :

+ lorsqu'un objet blanc est trouvé, il est ramassé ;

+ lorsqu'un objet noir est trouvé, il est remis à blanc pour préparer le cycle suivant.

L'algorithme est écrit ci-dessous :

```
{initialisation du cycle de ramasse-miettes}
{p est la pile des références d'objets à parcourir}
{pour tout objet O, O.couleur = BLANC}
empile(p, RA) ; RA.couleur := GRIS ;
pourtout Oact faire        {Oact est une pile d'une activité}
       empile(p, Oact) ; Oact.couleur := GRIS ;
fait

{phase de parcours-marquage}
tantque non pile_vide(p) faire
       dépile(p,O) ;
       pourtout Oi successeur de O faire
          si Oi.couleur = BLANC alors
               empile(p,Oi) ;
               Oi.couleur := GRIS
          fsi
       fait
fait

{pour tout objet O :       O.couleur = blanc ⇔ O inaccessible,

                           O.couleur = noir ⇔ O accessible}
{début de la phase de balayage}
pourtout objet O de la mémoire
faire

       si O.couleur := BLANC alors GL ⇐ GL ∪ {O} ;
       sinon O.couleur := BLANC ;
       fsi
fait
{pour tout objet O, O.couleur = BLANC}
```

```
{procédures  utilisées}
empile  (p,O)
{ p : la pile ; O : référence à un objet }
      si (pile est pleine) alors
            pile_débordée := TRUE ;
      sinon
            empiler (p,O);    {empilement proprement dit}
      fsi

dépile  (p,O)
{ p : la pile ; O : référence d'un objet à retourner}
{ cette fonction n'est appelée que si la fonction pile-vide ci-dessous}
{ a renvoyé la valeur FALSE (il y a encore des objets gris) }
      si (pile est vide) alors
            pile_débordée := FALSE ;
            parcours de l'espace et
            empile (p,O) pour chaque objet gris O;
      fsi
      dépiler (p,O) ;          {dépilement proprement dit}

pile_vide  (p)  :  Booléen
{ p : la pile }
      si (pile est vide) et pile_débordée = FALSE alors
            retourne (TRUE) ;
      sinon
            retourne (FALSE) ;
      fsi
```

Algorithme de type parcours-marquage pour Guide mono-site

Dans le cas particulier où aucune activité n'existe pendant le cycle de ramasse-miettes, la tâche de parcours des piles d'exécution devient nulle, ce qui raccourcit considérablement le temps d'un cycle de ramasse-miettes.

6.4 PARCOURS-MARQUAGE MONO-SITE AVEC ACTIVITES

Le modèle de mémoire d'objets pour cet algorithme est celui de l'algorithme précédent, mais nous laissons les activités s'exécuter en parallèle avec le ramasse-miettes. Comme nous l'avons présenté en 5.1.2, l'espace d'objets peut évoluer à cause des actions d'allocation d'objets et d'affectation de références des activités utilisateurs, et ainsi le graphe des objets accessibles G_A devient dynamique. Pour assurer un ramassage correct, il faut que le collecteur puisse construire un graphe G'_A qui est un sur-ensemble des graphes G_A évoluant dans le cycle (G'_A peut posséder des miettes récemment créées). Cela n'est possible que si les activités participent au marquage des objets lorsqu'elles affectent des références.

L'algorithme du collecteur reste inchangé. Lorsqu'une activité fait évoluer le graphe $G_A \cup G_L$ par une des actions d'allocation d'objets et d'affectation de références, elle doit marquer les objets :

- allouer (var_ref := Class.New) :

```
< crée le nouvel objet ;
  marque à noir ce nouvel objet ;
  affecte sa référence à var_ref ; >
```

- affecter une référence à une variable de référence (var_ref := une_ref) :

```
si l'objet référencé est blanc alors
  < marque à gris l'objet référencé ;
    empile cette référence ;
    affecte cette référence à var_ref ; >
sinon
  affecte cette référence à var_ref ;
```

Le code des actions ci-dessus est généré automatiquement par le compilateur.

Optimisations

Deux problèmes majeurs sont associés à l'algorithme de type parcours-marquage : l'indivisibilité du code associé à l'affectation de références et le parcours de la pile d'exécution des processus (activités dans Guide). Dans ce paragraphe, nous présentons *nos solutions* proposées et implémentées dans Guide pour résoudre ces deux problèmes. Ces optimisations sont basées sur la remarque suivante : l'espace d'objets Guide dont il faut ramasser les miettes est uniquement la MPO qui est un grand espace et qui peut contenir un très grand nombre d'objets. Pratiquement, cet espace n'est rempli totalement qu'après plusieurs heures de travail. Lorsqu'un objet devient miette, le temps nécessaire pour le ramasser peut être retardé de plusieurs heures.

1. Problème d'indivisibilité du code associé à l'affectation de références

Comme nous l'avons présenté au paragraphe précédent, un des inconvénients de l'algorithme du type parcours-marquage fonctionnant en parallèle avec des activités des utilisateurs est le fait que quand une activité affecte une référence d'un objet à l'état d'un autre objet (la pile d'exécution d'une activité fait partie de son état), elle doit marquer à gris l'objet référencé. De plus, l'action d'affectation et l'action de marquage doivent s'effectuer en exclusion mutuelle avec le collecteur. Généralement, on peut établir une section critique (en utilisant les sémaphores ou les moniteurs...) pour réaliser cette exclusion. Cette solution est lourde et on cherche une autre solution.

Nous reprenons la situation présentée au chapitre 5 qui provoque le ramassage incorrect des objets accessibles. En regardant le schéma 5.6, on peut constater que l'indivisibilité du code associé à une affectation de références est violée uniquement dans le cas où le collecteur, en remettant à blanc tous les objets (dans la phase de ramassage), défait l'action de marquage à gris d'une activité et entre dans la phase de parcours-marquage du cycle suivant avant que l'activité puisse affecter la référence.

Supposons qu'on dispose d'un majorant `'temps_max_affectation'` du temps d'exécution du code associé à une affectation de références. Si l'intervalle de temps séparant

la dernière action de remise à blanc effectuée par le collecteur (dans la phase de ramassage d'un cycle) du début de la phase de parcours-marquage du cycle suivant est au moins 'temps_max_affectation', alors le problème du schéma 5.6 est résolu. Pour cela, on enregistre le moment où la phase de ramassage se termine dans une variable 'fin_cycle_précédent' et avant d'entrer dans la phase de parcours-marquage du cycle suivant, on attendra si nécessaire pour assurer l'intervalle voulu.

Le coût de cette solution : on n'a pas besoin de mettre deux actions d'affectation et de marquage dans une section critique mais le temps nécessaire pour un cycle de ramasse-miettes peut augmenter (au maximum 'temps_max_affectation'). En pratique, on n'exécute le ramasse-miettes qu'à une fréquence relativement faible (toutes les heures en cas d'utilisation massive), la condition annoncée précédemment est toujours vérifiée.

2. Problème de parcours de la pile des activités

Comme nous l'avons présenté en 6.3, les objets 'activités' doivent être considérés comme les racines et on doit parcourir leur pile d'exécution pour marquer à gris les objets référencés. Ces parcours sont très coûteux mais doivent être faits à chaque cycle de ramasse-miettes. Existe-t-il un moyen de supprimer ces parcours ou au moins de diminuer leur fréquence ?

Nous en avons trouvé un. Le principe de notre solution est de remplacer le parcours des piles lors de l'exécution de chaque cycle de ramasse-miettes par un parcours de la pile d'une activité lorsque cette dernière dépasse un certain temps d'exécution (fixé ou modifiable par le système). Si ce temps est supérieur à la plupart des temps d'exécution des activités, le parcours de piles ne sera presque plus jamais effectué. En contre partie, la détection des miettes et leur récupération seront retardées d'un temps comparable.

La durée de vie des activités est normalement limitée. On suppose que le temps de vie maximal des activités est de 'durée_de_vie_max_activité'. Plus cette valeur est grande, plus le nombre des activités qui demeurent vivantes après cette valeur est faible. Chaque fois qu'une activité se termine, sa pile d'exécution est libérée et on n'a pas besoin d'examiner des références à des objets qui y existent. D'où vient l'optimisation suivante concernant le parcours des piles :

- chaque activité possède un champ qui enregistre sa date de création. Ce champ sert à calculer l'âge de l'activité ;
- chaque objet possède un champ qui enregistre le dernier moment où il devient blanc à partir d'une autre couleur (cette action est faite uniquement par le collecteur dans la phase de ramassage). Ce champ sert à calculer l'âge "inaccessible" des objets ;
- au début de chaque cycle de ramasse-miettes, au lieu de parcourir toutes les piles d'exécution des activités, on vérifie leur âge. Pour une activité dont l'âge dépasse le seuil 'durée_de_vie_max_activité', on considère qu'elle est recréée en mettant à zéro son âge, puis on parcourt sa pile et on marque à gris les objets référencés.

- dans la phase de ramassage, on ne ramasse que les objets blancs dont l'âge dépasse le seuil "durée_de_vie_max_activité" ;

Remarque : l'âge d'une activité ou d'un objet est compté relativement au début du cycle de ramasse-miettes en question.

Donnons de façon informelle la preuve de cette optimisation. Les trois prédicats suivants sont maintenus vrais :

a) La pile de toute activité ne référence que des objets ayant un âge inférieur à l'âge de l'activité. Cela vient du fait que la première fois qu'on affecte une référence dans la pile, l'objet référencé est marqué à gris et s'il devient blanc après, son âge est remis à zéro.

b) A la phase de ramassage, il n'y a aucune activité dont l'âge dépasse le seuil "durée_de_vie_max_activité".

c) tous les objets référencés par les piles d'exécution des activités ont leur âge qui est inférieur au seuil. Ce prédicat est déduit directement des deux prédicats précédents.

Après la phase de parcours-marquage du collecteur, les objets blancs ne sont pas référencés par d'autres objets accessibles. Ils peuvent être référencés uniquement par les références qui existent dans les piles des activités. Mais si leur âge dépasse le seuil "durée_de_vie_max_activité", le prédicat c) ci-dessus nous montre qu'ils ne sont également référencés par aucune activité, donc ils sont vraiment miettes.

Le coût de cette solution :

- chaque objet possède un champ de plus qui enregistre le dernier moment où il devient blanc ;
- les objets sont ramassés après un temps différé qui est égal au seuil "durée_de_vie_max_activité". Cette valeur est à notre choix : plus grande est cette valeur, plus faible est le nombre des activités dont on doit parcourir la pile d'exécution (et plus grand est le nombre des miettes non ramassées à temps).

6.5 PARCOURS-MARQUAGE REPARTI

L'idée de ce ramasse-miettes est de construire le graphe de dépendance global des objets accessibles en utilisant des ramasse-miettes locaux qui construisent la partie locale du graphe global. Pour la détection de la terminaison de la construction du graphe global, nous utilisons l'algorithme de MISRA [Misra 83] parce que cet algorithme est bien adapté à l'environnement de Guide et qu'il est simple à réaliser. Dans ce paragraphe, nous résumons tout d'abord l'algorithme de détection de la terminaison répartie de MISRA, puis nous présentons le modèle de mémoire d'objets répartie de Guide et enfin nous présentons le ramasse-miettes réparti lui-même.

6.5.1 ALGORITHME DE DETECTION DE TERMINAISON REPARTIE DE MISRA

Un algorithme réparti est construit par les processus locaux. Un processus sera dit actif lorsqu'il exécute son texte de programme et passif dans les autres cas. Un processus passif peut être soit terminé, soit en attente de messages en provenance d'autres processus. Lorsque tous les processus sont passifs et qu'il n'y a pas de message en transit, l'algorithme distribué est dit terminé. Tous les algorithmes proposés commencent par structurer l'ensemble des processus de façon à pouvoir le parcourir pour vérifier la propriété.

a. Hypothèses

Pour que l'algorithme de MISRA fonctionne correctement, les hypothèses suivantes doivent être respectées :

- la structure des processus est un circuit C prédéfini auquel on peut accéder par les deux fonctions suivantes :

```
fonction taille (C : Circuit) : entier ;
fonction successeur (C : Circuit, i : 1..n) : 1..n ;
```

  La fonction `taille` donne la taille du circuit, et `successeur` donne, pour le processus P_i qui l'exécute, le numéro du successeur sur le circuit.

- la fiabilité de communication : les messages ne se perdent pas ;
- le séquencement de messages : entre deux processus communicants, les messages sont reçus dans leur ordre d'émission.

La terminaison globale est réalisée lorsque tous les processus sont passifs et qu'il n'y a plus de message en transit. Pour détecter la passivité de tous les processus, MISRA propose de faire visiter les processus dans le circuit par un jeton. Mais la constatation par le jeton que tous les processus étaient passifs lorsqu'ils ont été visités ne permet pas de conclure que la terminaison s'est réalisée : des processus ont pu être réactivés et des messages peuvent être en transit. En conséquence, on ne peut conclure à la terminaison que lorsque le jeton a effectué deux tours en ne visitant que des processus passifs.

b. Comportement du jeton

Des couleurs associées aux processus vont permettre de résoudre le problème de capter le fait qu'un processus est resté passif en permanence depuis la dernière visite du jeton. Lorsqu'il devient actif un processus devient noir, c'est le jeton qui le peint en blanc lorsqu'il quitte le processus. Ainsi si le jeton retrouve blanc un processus c'est que ce dernier est resté passif en permanence depuis la dernière visite. Lorsque le jeton a visité tous les processus et qu'il les a tous trouvés blancs il peut en conclure la terminaison.

Tous les processus ont des comportements identiques. Tout processus peut initialiser une détection avec son propre jeton.

c. Algorithme

Le jeton véhicule une valeur j qui indique que les j derniers processus visités sont restés en permanence passifs (ces processus étaient blancs lorsque le jeton les a atteint). Le jeton détectera la terminaison lorsque l'on aura `j = taille (C)`.

Chaque processus P_i est doté des déclarations suivantes :

```
var    couleur : (blanc, noir) initialisé à noir ;
       état : (actif, passif) initialisé à actif ;
       jetonprésent : booléen initialisé à faux ;
       nbprocpassif : entier ;
```

où `nbprocpassif` sert à mémoriser la valeur associée au jeton entre sa réception et sa réémission. Les messages sont de deux types distincts (`messages/jeton`) selon qu'il s'agit des messages de l'algorithme contrôlé ou de son contrôle.

Le texte de l'algorithme pour chaque processus P_i est le suivant :

```
lors de
       réception de (message,m)
         faire état ← actif ;
              couleur ← noir ;
         fait ;

       attente de (message, m)
         faire état ← passif fait ;

       réception de (jeton, j)
         faire nbprocpassif ← j ; jetonprésent ← vrai ;
              si nbprocpassif = taille (C) et couleur = blanc
                   alors terminaison détectée
              fsi ;
         fait ;

       émission de (jeton, j)
         possible seulement si jetonprésent et état = passif ;
         faire si couleur = noir alors nbprocpassif ← 0
                                  sinon nbprocpassif ← nbprocpassif + 1
              fsi ;
              envoyer (jeton, nbprocpassif) à successeur (C, i) ;
              couleur ← blanc ;
              jetonprésent ← faux ;
         fait ;
```

Comme le texte de l'algorithme le montre, la couleur noire d'un processus indique qu'il a reçu au moins un message depuis la dernière visite du jeton. Initialement, un processus quelconque possède le jeton et est donc le seul à avoir `jetonprésent` à vrai. Ce processus n'est pas nécessairement celui qui détectera la terminaison. En effet la valeur associée au jeton peut être remise à zéro par n'importe quel processus qui n'est pas resté en permanence passif.

6.5.2 MODELE DE MEMOIRE D'OBJETS GUIDE POUR CET ALGORITHME

L'algorithme de ramasse-miettes global fonctionne sur le modèle de mémoire d'objets Guide qui possède les caractéristiques suivantes :

- L'espace global des objets du système est partitionné en différents sous-espaces locaux et distincts ; chacun possède un modèle comme celui présenté en paragraphe 6.3.2.
- chaque objet possède uniquement un exemplaire de son état, cet exemplaire existe à un moment donné sur un seul site ;
- A chaque espace local est associé un collecteur local (cf. 6.3.2) qui exécute :
  + la phase d'initialisation et de ramassage uniquement sur les objets contenus dans son espace local,
  + la phase de parcours-marquage uniquement sur les objets contenus dans son espace local. Lorsqu'il rencontre une référence à un objet distant pendant cette phase, il envoie un message contenant la référence de l'objet distant au ramasse-miettes distant associé à l'espace de stockage de cet objet pour lui demander le marquage à gris de l'objet. Ce message de demande de marquage d'un objet est asynchrone sans acquittement.
- Le réseau de communication entre sites doit supporter les hypothèses demandées par l'algorithme de détection de terminaison globale de Misra (cf. 6.5.1.a) ;
- Tous les sites doivent fonctionner sans panne pendant le cycle de ramasse-miettes.

6.5.3 PRINCIPE DE L'ALGORITHME DE PARCOURS-MARQUAGE

Il est évident que si chaque collecteur local fonctionne isolément, il ne peut pas savoir quand il peut terminer la tâche et par conséquent quelles sont les miettes qui existent sur son site. En effet, à tout instant un site peut posséder un objet référençant un autre objet à distance qui y est considéré localement comme une miette. En résultat, tous les collecteurs doivent travailler ensemble et un protocole de détection de terminaison doit être utilisé.

Au début, les collecteurs sont tous activés soit par une autre activité, soit à la main. Comme la façon d'activer les collecteurs n'est pas importante, nous ne la précisons pas ici.

Pour chaque collecteur local, la tâche de la phase d'initialisation est de marquer à gris les racines et de les mettre dans la pile de références. Ces tâches, qui ne traitent que des objets locaux, sont indépendantes.

Par contre, la phase de parcours-marquage des sites doit être synchronisée.

Condition de terminaison de la phase de parcours

La phase de parcours global est terminée correctement s'il ne reste aucun objet gris sur le système et aucun message de demande de marquage à gris en circulation. L'algorithme de détection de terminaison de Misra est utilisé pour détecter la terminaison de cette phase :

- Au début, il y a un site (par exemple le site 0) qui possède le jeton ;
- le collecteur est dit passif lorsqu'il termine sa propre phase de parcours. Dans le cas contraire, il est dit actif ;

- lorsqu'un message de mise à gris d'un objet est reçu sur un site où son collecteur est dans l'état passif, il est réactivé pour continuer à faire la phase de parcours.

Lorsqu'un site détecte la terminaison globale de la phase de parcours-marquage, il envoie aux autres sites cet événement pour qu'ils puissent entrer dans la phase de ramassage. Cette phase est faite de façon indépendante mais la détection de terminaison de tous les sites est indispensable dans le cas où on veut réactiver un autre cycle de ramasse-miettes.

L'algorithme du collecteur local de chaque site est écrit ci-dessous.

```
{initialisation du cycle de ramasse-miettes}
{p est la pile des références d'objets à parcourir}
{pour tout objet O, O.couleur := BLANC}
état_collecteur := ACTIF ;
couleur_collecteur := BLANC ;
empile(p, RA) ; RA.couleur := GRIS ;
pourtout Oact faire        {Oact est une pile d'une activité}
      empile(p, Oact) ;
      Oact.couleur = GRIS ;
fait

{phase de parcours-marquage}
tantque non pile_vide(p) faire
      dépile(p,O) ;
      pourtout Oi successeur de O
        faire
            si Oi est local alors
                si Oi.couleur = BLANC alors
                    empile(p,Oi) ;
                    Oi.couleur = GRIS
                fsi
            sinon       { Oi se situe sur autre site k }
                envoi_demande_marquage (Oi,k)
            fsi
        fait
fait
état_collecteur := PASSIF ;
émission_jeton();
attente/détection de la terminaison globale

{début de la phase de balayage}
{pour tout objet O :      O.couleur = BLANC ⇔ O inaccessible,
                          O.couleur = NOIR ⇔ O accessible}
pourtout objet O de la mémoire
faire
      si O.couleur = BLANC alors GL ⇐ GL ∪ {O} ;
      sinon O.couleur = BLANC ;
      fsi
```

```
fait
{pour tout objet O, O.couleur = BLANC}

{procédures utilisées}
empile (p,O)
{ p : la pile ; O : référence d'un objet }
        si (pile est pleine) alors
                pile_débordée := VRAI ;
        sinon
                empiler (p,O);    {empilemment proprement dit}
        fsi

dépile (p,O)
{ p : la pile ; O : référence d'un objet à retourner}
{ cette fonction n'est appelée que si la fonction pile-vide ci-dessous}
{ a renvoyé la valeur FALSE (il y a encore des objets gris) }
        si (pile est vide) alors
                pile_débordée := FAUX ;
                parcours de l'espace et
                empile (p,O) pour chaque objet gris O;
        fsi
        dépiler (p,O) ;          {dépilemment proprement dit}

pile_vide (p) : Booléen
{ p : la pile }
        si (pile est vide) et pile_débordée = FAUX alors
                retourne (VRAI) ;
        sinon
                retourne (FAUX) ;
        fsi
```

Algorithme de type parcours-marquage pour Guide réparti

Traitement des messages

Il y a trois catégories de messages :

- Jeton (sa valeur) ;
- Demande de marquage à gris d'un objet ;
- Signal de la terminaison globale de la phase de parcours-marquage ;

Pour chaque catégorie de messages, on a deux primitives pour soit envoyer soit recevoir chaque message. La primitive d'envoi d'un message est appelée à la demande. Par contre, la primitive de réception d'un message est appelée lorsque le message arrive de façon asynchrone avec le déroulement du collecteur.

```
JETON

    réception de (jeton, j)
```

```
        faire nbprocpassif ← j ; jetonprésent ← VRAI ;
              si nbprocpassif = taille (C) et couleur = BLANC alors
                    { terminaison détectée }
                    envoi_terminaison(autres sites);
                    entre dans la phase de ramassage ;
              fsi ;
        fait ;

  émission de (jeton, j)
        possible seulement si jetonprésent et état = PASSIF ;
        faire si couleur = noir alors nbprocpassif ← 0
                                  sinon nbprocpassif ← nbprocpassif + 1
              fsi ;
              envoyer (jeton, nbprocpassif) à successeur (C, i) ;
              couleur ← BLANC ;
              jetonprésent ← FAUX ;
        fait ;
```

DEMANDE DE MARQUAGE

- envoi_demande_marquage (référence, site)
- reçoit_demande_marquage (référence)

```
  si référence.couleur = BLANC alors
     marque à gris l'objet référencé et l'empile ;
     si couleur_collecteur = PASSIF alors
       active le collecteur pour refaire le parcours
     fsi
  fsi
```

MESSAGE DE TERMINAISON GLOBALE

- envoi_terminaison (autres sites)
- reçoit_terminaison(message de terminaison)

```
  se réveille et entre dans la phase de ramassage
```

Quand on permet à des activités utilisateurs de fonctionner en parallèle avec les collecteurs, les activités doivent participer au marquage. Le collecteur reste tel que nous l'avons décrit en 6.5.3.

6.6 RAMASSE-MIETTES LOCAL SUR CHAQUE SITE

Avec le ramasse-miettes global présenté au paragraphe précédent, chaque collecteur local ne peut pas reconnaître les miettes sur son site après sa propre phase de parcours-marquage. En effet, après la terminaison de la phase de parcours-marquage locale d'un site, les objets blancs de ce site ne sont pas référencés localement, mais ils peuvent être référencés par des objets d'autres sites. Si l'on a un moyen de reconnaître quels sont les objets d'un site référencés par des objets d'autres sites, on peut détecter les miettes parmi les objets blancs immédiatement après la terminaison locale de la phase de parcours-marquage sans avoir besoin de communiquer avec les autres sites du système.

Dans le système Guide, les sites communiquent entre eux par des envois des messages. La référence à un objet d'un site est connue par un autre site si ce dernier a reçu un message qui porte la référence de l'objet en question. Pour savoir quels sont les objets éventuellement référencés à distance, chaque site maintient une Table de Références d'Objets Locaux référencés à Distance (TROLD), et chaque fois qu'il envoie une référence d'un objet de son site à autre site, il doit ajouter cette référence dans la table si elle n'y existe pas encore.

Avec la table TROLD maintenue par le système, on peut utiliser le ramasse-miette local présenté en 6.3 avec les modifications suivantes pour ramasser les miettes locales sur un site :

- tous les objets référencés par les entrées de la table TROLD sont des racines.
- lorsque l'on rencontre une référence d'un objet à distance dans la phase de parcours-marquage, on ne fait rien.

Avec les modifications ci-dessus, les objets qui restent blancs après la phase de parcours-marquage sont vraiment des miettes. Par contre, il peut y avoir des miettes colorées non détectées. Cela vient du fait que la table TROLD est un sur-ensemble de références existant à distance. Le ramasse-miettes réparti présenté dans le paragraphe 6.5 reste encore essentiel parce qu'il est le seul à pouvoir ramasser toutes les miettes dans le système entier.

6.7 CONCLUSION

Dans le système Guide où la mémoire d'objets est répartie sur plusieurs sites, on doit avoir un algorithme du type parcours-marquage global pour assurer le ramassage total des miettes sur le système entier. L'algorithme qui utilise le protocole de détection de terminaison globale de Misra (cf. 6.5.1) est le plus simple, le plus efficace et le mieux adapté à l'environnement matériel supposé pour Guide. Le point faible de ce ramasse-miettes réparti est la panne d'un site pendant un cycle de ramasse-miettes qui bloque le travail de ramassage d'autres sites. Actuellement, nous prenons la solution la plus simple (mais la plus sûre) de suspendre les autres collecteurs et de relancer le ramasse-miettes plus tard quand tous les sites sont disponibles.

Avec l'expérience du système Guide, les références entre sites sont beaucoup moins fréquentes que les références locales et dans ce cas, nous utilisons fréquemment le ramasse-miettes local sur chaque site présenté dans le paragraphe 6.6 qui peut ramasser les miettes locales sans avoir besoin de communiquer avec les autres sites.

On trouvera dans la conclusion générale (chapitre 7) quelques résultats des mesures effectuées sur les ramasse-miettes.

CHAPITRE 7 : CONCLUSION

7.1 RAPPELS DES OBJECTIFS

Le projet Guide a pour vocation d'être le support d'un ensemble de recherches sur la programmation d'applications réparties. Son objectif principal est le développement d'un système d'exploitation à haut degré d'intégration basé sur le modèle à objets. Le degré d'intégration souhaité fait de la propriété de transparence un aspect clé de ce système : transparence de la localisation, de la réalisation et de la conservation des objets.

Le langage à objets Guide est défini pour fournir aux utilisateurs un moyen de programmer leurs applications réparties en exploitant le plus facilement possible tous les éléments du système. Ecrire une application en Guide consiste à écrire un nouvel ensemble de types et de classes en utilisant des types et des classes existants.

Un des objectifs de ma thèse était de concevoir et de réaliser un compilateur pour traduire les sources des types et des classes en des objets exécutables.

Lorsque les applications s'exécutent, elles créent dynamiquement des objets au fur et à mesure des besoins. Par conséquent, un ramasse-miette doit être présent pour assurer la destruction automatique des miettes tout en garantissant la cohérence de la mémoire d'objets du système. Le deuxième objet de ma thèse était d'étudier les algorithmes existants et de les adapter pour le système Guide.

7.2 RAPPELS DES RESULTATS OBTENUS

Un prototype du système Guide réalisé au-dessus d'Unix est disponible depuis la fin 1988. Actuellement, ce prototype peut fonctionner sur plusieurs types de machines : SUN-3, SUN-386, SUN-4, DEC 3100, DPX-1000 et DPX-2000. Il est constitué d'un système de gestion d'objets, d'un système d'exécution et d'un compilateur du langage à objets Guide.

Le choix d'un modèle à objets comme structure de base du système et comme modèle pour la programmation des applications a permis d'obtenir le niveau d'intégration et d'abstraction souhaité. Les types et les classes (les composants des applications) sont réutilisables et extensibles grâce aux mécanismes de sous-typage et d'héritage. Le contrôle statique (qui utilise la relation de conformité entre types) permet d'allier sécurité et puissance d'expression. Le contrôle dynamique demandé explicitement par l'utilisateur permet de remédier à certaines limitations du contrôle statique.

7.2.1 Résultats obtenus concernant la compilation des types et des classes Guide

Le compilateur du langage Guide est disponible depuis la fin 1988 ; il a évolué jusqu'à maintenant (version 1.5) pour prendre en compte tous les éléments définis dans le modèle.

Le compilateur Guide a permis la mise en œuvre de plusieurs applications écrites dans le langage Guide ; une centaine de milliers de lignes de code Guide ont été écrites et mises au point, ce qui nous permet d'étudier les performances du langage et du système Guide, et d'évaluer l'adéquation du modèle d'objets de Guide pour l'écriture d'applications réparties.

Dans les tables de mesures présentées en 4.6, nous constatons que la performance de la compilation du compilateur Guide est voisine de celle du compilateur Eiffel, à l'exception de la taille du binaire final. En Guide, on partage le code binaire d'une classe avec toutes ses sous-classes, ce qui rend la taille des classes exécutables la plus petite possible.

Nous résumons ci-dessous les principales expériences acquises avec notre modèle d'objets :

- **Programmation par objets** : elle est un intérêt fondamental, en particulier dans un contexte réparti tel que Guide, comme nous l'avons dit dans les paragraphes précédents.
- **Types et classes** : Même s'il existe très peu de cas où un type est réalisé par plusieurs classes et où ces dernières sont utilisées dans une même application, la séparation entre types et classes est un moyen d'abstraction de haut niveau séparant l'interface et la réalisation concrète. Grâce aux types, on n'a pas besoin de la notion de classe différée (dans Eiffel) : on peut utiliser les objets dont le type est connu mais dont la classe n'est éventuellement pas réalisée.
- **Conformité** : il existe peu de cas où deux types indépendants sont liés par la relation de conformité. Par contre, l'utilisation de la conformité entre super-type et sous-type est d'un intérêt indiscutable. Un des buts de rendre explicite la relation de conformité est de permettre aux utilisateurs de réaliser manuellement le sous-typage et l'héritage multiple, qui ne sont pas fournis actuellement par Guide. L'autre but est de permettre la restriction des interfaces.
- **Héritage** : c'est un moyen de réutilisation de programmes fréquemment utilisé pour partager des constantes, des variables et du code de méthodes. En comptant la classe '`Top`', les différentes applications utilisent 5 niveaux de hiérarchie en moyenne.

  Il y a actuellement des incohérences au niveau de la réutilisation des éléments dans les super-classes : on peut réutiliser les variables d'état et les méthodes, mais on ne peut pas réutiliser les procédures définies dans les super-classes. Ces incohérences seront supprimées dans la prochaine version.

  Le besoin de l'héritage multiple est assez fréquent dans les applications et une étude sur ce problème, surtout sur le mécanisme de réalisation sera faite. Il est cependant très lourd, voire impossible, d'implémenter l'héritage multiple dans un

système à objets dans lequel les classes utilisées par une application ne peuvent pas être déterminées à la compilation.

- **Généricité** : c'est un autre moyen de réutilisation de programmes fréquemment utilisé. Le type et la classe générique '`List`' (cf. annexe 5) sont utilisés dans la plupart des applications.
- **Synchronisation** : le choix du contrôle d'accès aux objets partagés au niveau des appels des méthodes est très intéressant et permet de contrôler facilement la cohérence des objets. Pourtant, la synchronisation non bloquante (être averti qu'un appel de méthode ne peut pas s'exécuter à cause des conditions de synchronisation) est nécessaire dans plusieurs cas, mais n'est pas encore supportée par Guide.

7.2.2 Résultats obtenus concernant le ramassage des miettes

Un ramasse-miettes de type compteur de références a été réalisé en ajoutant automatiquement les codes de manipulation des compteurs d'objets dans le code généré lors de la compilation des applications (les types et les classes). Grâce au modèle à objets avec le sous-typage et l'héritage, la réalisation de ce ramasse-miettes est rendue très simple : pour chaque objet on réalise (automatiquement par le compilateur) deux méthodes, l'une pour incrémenter le compteur et l'autre pour le décrémenter (et parcourir éventuellement les références contenues dans l'objet en question pour décrémenter le compteur des objets référencés). Ces méthodes s'exécutent en mode exclusif en utilisant la clause de contrôle de synchronisation EXCLUSIVE. Leur interface entre dans le type "`Top`" et est ainsi héritée par tous les types.

Avec ce type d'algorithme, la manipulation des compteurs est considérée comme un travail des programmes d'applications (bien qu'elle soit faite automatiquement par le compilateur). La transparence de la localisation des objets permet au compilateur d'engendrer facilement le code concernant la manipulation des compteurs d'objets (uniforme et indépendante du fait que l'objet soit local ou distant).

Au niveau de la performance de ce ramasse-miettes, après avoir testé et mesuré l'algorithme sur les programmes de démonstration, nous obtenons un taux de 0% (pour les programmes qui ne manipulent pas de références) à 25% (pour les programmes qui manipulent des références très fréquemment) de temps occupé par la manipulation des compteurs par rapport au temps effectif d'exécution. Ce taux est comparable aux chiffres fournis par les autres systèmes ([Stamos 82][Deutsch 84]). Cette performance est tout à fait acceptable pour ce genre d'algorithmes (remarquons que nous avons optimisé dans une certaine mesure en mettant le compteur dans le descripteur de l'objet et non pas dans l'état de l'objet).

Nous avons aussi un ramasse-miettes de type parcours-marquage qui fonctionne indifféremment en 'off-line' ou en parallèle avec les activités des utilisateurs. C'est un ramasse-miettes global et réparti qui utilise le protocole de détection de terminaison de Misra. Dans le cas où le système Guide fonctionne sur un seul site, ce ramasse-miettes devient

centralisé. Lorsque les activités s'exécutent en parallèle avec le ramasse-miettes, elles doivent obligatoirement participer au marquage des objets. Après avoir mesuré le temps, nous obtenons un taux de 0% à 12% de temps occupé par le marquage par rapport au temps effectif d'exécution des programmes. Ce coût en termes de temps d'exécution est moins important que celui du ramasse-miettes de type compteurs de références, mais on doit payer en plus le coût du collecteur. En conséquence, le choix entre les deux ramasse-miettes est très délicat.

Actuellement, nous utilisons le ramasse-miettes de type compteurs de références (le compilateur génère automatiquement le code de manipulation des compteurs) pour ramasser la plupart des miettes immédiatement. Ce sont des objets temporaires, créés et utilisés dans une seule application (95% des objets créés selon les mesures). Afin de ramasser les miettes restantes, nous exécutons de temps en temps le ramasse-miettes de type parcours-marquage en "off-line" pour ne pas payer le prix de marquage des objets fait par les activités des utilisateurs.

7.3 PERSPECTIVES

La première phase du projet Guide est terminée et nous avons atteint notre objectif prévu : la réalisation d'un prototype d'un système réparti à objets. La deuxième phase du projet est en cours avec son objectif : à partir des expériences acquises, reconçevoir le modèle d'objets et le système, puis implémenter le système en utilisant un autre noyau qui est plus adapté qu'Unix afin d'arriver à un produit utilisable dans l'insdustrielle. Les orientations ci-dessous que je propose sont issues de l'expérience acquise dans Guide pour le compilateur, le ramasse-miettes, ainsi que pour le système.

PERSPECTIVES IMMEDIATES CONCERNANT MON TRAVAIL

- Le problème du contrôle de la cohérence des objets est un problème fondamental dans un système dans lequel les objets sont permanents et partageables. Si la manipulation des objets normaux est bien contrôlée par la relation de conformité et par des clauses de contrôle d'accès concurrent, ce n'est pas le cas pour les types et les classes qui sont actuellement modifiés sans aucun contrôle par le compilateur à la demande de l'utilisateur. Comme il existe des relations délicates entre les types, entre les classes, entre les types et les classes ainsi qu'entre les objets et leur type et leur classe, la gestion de l'évolution des types et des classes est un des problèmes à résoudre. La conception d'un tel gestionnaire est actuellement à l'étude.
- Tous les ramasse-miettes qui fonctionnent actuellement sont réalisés de façon à éviter le problème de parcours des piles des activités. Or pour avoir des ramasse-miettes plus satisfaisants, ce problème de parcours doit être résolu. L'étude des stratégies de parcours des piles fait aussi l'objet d'un travail de recherche dans la seconde phase de conception du système Guide.

PERSPECTIVES CONCERNANT LE LANGAGE

- Le langage Guide a été défini et mis en œuvre dans le but de pouvoir disposer rapidement d'un support d'expérimentation des modèles de programmation et

d'exécution. L'objectif a été atteint mais je pense cependant qu'il est nécessaire que le langage Guide ait une définition propre et notamment qu'il soit complété par une sémantique précise. Cela devrait être l'objet d'un travail dans la seconde phase de conception qui est en cours.

PERSPECTIVES CONCERNANT LE SYSTEME

- Bien qu'il n'existe pas encore, un environnement de développement à la hauteur des possibilités du système est vraiment nécessaire. Cet environnement doit fournir notamment des outils d'aide à l'édition (éditeur guidé par la syntaxe et éventuellement par l'état actuel des objets dans le système) et de mise au point.
- Si la construction des objets par liaison (en utilisant des références) facilite beaucoup la construction des objets complexes et le partage des objets (c'est la motivation), elle engendre des difficultés dans le déplacement d'un objet d'un système à un autre, et notamment dans le montage ou le démontage d'un système d'objets (la mémoire d'objets du système se compose des différents systèmes d'objets, chacun utilise normalement une zone consécutive de mémoire secondaire). Actuellement, on ne peut pas monter ni démonter dynamiquement les systèmes d'objets (ils sont montés une fois pour toutes lors du lancement du système). Il me semble que le montage et le démontage dynamique des systèmes d'objets sont nécessaires et l'étude des conditions et des stratégies de montage et démontage doit être faite. Une fois les problèmes de montage et de démontage résolus, le déplacement des objets d'un système à un autre devient facile à résoudre.

ANNEXE 1

CONVENTIONS DE PROGRAMMATION EN GUIDE

1. Jeu de caractères

Tous les caractères de l'alphabet IA5 sont autorisés. Le code '^L' (touche de contrôle + L) peut être utilisé pour provoquer le saut de page explicite à l'impression et il est ignoré par le compilateur.

2. Identificateurs

Les identificateurs doivent être alphanumériques (le premier symbole étant une lettre) et peuvent contenir le caractère souligné '_'. Les chaînes de caractères correspondant aux mots-clé et aux mots réservés ne doivent pas être utilisées. Distinction est faite entre minuscules et majuscules ; par exemple, `'MyIdent'` et `'myident'` sont deux identificateurs différents.

Les identificateurs des types et des classes forment deux espaces indépendants. Il est donc possible d'utiliser un même identificateur pour désigner un type et une classe. C'est ce qui est souvent fait lorsqu'un type est implémenté par une seule classe.

Syntaxe :

<ident> = <letter> { <letter> | <digit> | '_' }

<letter> = 'A'..'Z' | 'a'..'z'

<digit> = '0'..'9'

Exemple :

`Count`, `PERSON_1` et `my_name` sont des identificateurs corrects ;

`string?`, `12age` et `_name` sont des identificateurs incorrects.

3. Mots-clés

ALL	AND	ASSERTYPE	ATTRIBUTE
BEGIN	BY	BLOCK	CASE
CLASS	CONTROL	CO_BEGIN	CO_END
CONST	CONSTRUCTOR	DIV	DO
ELSE	END	EXCEPT	EXCLUSIVE
FIRST	FOR	FROM	IF
IMPLEMENTS	IN	IN_OUT	IS
MOD	METHOD	NOT	OF
OR	OTHERS	OUT	PROCEDURE

RAISE	READER	REF	REPLACE
REPEAT	RETRY	RETURN	SELF
SIGNALS	SUBCLASS	SUBTYPE	SUPER
SYNONYM	SYSTEM	THEN	TO
TYPE	TYPECASE	UNTIL	UNMOVABLE
WHILE	WRITER	XOR	

Les mots-clé peuvent être écrits indifféremment soit tout en minuscules soit tout en majuscules.

4. Symboles

= (égale)	# (différence)	<
>	<=	>=
+	-	*
/	|	&

5. Mots réservés

Pour désigner les types élémentaires :

Array[n] OF ...	Boolean	Char
Integer	LongInt	ShortInt
Record ... END	String[n]	

Pour désigner les constantes :

FALSE	false	TRUE	true
NIL	nil		

Pour désigner les variables d'environnement prédéclarées :

input	output	catal
myActivity	myDomain	

Mots interdits :

Pour faciliter l'utilisation d'une version ultérieure du compilateur Guide, nous donnons ci-dessous la liste des mots qu'il est interdit d'employer comme identificateurs dans les programmes Guide.

ADDRESS	ATOMIC	BEGIN_TRANSACTION
END_TRANSACTION	ABORT_TRANSACTION	BEGIN_C
END_C BEGIN_LOCK	END_LOCK	INHERIT OPERATOR

6. Commentaires

Les commentaires sur une ligne sont introduits par la chaîne '//' et terminés par le caractère de fin de ligne. Les longs commentaires sont maintenant autorisés en les plaçant entre "/*" et "*/".

Exemple :

```
toto.Print; // voici un commentaire
// et un autre
// commentaire
/* voici un commentaire long. Il peut s'étendre sur plusieurs lignes. Ces
commentaires ne sont pas imbriqués */
```

ANNEXE 2

SYNTAXE DU LANGAGE GUIDE

Conventions d'écriture des éléments de syntaxe :

- les terminaux sont notés en majuscules ou entre guillemets,
- les non-terminaux sont notés entre < et >,
- un nombre quelconque de x s'écrit { x },
- au moins un x s'écrit { x }*,
- au plus un x s'écrit [x],
- exactement un x s'écrit (x),
- l'alternative s'écrit |.

1. DESIGNATION DE TYPE

```
<désignateur_type> =
    <désignateur_type_élémentaire> | <désignateur_type_construit>

<désignateur_type_élémentaire> =
    'Integer' | 'ShortInt' | 'LongInt' | 'Char'  | 'Boolean'
    | <désignateur_type_structuré>

<désignateur_type_structuré> =
    <désignateur_type_record>              (cf. annexe 4)
    | désignateur_type_array>              (cf. annexe 4)
    | désignateur_type_string>             (cf. annexe 4)

<désignateur_type_construit> =
    REF <ident_type>
    [ OF '[' <désignateur_type> { ',' <désignateur_type> } ']' ]
```

2. DEFINITION DE TYPE CONSTRUIT

```
<def_type> =
    <en-tête_type>
    { <synonyme> }
    { <variable d'état visible> }
    { <signature_méthode> }
    <fin_type>
```

```
<def_type_générique> =
    <en-tête_type_générique>
    { <synonyme> }
    { <variable d'état visible> }
    { <signature_méthode> }
    <fin_type>

<en-tête_type> =
    TYPE <ident_type> [ SUBTYPE OF <ident_type> ] IS

<en-tête_type_générique> =
    <en-tête_type_générique_contraint>
    |  <en-tête_type_générique_non_contraint>

<en-tête_type_générique_contraint> =
    TYPE CONSTRUCTOR <ident_type> OF
       '[' <param_générique_contraint>
            { ',' <param_générique_contraint> }
       ']'
    IS

<en-tête_type_générique_non_contraint> =
    TYPE CONSTRUCTOR <ident_type> OF
       '[' <param_générique_non_contraint>
            { ',' <param_générique_non_contraint> }
       ']'
    IS

<param_générique_contraint> =
    <ident> : REF <ident_type>

<param_générique_non_contraint> = <ident>

<fin_type> =
    END [ <ident_type> ] '.'

<synonyme> =
    SYNONYM <ident> '=' <désignateur_type> ';'

<variable d'état visible> = <décl_variable>

<décl_variable> =
    <ident> { ',' <ident> } ':' <désignateur_type>
                              [ '=' <expr_constante> ])  ';'
    | CONST <ident> { ',' <ident> } ':' <désignateur_type>
                              '=' <expr_constante> ';'
```

```
<signature_méthode> =
    METHOD <ident_methode>
    [ '(' <liste_param> ')' ]
    [ ':' <type_retour> ] ';'
    [ <liste_signaux> ]

<liste_param> =
    ( IN  | OUT | IN_OUT ) <décl_param>
    { ';' ( IN  | OUT | IN_OUT ) <décl_param> }

<décl_param> =
    [ <ident> { ',' <ident> } ':' ] <désignateur_type>

<type_retour> =
    <désignateur_type>

<liste_signaux> =
    SIGNALS <ident> { ',' <ident> } ';'
```

3. DEFINITION DE CLASSE

```
<def_classe> =
    <en-tête_classe>
    [ <caractéristiques> ]
    { <synonyme> }
    { <variable_d'état> }
    { <def_méthode> | <def_procédure> }
    [ <bloc_traitement_exception> ]
    [ <bloc_de_restauration> ]
    [ <contrôle> ]
    <fin_classe>

<def_classe_générique> =
    <en-tête_classe_générique>
    [ <caractéristiques> ]
    { <synonyme> }
    { <variable_d'état> }
    { <def_méthode> | <def_procédure> }
    [ <bloc_traitement_exception> ]
    [ <bloc_de_restauration> ]
    [ <contrôle> ]
    <fin_classe>

<en-tête_classe> =
    CLASS <ident_classe> [ SUBCLASS OF <ident_classe> ]
    IMPLEMENTS <ident_type> IS

<fin_classe> = END [<ident_classe>] '.'
```

```
<en-tête_classe_générique> =
    <en-tête_classe_générique_contrainte>
    | <en-tête_classe_générique_non_contrainte>

<en-tête_classe_générique_contrainte> =
    CLASS CONSTRUCTOR <ident_classe>
    [ '[' <param_dim> { ',' <param_dim> } ']' ]
    [ OF '[' <param_générique_contraint>
          { ',' <param_générique_contraint> } ']' ]
    IMPLEMENTS <ident_type> IS

<en-tête_classe_générique_non_contrainte> =
    CLASS CONSTRUCTOR <ident_classe>
    [ '[' <param_dim> { ',' <param_dim> } ']' ]
    [ OF '[' <param_générique_non_contraint>
          { ',' <param_générique_non_contraint> } ']' ]
    IMPLEMENTS <ident_type> IS

<param_dim> = <ident>

<caractéristiques> =
    ATTRIBUTE <caractéristique> { ',' <caractéristique> } ';'

<caractéristique> =
    UNMOVABLE

<variable_d'état> = <décl_variable>

<def_méthode> =
    <en-tête_méthode>
    { <variable_de_travail> }
    BEGIN
          <liste_d'instructions>
          [ <bloc_traitement_exception> ]
          [ <bloc_de_restauration> ]
    END [<ident_méthode>] ';'

<en-tête_méthode> = <signature_méthode>

<variable_de_travail> = <décl_variable>

<procédure> =
    <en-tête_procédure>
    { <variable_de_travail> }
    BEGIN
          <liste_d'instructions>
          [ <bloc_traitement_exception> ]
          [ <bloc_de_restauration> ]
```

```
    END [<ident_procédure>] ';'

<en-tête_procédure> =
    PROCEDURE <ident_procédure>
    [ '(' <liste_param> ')' ]
    [ ':' <type_retour> ] ';'
    [ <liste_signaux> ]

<contrôle> =
    CONTROL
    {<ident> { ',' <ident> } ':' <condition_d'activation> ';' } *

<condition_d'activation> =
    < expression_booléenne>
    | EXCLUSIVE
    | WRITER
    | READER

<liste_d'instructions> =
    <instruction> { <instruction> }

<instruction> =
    ( <affectation>
       | <appel_de_méthode>
       | <appel_de_procédure>
       | <instruction_IF>
       | <instruction_CASE>
       | <instruction_TYPECASE>
       | <instruction_ASSERTYPE>
       | <instruction_FOR>
       | <instruction_WHILE>
       | <instruction_REPEAT>
       | <instruction_CO_BEGIN/CO_END>
       | <instruction_BLOCK>
       | <instruction_RETURN>
       | <instruction_RAISE>
       | BEGIN <liste_d'instructions>
            [ <bloc_traitement_exception> ]
         END
      )  ';'

<affectation> =
    <variable> ':=' <expression>

<expression> =
    <ident>
    | <appel_de_méthode>
    | <appel_de_procédure>
```

```
<variable> = <ident>

<appel_de_méthode> =
    <variable_référence>'.'<ident_méthode>
    [ '('<expression> { ',' <expression> } ')' ]

<variable_référence> = <ident>

<appel_de_procédure> =
    <ident_procédure> [ '('<expression> { ',' <expression } ')' ]

<instruction_IF> =
    IF <expression_booléenne> THEN
          <liste_d'instructions>
    [ ELSE <liste_d'instructions> ]
    [ <bloc_traitement_exception> ]
    END

<instruction_CASE> =
    CASE <expression> OF
    { <liste_d'étiquettes> ':' <liste_d'instructions> END ';'  }*
    [ OTHERS ':' <liste_d'instructions> ]
    [ <bloc_traitement_exception> ]
    END

<instruction_TYPECASE> =
    TYPECASE <variable_référence> OF
    { <ident_type> ':' <liste_d'instructions> END ';' }*
    [ OTHERS ':' <liste_d'instructions> ]
    [ <bloc_traitement_exception> ]
    END

<instruction_ASSERTYPE> =
    <variable_référence> ':=' "ASSERTYPE" '(' <expression> ')'

<instruction_FOR> =
    FOR <ident> ':=' <exp_entière> TO <exp_entière> [BY <exp_entière>]
    DO
       <liste_d'instructions>
       [ <bloc_traitement_exception> ]
    END

<instruction_WHILE> =
    WHILE <expression_booléenne> DO
       <liste_d'instructions>
       [ <bloc_traitement_exception> ]
    END

<instruction_REPEAT> =
```

```
    REPEAT
       <liste_d'instructions>
       [ <bloc_traitement_exception> ]
    UNTIL <expression_booléenne>

<instruction_CO_BEGIN/CO_END> =
    CO_BEGIN
       { <ident_activité_fille> ':' <appel_de_méthode> ';' }*
       [ <bloc_traitement_exception> ]
    CO_END [ <expression_booléenne> | ALL | FIRST ]

<instruction_BLOCK> =
    <ident> ':=' BLOCK
       {<variable_de_travail>}
    BEGIN
       <liste_d'instructions>
       [<bloc_traitement_exception>]
    END

<instruction_RETURN> =
    RETURN [ <expression> ]

<instruction_RAISE> =
    RAISE <ident_exception> ';'

<bloc_traitement_exception> =
    EXCEPT
       { <masque_d'exception> ':' <traitant> }*
    END

<masque_d'exception> =
    <ident_exception> | ALL
    [ FROM
      ( [<ident_méthode> '.' ] <ident_type> | SYSTEM | ALL )
    ]

<traitant> = <instruction> ';'
```

ANNEXE 3

ANALYSE LEXICALE ET SYNTAXIQUE DU COMPILATEUR GUIDE

1. ANALYSE LEXICALE

Nous utilisons actuellement, pour des raisons de simplicité d'écriture et de mise à jour de notre analyse lexicale, LEX comme un moyen de construction de l'analyseur lexical des objets sources. La fonction essentielle de cette partie du compilateur est de transformer la chaîne de caractères d'un objet source en une chaîne de symboles lexicaux et de détecter les caractères illégaux. Pour utiliser LEX, nous écrivons un source LEX contenant un ensemble de règles (expressions régulières en termes de LEX), dont chaque règle exprime un symbole particulier. Une action particulière codée sous forme d'un bloc d'instructions C est associée à chaque règle ; cette action est normalement terminée par une instruction de retour pour rendre à la partie suivante un identificateur (numérique) du symbole. En conséquence, le squelette du source LEX qui joue le rôle de l'analyse lexicale du compilateur Guide est le suivant :

```
/* définition des identificateurs numériques associés aux symboles */
#define TYPE      256
#define IDENT     257
...
%% Partie de définition des règles pour la reconnaissance des symboles
"TYPE" | "type"          { return (TYPE); }
[a-zA-Z][a-zA-Z0-9_]*    { return (IDENT) ; }
....
```

Le générateur LEX compile ce programme source LEX et produit un module (fichier) C qui contient les variables globales et les fonctions accessibles à d'autres modules C. Parmi les fonctions produites par LEX, il y a une fonction principale qui s'appelle `yylex()`, c'est l'analyseur lexical dont le fonctionnement est spécifié dans le source LEX. Le fait de produire des variables globales et des fonctions accessibles à d'autres modules oblige un programme donné à utiliser un seul module résultat de LEX, sinon des conflits entre ces variables et ces fonctions se produisent certainement. C'est le cas dans le compilateur Guide actuel où on utilise deux modules différents qui sont résultats de LEX : l'un se charge d'analyser l'objet source, l'autre est utilisé par le gestionnaire de types et de classes (intégré dans le compilateur) pour saisir les objets description de types ou de classes au fur et à mesure des besoins.

En résumé, lorsque l'on veut utiliser deux ou plusieurs modules produits par LEX dans un même programme (c'est notre cas), on doit chercher une solution pour résoudre les conflits de noms symboliques entre les variables et les fonctions produites par LEX.

Il y a plusieurs solutions pour résoudre ce problème et la solution la plus agréable est d'étendre le générateur LEX lui-même pour qu'il puisse générer le module résultat dans lequel les variables et les fonctions sont soit globales, soit statiques selon l'option à la compilation. Pour utiliser plusieurs modules sources de LEX dans un même programme, on les compile avec l'option de production des variables et des fonctions statiques, puis on inclut chaque module résultat dans le module qui l'utilise. C'est le choix qui a été fait pour le compilateur Guide actuel.

Traitements des erreurs lexicales

Il y a deux types d'erreurs lexicales que l'on doit traiter :

- utilisation des caractères interdits ;
- les fautes d'orthographe sur un mot-clé ou sur un opérateur.

La détection du premier type d'erreurs est très facile à mettre en œuvre dans notre contexte où nous utilisons le générateur LEX : après avoir construit les règles qui reconnaissent et qui traitent les symboles valides du langage, on ajoute une règle particulière (fournie par LEX) pour reconnaître les caractères illégaux et dans le bloc d'instructions associé à cette règle on peut coder les traitements voulus. Nous réalisons actuellement le traitement le plus simple pour ce type d'erreurs : afficher un message d'erreur, ignorer le caractère illégal (sans le donner à l'analyseur syntaxique) et continuer à traiter le symbole suivant.

Les fautes d'orthographe sur un mot-clé ou sur un opérateur transforment généralement des mots-clés en des identificateurs et il est difficile, voire impossible, de les détecter dans la phase d'analyse lexicale dans la mesure où cette phase ne dispose d'aucune d'information hiérarchique des symboles lexicaux. Nous choisissons le traitement le plus fréquemment utilisé pour ce type d'erreurs : laisser le traitement de ces erreurs à la phase d'analyse syntaxique dans laquelle elles sont détectées car elles provoquent sûrement des violations des règles grammaticales du langage.

2. ANALYSE SYNTAXIQUE

Pour les mêmes raisons de simplicité d'écriture et de souplesse de mise à jour, nous utilisons actuellement YACC pour construire notre analyseur syntaxique. La fonction essentielle de cette partie du compilateur est de vérifier si la chaîne de symboles rendue par `yylex` (fonction d'analyse lexicale créée par LEX) obéit ou non aux règles grammaticales du langage. Pour utiliser YACC, nous écrivons un source YACC contenant un ensemble de règles (`productions` en termes de YACC), dont chacune exprime une phrase particulière de la grammaire. Une action particulière codée sous forme d'un bloc d'instructions C est associée à chaque règle ; cette action sert à construire l'arbre syntaxique associé à l'élément exprimé par la règle. En conséquence, le squelette du source YACC qui joue le rôle de l'analyse syntaxique du compilateur Guide est le suivant :

```
/* définition des identificateurs numériques associés aux symboles */
/* rendus par yylex. On les appelle tokens */
```

```
#TOKEN  TYPE
#TOKEN  IDENT
#TOKEN  METHOD
#TOKEN  POINT_VIRGULE
...
%% Partie de définition des règles pour la reconnaissance des phrases
SignatureMéthode :      METHOD IDENT DeclarationParamètres
                        DefException POINT_VIRGULE
          { /* action associée à la règle : construction de l'arbre
               syntaxique pour cet élément */       }
DeclarationParamètres : ...
          { /* action associée à la règle : construction de l'arbre
               syntaxique pour cet élément */       }
DefException :          ...
          { /* action associée à la règle : construction de l'arbre
               syntaxique pour cet élément */       }
            | error
          { /* action associée à la règle : traitement d'erreurs
                 et resynchronisation de l'analyseur */    }
....
```

Avec la même méthode de compilation que LEX, YACC compile le programme source YACC et produit un module (fichier) C qui contient les variables globales et les fonctions accessibles aux autres modules C ; parmi les fonctions produites, on obtient la fonction `yyparse()` qui constitue l'analyseur syntaxique du langage dont la grammaire est spécifiée dans le source. Comme les modules produits par LEX, un module produit par YACC contient des variables globales et des fonctions accessibles à d'autres modules ; cela provoque certainement des conflits entre les noms lorsque l'on utilise deux ou plusieurs modules, produits par YACC dans un même programme (c'est notre cas). Nous choisissons et réalisons la même solution appliquée pour LEX (voir le paragraphe précédent) afin d'éviter les conflits.

Avec la chaîne de symboles produite par l'analyseur lexical `yylex()` en entrée, les règles définies dans le source YACC permettent à l'analyseur syntaxique produit de reconnaître les éléments syntaxiques. Chaque fois qu'un élément syntaxique est reconnu, l'action qui lui est associée est exécutée ; cette action construit généralement l'arbre syntaxique correspondant.

Après la phase d'analyse syntaxique, on obtient un arbre syntaxique pouvant contenir des erreurs sémantiques et non suffisamment documenté pour permettre la génération du code. C'est le rôle de la phase d'analyse sémantique de détecter les erreurs sémantiques et de compléter l'arbre.

Traitements des erreurs syntaxiques

Une erreur syntaxique est détectée chaque fois que la chaîne des symboles produite par l'analyse syntaxique n'obéit pas aux règles grammaticales du langage. Il y a actuellement quatre méthodes de traitement d'erreurs syntaxiques et de resynchronisation [Aho 86] que nous présentons ci-dessous ainsi que notre réalisation.

En ce qui concerne le traitement d'erreurs, YACC fournit un élément particulier appelé `error`. Le programmeur peut utiliser cet élément autant de fois que et où il la désire pour construire les règles de grammaire. A l'exécution, lorsqu'il y a réellement erreur correspondant à un élément x, ce dernier est activé et l'action qui lui est associée est exécutée. Le programmeur peut mettre dans l'action le traitement d'erreur désiré qui, le plus souvent, consiste en une resynchronisation du travail de l'analyseur.

a. Synchronisation en mode "`panique`"

C'est la méthode la plus simple à mettre en œuvre. Lorsqu'une erreur est détectée, le compilateur continue de saisir les symboles suivants mais il ne traite aucun symbole jusqu'au moment où il rencontre un des symboles particuliers que l'on appelle **symbole de synchronisation.** Les symboles de synchronisation sont choisis de deux manières :

- soit il existe un ensemble de symboles de synchronisation par règle (ou groupe de règles) grammaticale ;
- soit il existe un seul ensemble de symboles de synchronisation commun à toutes les règles.

Généralement les symboles de synchronisation sont les séparateurs et les terminateurs des entités syntaxiques. Par exemple, des points-virgules qui sont utilisés comme des terminateurs des instructions dans la plupart des langages sont utilisés comme symboles de synchronisation. Bien que cette méthode suspende l'analyse de la grammaire pendant plusieurs symboles, elle est fréquemment utilisée car :

- elle est facile à mettre en œuvre ;
- elle assure qu'aucune boucle infinie ne sera jamais introduite dans l'analyse ;
- en pratique, il est rare d'avoir plusieurs erreurs dans une même entité syntaxique.

Dans Guide, nous implémentons cette méthode pour les entités les plus utilisées comme les expressions et les instructions. Par exemple, nous choisissons le point-virgule et le symbole "END" comme symboles de synchronisation pour les instructions. Lorsqu'une erreur est détectée dans une instruction, cette dernière est abandonnée et le compilateur fait l'analyse pour l'instruction suivante.

b. Synchronisation au niveau des phrases

Dans cette méthode, lorsqu'une erreur est détectée, l'analyseur fait, si possible, une correction locale sur les symboles suivants (y compris le symbole qui provoque l'erreur) afin de construire une règle de grammaire correcte. Par exemple, il peut remplacer un ou plusieurs symboles par d'autres (remplacer un point-virgule par une virgule ou vice versa).

Le choix des corrections locales possibles à effectuer est sous la responsabilité du programmeur du compilateur, il n'existe pas de règle universelle ; de plus, cette méthode peut produire des boucles infinies.

Avec YACC, lorsqu'une erreur a été détectée par l'analyseur (à cause du dernier symbole pendant la reconnaissance d'une production), on ne peut pas corriger le symbole incorrect et

refaire la reconnaissance de la production en question. C'est une faiblesse de YACC et pour remédier à ce problème, on doit corriger les symboles qui provoquent les erreurs syntaxiques avant de les donner à l'analyseur syntaxique produit par YACC.

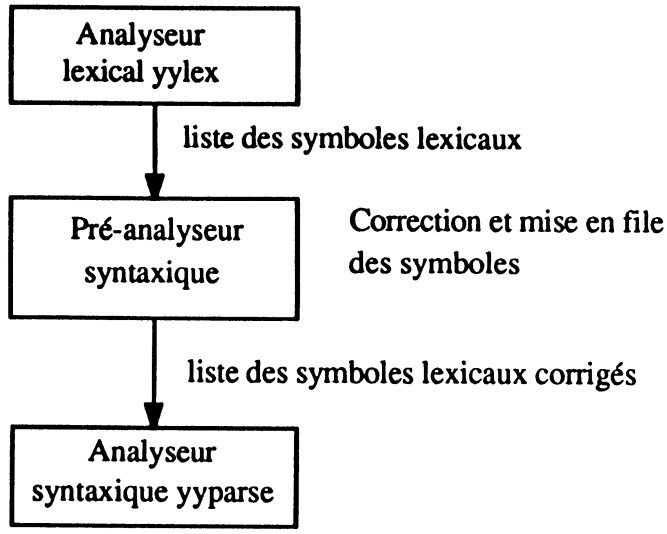


Figure A3.1 : Un pré-analyseur syntaxique entre yylex et yyparse

La solution que nous avons choisie est d'ajouter un mini-analyseur ou pré-analyseur syntaxique qui se situe entre LEX et YACC et qui est chargé de détecter et de corriger localement des symboles incorrects (Fig. A3.1). Nous désirons corriger à ce niveau les points-virgules, les parenthèses et les symboles 'END' oubliés ou superflus. Par exemple, avec la syntaxe du langage Guide [ROI90], lorsque les mots-clés marquant le début d'une instruction comme IF, CASE, FOR, WHILE, REPEAT, BEGIN, ... sont rencontrés, il faut que le symbole qui les précède soit un des symboles ELSE, BEGIN, THEN, DO, REPEAT, ';' (point-virgule). Sinon, une erreur est certainement commise et on peut ajouter avant ces mots-clés un point-virgule. Cette correction permet de corriger éventuellement l'entité qui précède l'instruction en question. De toute façon, elle permet au mécanisme "panique" de synchroniser correctement l'analyseur pour continuer à traiter correctement l'instruction en question (l'analyseur laisse tomber les symboles avant le point-virgule ajouté et recommence à traiter l'instruction).

c. Productions d'erreurs

Si l'on peut prévoir les erreurs couramment commises, on peut augmenter la grammaire de nouvelles règles en y introduisant celles correspondant à ces erreurs. Cette méthode permet de traiter de façon idéale les erreurs les plus courantes.

Dans Guide, nous utilisons cette méthode pour corriger les erreurs habituelles liées généralement à la modification de la syntaxe du langage entre deux versions. Pour cela, le travail à faire est tout simplement d'ajouter les productions exprimant les erreurs couramment commises ainsi que les actions associées (affichage des messages d'erreurs) dans le source YACC.

d. Corrections globales

Le programme source est considéré comme une liste de symboles lexicaux. Lorsque cette liste n'est pas correcte (elle contient des séquences de symboles qui n'obéissent pas aux règles grammaticales), cette méthode fait le moins de modifications nécessaires pour transformer la liste incorrecte en une liste correcte. En théorie, il existe des algorithmes de correction qui choisissent une séquence minimale de changements pour obtenir une correction globale avec le coût le plus faible ; ces algorithmes sont cependant très chers en temps et en espace de mémoire. En conséquence, ils sont rarement utilisés dans les compilateurs actuels et nous ne les utilisons pas pour le compilateur de Guide.

ANNEXE 4

TYPES ELEMENTAIRES DE GUIDE

Les types élémentaires (types de base) sont constitués d'une part des types primitifs et d'autre part des types structurés.

Les objets dont le type est un type de base (appelés aussi objets de base) ne peuvent pas être permanents, ils sont toujours créés à l'intérieur d'un autre objet et ne sont pas gérés par le système. Il est donc interdit de déclarer des variables de la forme `x : REF Integer ;` un objet de base est donc toujours interne à un autre ou à un objet de travail. A partir des types (et des classes) de base, il n'est pas possible de définir d'autres types (et classes) par le biais des mécanismes de sous-typage et d'héritage.

1. TYPES DE BASE PRIMITIFS

Un objet Guide est toujours construit par composition d'autres objets sauf les objets primitifs dont les types sont décrits ci-dessous. Les méthodes des types primitifs sont définies à l'aide d'opérateurs, ce qui rend la syntaxe d'appel plus simple et similaire à celle de la programmation classique. Cette syntaxe n'est autorisée que pour les types et les classes de base.

Par exemple, on écrira `z := x + y` et non `z := x.+(y)`

1.1 Boolean

```
TYPE Boolean IS
    PREFIX OPERATOR NOT : Boolean ;
    OPERATOR AND (IN Boolean) : Boolean ;
    OPERATOR OR  (IN Boolean) : Boolean ;
    OPERATOR XOR (IN Boolean) : Boolean ;
    OPERATOR = (IN Boolean) : Boolean ;
    OPERATOR # (IN Boolean) : Boolean ;
END Boolean.
```

Deux objets constants de classe `Boolean` sont prédéfinis ; leurs valeurs sont respectivement 'vrai' et 'faux'. Les mots réservés 'TRUE' et 'FALSE' permettent au programmeur de les désigner.

1.2 Char

Le type `Char` permet de déclarer des objets caractères codés en ASCII sur 7 bits. Une constante caractère est désignée entre apostrophes.

```
TYPE Char IS
   OPERATOR = (IN Char) : Boolean ;
   OPERATOR # (IN Char) : Boolean ;
   OPERATOR < (IN Char) : Boolean ;
   OPERATOR > (IN Char) : Boolean ;
   OPERATOR <= (IN Char) : Boolean ;
   OPERATOR >= (IN Char) : Boolean ;
   OPERATOR + (IN Integer) : Char ;
      // successeur selon l'ordre ASCII
   OPERATOR - (IN Integer) : Char ;
      // prédécesseur selon l'ordre ASCII
      // ces 2 opérateurs rendent NIL hors des bornes
END Char.
```

1.3 Integer

Le type `Integer` permet de définir des entiers codés sur 32 bits.

```
TYPE Integer IS
   OPERATOR + (IN Integer) : Integer ; // addition
   OPERATOR - (IN Integer) : Integer ; // soustraction
   OPERATOR POWER (IN Integer) : Integer ; // puissance
   OPERATOR * (IN Integer) : Integer ; // multiplication
   OPERATOR / (IN Integer) : Integer ; // i.e. DIV
   OPERATOR DIV (IN Integer) : Integer ; // division entière
   OPERATOR MOD (IN Integer) : Integer ;
      // reste de la division entière
   OPERATOR = (IN Integer) : Boolean ;
   OPERATOR # (IN Integer) : Boolean ;
   OPERATOR < (IN Integer) : Boolean ;
   OPERATOR > (IN Integer) : Boolean ;
   OPERATOR <= (IN Integer) : Boolean ;
   OPERATOR >= (IN Integer) : Boolean ;
   OPERATOR | (IN Integer) : Integer ;  // opération 'ou' bit à bit
   OPERATOR & (IN Integer) : Boolean ;  // opération 'et' bit à bit
END Integer.
```

1.4 ShortInt

Ce type offre les mêmes opérateurs que le type `Integer`, mais la taille de ses objets est de 16 bits.

1.5 LongInt

Ce type offre les mêmes opérateurs que le type `Integer` et la taille de ses objets est de 32 bits.

1.6 Règles d'affectation entre les types primitifs

Le contrôle de types effectué par le compilateur sur les types élémentaires est basé sur l'identité des types. Cependant, les affectations définies dans le tableau ci-dessous sont autorisées. Par contre, les affectations entre booléens et caractères ou entiers sont interdites. Il est enfin possible d'affecter un caractère (ou un entier) à un élément de chaîne.

type de la variable	type de l'expression	effet de l'affectation
Integer/LongInt/ShortInt	Char	Affectation du code ASCII du caractère à la variable entière
Char	Integer/LongInt/ShortInt	Les 7 bits de poids faible de l'entier définissent le code ASCII de la variable caractère
Integer/LongInt	ShortInt	Affectation des 16 bits du ShortInt à la variable entière, et extension de signe
ShortInt	Integer/LongInt	Troncature des bits de poids fort de l'expression entière
Integer/LongInt	Integer/LongInt	pas de problème car même longueur

2. LES TYPES STRUCTURES

2.1 Array[n] OF ...

Le type structuré `Array` permet de construire des tableaux multidimensionnels de taille fixe. Les paramètres de taille précisent le nombre d'éléments de chaque dimension (trois dimensions au maximum sont autorisées), les bornes étant toujours de 0 à `taille - 1`. La syntaxe de déclaration est :

<désignateur_type_array> =
'Array' '[' <const_entière> { ',' <const_entière> } ']'
OF <désignateur_type>

Les opérations définies sont l'affectation de tableaux et l'accès aux éléments de tableaux en lecture et écriture avec la syntaxe classique sur les tableaux.

Exemple :

```
tab : Array[20,20] OF Integer ;
tab[5,5] := 10;    // opération d'écriture
elem := tab[5,5];  // opération de lecture
```

2.2 Record

Le format de déclaration de type `Record` est :

<désignateur_type_record> =
'Record'
{ <ident> ':' <désignateur_type ';' }*
END

Dans le cas où l'on utilise un type `Record` plusieurs fois, il vaut mieux nommer cette structure en utilisant la clause SYNONYM.

Exemple :

```
SYNONYM livre =  Record
                   titre : String [15] ;
                   auteur : String [10] ;
                   contenu : REF List OF REF Chapitre;
                 END ;
...
x, y : livre;
z : Record
    a : Integer;
    b : String[10];
u, v : livre;
END;
```

Règle d'affectation

Il est possible d'affecter un `Record` à un `Record` si l'affectation champ à champ est correcte du point de vue du contrôle de types : conformité pour les champs de type référence vers un type construit, et égalité pour les champs de type élémentaire.

Remarque : Il est très facile de définir des types et des classes construits qui ressemblent au type `Record` (mais leurs objets sont permanents). En effet, on peut déclarer un type (et la classe correspondante) ne comportant que des variables visibles (pas de méthodes explicites). Les seules opérations définies sur les objets de ce type sont des lectures et des écritures sur ces variables visibles : cette définition correspond bien à celle d'une structure de `Record` classique.

2.3 String[n]

Le type structuré `String[n]` définit l'interface d'accès aux chaînes de caractères de taille fixe. Le paramètre dans les crochets en précise la taille, mais il n'est pas obligatoire dans les paramètres formels des méthodes et des procédures.

Il est possible d'accéder aux caractères d'une variable de type `String` en sélectionnant le rang du caractère dans la chaîne. Les bornes d'une chaîne sont `0` et `n-1` (la longueur n'est pas codée dans la structure).

Les constantes chaînes de caractères sont désignées entre guillemets et ne doivent pas dépasser la taille d'une ligne. Un guillemet à l'intérieur d'une chaîne doit être précédé du caractère \'.

La syntaxe du type `String` est :

```
<désignateur_type_string> =
    'String' [ ' ['<expr_entière_constante> ']' ]

<constante_string> = ' " ' <suite_de_caractères> ' " '
```

La définition de ce constructeur de types est :

```
TYPE CONSTRUCTOR String IS
    OPERATOR = (IN String) : Boolean ;
    OPERATOR # (IN String) : Boolean ;
    OPERATOR < (IN String) : Boolean ;
    OPERATOR > (IN String) : Boolean ;
    OPERATOR <= (IN String) : Boolean ;
    OPERATOR >= (IN String) : Boolean ;
    OPERATOR +  (IN s1, s2 : String) : String;
        // Concaténation de chaînes
 END String.
```

Exemple :

```
string1 : String[20] ;
string2 : String[10] = "toto";
c : Char;
CONST string3 : String[4] = "tata";

c := string2[0];   // premier caractère de string2, c'est-à-dire 't';

METHOD Search (IN nom : String) : REF Element;
```

2.4 Règles d'affectation entre les types structurés

Il est possible d'affecter des tableaux (d'éléments de même type) ou des chaînes de dimensions différentes : si l'affectation a lieu dans le sens du plus grand vers le plus petit, il y a troncature. Un message d'avertissement est affiché par le compilateur lorsque les dimensions ne sont pas égales.

Il est possible d'affecter un `Array OF Char` à une variable de type `String` et vice versa.

ANNEXE 5

BIBLIOTHEQUE DES TYPES GUIDE

Dans cette annexe, nous présentons les différents types construits Guide définis dans la bibliothèque actuellement disponible et qui sont accessibles par le programmeur. Nous donnerons pour chaque type une description des méthodes accessibles à l'utilisateur.

1. Top

Le type `Top` est le super-type implicite de tous les autres, il définit les méthodes applicables à n'importe quel objet dans le système.

```
TYPE Top IS
    SYNONYM Predicat = Record
        caller : Top ;
        methode : String ;
    END;

    METHOD InitOnNew;
    METHOD ObjectSize : Integer;
    METHOD ObjectTypeName : String;
    METHOD ObjectClassName : String;
    METHOD ObjectTypeRef : REF Top;
    METHOD ObjectClassRef : REF Top;
    // les deux méthodes qui suivent sont utilisées pour la mise au point
    METHOD PrintState;
    METHOD PrintCounter;
    // les cinq méthodes suivantes sont utilisées par le ramasse-miettes
    METHOD GetRef : Array [500] OF REF Top;
    METHOD Destroy;
    METHOD IncrementeCompter;
    METHOD DecrementeCompterNonRecursif;
    METHOD DecrementeCompterRecursif;

END Top.
```

2 Activity

Le type `Activity` définit l'interface des activités Guide accessible aux utilisateurs. La variable constante d'environnement `MyActivity` contient en permanence une référence à l'activité qui exécute le code de la méthode en question.

```
TYPE Activity IS
    METHOD Kill;
        // destruction de l'activité
```

```
    METHOD  Suspend;
        // arrêt de l'activité
    METHOD  Resume;
        // réveil de l'activité
    METHOD  Status  :  Boolean;
        // état de l'activité. Si TRUE, l'activité est active ;
        // FALSE sinon.
END  Activity.
```

3. Domain

Le type `Domain` permet de rendre visible dans un programme Guide, certaines primitives de la machine virtuelle. Cela permet notamment de détruire son propre domaine (fin d'exécution de son application) par l'instruction :

```
MyDomain.Kill ;
```

où `MyDomain` est la variable constante d'environnement qui contient en permanence une référence au domaine en question.

```
TYPE  Domain  IS
    chanIn  :  REF  ChanIn;
    chanOut  :  REF  ChanOut;

    METHOD  Kill;
        // destruction du domaine
    METHOD  Start(IN  block  :  REF  Block);
        /* création de l'activité initiale du domaine.
             Celle-ci a comme canaux d'entrée/sortie les canaux
             d'entrée / sortie de l'activité qui exécute la méthode start.
             L'activité principale créée exécute le bloc passé en
             paramètre */
    METHOD  Suspend;
        // arrêt du domaine
    METHOD  Resume;
        // réveil du domaine
    METHOD  Status  :  Boolean;
        // état du domaine. Si TRUE, le domaine est actif ;
        // FALSE sinon
END  Domain.
```

4. Catal

Le type `Catal` et la référence constante `catal` permettent d'accéder au catalogue des objets : ce catalogue a pour fonction d'associer à un objet permanent un nom symbolique fourni par le programmeur, de durée de vie indépendante des programmes.

```
TYPE Catal IS

   METHOD ChangeOwner(IN newownername, localname : String) : Boolean;
 // changer le propriétaire d'un objet ou d'un répertoire

   METHOD CreateDir (IN localname, ownername : String;
            OUT dir : REF Catal) : Boolean;
 // créer un nouveau répertoire dans le répertoire courant

   METHOD Delete(IN localname : String; OUT object : REF Top)
          : Boolean;
   METHOD Delete(IN localname : String; OUT object : REF Top);
   METHOD Delete(IN localname : String);
 // détruire l'association entre un objet et un nom symbolique

   METHOD DeleteDir(IN localname : String; OUT dir : REF Catal)
            : Boolean;
 // détruire le répertoire de nom "localname" dans le répertoire courant

   METHOD Find(IN object : REF Top) : String;
   METHOD Find(IN object : REF Top; OUT name : String) : Boolean;
   METHOD RecursiveFind(IN object : REF Top; pathname : String;
            pathtable : Array[10] OF Catal): String;
 // retrouver le nom symbolique de l'objet dont la référence est passée
 // en paramètre

   METHOD Init(IN dad, slash : REF Catal);
 // initialiser ou modifier le père et le répertoire du catalogue

   METHOD Insert(IN localname : String; object : REF Top);
   METHOD Insert(IN localname : String; object : REF Top) : Boolean;
   METHOD Insert(IN localname : String; owner : String;
            object : REF Top);
   METHOD Insert(IN localname : String; owner : String;
            object : REF Top) : Boolean;
 // insérer une nouvelle entrée dans le catalogue

   METHOD List;
   METHOD List(IN filter : String);
 // lister le contenu du répertoire courant
 // le filtre permet d'en sélectionner un sous-ensemble

   METHOD SearchClass(IN localname : String) : REF Top;
 // rechercher la référence de la classe dont le nom symbolique est passé
 // en paramètre

   METHOD Search(IN localname : String) : REF Top;
 // rechercher la référence de l'objet dont le nom symbolique est passé
 // en paramètre
```

```
    METHOD  RecursiveSearch(IN  localname,  pathname  :  String  ;
                pathtable  :  Array[10]  OF  REF  Catal);
  // lister tous les objets d'une arborescence qui ont le même nom symbolique

    METHOD  SearchDir(IN  localname  :  String)  :  REF  Catal;
  // rechercher la référence d'un répertoire local dont le nom symbolique
  // est passé en paramètre.

END  Catal.
```

5. Chan

Le type Chan permet d'accéder aux entrées/sorties du système depuis les programmes écrits en Guide. Les méthodes définissent les accès d'entrées/sorties des types élémentaires. Suivant l'implémentation du type Chan choisie, certaines des méthodes suivantes n'ont aucun effet.

```
TYPE  Chan  IS
  /* Les méthodes d'initialisation renvoient le signal NOT_EXIST, si le fichier
     n'existe pas */
    METHOD  Init  (IN  fichier  :  String);
  // ouvre le fichier spécifié en lecture/écriture
    METHOD  Init(IN  fichier  :  String;  beginRead,  beginWrite  :  Boolean);
  // ouvre le fichier spécifié selon le mode suivant :
  // si beginRead = TRUE, le pointeur de lecture est positionné au début.
  // A la fin sinon.
  // si beginWrite = TRUE, le pointeur d'écriture est positionné au début.
  // A la fin sinon.

  /* toutes les méthodes de lecture renvoient le signal EOF, si l'on est en
     fin de fichier */
    METHOD  ReadInteger  :  Integer;  SIGNALS  EOF;
  // lire le prochain entier rencontré sous forme décimale
    METHOD  ReadString  :  String;  SIGNALS  EOF;
  /* lire la prochaine chaîne de caractères (jusqu'au caractère '\n').
     Celui-ci n'est pas délivré */
    METHOD  ReadChar  :  Char;  SIGNALS  EOF;
  // lire le prochain caractère rencontré
    METHOD  ReadBool  :  Boolean;  SIGNALS  EOF;
  // lire le prochain booléen rencontré. On renvoie TRUE si la chaine
  // de caractères rencontrée est égale "TRUE" ou "true" ; FALSE sinon
    METHOD  Sync;
  // vider les buffers d'entrée et de sortie
    METHOD  Clear;
     effacer l'écran si celui-ci est compatible vt100
    METHOD  Goto  (  IN  Integer;  Integer  );
    positionner le curseur à la ligne i et la colonne j si l'écran est
```

```
    compatible vt100 */

   METHOD WriteRef( IN r : REF Top );
  // écrire une reference sous la forme : <numéro de container>.<oid>
   METHOD WriteInteger( IN i : Integer );
  // écrire un entier sous forme décimale
   METHOD WriteString( IN s : String );
  // écrire une chaine de caractère
   METHOD WriteChar( IN c : Char );
  // écrire un caractère
   METHOD WriteBool( IN b : Boolean );
  // écrire un booléen.

END Chan.
```

6. ChanIn

Le type `ChanIn` est une restriction du type `Chan`, ne comportant que les méthodes d'entrée. La variable d'environnement `input` est déclarée de ce type.

```
TYPE ChanIn IS
  /* toutes les méthodes de lecture renvoient le signal EOF, si l'on est en
     fin de fichier */
   METHOD ReadInteger : Integer;
  // lire le prochain entier rencontré sous forme décimale.
   METHOD ReadString : String;
  /* lire la prochaine chaine de caractères (jusqu'au caractère '\n').
    Celui-ci n'est pas délivré. */
   METHOD ReadChar : Char;
  // lire le prochain caractère rencontré
   METHOD ReadBool : Boolean;
  // lire le prochain booléen rencontré. On renvoie TRUE si la chaine
  // de caractères rencontrée est égale "TRUE" ou "true" ; FALSE sinon.
END ChanIn.
```

7. ChanOut

Le type `ChanOut` est une restriction du type `Chan`, ne comportant que les méthodes de sortie. La variable d'environnement `output` est déclarée de ce type.

```
TYPE ChanOut IS
   METHOD Sync;
  // vider les buffers d'entrée et de sortie
   METHOD Clear;
  // effacer l'écran si celui-ci est compatible vt100
   METHOD Goto ( IN Integer; Integer );
  /* positionner le curseur à la ligne i et la colonne j si l'écran est
     compatible vt100 */
```

```
    METHOD WriteRef( IN r : REF Top );
   // écrire une référence sous la forme : <numéro de container>.<oid>
    METHOD WriteInteger( IN i : Integer );
   // écrire un entier sous forme décimale
    METHOD WriteString( IN s : String );
   // écrire une chaine de caractère
    METHOD WriteChar( IN c : Char );
   // écrire un caractère
    METHOD WriteBool( IN b : Boolean );
   // écrire un booléen.
END  ChanOut.
```

8. Class

Le type `Class` définit l'interface des objets 'classe exécutable'.

```
TYPE  Class  IS
  /* toutes les méthodes renvoient le signal NOT_EXIST si la classe spécifiée
     n'existe pas */
    METHOD New : REF Top; SIGNALS NOT_EXIST;
  /* créer un objet de la classe spécifiée sur le site d'exécution de
     l'activité */
    METHOD New (IN site : String) : REF Top; SIGNALS NOT_EXIST;
  /* créer un objet de la classe spécifiée sur le site spécifié */
    METHOD New (IN site : String; taille : Integer) : REF Top;
     SIGNALS  NOT_EXIST;
  /* créer un objet de la classe spécifiée sur le site spécifié et de la taille
     donnée (utilisé essentiellement par le compilateur). Si la taille spécifiée
     est inférieure à la taille d'une instance de la classe on prend la taille
     de la classe */
    METHOD New (IN site : String; taille : Integer; flag : Integer)
     : REF Top; SIGNALS NOT_EXIST;
  /* créer un objet de la classe spécifiée sur le site spécifié, de la taille
     donnée (utilisé essentiellement par le compilateur) avec les
     caractéristiques spécifiées. Si la taille spécifiée est inférieure à la
     taille d'une instance de la classe on prend la taille de la classe */
    METHOD NewOnSite (IN site : String) : REF Top; SIGNALS NOT_EXIST;
  /* créer un objet de la classe spécifiée sur le site spécifié */
    METHOD NewWithSize (IN taille : Integer) : REF Top;
     SIGNALS  NOT_EXIST;
  /* créer un objet de la classe spécifiée de la taille donnée (utilisé
     essentiellement par le compilateur). Si la taille spécifiée est inférieure
     à la taille d'une instance de la classe, on prend la taille de la classe */
    METHOD NewWithFlag (IN flag : Integer) : REF Top;
     SIGNALS  NOT_EXIST;
  /* créer un objet de la classe spécifiée avec les caractéristiques spécifiéee */
END  Class.
```

BIBLIOGRAPHIE

Liste des références sur les langages et les systèmes répartis

[Aho 76] Aho A. V., Sethi R., Ullman F. D., "Compilers, Principles, Techniques, and Tools", Addison-Wesley Publishing Company, 1986

[Allchin 83] Allchin J.E., McKendry M.S., "Synchronization and recovery of actions", Proceedings Second Symposium on principles of distributed computing (ACM SIGACT/SIGOPS), Montreal, August 1983.

[Black 86a] Black A.P., Hutchinson N., Jul E., Levy H., Carter L., "Distribution and abstract types in Emerald", IEEE Transactions on Software Engineering, Volume SE-12, December 1986.

[Black 86b] Black A.P., Hutchinson N., Jul E., Levy H., "Object Structure in the Emerald System", Proceedings ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, SIGPLAN Notices, Volume 21, N° 11, November 1986, pp. 78-86.

[Black 88] Black A.P., Hutchinson N., Jul E., Levy H., "Fine-Grained Mobility in the Emerald System", ACM Transactions on Computer Systems, Volume 6, N° 1, February 1988, pp. 109-133.

[Brownbridge 82] Brownbridge D.R., Marshall L.F, Randell B., "The Newcastle Connection - or UNIXes of the World Unite", Software Practice and Experience, Volume 12, December 1982, pp. 78-86.

[Cohen 74] Cohen E., Wulf Wm. A., Corwin W., Jones A., Levin R., Pierson C. and Pollack F., "Hydra: The Kernel of a Multiprocessing Operating System", Communications of the Association for Computing Machinery, Volume 17, N° 6, June 1974, pp. 337-345.

[Donahue 85] Donahue J., "Integration mechanisms in Cedar", Proceeding ACM Symposium on Language issues in programming environments, SIGPLAN Notices, Volume 20, N° 7, July 1985, pp. 245-251.

[Decouchant 88a] Decouchant D., Krakowiak S., Meysembourg M., Riveill M., Rousset de Pina X., "A synchronization mechanism for typed objects in a distributed system", Workshop on Object-Oriented Concurrent Programming, San Diego, September 1988.

[Decouchant 88b] Decouchant D., Duda A., Freyssinet A., Paire E., Riveill M., Rousset de Pina X., Vandôme G., "Guide: an implementation of the Comandos object-oriented architecture on Unix", Proceedings EUUG Conference, Lisbonne, October 1988.

[Dijkstra 80] Dijkstra E. W., Scholten C.S., "Termination Detection for Diffusing Computation", Information Processing Letters, Volume 11, N° 1, August 1980.

[Freyssinet 90] Freyssinet A., "Architecture et Réalisation d'un système réparti à objets", Thèse de Docteur de l'INP de Grenoble, 1991 (à paraître).

[Goldberg 85] Goldberg A., "Smalltalk-80: The Language and its Implementation", Addison Wesley, Reading (Mass.), 1985.

[Krakowiak 90] Krakowiak S., Meysembourg M., Nguyen Van H., Riveill M., Roisin C., Rousset de Pina X., "Design and Implementation of an object-oriented, strongly typed language for distributed applications", "JOOP", Volume 3, N° 3, October 1990, pp. 11-22.

[Leach 83] Leach P.J., Levine P.H., Douros B.P., Hamilton J.A., Nelson D.L., Stumpf B.L., "The architecture of an integrated local network", IEEE Journal on Selected Areas in Communication, November 1983, pp. 842-856.

[Meyer 88] Meyer B., "Object-oriented Software Construction", Prentice Hall, 1988.

[Meysembourg 89] Meysembourg M., "Modèle et Langage à objets pour la programmation d'applications réparties", Thèse de Docteur de l'INP de Grenoble, juillet 1989.

[Misra 83] Misra J., "Detecting Termination of Distributed Computation using Markers", Proceedings of the 2nd annual ACM Symposium on Principles of DC, Montréal, August 1983, pp. 290-294.

[Organick 72] Organick E. I., "The Multics system : an examination of its structure", MIT Press, 1972.

[Popek 85] Popek G. J. and Walker B. J., "The LOCUS Distributed System Architecture", MIT Press, Cambridge Massachusetts, 1985.

[Raynal 85] Raynal M., "Algorithmes distribués et Protocoles", EYROLLES, 1985.

[Robert 77] Robert P., Verjus J.-P., "Toward autonomous descriptions of synchronization modules", Proceedings IFIP. Congress (B. Gilchrist, ed.), North-Holland, 1977, pp. 981-986.

[Rozier 88] Rozier M., Abrossimov V., Armand F., Boule I., Gien M., Guillemont M., Herrmann F., Kaiser C., Langlois S., Leonard P. and Neuhauser W., "Chorus Distributed Operating Systems", USENIX Computing Systems, Volume 1, N° 4, Fall 1988.

[Sandberg 86] Sandberg R., "The Sun Network File System: design, implementation and experience", European Unix Systems Users Group (EUUG) Spring Conference Proceedings, Florence, April 1986.

[Schaffert 85] Schaffert C., Cooper T., Carrie W., "Trellis Object-Based Environment, Language Reference Manual", DEC-TR-372, v1.1, November 1985.

[Scioville 89] Scioville R., "Gestion des informations persistantes dans un système réparti à objets", Thèse de Docteur de l'Université J. Fourier de Grenoble, 1989.

[Stroustrup 86] Stroustrup B., "The C++ programming langage", Addison Wesley, March 1986.

[Wirth 85] Wirth N., "Programming in Modula-2", Springer-Verlag, 1982 ; 3rd ed. (1985).

[Tanenbaum 86] Tanenbaum A. S., Mullender S. J., "The design of a Capability-Based Distributed Operating System", The Computer Journal, Volume 29, N° 4, March 1986, pp. 289-300.

Liste des notes et des Rapports Guide (référencés dans cette thèse)

[Guide-R1] Balter R., Krakowiak S., Meysembourg M., Roisin C., Rousset de Pina X., Scioville R., Vandôme G., "Principes de conception du système d'exploitation réparti Guide", Rapport de recherche Guide n°1, Bull-LGI, (avril 1987). Version préliminaire dans BIGRE+Globule, n°52 (décembre 1986).

[Guide-R2] S. Krakowiak, M. Meysembourg, M. Riveill, C. Roisin, "Modèle d'objets et langage du système Guide", (novembre 1987). Version d'une communication aux Journées Francophones sur l'Informatique, Genève, (janvier 1988).

[Guide-R3] Balter R., Bernadat J., Rousset de Pina X., "Modèle d'exécution du système Guide", Rapport de recherche Guide n°3, Bull-LGI, (décembre 1987).

[Guide-R4] Freyssinet A., Scioville R., Vandôme G., "Gestion des objets dans le système Guide", Rapport de recherche Guide n°4, Bull-LGI, (juin 1988).

[Guide-R5] Decouchant D., Rousset de Pina X., "Principes de réalisation du noyau d'exécution de Guide sur Unix system", (juin 1988).

[Guide-R6] Freyssinet A., Scioville R., Vandôme G., "Réalisation de la mémoire permanente d'objets du système Guide", (juin 1988).

[Lacourte 90] Lacourte S., "Le modèle d'exception de Guide et son implémentation", Note interne Guide, n°. 61, Juillet 1990.

[Roisin 90] Roisin C., Riveill M., Nguyen Van H., "Manuel du langage Guide", Rapport Bull-IMAG 3-90.

Liste des références sur les ramasse-miettes centralisés

[Appel 87] Appel A. W., "Garbage Collection can be faster stack allocation", Information Processing Letters, Volume 25, N° 4, June 1987, pp. 275-279.

[Baker 78] Baker H. G., "List Processing in real-time on Serial Computer", Communications ACM, Volume 21, N° 4, April 1978, pp.280-294.

[Ben-Ari 84] Ben-Ari M., "Algorithms for on-the-fly garbage collection." ACM Transactions on Programming Languages and Systems , Volume 6(3), July 1984, pp. 333- 344.

[Cohen 81] Cohen J., "Garbage Collection of Linked Data Structures." Computing Surveys, Volume 13, N° 3, September 1981, pp. 341-367.

[Ungar 84] Ungar D., "Generation Scavenging : A Non-Discruptive High Performance Storage Reclamation Algorithm." Proceedings of the ACM Symposium on Practical Software Development Environments, Pittsburgh, PA, April 1984, pp. 157-167.

[Ungar 88] Ungar D., Jackson F., "Tenuring Policies for Generation-Based Storage Reclamation", OOPSLA'88 Proceedings, September 1988, pp. 1-17.

[Deutsch 76] Deutsch L. P., Bobrow D. G., "An Efficient, Incremental, Automatic Garbage Collector", Communications of ACM, Volume 19, N° 9, September 1976, pp. 522-526.

[Deutsch 84] Deutsch L. P., Schiffman A. M., "Efficient Implementation of the Smalltalk-80 System", Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages, Salt Lake City, Utah, January 1984.

[Dijkstra 78] Dijkstra E.W, Lamport L., Martin A.J., Scholten C.S., Steffens E.F.M., "On-the-fly Garbage Collection : an exercise in cooperation." Communications of ACM, Volume 21, N° 11, November 1978, pp. 966-975

[Ellis 88] Ellis R., Kai Li, Appel A. W., "Real-time Concurrent Collection on Stock Multiprocessors", Systems Research Center, February 1988.

[Lieberman 83] Lieberman H., Hewitt C., "A Real-Time Garbage Collector Based on the Lifetimes of Objects." Communications of ACM, Volume 26, N° 6, June 1983, pp. 419-429.

[Kung 77] Kung H.T. & Song S.W., "An Efficient Parallel Garbage Collection System and its correctness proof." Proceedings of the IEEE Symposium on Foundations of Computer Science (Providence, R.I.). IEEE, New York, 1977, pp. 120-131.

[Morris 78] Morris F. L., "A Time- and Space- Efficient Garbage Compaction Algorithm." Communications of ACM, Volume 21, N° 8, August 1978, pp. 662-665.

[Stamos 82] Stamos J. W., "A Large Object-Oriented Virtual Memory: Grouping, Measurements, and Performance" Xerox technical report, SCG-82-2, Xerox, PARC, Paolo Alto, CA, May 1982.

Liste des références sur les ramasse-miettes répartis

[Ali 85] Ali K. M., Haridi S., "Global Garbage Collection for Distributed Heap Storage Systems", IBM Research Report RC 11082 (#49769), April 1985.

[Almes 82] Almes G. T., "Garbage Collection in Object-Oriented System", Thesis of Doctor of Philosophy, Carnegie-Mellon University, June 1980.

[Augusteijn 87] Augusteijn L., "Garbage Collection in a Distributed Environment", Parle LNCS, N° 259, June 1987, pp. 75-93.

[Beckerle 86] Beckerle M. J., Ekanadham K., "Distributed Garbage Collection with no Global synchronization", IBM Research Report RC 11667 (#52377), January 1986.

[Bevan 87] Bevan D. I., "Distributed Garbage Collection Using Reference Counting", LNCS N°259, Parallele Architectures and Languages Europe Volume II, Eindhoven, Springer Verlag, June 1987, pp. 176-187.

[Couvert 89] Couvert A., Maddi A., Pedrono R., "Partage d'objets dans les systèmes distribués - Principes des ramasse-miettes", Rapports de Recherche, INRIA, janvier 1989.

[Hudak 82] Hudak P. and Keller R. M., "Garbage Collection and Task Deletion in Distributed Applicative" Processing Systems, Conf. Record 1982 ACM Symp. on LISP and Functional Programming, ACM, 1982, pp.168-178.

[Hughes 85] Hughes J., "A distributed Garbage Collection Algorithm", LNCS n° 201, Functional Programming Languages and Computer Architecture, Nancy, France, September 1985, Springer Verlag, pp. 256-272.

[Lermen 86] Lermen C. W., Maurer D., "A protocol for distributed reference counting", Proceedings of the ACM Conference on LISP and

Functional Programming, Cambridge, Massachusetts, August 1986, pp. 364-372.

[Pixley 88] Pixley C., "An Incremental Garbage Collection Algorithm for multi-mutator systems", Distributed Computing, N° 3, 1988, pp. 41-50.

[Rudalics 86] Rudalics M., "Distributed Copying Garbage Collection", Proceedings of the ACM Conference on LISP and Functional Programming, Cambridge, Massachusetts, August 1986, pp. 364-372.

[Vestal 87] Vestal S. C., "Garbage Collection : An Exercise in Distributed, Fault-Tolerant Programming", Technical Report 87-01-03, January 1987, Departement of Computer Science, University of Washington.

[Tel 87] Tel G., Tan R. B., Van Leeuwen J., "The derivation of the on-the-fly garbage algorithms from distributed termination detection protocols", LNCS n°247, 4th annual Symposium on Theoretical Aspects of Computer Science, Passau, Federal Republic of Germany, February 1987, pp. 445-455.

COMPILATION ET ENVIRONNEMENT D'EXECUTION D'UN LANGAGE A OBJETS

RESUME

Cette thèse a été effectuée dans le cadre du projet Guide mené par le Laboratoire BULL-IMAG/Systèmes depuis mi 1986. Guide est le support d'un ensemble de recherches sur la programmation des applications réparties. Ces recherches sont entreprises sur la base du développement d'un système d'exploitation réparti à objets Guide qui fournit un haut niveau d'intégration (invisibilité de la répartition notamment). Le langage à objets Guide défini spécifiquement permet la programmation et la mise en œuvre d'applications réparties le plus facilement possible en utilisant tous les éléments du modèle d'objets et du système Guide. La thèse présente les principes de compilation des sources Guide en objets exécutables. Les objets sont créés dynamiquement lors de l'exécution des applications et ils restent permanent même dans le cas où ils ne sont plus utilisables (on ne peut pas y accéder). Donc, un ramasse-miettes doit être présent pour ramasser automatiquement ces miettes tout en garantissant la cohérence de la mémoire d'objets. La thèse présente aussi notre étude sur les familles d'algorithmes de ramasse-miettes existant ainsi que leur adaptation pour le système Guide.

ABSTRACT

This thesis was prepared within the Guide project, designed and implemented by BULL-IMAG/System Laboratory at Grenoble since summer 1986. The Guide project is the framework for research in the area of distributed applications and is based on the development of an experimental distributed object-based operating system. It provides a high degree of integration, and especially location transparency. The Guide object-based language defined specifically allows distributed applications to be written and executed easily using all elements of the model and the system. This thesis presents compiling principles of Guide sources in executable objects. Objects are dynamically created while executing applications and they remain permanent. Therefore, a garbage-collector must be present to automatically gather garbages. This thesis presents also our study of garbage-collector algorithm families and their adaptation for the Guide system.

MOTS-CLES

objet, type, classe, persistance, héritage, conformité, répartition, ramasse-miettes