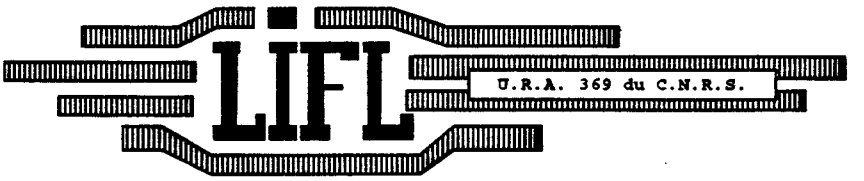
NUMERO D'ORDRE : **978**

ANNEE : 1992



LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

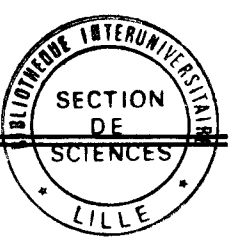
pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

Luc COURTRAI

Les Composants Actifs de Communication: Outils pour la conception et l'implantation de langages parallèles à objets actifs pour machines MIMD



La thèse sera soutenue le 29 Octobre 1992, devant la commission d'examen

Membres du Jury :

Président :	J-P. DELAHAYE	LIFL
Rapporteurs :	L. FERAUD	IRIT
	F. ANDRE	IRISA
	J-M. JEZEQUEL	IRISA
Examinateurs :	D. CAROMEL	SOPHIA ANTIPOLIS
	J-M. GEIB	LIFL
Invités :	M. RIVEILL	BULL-IMAG
	M. MAKPANGOU	INRIA
	J-F. MEHAUT	LIFL

UNIVERSITE DES [illegible]LOGIES DE LILLE
U.F.R. d'I.E.E.A. [illegible] d'Ascq CEDEX
Tél. [illegible] 65.66

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL. FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE. LELONG. LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU. ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PARREAU, J. LOMBARD, M. MIGEON. J. CORTOIS. A.DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphériques

M. MIGEON Michel	EUDIL
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel	Informatique
M. LAVEINE Jean Pierre	Paléontologie
Mme LECLERCQ Ginette	Catalyse
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHENAFF René	Géographie
M. LHOMME Jean	Chimie organique biologique
M. LOUAGE Françis	Electronique
M. LOUCHEUX Claude	Chimie-Physique
M. LUCQUIN Michel	Chimie physique
M. MAILLET Pierre	Sciences Economiques
M. MAROUF Nadir	Sociologie
M. MICHEAU Pierre	Mécanique des fluides
M. PAQUET Jacques	Géologie générale
M. PASZKOWSKI Stéfan	Mathématiques
M. PETIT Francis	Chimie organique
M. PORCHET Maurice	Biologie animale
M. POUZET Pierre	Modélisation - calcul scientifique
M. POVY Lucien	Automatique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. RAMAN Jean Pierre	Sciences de gestion
M. SALMER Georges	Electronique
M. SCHAMPS Joël	Spectroscopie moléculaire
Mme SCHWARZBACH Yvette	Géométrie
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. THIEBAULT François	Sciences de la Terre
M. THOMAS Jean Claude	Géométrie - Topologie
M. THUMERELLE Pierre	Démographie - Géographie humaine
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. TREANTON Jean René	Sociologie du travail
M. TURRELL Georges	Spectrochimie infrarouge et raman
M. VANEECLOO Nicolas	Sciences Economiques
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. VIDAL Pierre	Automatique
M. WALLART Françis	Spectrochimie infrarouge et raman
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	
M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BALL Steven	Génétique
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BAUSIERE Robert	Systèmes électroniques
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERNAGE Pascal	Physique atomique, moléculaire et du rayonnement
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Sciences Economiques
M. BERZIN Robert	Analyse
M. BISKUPSKI Gérard	Physique de l'état condensé et cristallographie
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOHIN Jean Pierre	Biochimie métabolique et cellulaire
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUCHER Daniel	Physique
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BROCARD Jacques	Chimie
Mme BROUSMICHE Claudine	Paléontologie
M. BUISINE Daniel	Mécanique
M. CAPURON Alfred	Biologie animale
M. CARRE François	Géographie humaine
M. CATTEAU Jean Pierre	Chimie organique
M. CAYATTE Jean Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie structurale
M. CHIVE Maurice	Composants électroniques optiques
M. COMYN Gérard	Informatique théorique
Mme CONSTANT Monique	Composants électroniques et optiques
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique géodynamique
M. CURGY Jean Jacques	Biologie
M. DANGOISSE Didier	Physique théorique
M. DE PARIS Jean Claude	Analyse
M. DECOSTER Didier	Composants électroniques et optiques
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean Paul	Informatique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
Mme DEMUYNCK Claire	Physique atomique, moléculaire et du rayonnement
M. DENEL Jacques	Informatique
M. DEPREZ Gilbert	Physique du solide - christallographie

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation,calcul scientifique,statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique,moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation,calcul scientifique,statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO Françis	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarette	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
M. GOURIEROUX Christian	Probabilités et statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie organique
M. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

Remerciements

Avant d'entamer les traditionnels remerciements, je voudrais rendre hommage à **Eric DELATTRE**, Professeur à l'Université de Lille I, décédé subitement en janvier de l'année 1991. Il m'a accueilli dans son équipe et n'a cessé de m'encourager pendant ma première année de thèse. Il a été l'instigateur des travaux de cette thèse, mais aussi ceux de l'équipe.

Je remercie les membres du jury :

- Monsieur **Jean-Paul DELAHAYE**, Professeur à l'Université de Lille I, pour m'avoir fait l'honneur de présider le jury de cette thèse.
- Monsieur **Louis FERAUD**, Professeur à l'IRIT de l'Université de Toulouse, d'avoir rapporté cette thèse et pour l'intérêt qu'il manifeste envers nos travaux.
- Madame **Françoise ANDRE**, Professeur à l'IFSIC - IRISA de Rennes, et Monsieur **Jean-Marc JEZEQUEL** Chargé de Recherches CNRS à l'IRISA, pour avoir corapporté la thèse avec le plus grand soin. Leurs remarques m'ont permis d'améliorer la rédaction de cette thèse.
- **Jean-Marc GEIB**, Maître de Conférences à l'Université de Lille I, pour m'avoir dirigé dans mon travail de thèse et pour son aide importante à la rédaction de ce document.
- **Jean-François MEHAUT**, Maître de Conférences à l'Université de Lille I, pour sa contribution essentielle à cette rédaction.
- Monsieur **Denis CAROMEL**, Maître de Conférences à l'Université de NICE SOPHIA ANTIPOLIS, d'avoir accepté d'examiner cette thèse.
- Monsieur **Michel RIVEILL**, Chercheur à BULL- IMAG, de participer au jury et pour sa lecture critique et constructive du manuscrit.
- Monsieur **Mesaac MAPANGOU**, Chercheur à l'INRIA, pour sa participation au Jury et pour son intérêt à ce travail.

Je remercie de nouveau et plus particulièrement :

- **Jean-Marc GEIB** pour son aide durant ces années de thèse et d'avoir accepté de diriger mes recherches, ceci dans des circonstances difficiles. De plus, les nombreuses discussions que nous avons eues, ont été source d'idées et d'inspiration et m'ont permis de mener à terme ce travail.
- **Jean-François MEHAUT** pour son soutien moral durant l'ensemble de mes travaux et pour ses conseils tant sur un plan technique que conceptuel. Son expérience dans le domaine des systèmes répartis et son aide à la conception et à la réalisation des différents prototypes ont été déterminants. D'autre part, ses qualités humaines et son amitié m'ont aidé tout au long de la thèse.

Je tiens également à remercier :

- **Jean-François ROOS** pour sa participation à la réalisation des prototypes. Nos houleuses discussions ont contribué à l'avancement et à la qualité des outils proposés; tout ceci dans un climat d'amitié.
- Les autres membres de l'équipe et plus particulièrement **Fred HEMERY, Christophe GRANSART, Raymond NAMYST, Cédric DUMOULIN, Chry(i)stel(le) GRENOT, Philippe MERLE** et **Lenneke DEKKER**, pour l'ambiance nécessaire à un tel travail.
- **Bernard CARRE**, Maître de Conférences à l'Université de Lille I, pour sa lecture critique de la thèse. Ses remarques ont permis une réorganisation positive de ce document.
- Les nombreuses personnes, dont **Chrystel GRENOT**, Monsieur **SAVINA** et **Valérie HUET**, qui ont lu et relu le document, à la chasse aux fautes d'orthographe.
- **Henri GLANC** pour avoir assuré efficacement la reprographie de ce document, mais aussi des multiples versions antérieures.

Je tiens enfin à remercier **Nathalie**, ma future femme, qui m'a toujours soutenu et supporté avec beaucoup de patience, tout au long de ces dernières années et ceci malgré les périodes d'incertitude. Je lui dois beaucoup. **Ce travail lui est dédié.**

Table des matières

Chapitre -III- Implantations d'environnements d'Objets Actifs __ 107

Liste des figures

Introduction

Environnement

Le projet Pvc a débuté au cours du premier semestre 1989 au LIFL (Laboratoire Informatique Fondamentale de Lille) à l'initiative d'Eric DELATTRE responsable de l'équipe *système*. L'objectif de ce projet portait sur la conception et la réalisation d'une architecture logicielle support de langages parallèles à objets. Depuis le décès d'Eric DELATTRE, janvier 1991, l'équipe est encadrée par Jean-Marc GEIB secondé par Jean-François MEHAUT, tous deux maîtres de conférence au laboratoire. Quatre thèses, autres que celle-ci, sont actuellement en cours sur le projet Pvc et son environnement. Voici les sujets de ces thèses:

- 1- Fred HEMERY, «Définition et réalisation d'un outil de répartition d'objets sur un multi-ordinateur»
- 2- Jean François ROOS, «Contribution à la définition et à la réalisation d'outils de trace et de "déverminage" pour machines parallèles»
- 3- Jean François COLIN, «Définition de mécanismes de coopération des activités liées à la fragmentation des instances introduites dans l'architecture logicielle Pvc»
- 4- Christophe GRANSART, «Fragmentation d'objets pour machines fortement parallèles»

Problématique

Cette étude est une contribution à la définition et l'implantation de langages parallèles à objets sur «multi-ordinateur». Elle a pour cadre la programmation parallèle au confluent des domaines très actifs et prometteurs que sont: l'exploitation des multi-ordinateurs, la programmation parallèle et les langages à objets.

Les multi-ordinateurs

Les multi-ordinateurs (multicomputer) sont des machines formées de noeuds puissants fortement connectés et qui se caractérisent par l'absence de mémoire commune. Ces machines MIMD (Multiple Instructions - Multiple Data) sont constituées d'un ensemble de processeurs où chaque processeur possède sa propre mémoire locale (généralement de taille importante, plusieurs MOctets). Un processeur et sa mémoire locale constituent un noeud de la machine. Un réseau de communication performant relie les noeuds du multi-ordinateur (type *Crossbar, Benes...*).

Ces machines MIMD peuvent être vues comme des ensembles de composants processeur(s) et mémoire(s).

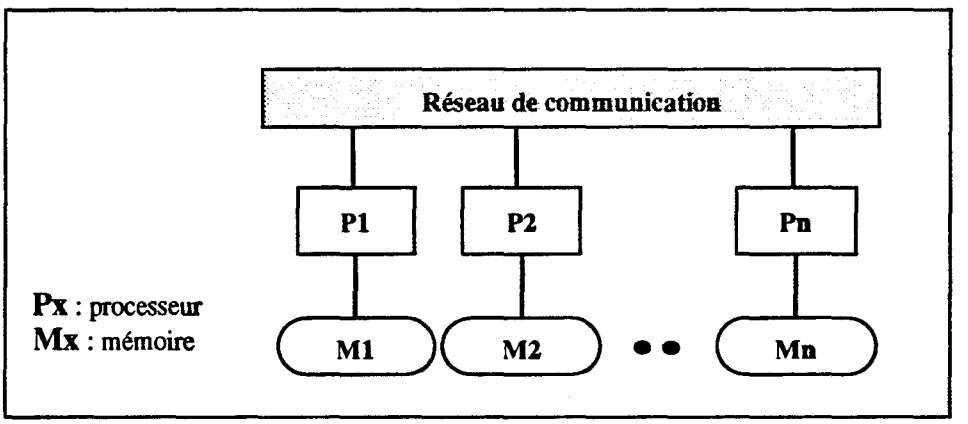


figure 1.0 *Architecture d'un multi-ordinateur*

Le réseau de communication est localisé physiquement dans la machine. Il présente des temps de communication performants (80 Mbytes/s pour un *Transputer* T9000 [T9000-91]). Une durée de transmission de message, de l'ordre de 50 fois le temps d'exécution d'une instruction élémentaire, est envisagée par W. C. Athas [Athas88].

Les multi-ordinateurs sont une réponse prometteuse à la demande en puissance de calcul de plus en plus importante.

La programmation parallèle

Le parallélisme fait l'objet de nombreuses recherches et couvre des domaines très vastes qui vont de la conception de nouvelles architectures matérielles à celui de la programmation et l'utilisation des architectures parallèles. Nos travaux découlent de deux constats:

- 1 - Les machines parallèles restent faiblement exploitées et fortement démunies d'outils.

Le développement de la haute intégration des composants électroniques permet maintenant la réalisation de machines essentiellement MIMD, demandant la prise en charge d'un «parallélisme de tâches» très élevé, obligeant à une distribution des activités et des données et à l'abandon de solutions centralisées toujours plus faciles à mettre en oeuvre. De plus, il faut ici ne pas oublier l'effet d'échelle: les modèles dits distribués (s'intéressant à un ensemble de machines reliées par un réseau local de communication) ne sont pas directement applicables car d'un grain trop important. Il est en effet difficile d'appliquer un modèle d'applications coopérant épisodiquement à une machine privilégiant justement la coopération. Il est aussi difficile d'utiliser un partitionnement large d'applications sur une machine prônant une fragmentation beaucoup plus importante.

Ainsi peu de modèles adaptés et d'outils efficaces existent pour ces machines. *Occam* [INMOS88], et son modèle de processus communiquant, a fait le succès du Transputer, mais comme on l'a dit au paragraphe précédent, il ne permet pas l'abstraction de donnée et reste ainsi très difficile à utiliser à grande échelle. D'autres systèmes n'offrent que des bibliothèques spécialisées à utiliser dans des environnements de programmation classique (*C* par exemple). Ils n'offrent donc aucune méthodologie particulière pour la conception d'applications parallèles. Là encore le développement à grande échelle demeure un travail trop ardu pour répondre aux besoins en logiciels de ces machines.

- 2 - Les applications parallèles sont délicates à programmer.

Les modèles simples, de mise en oeuvre efficace (s'il en existe) restent à découvrir. Le niveau de complexité de la conception parallèle semble beaucoup plus élevé que celui de la conception séquentielle. Les premières techniques utilisées, telles que les sémaphores, verrous etc..., ont permis l'expérimentation dans ce domaine. Elles ont montré aussi leurs limites pour le développement à grande échelle d'applications parallèles. Plus récemment, des modèles de processus communiquant ont ouvert de nouvelles perspectives. La notion centrale de processus est bien mieux appréhendée et les interactions entre processus plus formellement spécifiées. Ces modèles s'intéressent malheureusement peu aux données pour se concentrer sur les activités. Le rôle du concepteur se situe dans la recherche des activités indépendantes et de leurs interactions, mais très faiblement dans l'utilisation des principes d'abstraction et d'encapsulation chers au génie logiciel pour la conception de grands logiciels.

Plus récemment, donc, sont apparus les modèles d'acteurs ou d'objets actifs qui allient la description d'activité parallèle et la structuration en objets du domaine considéré. Ces modèles sont au coeur de la recherche actuelle sur la prise en compte du parallélisme. D'un côté, les modèles d'acteurs offrent un cadre puissant à des études théoriques du parallélisme mais proposent peu d'outils effectifs de programmation à grande échelle. De l'autre côté, les modèles d'objets actifs se veulent ambitieux sur un plan réalisation, mais ne semblent pas être à maturité et ne savent pas «réutiliser» toutes les notions des modèles objets (séquentiels) qu'ils sont sensés justement se

réapproprier. On peut citer l'héritage qui doit ici inclure l'héritage des contraintes de synchronisation et de la spécification. Comment spécifier simplement le comportement d'une entité autonome ?

L'approche Objet

Les langages à objets sont des langages fortement modulaires dont les qualités sont celles prônées par le génie logiciel [Meyer88]. L'approche Objet répond aux problèmes de modélisation du monde réel. L'**objet** est alors l'élément de construction d'une application qui se conçoit uniquement en termes d'objets. Les objets sont eux-mêmes décrits dans des **classes** où sont spécifiées les données locales des objets ainsi que le code permettant de manipuler les objets de cette classe. Les objets d'une classe sont appelés **instances** de la classe. (Les concepts objets sont décrits dans [Wegner87,90]).

Un objet est défini par:

- ses **variables d'instance** (ou **attributs**) qui représentent les données locales de l'objet. Une variable d'instance peut contenir une valeur ou une référence sur un autre objet du système. Chaque objet de la classe possède ses propres variables d'instance. Les variables d'instance sont privées et locales à l'objet.
- les **méthodes** (ou **routines**). Chaque classe d'objets comporte un ensemble de méthodes qui s'appliquent sur les instances de la classe. Une méthode peut manipuler les variables d'instance de l'objet et communiquer avec les autres objets du système. Pour cela, elle déclenche l'exécution d'une méthode sur l'objet cible, objet connu par l'objet de la méthode appelante (mécanisme d'invocation de méthode).

L'exécution d'une application à objets est matérialisée par une suite de déclenchements de méthodes sur les objets de l'application. Chaque invocation de méthode s'exprime par la construction **o.m(args)** qui provoque l'exécution de la méthode de nom *m* sur l'objet de référence *o*.

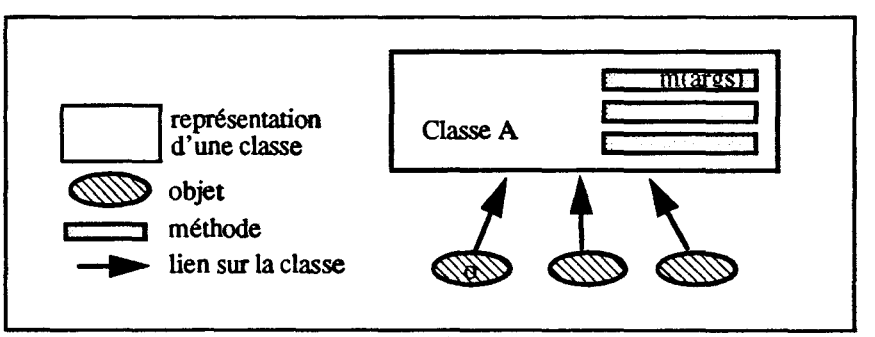


figure 1.1 *Les objets, méthodes et les classes*

Rappelons rapidement que les langages à objets semblent être un élément de réponse à la crise du logiciel (faible productivité des développeurs, «mauvaise» qualité des produits, difficulté de maintenance et d'évolution des programmes).

Le parallélisme et l'approche Objet

L'intersection de ces solutions, l'une logicielle (les langages à objets) et l'autre matérielle (les multi-ordinateurs) sera les langages parallèles à objets pour multi-ordinateurs. L'objectif est d'appliquer les bonnes propriétés de l'approche objet à la programmation parallèle, telle que la modélisation naturelle des applications en termes d'objets.

Parmi ces langages, les langages à objets actifs intègrent dans la structure même de l'objet une activité autonome. Cette propriété parallélise implicitement des applications. Le courant de recherche sur les «objets actifs» [Yonezawa87] montre que parallélisme et approche par objets peuvent être harmonieusement conjugués. Certains de ces langages cherchent à augmenter le parallélisme dans l'objet; pour cela les objets actifs sont multi-programmés et même fragmentés [Banâtre91]. L'objet peut alors traiter plusieurs requêtes simultanément.

Les langages parallèles à objet sont différents des langages séquentiels sur trois points:

- 1 - L'exécution d'une application parallèle à objets crée plusieurs flots d'exécution pris en charge par des processus distincts. La création du parallélisme s'exprime facilement dans les langages de programmation.
- 2 - L'existence de plusieurs flots d'exécution nécessite la définition d'une synchronisation. Son expression définit la synchronisation comportementale des objets de l'application et une synchronisation interne aux objets (pour protéger leurs variables locales des objets multi-programmés).
- 3 - L'implantation de tels langages sur un multi-ordinateur nécessite une stratégie de répartition des éléments issus du langage sur les noeuds de la machine cible. Il faut répartir les objets de l'application et le code défini sur ces objets en exploitant au mieux la machine cible.

La multitude des solutions nous conduit à effectuer un réexamen des concepts proposés pour la définition de langages parallèles à objets; ceci pour juger de leurs adéquations avec les multi-ordinateurs visés.

Il s'agit donc essentiellement de

- la création et l'expression du parallélisme dans le modèle objet,
- la synchronisation des activités parallèles,
- la répartition des éléments du langage à l'exécution.

Sur ces deux derniers points, nous proposons de nouvelles solutions.

Nous proposons un nouveau modèle de gestion du parallélisme massif dans les langages parallèles à objets, basé sur l'implantation même des objets. Notre proposition éclate les objets sur les différents noeuds de la machine. Elle crée un réel parallélisme intra-objet et une meilleure exploitation des possibilités physiques des machines parallèles que sont les multi-ordinateurs. La plupart des langages n'offrent que du parallélisme inter-objet et des objets sérialisés. Le parallélisme intra-objet, et, plus particulièrement celui de données, a été peu étudié jusqu'à récemment [Dally87, Banâtre91, Shapiro89-91].

Réalisation de langages parallèles à objest sur architecture distribuée.

La réalisation de langages parallèles à objets sur un multi-ordinateur se révèle rapidement difficile en utilisant le logiciel de base fourni par le constructeur. Les reproches que l'on puisse faire, outre ceux déjà mentionnés dans la section programmation parallèle, sont liés au fait que ces outils, proches du matériel, sont de trop bas niveau et donc non portables.

Face à ce constat, notre approche se veut très pragmatique. Notre premier objectif a été de définir un environnement de programmation qui :

- 1 - introduise un grain unique de distribution des activités et des données. Le grain devant être assez fin pour s'adapter au contexte des machines MIMD.

- 2 - se fondant sur les caractéristiques générales des machines visées, en cache néanmoins les détails de bas-niveau et permette ainsi la création de plates-formes efficaces de conception d'applications parallèles.

-3 - autorise et de surcroît prône la conception par la combinaison d'entités actives fortement réutilisables.

- 4 - permette à faible coût l'utilisation des modèles existants (comme celui des acteurs ou des objets actifs) pour la construction effective d'applications parallèles.

Nous proposons un outil pour la conception d'applications parallèles sur machine parallèle. Cet environnement doit être suffisamment évolué pour être facilement programmable. Mais il doit aussi être suffisamment proche du logiciel de base de la machine cible pour rester efficace. La proposition possède un parallélisme implicite pour exploiter pleinement les possibilités de la machine cible et l'environnement intègre un outil souple de répartition du code sur les noeuds de la machine cible.

La plate-forme de développement Pvc

L'environnement de développement Pvc est ainsi construit en couches. Tous les prototypes sont réalisés sur deux **architectures** différentes: un multi-ordinateur(MultiCluster II de chez *Parsytec* constitué de 32 *transputers* T800) et un réseau de Sparc-stations (SUN) reliées par *Ethernet*. Au dessus des deux **systèmes d'exploitation** (Hélios et SunOs), nous avons réalisé le run-time des **Composants Actifs de Communication** lui aussi construit en couches. Les **applications Cac** sont développées au dessus de ce support de programmation. Celles ci sont constituées des applications Cac (directement développées à partir des primitives Cac) dont plusieurs implantations d'objets actifs fragmentés et multi-programmés.

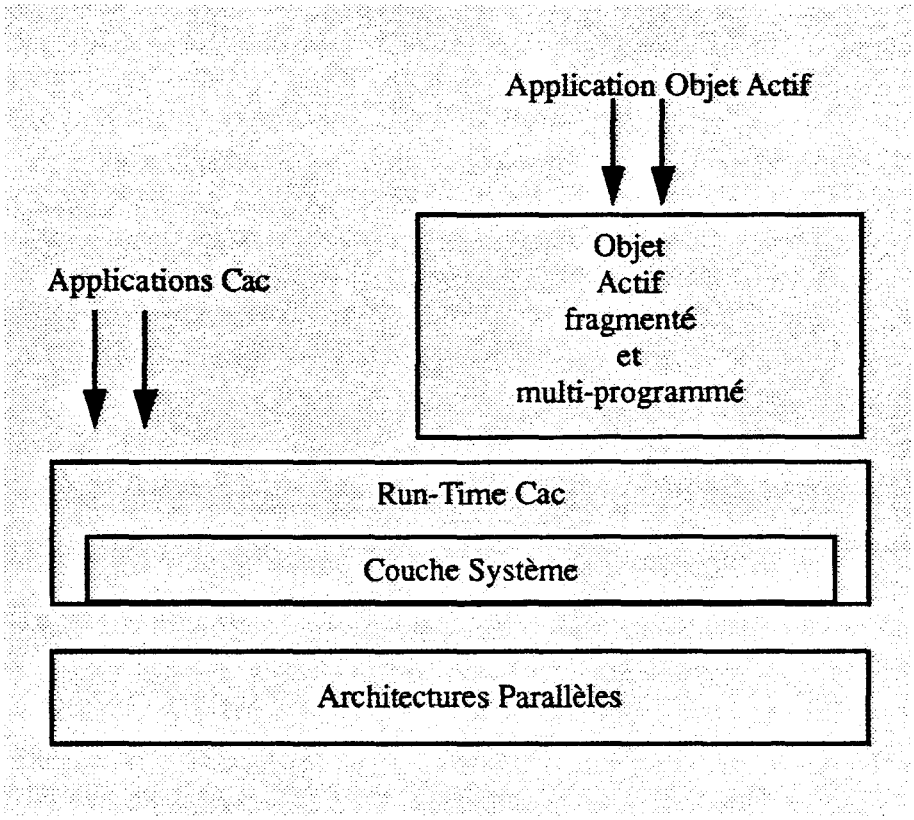


figure 1.2 *L'environnement de développement du projet Pvc*

Plan de la thèse

Cette thèse se découpe en trois chapitres:

- 1 - Un premier chapitre présente un **réexamen des langages parallèles à objets.** Nous verrons successivement: i) le parallélisme et son expression dans les différents langages existants, ii) la synchronisation et son expression dans ces langages parallèles à objets, iii) les modèles de répartition des entités du langage à l'exécution pour exploiter le parallélisme de la machine cible.

- 2 - Le chapitre II décrit notre support de développement. Il présente les **Composants Actifs de Communication**, outils pour la conception d'applications parallèles. Un composant est une structure active qui intègre une activité propre, une boite aux lettres (élément de communication) et un environnement local. La structure d'un Cac répond naturellement aux problèmes rencontrés dans les applications parallèles que sont i) l'expression du parallélisme ii) faire communiquer des processus distants. Les composants évoluent dans les modules qui sont les entités de partage de code du modèle Cac. Le Cac est facilement réalisable au dessus des systèmes d'exploitation. Nous montrerons l'implantation du run-time Cac sur un multi-ordinateur, ainsi que quelques applications. L'environnement des Cac/s sert de plate-forme au développement de langages parallèles à objets.

- 3 - Le troisième chapitre présente plusieurs implantations de langages parallèles à objets au dessus des Composants Actifs de Communication. Une première partie présente une extension des Cac/s vers le modèle d'exécution **d'un modèle Acteur** [Hewitt77, Agha86, Liberman86]. L'extension utilise la notion de classes d'acteurs. Nous décrirons une réalisation d'un modèle d'acteurs sur un multi-ordinateur. Nous proposerons une synchronisation «a priori», reprise du langage *Act* ++ [Kafura89,90]. Cette synchronisation, appliquée à notre modèle d'exécution, prend en compte la localisation de chaque acteur. Dans une seconde partie, nous proposons plusieurs **représentations distribuées d'objets actifs** multi-programmés à partir des Composants Actifs de Communication. La proposition permet à l'exécution d'exploiter un réel parallélisme intra-objet. Ce parallélisme d'exécution est complété par un parallélisme de données lié à l'éclatement des objets. Les représentations distribuées d'objets actifs sont intégrées à un environnement de programmation associé à plusieurs modèles de répartition de code.
- Une première annexe présente **une fragmentation des objets actifs en fonction de l'héritage des classes**. Nous détaillerons les caractéristiques de la proposition et montrerons la faisabilité de la solution sur une machine parallèle.
- Une seconde annexe décrit les **primitives** de l'environnement de programmation des **Composants Actifs de Communication**. Nous détaillerons les différentes primitives de l'interface de programmation de l'environnement des Cac/s.
- Une dernière annexe présente des exemples de programmes définissant des objets ayant des représentations réparties.

Chapitre - I -

Les Langages Parallèles à Objets

L'objectif de nos travaux est la conception et l'implantation de langages parallèles à objets sur un multicomputer. Nous nous intéressons plus particulièrement à leur implantation sur architecture distribuée.

Dans cette étude, nous avons privilégié trois points:

- Le **parallélisme**: Pour implanter des applications parallèles, le modèle d'exécution d'un langage doit intégrer la notion d'entité parallèle. Nous montrerons comment ces entités sont intégrées au modèle d'exécution et comment elles s'expriment dans le langage de programmation. Nous essayerons d'établir une classification des principaux langages parallèles à objets.

- La **synchronisation**: Le parallélisme nécessite une synchronisation des flots d'exécutions. Nous montrerons les différents types de synchronisation et leur expression dans les langages de programmation.

- La **répartition des entités du langage à l'exécution**: Pour exploiter le parallélisme des machines cibles et créer un réel parallélisme dans une application, il faut répartir l'exécution de l'application sur différents processeurs. Nous montrerons les différents modèles d'exécution répartis existant dans les langages à objets et leur réalisation.

Nous présentons le graphisme utilisé dans ce chapitre. Une application issue d'un langage parallèle à objets met en oeuvre un ensemble d'objets (représentés par des ovales) traversés par différents flots d'exécution (les flèches). Ces différents flots matérialisant le *parallélisme* de l'application se synchronisent sur des points de *synchronisation* (disque hachuré). L'ensemble des éléments de l'application doit à l'exécution être *réparti* sur les noeuds (rectangle) de la machine cible.

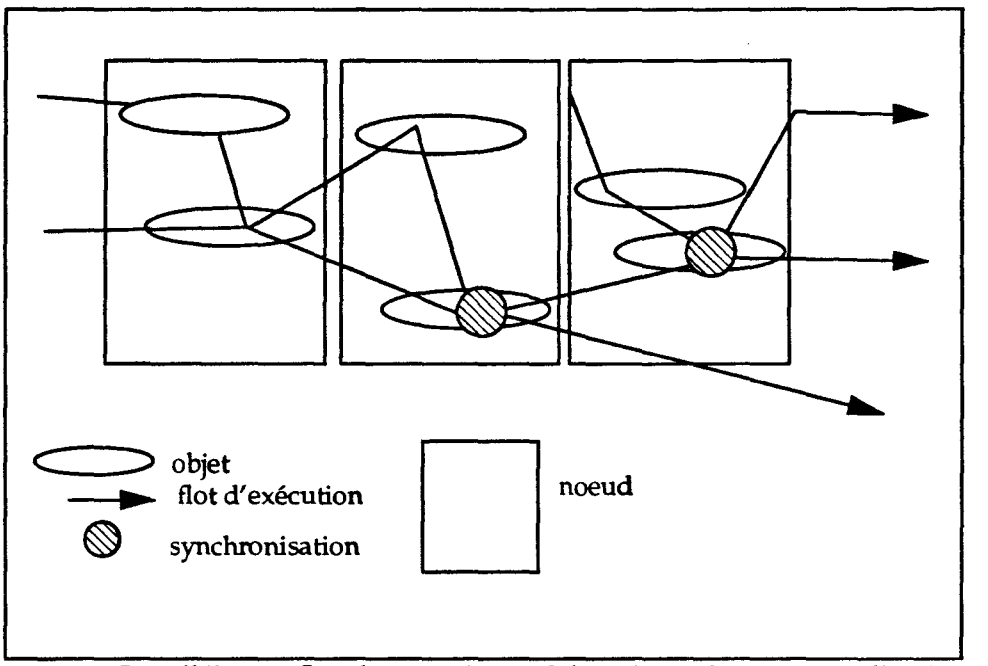


figure 1.0 *Parallélisme, Synchronisation et Répartition dans une application*

Les langages et systèmes référencés dans ce chapitre sont les suivants: *Abcl/1* [Yonezawa86,86b,87,90,Takara90], *Actra* [Thomas89, McAffer90], *Act1* [Liberman81], *Act3* [Agha86], *Act++* [Kafura89,90,91], *Ada* [Ada83], *Actalk* [Briot89a,89b], *Amoeba* [Mullender86, Mullender90, Tanenbaum85], *Clix* [Hur87], *Concurrent Aggregates* [Dally87, Dally89, Chien91], *Concurrent C++* [Gehani88], *Concurrent Smalltalk* [Yokote86,87], *C++* [Stroustrup86], *Dragoon* [Genolini89, Atkinson91], *Distributed-Smalltalk* [Bennet87,90], *Distributed-Smalltalk* [Shelvis88], *Edoms* [Burke90], *Eiffel* [Meyer88], *Emerald* [Black86,87, Jul88], *Eiffel Parallèle* [Caromel89,90,90b,91], *Gothic* [Banâtre91], *Guide* [Decouchant89, Balter90, Krakowiak90, Nguyen90], *Hybrid* [Nierstrasz87], *Parallel Eiffel* [Colin89,90, Gransart91,92], *Po* [Corradi87,89,91, Ciampolini89], *Pool* [America87,87b,90], *Presto* [Bershad88, Faust90], *Rosette* [Tomlinson89], *Sloop* [Lucco87], *Smalltalk80* [Goldberg83], *Sos* [Shapiro89,91, Makpangou91], *Trellis/Owl* [Moss87]. Tous ces langages et systèmes constituent un éventail très étendu et significatif du domaine étudié.

- I - 1. Objets et Parallélisme

L'objectif des langages parallèles à objets, (c'est à dire incluant à la fois description par objets et expression du parallélisme,) est de faciliter la programmation des machines parallèles en exprimant naturellement le parallélisme du monde réel. Les langages à objets sont modulaires par essence et doivent ainsi favoriser la modélisation des applications parallèles. L'expression du parallélisme doit bénéficier de ces bonnes propriétés.

Une application développée dans un langage à objets se décompose en un ensemble d'objets communiquant entre eux. Une application séquentielle dans un tel langage s'exécute en un enchaînement d'appels aux méthodes sur les objets du système. Un appel de méthode sur un objet s'apparente à un appel procédural classique et un seul flot d'exécution évolue donc dans le système. Le flot d'exécution doit être pris en charge par un processus[1]. Dans ce contexte séquentiel, l'unique processus associé au flot d'exécution saute d'objet en objet au gré des déclenchements des méthodes de l'application.

L'introduction du parallélisme dans un langage à objets entraîne la présence simultanée de multiples flots d'exécution. A chaque flot d'exécution est associé au moins un processus distinct. Le degré de parallélisme d'une application à objets peut être mesuré par le nombre de processus dans le système à un moment donné (figure 1.1 «Application parallèle à objets»).

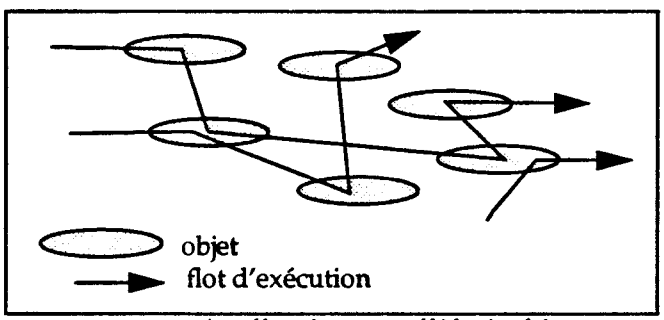


figure 1.1 *Application parallèle à objets*

Dans cette synthèse nous établissons une classification des principaux langages parallèles à objets en fonction des critères suivants:

- L'intégration des processus dans le modèle Objet. On distinguera l'approche «objets + processus» de celle des «objets actifs».

- Le nombre possible de processus concurrents dans un objet. On distinguera les objets mono-programmés des objets multi-programmés.

- La représentation de l'objet: localisé, dupliqué ou fragmenté.

- Le domaine d'un processus (l'espace accessible par le processus).

- Le mode de création du parallélisme.

---

1. *processus* : Toute activité séquentielle à laquelle est alloué un processeur [Cornafion81].

- I - 1.1 La notion de processus dans le modèle objet

Les flots d'exécution créés par un programme issu d'un langage parallèle à objets sont pris en charge par un ensemble de processus. Deux approches sont rencontrées dans les langages parallèles à objets: 1- les processus sont des entités distinctes des objets (approche «objet + processus») 2 - Les processus et les objets sont unifiés en une seule entité appelée «objet actif».

- I - 1.1.1 L'approche «objet + processus»

Dans l'approche «objet + processus», les objets sont passifs (au sens classique: un objet représente l'encapsulation de données) et les processus (entités actives) évoluent parmi ces objets. Les processus ne sont pas des objets et leur contrôle doit être explicitement exprimé par le programmeur à l'aide d'opérateurs spécifiques.

Le degré de parallélisme est égal au nombre de processus créés et évoluant simultanément dans l'application à un instant donné. Il est totalement déconnecté du nombre d'objets présents à cet instant. Deux mondes distincts coexistent.

Suivant le langage, le contrôle des processus est plus ou moins bien intégré au modèle objet:

- 1- Les processus forment un monde totalement différent de celui des objets. Le programmeur définit les objets de son application et les processus qui manipuleront ces objets. Les processus sont gérés en dehors de la partie objet. Le langage *Concurrent C++* se rattache à cette approche. Les processus sont déclarés séparément des classes d'objets et le programmeur doit lancer explicitement les processus décrits dans le programme principal.

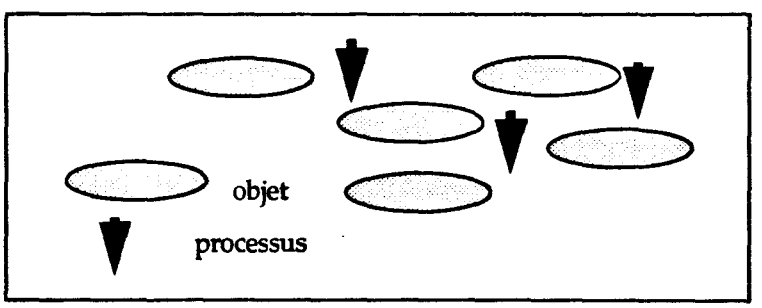


figure 1.2 *Des objets et des processus*

- 2 - Les processus sont manipulés par l'intermédiaire d'objets qui les représentent dans le monde objet. Le langage fournit généralement une classe particulière minimale pour ces objets. Cette classe définit les méthodes de manipulation de processus (lancement, suspension, reprise, destruction, etc). Une instance de cette classe représente un processus dans le monde objet. La gestion des processus peut donc être développée par une approche objet. Les langages *Smalltalk80, C++, Presto, Trellis/Owl* et l'extension parallèle de J.F.Colin du langage *Eiffel* font partie de cette famille de langages.

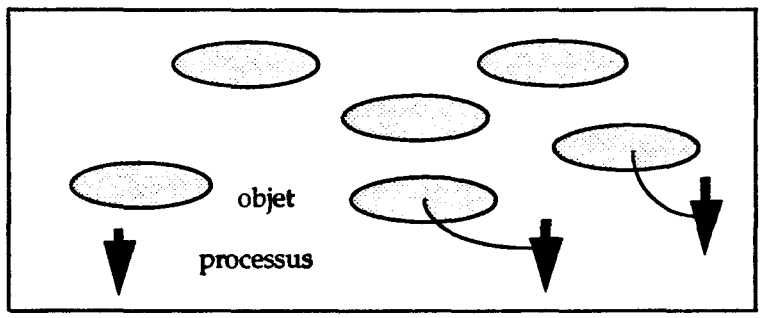


figure 1.3 *Des objets et des objets qui représentent des processus*

Dans l'extension parallèle d'*Eiffel* [Colin90,91], chaque instance de la classe *Thread* sert au contrôle d'un processus. Par l'intermédiaire d'une telle instance, on peut créer un processus en fournissant un objet hôte et la routine de cet objet à déclencher. Ensuite et toujours par l'intermédiaire de l'instance de la classe *Thread*, on peut suspendre, relancer ou détruire le processus évoluant parmi les objets.

Les instances de la classe *Thread* cachent dans un objet, un processus. Chaque instance contient dans une de ses variables d'instance le nom de la routine associée au processus. L'interface de la classe propose une série de primitives de manipulation de processus.

- 3 - La description des processus est une section particulière des objets. Dans le langage *Emerald*, une classe peut contenir une «*Process Section*». A la création d'un tel objet, un nouveau processus va prendre en charge cette section et évoluer dans l'ensemble des objets au gré des appels de méthodes. L'introduction de la notion de processus dans les objets ne joue donc ici qu'un rôle de création de nouveaux flots d'exécution.

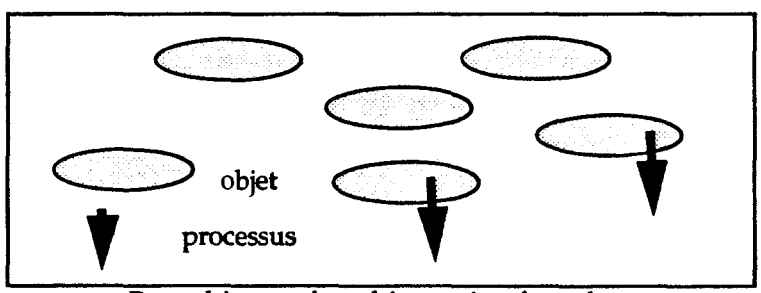


figure 1.4 *Des objets et des objets qui créent des processus*

Toutes ces approches se caractérisent par l'introduction de processus dans le modèle objet comme des entités particulières qu'il faut gérer de manière spécifique. De cette manière, le modèle objet apporte peu à la conception des programmes fortement parallèles (nombreux flots d'exécution), ce qui constitue pourtant l'objectif initial.

L'approche 2 (et plus faiblement la 3) permet cependant d'exprimer la gestion des activités parallèles sur un mode objet. On bénéficie donc des possibilités d'abstraction du modèle objet, ce qui autorise la conception d'entités de plus haut niveau (comme les objets actifs) à partir d'objets de bas-niveau représentant directement les processus (voir le travail de C. Gransart dans *Parallel Eiffel* [Gransart91,92]).

Toutes ces approches se caractérisent aussi par le fait que les processus peuvent sauter d'objets en objets suivant les appels de méthodes, déconnectant ainsi les deux notions. Une implantation sur Multicomputer, (où les objets sont distribués sur les noeuds) nécessite alors des techniques de migration de processus ou d'appels de procédure à distance «Remote Procedure Call» [Birell83] rendant encore plus difficile la maîtrise globale d'une application. Cela mène souvent à restreindre les possibilités de «sauts» des processus en les limitant à un petit groupe d'objets que l'on appelle *domaine* du processus (voir section - I - 1.5 «Domaine»). A l'extrême, un processus est limité à un objet, et élimine donc toutes questions de migration de processus.

Ces deux remarques montrent la nécessité d'une liaison plus intime entre objets et processus. C'est l'approche «objets actifs», qui est maintenant généralement reconnue comme la bonne approche pour allier objets et processus. Les recherches dans ce domaine font apparaître plusieurs types d'objets actifs que nous détaillons maintenant.

- I - 1.1.2 Objets Actifs (les objets serveurs et les Acteurs)

La notion d'«objet actif» permet d'intégrer intimement des processus dans la structure de l'objet (figure 1.5 «Les objets actifs»). On parle d'objet actif lorsqu'un objet représente (et exécute) une tâche[1] indépendante de l'application. Cette tâche est naturellement prise en charge par un processus interne à l'objet. Ce processus n'existe qu'à partir de la création de l'objet et travaille jusqu'à la destruction de l'objet. A un plus haut niveau de description, les objets et les processus ne sont plus distingués, et le modèle d'objet actif sert donc de manière unique à la structuration des données et des activités d'une application parallèle.

Contrairement au modèle séquentiel, l'instance est capable d'effectuer des traitements locaux sans être sollicitée par une invocation de méthode. L'objet est réellement actif, au lieu d'être seulement «réactif».

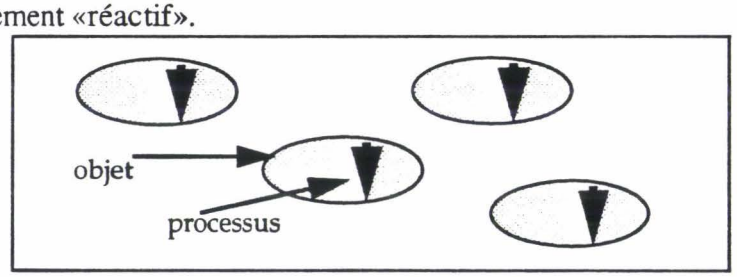


figure 1.5 *Les objets actifs*

Nous pouvons scinder la famille des langages à objets actifs en fonction du rôle du processus de l'objet. On peut ainsi isoler deux types d'objets actifs:

- 1 - **Les objets serveurs.** Le processus de l'objet gère le fait que cet objet est plongé dans un univers parallèle: L'objet reçoit des requêtes concurrentes venant d'autres objets et doit gérer l'ordonnancement des traitements. Dans *Pool-T*, les objets sont actifs et une partie de code spécifique, nommée «body» est lancée dès l'instanciation de l'objet. Le «body» accepte sélectivement les requêtes par une instruction spécifique «Answer». Il va ainsi sérialiser de manière ad-hoc les traitements et implanter la synchronisation nécessaire. Un mécanisme identique est implanté dans *Eiffel Parallèle.*

1. Nous verrons dans les objets multi-programmés qu'un objet peut représenter plusieurs tâches

- 2 - **Les Acteurs**: Dans les langages d'acteurs le processus d'un acteur prend en charge directement l'exécution du traitement d'un message défini dans le «*script*» de l'acteur [Agha86]. Chaque «invocation» d'un acteur donne lieu à un envoi asynchrone de message dans la boîte aux lettres de l'objet cible. Le code du «*script*», écrit par le programmeur, traite les messages déposés dans la boîte aux lettres de l'acteur. *Act1*, *Act3*, *Abcl/1*, *Actalk*, *Actra* se classent dans cette approche.

Les acteurs sont un outil de bas niveau pour la construction d'applications parallèles. Ils se prêtent bien à des études fondamentales sur le parallélisme, mais se prêtent mal à une utilisation à grande échelle.

Les objets actifs de type serveur ont donné des langages plus opérationnels qui récupèrent mieux les qualités de la programmation objet comme la réutilisation et l'abstraction [Carome190].

L'approche «objets actifs» induit intuitivement une démarche d'implantation bien adaptée aux multicomputers: Les objets (et leurs processus) sont situés sur les noeuds; ils communiquent par messages grâce au réseau de communication de la machine. Ainsi le parallélisme du modèle sera bien relayé par le parallélisme physique de la machine, idem pour la communication. Dans cette thèse nous reprendrons cette idée pour les composants actifs de communication (voir chapitre II « Les Composants Actifs de Communication»)

- I - 1.2 Le nombre possible de processus concurrents dans un objet

- I - 1.2.1 Les objets mono-programmés

La plupart des langages à objets actifs, *Pool*, *Act++*, *Eiffel Parallèle*, proposent des objets mono-programmés, bien que sollicités par des requêtes simultanées, un seul flot d'exécution est actif à un moment donné dans un objet. Cela peut signifier que:

- seul un des processus client est autorisé à entrer dans l'objet lorsque celui-ci est libre (c'est le cas de Pool, et c'est le «body» qui décide de celui qui entre).

- ou qu'il n'existe qu'un seul processus qui traite séquentiellement toutes les requêtes (c'est le cas de *Eiffel Parallèle*, et c'est le «*Live*» qui décide de l'ordre des traitements). C'est le cas aussi du modèle acteur.

Dans les deux cas, le degré de parallélisme est égal au nombre d'objets actifs à l'instant donné.

L'avantage immédiat des objets mono-programmés est d'éviter totalement les accès simultanés à la représentation interne de l'objet, accès qui pourraient rendre cette structure incohérente.

L'inconvénient de cette solution est la forte (et parfois inutile) sérialisation des traitements qu'elle impose. Un objet actif peut alors devenir un goulot d'étranglement pour l'application.

- I - 1.2.2 Les objets multi-programmés

Pour augmenter le degré de parallélisme et éliminer les goulots d'étranglement, le langage peut permettre un parallélisme dans l'objet (parallélisme intra-objet). Plusieurs méthodes peuvent alors s'exécuter sur le même objet et plusieurs processus sont actifs simultanément dans l'objet. Le degré de parallélisme est ici supérieur au nombre d'objets actifs.

Le goulot d'étranglement détecté dans le modèle des objets mono-programmés disparaît ici en permettant des activités parallèles dans l'objet. Les objets fortement sollicités peuvent travailler à la façon d'un serveur qui répartit les traitements sur plusieurs processus.

Prenons l'exemple du langage *Po*, basé sur les objets actifs permettant le parallélisme intra-objet. Un processus de l'objet exécute la «Schedulling Part»; il est toujours actif et fonctionne à la façon d'un serveur. Plusieurs invocations de méthode peuvent être autorisées simultanément par ce processus. Le processus évalue pour toutes les méthodes appelées une condition dite d'ordonnancement qui implémente partiellement la fonction de serveur. Elle filtre les requêtes et lance les méthodes. Un mécanisme similaire existe dans le langage *Dragoon*.

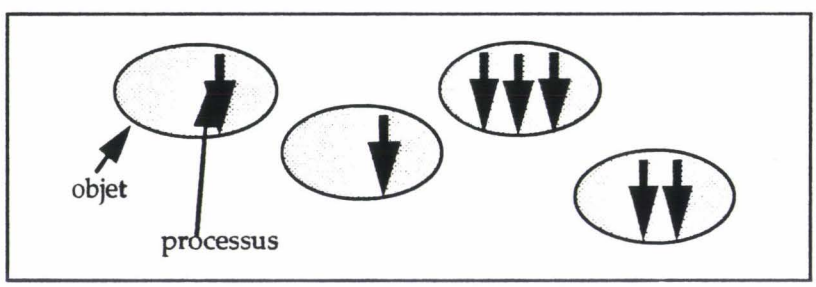


figure 1.6 *Parallélisme dans l'objet*

La programmation des objets multi-programmés peut être délicate. Des outils de synchronisation intra-objet sont nécessaires. Ces outils ne doivent pas entraîner la dispersion du code de synchronisation dans les méthodes (comme le feraient de simples sémaphores) pour ne pas perdre les possibilités d'extensibilité et de réutilisabilité du modèle Objet. Les langages *Po, Guide, Dragoon* et *Act ++* qui permettent le parallélisme intra-objet utilisent tous une synchronisation «a priori»; c'est à dire que la synchronisation exprime les conditions dans lesquelles une méthode peut être lancée. Une fois lancée, les méthodes ne sont plus synchronisées. Cette synchronisation «a priori» peut dans certains cas imposer une trop forte sérialisation des traitements dans l'objet.

Les objets multi-programmés semblent intéressants pour les multicomputers. D'une part, les noeuds d'une telle machine utilisent des processeurs puissants permettant la multiprogrammation. D'autre part si les différents processus internes peuvent être pris en charge par des noeuds différents, le parallélisme intra-objet sera un parallélisme réel. On voit que dans ce dernier cas, un objet s'étend logiquement sur plusieurs noeuds, ce qui pose le problème de l'unicité de sa représentation. Cette thèse s'intéresse particulièrement à ce problème.

- I - 1.3 Représentation des objets

Comme on vient de le voir, le grain de parallélisme peut être plus fin que l'objet. La représentation de l'objet influence alors fortement les possibilités de parallélisme.

- I - 1.3.1 Les objets localisés

La plupart des langages parallèles à objets implante les objets par des entités «mono-bloc» à l'exécution. Toutes les variables d'instance de l'objet sont regroupées en un seul bloc mémoire et les traitements dans l'objet doivent s'effectuer où se trouve l'objet.

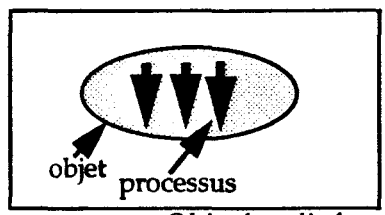


figure 1.7 *Objet localisé*

L'avantage d'une représentation localisée sur un noeud est la simplicité du modèle d'exécution. Un traitement est pris en charge par un processus du noeud d'accueil. Cependant, si on veut un réel parallélisme dans l'objet, en distribuant les traitements (et donc les processus) sur différents noeuds, l'accès à la structure localisée par les processus distants pose un pénalisant problème d'implantation.

Si on veut un réel parallélisme intra-objet, les objets multi-programmés ne peuvent être construits avec cette approche.

- I - 1.3.2 Les copies d'objets

Certains langages à objets gèrent des copies d'objets. Chaque copie peut être exploitée par un processus distinct. Le système doit alors gérer la cohérence des copies et mettre à jour les éventuelles modifications des variables d'instance. *Distributed Smalltalk* [Schevis88] permet cette possibilité mais ici la copie d'objet est essentiellement utilisée pour faire coopérer plusieurs sites *Smalltalk* faiblement couplés.

L'opération *«become»* [Agha86] définie dans les langages d'acteurs peut générer plusieurs flots d'exécution dans un acteur non-sérialisé. L'acteur traite le message suivant avant la fin du traitement du message précédent. Les traitements parallèles de deux messages sur le même acteur s'effectuent sur des copies des variables locales (accointances de l'acteur). On s'écarte ici fortement du modèle objet.

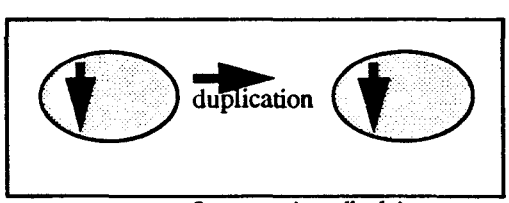


figure 1.8 *Les copies d'objets*

- I - 1.3.3 Les objets distribués (fragmentés)

Pour créer un réel parallélisme intra-objet, il est intéressant de découper les objets en fragments qui seront à l'exécution distribués sur les noeuds de la machine cible. Chaque fragment contient une partie des variables d'instances de l'objet et l'ensemble des fragments représente l'objet. Les traitements peuvent alors s'exécuter réellement en parallèle sur les différents fragments de l'objet.

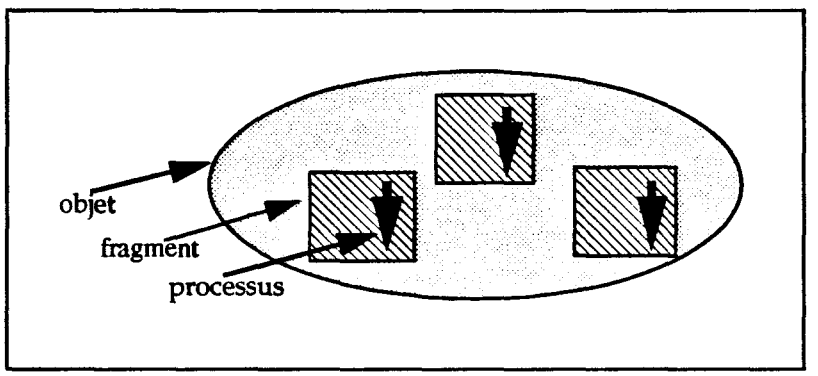


figure 1.9 *Les objets distribués*

Dans le langage *Concurrent Aggregate*, un agrégat est une collection d'objets. L'agrégat évolue dynamiquement et peut être identifié par un nom. Le langage propose des primitives permettant directement de manipuler les agrégats. La fragmentation est ici réalisée à un niveau supérieur: un objet peut être considéré comme un fragment et un agrégat comme un objet.

Les objets dans le système *Gothic* peuvent être fragmentés pour être répartis sur les processeurs de la machine cible. Les *multiprocédures* introduites dans le langage permettent de lancer des traitements sur les différents fragments. Les traitements sur les fragments peuvent être différents. L'objectif est ici de pouvoir écrire des méthodes pour des algorithmes parallèles. Le langage de programmation associé, *PolyGoth*, introduit les *multiprocédures* qui sont des procédures qui définissent des blocs d'instructions pouvant être exécutés en parallèle sur les différents fragments. La multiprocédure est l'outil d'expression du parallélisme dans le système.

Les objets distribués peuvent avoir des liens dynamiques avec leurs fragments. Un objet peut ainsi détacher un de ses fragments ou le remplacer par un fragment du même type. C'est le cas pour le langage *Sos* et les «*Concurrent Aggregates*». Dans *Sos*, lors d'une invocation de méthode, le système crée un fragment mandataire sur le site du client. Un objet fragmenté de Sos peut donc évoluer dynamiquement: perdre un fragment ou en intégrer de nouveaux en fonction des invocations de méthodes. Un même fragment peut être partagé par deux objets différents. Le système permet la migration des objets et un objet fragmenté migre avec ses différents fragments. Le langage *FOG* [Gourhant90a,90b] décrit les classes d'objets d'une application, le langage est écrit au dessus du langage *C++*.

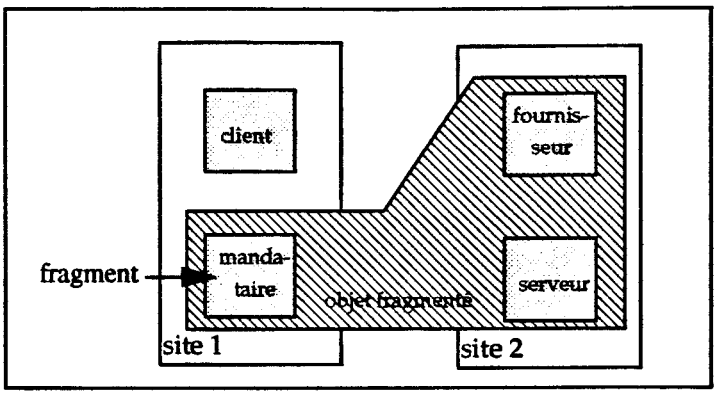


figure 1.10 *Fragmentation dynamique Sos*

Voici le tableau récapitulatif des sortes d'objets fragmentés

	Origine de la fragmentation	fragmentation dynamique	fragment partagé par plusieurs objets	migration des objets	parallélisme	expression du parallélisme
Gothic	en fonction des algorithmes	non	non	non	intra-objet mono-programmé dans les fragments	multi-procédure
Sos	explicite	oui	oui	oui	-	-
Concurent Aggregate	explicite	oui	-	non	intra-Aggregate	aggregate operations - multi-access

figure 1.11 *Tableau comparatif des objets fragmentés*

Les objets distribués nécessitent un modèle d'exécution évolué. La mise en place de communications entre les fragments d'un même objet est indispensable. La distribution des fragments peut entraîner un défaut de localité entre une variable d'instance et une méthode. Cette distribution doit être prise en charge par le modèle d'exécution.

Dans cette thèse, nous introduisons les Composants Actifs de Communication, véritables fragments actifs d'objets multi-programmés (voir chapitre II).

- I - 1.4 Tableaux récapitulatifs

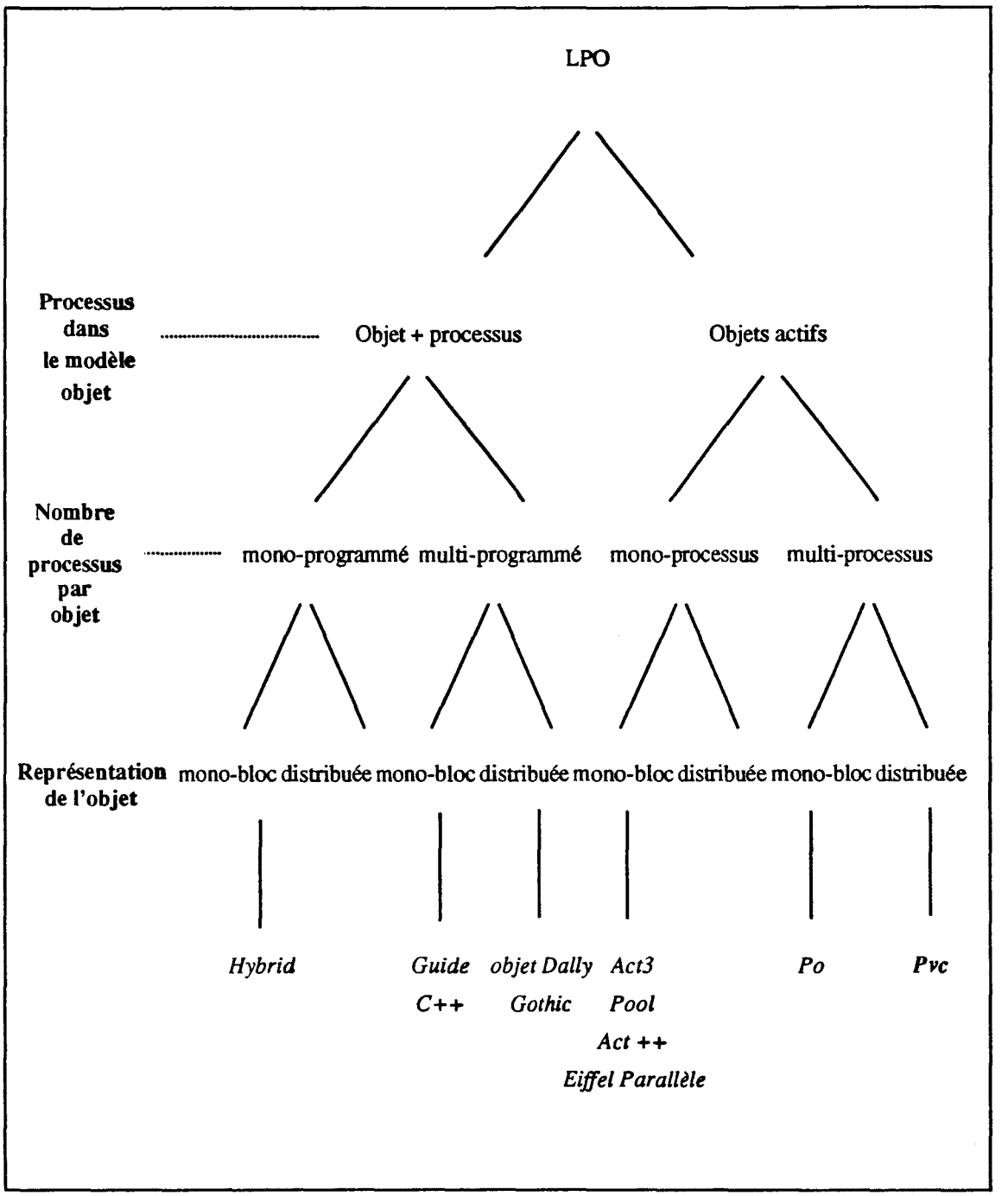


figure 1.12 *Classification des LPO/s*

NB: Le manque d'information sur certains langages (tels que Abcl, Dragoon, Sos) nous empêche de les placer dans ce tableau.

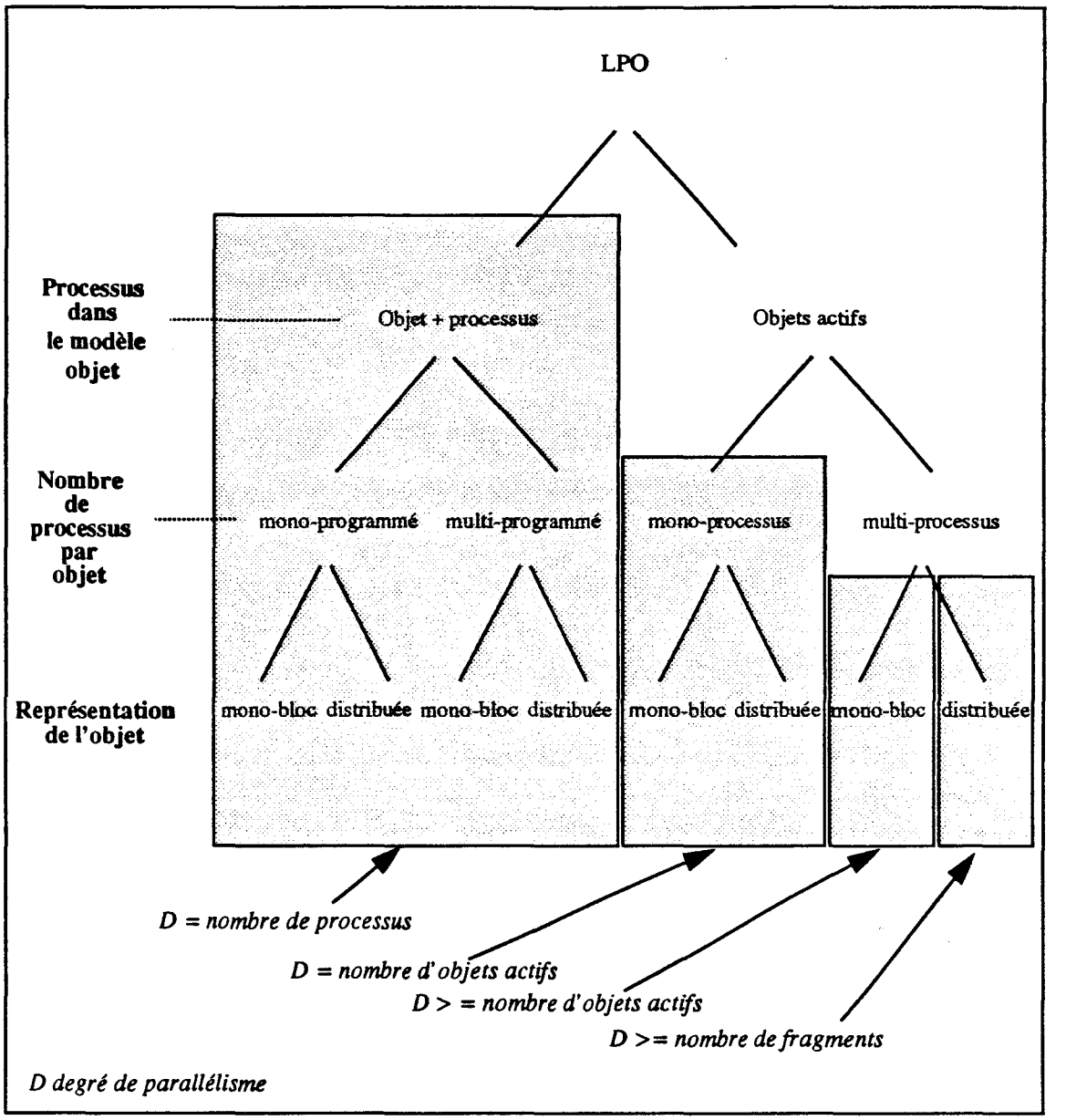


figure 1.13 *Degré de parallélisme*

- I - 1.5 Domaine

Nous pouvons compléter notre classification en introduisant une nouvelle caractéristique: le **domaine.**

Un domaine définit l'espace accessible par un processus.

La notion de domaine s'apparente à celle de contexte du processus. L'application se compose à l'exécution d'un ensemble dynamique de domaines. L'activité unique d'un domaine ne nécessite a priori [1] pas de synchronisation particulière dans un domaine. Les domaines sont de tailles variables selon le langage à objets. Nous détaillons ici les différents types de domaines rencontrés.

- I - 1.5.1 Un domaine = unité physique de parallélisme

La notion de domaine peut être directement liée à l'architecture de la machine. Tous les objets d'un même site se partagent le même processus et le domaine correspond exactement au site physique. Un domaine regroupe l'ensemble des objets et le processus qui évolue entre les objets du domaine. Un domaine est une entité parallèle et le nombre optimal de domaines est lié à celui des sites de la machine cible. Dans cette approche «domaine = site», le degré de parallélisme est égal au nombre de domaines définis.

Dans les différentes versions de *Distributed Smalltalk* (Shelvis, Bennet), la notion de domaine colle directement aux différentes machines constituant un réseau. Chaque machine possède une copie de la machine virtuelle et de l'image *Smalltalk* et chacune possède son propre flot d'exécution. Dans ces implémentations, le parallélisme n'est pas réellement exploité, l'objectif est ici d'établir la coopération entre plusieurs machines *Smalltalk.*

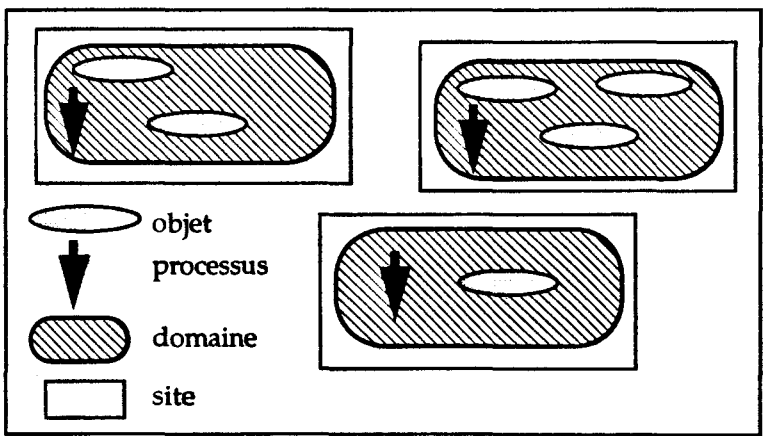


figure 1.14 *Un domaine = un Site*

Cette approche «domaine = site» est inadaptée aux multicomputers, machines fortement parallèles. Un noeud d'une telle machine est matériellement prévu pour accueillir plusieurs processus. Cette approche est d'un grain trop gros pour les multicomputers.

1. Nous verrons plus loin que les modèles basés sur le objets multi-programmés autorisent plusieurs processus dans un même domaine. Dans ce cas une synchronisation est nécessaire.

- I - 1.5.2 Un domaine = unité logique de parallélisme

La notion de domaine est souvent utilisée pour contrôler le parallélisme d'une application. Le programmeur définit ses domaines et agit ainsi directement sur le grain de parallélisme de son application. Le domaine est ici le grain logique de parallélisme.

Dans *Hybrid* les domaines sont directement définis dans le langage. Dans chaque domaine est associé un processus partagé par les objets du domaine. Il peut contenir un nombre variable d'objets et le plus haut degré de parallélisme est obtenu en associant à chaque domaine un seul objet (figure 1.15 «Un domaine, unité logique de parallélisme»).

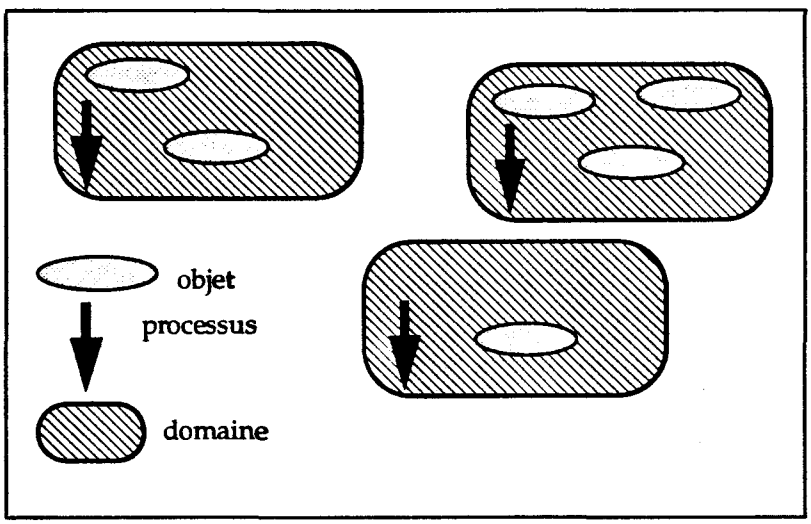


figure 1.15 *Un domaine, unité logique de parallélisme*

La notion de domaine «unité logique de parallélisme» est utilisée dans les langages mélangeant des objets actifs et des objets passifs. Les objets passifs ne sont normalement pas partageables par les objets actifs pour des raisons de protection. Pour cela, un domaine comprend un objet actif et un certain nombre d'objets passifs (c'est le cas d'*Eiffel Parallèle*).

Dans le système *Guide*, les domaines peuvent s'étendre ou se réduire dynamiquement au fur et à mesure des déclenchements de méthode. Les domaines se sont pas dépendants des sites et un domaine peut s'étendre sur un autre site. Les domaines ne sont pas disjoints et un objet peut être partagé par plusieurs domaines. La protection naturelle des objets dans les domaines n'est plus garantie par le modèle et le système propose des mécanismes de synchronisation ad-hoc.

La notion de domaine nuit à l'uniformité de la programmation objet. Les échanges entre les objets d'un même domaine et entre deux objets de deux domaines différents ne peuvent être identiques. La synchronisation n'est pas uniforme puisqu'il y a du parallélisme entre les domaines et pas de parallélisme intra-domaine (un seul flot d'exécution entre les objets d'un même domaine). Le programmeur doit connaître le domaine de l'objet destinataire pour assurer le bon fonctionnement de son application.

- I - 1.5.3 Un domaine = un objet

Dans les langages à objets actifs, les processus ne peuvent évoluer qu'à l'intérieur d'un objet. Le domaine d'un processus est limité à son objet hôte. Pour les objets mono-programmés un domaine est équivalent à l'objet.

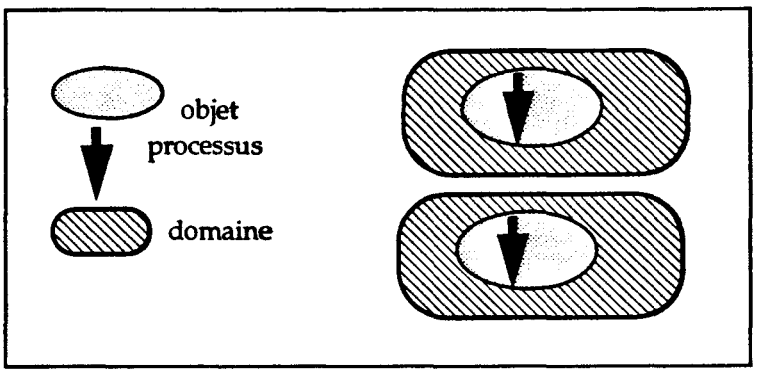


figure 1.16 *Les domaines des objets actifs*

L'approche «domaine = objet» rend uniforme la synchronisation d'une application entre les objets puisqu'il n'y a pas de notion de groupe d'objets.

Les objets actifs multi-programmés ont plusieurs processus qui se partagent le même domaine. Dans ce cas, le système doit proposer des outils de synchronisation des activités concurrentes d'un même domaine.

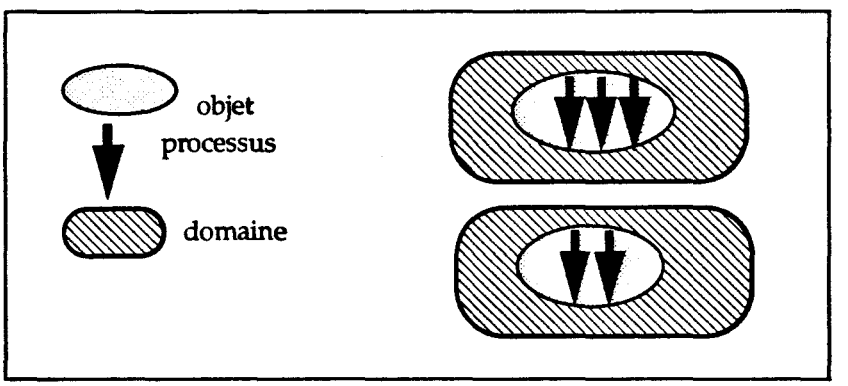


figure 1.17 *Les domaines des objets actifs multi-programmés*

- I - 1.5.4 Un domaine = un fragment

Dans les langages permettant une répartition de la représentation de l'objet en fragments, le domaine d'un processus est plus petit qu'un objet et se limite à un des fragments de l'objet.

Contrairement aux objets multi-programmés localisés, deux processus ne peuvent se partager un même domaine. La notion de fragment est alors un moyen de créer du parallélisme intra-objet tout en protégeant les données internes de l'objet des accès concurrents.

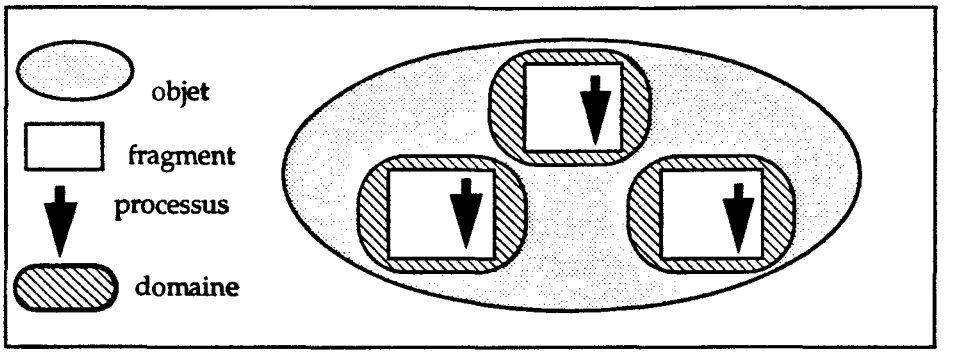


figure 1.18 *Les domaines des objets fragmentés*

Les composants actifs de communication introduits dans le chapitre - II - unifient la notion de domaine et de fragments.

- I - 1.5.5 Tableau récapitulatif

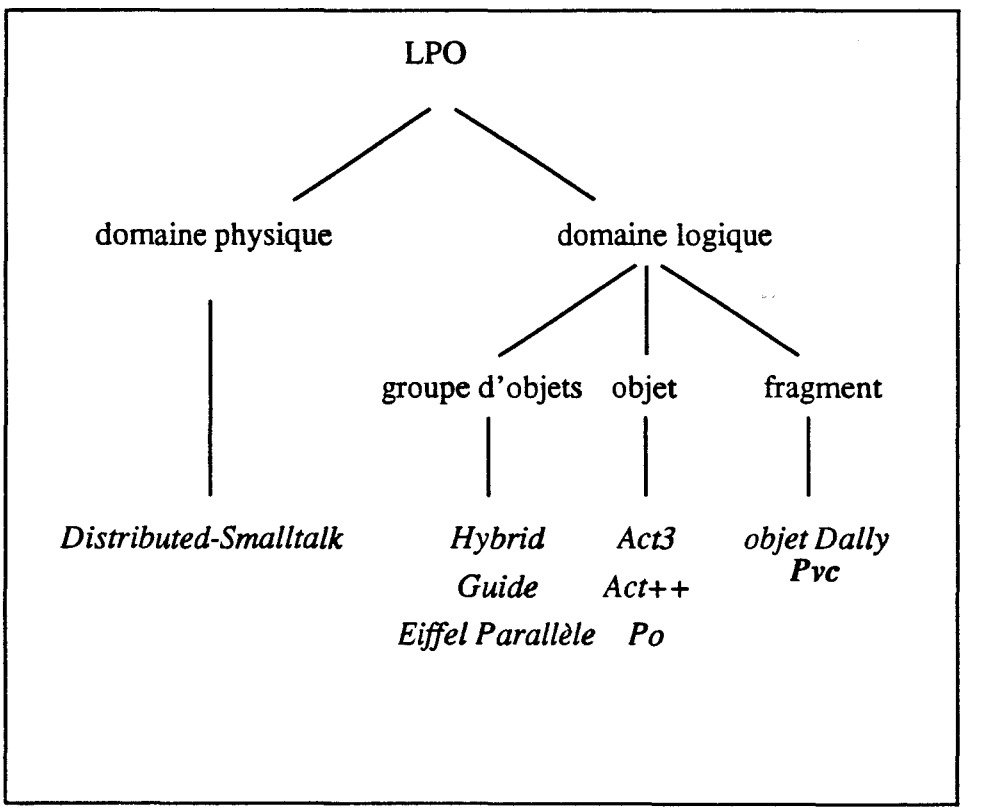


figure 1.19 *Les domaines dans les langages parallèles à objets*

- I - 1.6 Mode de création du parallélisme

Nous décrivons, dans une cette section les différents modes de création du parallélisme dans ces langages parallèles à objets.

La création et la manipulation des entités parallèles d'une application doivent s'exprimer dans les langages de programmation à parallèlisme explicite. Les langages parallèles à objets proposent en général plusieurs outils. L'expression du parallélisme est liée aux différents degrés de parallélisme du modèle d'exécution du langage support.

- I - 1.6.1 Rappel sur le traitement à distance

Il y a deux grandes techniques pour effectuer un déclenchement d'un traitement à distance:

- 1 - L'appel de procédure à distance (Remote Procédure Call, RCP) qui permet d'effectuer un traitement à distance sur le mode procédural. Du point de vue de l'appelant, c'est son processus qui exécute la procédure.

- 2 - L'envoi de message qui permet de déclencher à distance un traitement. Le message doit être reçu par une entité active qui effectue le traitement et retourne (éventuellement) un message pour le résultat.

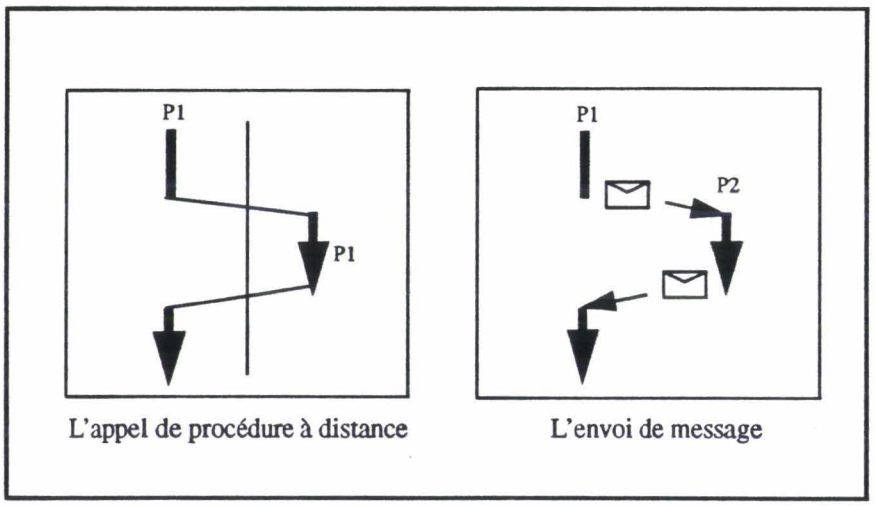


figure 1.20 *Outil de création du parallélisme*

L'envoi de message met en jeu deux processus et des dérivations de cette technique peuvent générer du parallélisme. Le RCP, lui, ne met en jeu sémantiquement qu'un seul flot de contrôle et est donc peu adapté à la création du parallélisme. Dans ce qui suit nous supposerons que la technique de l'envoi de message est utilisée. Les dérivations introduites sont l'appel asynchrone, la réponse anticipée, la libération de l'appelant, la délégation et la création d'objet actif.

- I - 1.6.2 L'appel asynchrone

- I - 1.6.2.1 L'appel asynchrone pur

L'invocation de méthode asynchrone est une source fréquente de parallélisme dans les langages parallèles à objets [Yonezawa87]. Ces invocations de méthode utilisent l'envoi asynchrone de message et la méthode appelante reprend son exécution sans attendre de réponse de la méthode appelée. L'appel asynchrone libère ainsi la méthode appelante. La méthode appelante et la méthode appelée s'exécutent alors en parallèle (figure 1.21 «l'appel asynchrone»).

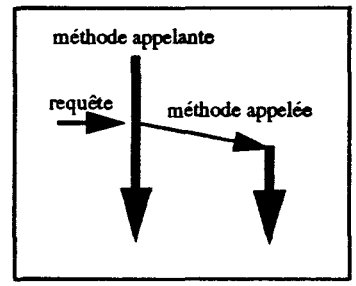


figure 1.21 *l'appel asynchrone*

Cette source de parallélisme est utilisée dans les langages *Sloop*, *Clix* et *Pool_I* [America90] et c'est le principal mode de création du parallélisme du modèle Acteur *Act1*, *Abcl/1*, *Actalk.*

Dans les langages d'acteurs, l'appel asynchrone s'exprime directement par un envoi de message asynchrone.

Dans Abcl/1, une construction syntaxique permet l'envoi multiple de messages. L'instruction s'achève après l'envoi de tous les messages. Ces envois multiples peuvent être assimilés à une suite d'envoi de messages asynchrones, ou ceux des Acteurs.

L'appel asynchrone est considéré comme le moyen "le plus pur" de création du parallélisme. Chaque appel asynchrone crée un nouveau flot d'exécution.

- I - 1.6.2.2 L'appel asynchrone avec récupération d'un résultat

L'appel de méthode en mode asynchrone peut retourner un résultat. La méthode appelante doit explicitement faire une demande de résultat.

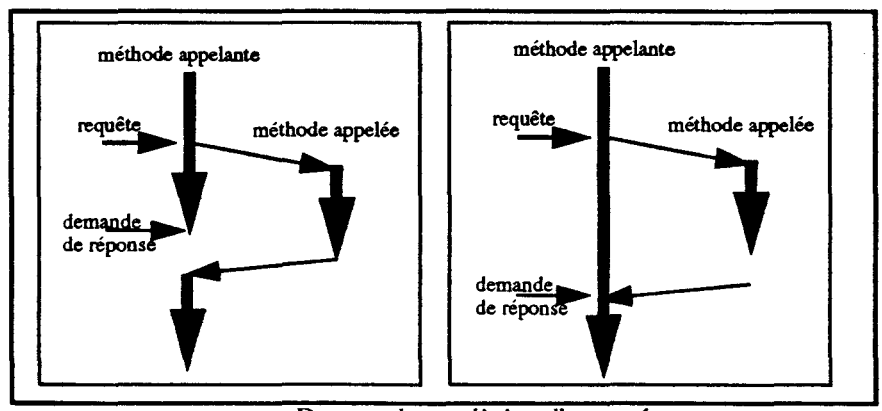


figure 1.22 *Demande explicite d'une réponse*

Dans ce contexte, les deux activités sont concurrentes après l'appel de la méthode et avant la demande de réponse ou le retour de celle-ci par la méthode appelée. Dans *Eiffel Parallèle*, le mécanisme d'attente par nécessité laisse au système la gestion de ses valeurs non encore retournées. La valeur n'est attendue que si elle doit être réellement utilisée. Ce mécanisme augmente la durée du parallélisme en repoussant au maximun la re-synchronisation des deux méthodes.

- I - 1.6.3 La réponse anticipée

Le mécanisme de réponse anticipée permet de continuer l'exécution de l'activité appelée après l'envoi de la réponse. La réponse est issue d'une invocation de méthode appelée en mode synchrone. La méthode appelée retourne une réponse avant la fin de son exécution. Le retour de la réponse débloque la méthode appelante et les deux activités s'exécutent en parallèle.

Le langage *Pool-T* introduit un parallélisme en décrivant des «*Post_sections*» dans la description des méthodes. Ces sections de code sont exécutées par la méthode appelée après le retour d'une réponse, en parallèle avec la reprise du traitement de la méthode appelante.

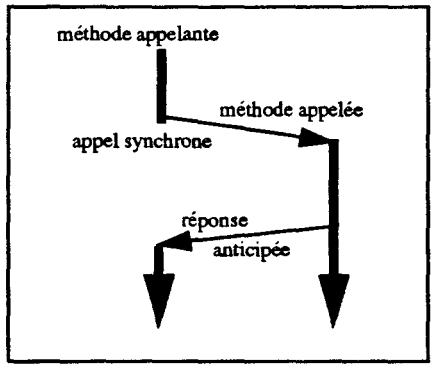


figure 1.23 *Le mécanisme de réponse anticipée*

Le mécanisme de réponse anticipée demande une bonne maîtrise du mécanisme du côté de l'appelé. Le programmeur doit pouvoir découper ses méthodes et décrire explicitement les actions pouvant s'effectuer en parallèle. Du côté de l'appelant, l'appel est classique (procédural), tout en créant de nouveaux flots d'exécution.

- I - 1.6.4 La libération explicite de l'appelant

La libération explicite de l'appelant permet de créer du parallélisme à partir d'un appel synchrone. Contrairement à la réponse anticipée la méthode appelée n'attend pas le calcul de la réponse pour libérer l'appelant. Le programmeur en utilisant une instruction spécifique (*release*), définit la section de programme pouvant s'exécuter sans danger en parallèle.

La méthode appelée et appelante peuvent ensuite se re-synchroniser sur l'envoi d'une réponse.

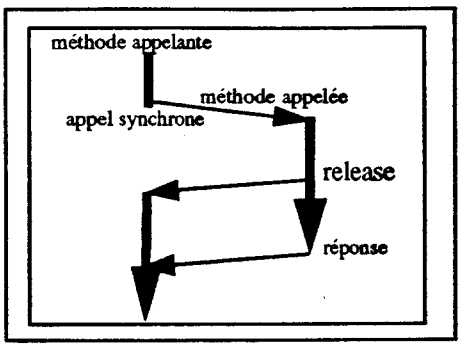


figure 1.24 *L'instruction release*

- I - 1.6.5 La délégation

Pour augmenter le parallélisme, certains langages tels que *Clix* ou *Act1*, où les acteurs non sérialisés, permet à un objet de déléguer l'exécution d'une méthode à un autre objet. L'objet est ainsi libre et peut traiter une autre demande. Le mécanisme de délégation libère l'objet et génère un parallélisme entre les objets.

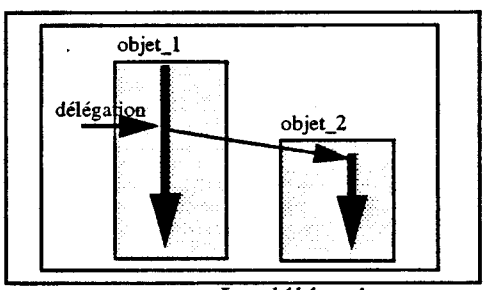


figure 1.25 *La délégation*

Un mécanisme de délégation similaire existe dans le langage *Hybrid.* La délégation d'une expression s'effectue par la création un nouveau flot d'exécution qui évaluera l'expression, (et non pas par un autre objet).

Si la délégation entraîne du parallélisme, elle n'est pas une manière naturelle d'effectuer deux traitements en parallèle. Une application parallèle ne s'exprime difficilement qu'avec la délégation. Si les deux méthodes d'un objet peuvent s'effectuer en parallèle, le programmeur doit construire deux objets différents. La délégation est un mécanisme de partage des connaissances qui permet à un objet ne sachant pas traiter une requête de déléguer le traitement à un autre objet.

- I - 1.6.6 La création d'objet actif

La création d'un objet actif est réalisée simplement par appel à la méthode d'instanciation d'un objet de la classe. Dès la création de l'objet, la méthode retourne le nom du nouvel objet, libérant ainsi l'activité appelante. Un nouveau flot d'exécution est associé à l'objet créé (figure 1.26 «La création d'objet actif»). Le processus de cet objet exécute un code spécifique défini dans la classe. Après le retour du nom du nouvel objet, l'objet créé ne communique plus a priori avec la méthode appelante, origine de l'instanciation.

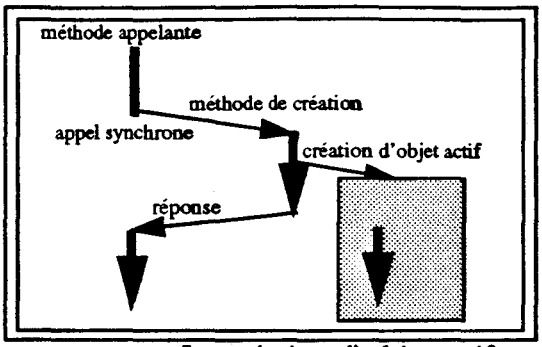


figure 1.26 *La création d'objet actif*

Cette source de parallélisme existe dans tout le modèle basé sur les objets actifs, voir section - I - 1.1.2 «Objets Actifs (les objets serveurs et les Acteurs)». Le mode de création du parallélisme par création d'objets actifs peut utiliser indifféremment l'appel synchrone ou l'appel asynchrone.

- I - 1.7 Conclusion sur «Objets et Parallélisme»

Pour intégrer la notion de processus dans le modèle objet, l'approche intéressante est celle des «objets actifs» où chaque objet est intrinsèquement parallèle. Le processus est directement intégré à la structure de l'objet. Le domaine d'évolution du processus est limité à l'objet associé. Le modèle d'exécution garantit ainsi une protection des variables locales dans l'objet.

Les objets actifs peuvent permettre un parallélisme intra-objet. Ils deviennent multi-programmés et chaque objet peut contenir ainsi plusieurs activités. La protection des variables locales n'est plus garantie par le modèle et le langage doit proposer des mécanismes de synchronisation de ces activités intra-objets.

Pour exploiter pleinement le parallélisme intra-objet des objets actifs sur une machine parallèle, il est nécessaire de fragmenter les objets. L'objectif visé est que chaque fragment soit localisé sur un noeud différent de la machine cible permettant un réel parallélisme dans l'objet. Chaque fragment est associé à un processus. Le domaine d'un processus est alors restreint au fragment et le modèle garantit une protection des données locales au niveau de chaque fragment malgré le parallélisme intra-objet.

Dans une approche «objet + processus», l'appel asynchrone est le plus pur moyen d'exprimer la création d'un nouveau flot d'exécution. Par contre dans l'approche «objets actifs», la création d'objet actif est plus naturelle. Le modèle d'exécution engendre une nouvelle entité parallèle à chaque création d'objet. De plus, différentes techniques peuvent exister pour se dégager de l'appel procédural aux objets et augmenter ainsi le parallélisme potentiel. Ces deux moyens d'expression et de création du parallélisme sont ici nécessaires.

L'existence en parallèle de plusieurs flots d'exécution nécessite des outils de synchronisation qui sont détaillés dans la partie suivante (- I - 2. «Objets et Synchronisation»).

- I - 2. Objets et Synchronisation

Dans une application parallèle, la synchronisation des flots d'exécution concurrents est une nécessité. Elle doit exprimer:

- 1 - L'ordonnancement nécessaire des tâches élémentaires. Cet ordonnancement est induit de la spécification de l'application et doit être respecté par l'application. (Par exemple, une tâche qui veut retirer un élément d'un «tampon» ne peut le faire que si un élément a été déposé par une autre tâche.)

- 2 - La protection des données communes à plusieurs flots, données inaccessibles simultanément par crainte de résultats imprévisibles.

Ces deux aspects de la synchronisation que sont l'ordonnancement et la protection nécessitent des outils adéquats dans un langage de programmation d'applications parallèles. D'une manière générale, on a besoin de pouvoir définir des points de synchronisation sur lesquels les tâches vont attendre la levée de contraintes de synchronisation.

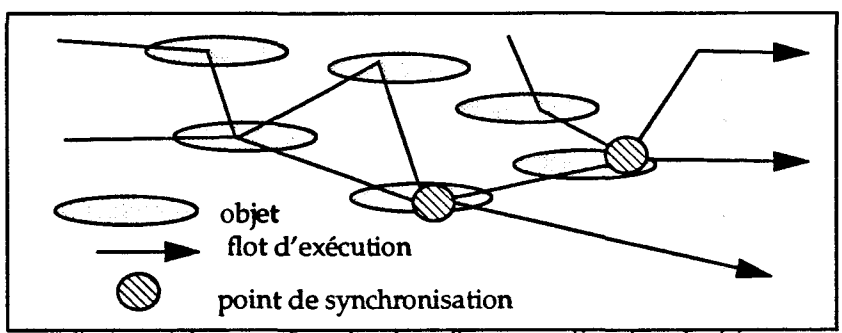


figure 2.0 *La synchronisation d'une application à objets*

Nous nous proposons ici de faire une synthèse des aspects synchronisation dans le monde des langages parallèles à objets.

L'expression de la synchronisation, au même titre que les méthodes et les variables d'instance, doit pouvoir être réutilisée dans le cadre d'un mécanisme de partage du source. L'association «parallélisme, synchronisation et héritage» pose des problèmes et P. America [America87] montre que l'héritage des contraintes de synchronisation est délicat, voire impossible dans *Pool*. Les contraintes de synchronisation liées aux nouvelles méthodes d'une sous-classe peuvent perturber les contraintes déjà définies dans les sur-classes. Or la déclaration d'une sous-classe ne doit pas remettre en cause les descriptions des sur-classes.

Dans une première section, nous introduirons les différents critères d'évaluation des types de synchronisation rencontrés dans les langages parallèles à objets. Nous établirons à partir de ces critères des fiches techniques des principaux langages à objets existants. Une dernière partie sera consacrée aux problèmes d'intégration de la synchronisation dans un héritage des classes.

- I - 2.1 Expression de la synchronisation

- I - 2.1.1 Les sémaphores et moniteurs

Une première approche des problèmes de synchronisation dans les objets, fut de réutiliser les solutions classiques de synchronisation de processus que sont les sémaphores [Dijkstra65] ou les moniteurs [Hoare75]. Historiquement ces outils ont été les premiers permettant d'aborder la programmation d'applications parallèles en notamment dans les premiers systèmes d'exploitation multi-tâches.

Ici ces approches peuvent être facilement utilisées dans une approche «objet + processus» (voir - I - 1.1 «La notion de processus dans le modèle objet») qui modélise l'exécution par un ensemble de processus parcourant un ensemble d'objets. Des points de synchronisation sont donc introduits dans les objets permettant l'ordonnancement des processus dans un objet et protégeant ses données. Par exemple un objet «tampon» de taille fixe peut être décrit de la manière suivante:

```
Objet Tampon[T]
var
     buf : Array[T] -- représentation interne du tampon
     full, empty, mutex : semaphore;
init()
     {
     full = 0;
     empty = n;
     mutex = 1;
     }
produire(e:T )
     {
     P(empty);
       P(mutex);
          -- deposer l'élement e
       V(mutex);
     V(full);
     }
consommer( ):T
     {P(full);
       P(mutex);
          -- sortir un élément
       V(mutex);
     V(empty);
     }
```

figure 2.1 *Programmation du tampon avec sémaphores*

On voit ici que *mutex* sert pour l'exclusion mutuelle qui protège les données internes, tandis que *full* et *empty* servent à ordonnancer de manière convenable les processus producteurs et consommateurs.

On aurait très bien pu utiliser tout aussi facilement la notion de moniteurs, d'autant que ces derniers protègent systématiquement les données qu'ils renferment, (un seul processus actif est dans un moniteur à un instant donné) et assurent l'ordonnancement par des conditions.

Des extensions de *C++* au parallélisme [Sun91] offrent ainsi des sémaphores comme outils de synchronisation. Le langage *Presto*[Bershad88] fournit aussi des moniteurs.

Ces approches peuvent être critiquées de plusieurs façons:

- 1 - Les outils sont de bas niveau et donc peu adaptés à une programmation à grande échelle. Une extension d'*Eiffel* [Colin91] qui offre au départ des classes *Monitor* et *Condition*, les réutilise immédiatement pour construire des outils de synchronisation de plus haut niveau.

- 2 - Les programmes parallèles qui utilisent des sémaphores ou moniteurs sont difficiles à valider. Par exemple, il est généralement difficile de montrer qu'un processus sortira bien d'une section critique.

- 3 - La synchronisation est dispersée dans le code, ce qui le rend peu adaptable et extensible (voir plus loin les problèmes d'héritage de la synchronisation).

- 4 - Ces outils sont essentiellement des outils de synchronisation en mémoire commune, aussi sont-ils peu adaptables aux multicomputers.

Pour ces raisons, ces outils sont actuellement peu utilisés dans les langages parallèles à objets au profit de notions plus adaptées que nous détaillons maintenant.

- I - 2.1.2 Les communications synchronisantes

De manière synthétique, on peut dire que la synchronisation dans le monde des objets a été réétudiée sur la base des constatations suivantes:

- 1 - Les objets sont essentiellement des entités communicantes et de ce fait, la synchronisation doit être étroitement liée à la communication. Elle sera essentiellement un ordonnancement des traitements des communications entre objets.

- 2 - Les objets sont essentiellement vus comme des objets actifs dont la synchronisation doit concerner celle qui se rattache à l'activité interne de l'objet avec les activités qui font appel à lui.

Ces deux points conduisent à retenir la technique générale des Rendez-Vous comme peut l'introduire *Ada* pour les tâches (task). La structuration plus forte des objets en méthodes par rapport aux tâches *Ada* donne une expression particulière des rendez-vous dans le monde des objets que nous détaillerons. La figure 2.2 «Le tampon borné de caractères en ADA» donne un exemple de tâche *Ada* pour la conception d'un buffer.

```
task BUFFER is
    entry READ(C : out CHARACTER);
    entry WRITE(C : in CHARACTER);
end;
task body BUFFER is
    POOL_SIZE : constant INTEGER : 100;
    POOL      : array (1.. POOL_SIZE) of CHARACTER;
    COUNT     :INTEGER range 0.. POOL_SIZE := 0;
    IN_INDEX, OUT_INDEX : INTEGER range 1.. POOL_SIZE := 1;
 begin
    loop
       select
         when COUNT < POOL_SIZE =>
              accept WRITE(C : in CHARACTER) do
                   POOL(IN_INDEX) := C;
              end;
              IN_INDEX := IN_INDEX mod POOL_SIZE + 1;
              COUNT := COUNT +1;
         or when COUNT > O =>
              accept READ(C : out CHARACTER) do
                   C := POOL(OUT_INDEX);
              end;
              OUT_INDEX := OUT_INDEX mod POOL_SIZE +1;
              COUNT := COUNT -1;
         or terminate;
       end select;
    end loop;
```

figure 2.2 *Le tampon borné de caractères en ADA*

Trois autres dimensions, autres que celle de l'expression des rendez-vous, nous servent à classifier les techniques objets de synchronisation. Ce sont la communication synchrone ou asynchrone, les objets mono ou multi-programmés, la synchronisation éclatée ou centralisée.

- I - 2.1.3 La communication synchrone ou asynchrone

Dans la communication synchrone, sur une base procédurale, la synchronisation de l'appelant avec l'objet appelé est liée de manière unique à l'appel. L'appelant reste immédiatement en attente du résultat.

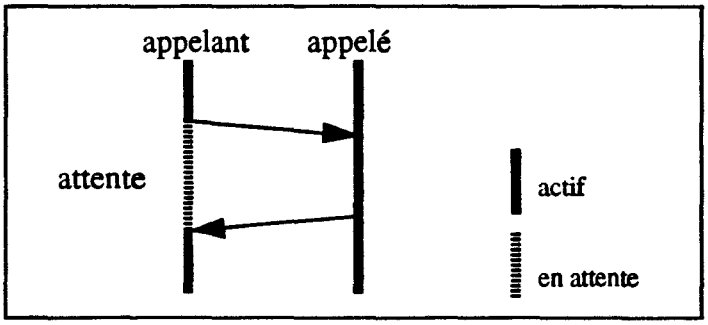


figure 2.3 *Communication synchrone*

Dans la communication asynchrone, basée sur l'envoi de message, la synchronisation de l'appelant avec l'objet appelé est liée au retour de résultat.

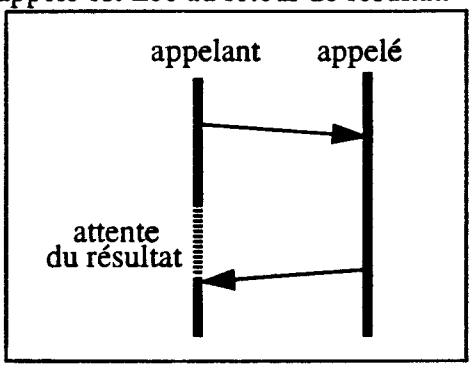


figure 2.4 *Communication asynchrone*

Ce type de communication, inconnu dans les langages séquentiels, utilise des outils particuliers pour la *resynchronisation* de l'appelant sur le résultat. Si l'appel explicite du résultat est l'outil le plus simple, on trouve des moyens plus évolués que sont les "*future*" variables d'*Abcl/1* et l'attente par nécessité de D. Caromel.

- I - 2.1.4 Les objets mono-programmés ou multi-programmés

De manière simple, on peut distinguer les objets ne pouvant contenir qu'une seule activité (objets mono-programmés) des objets pouvant contenir plusieurs activités concurrentes (objets multi-programmés) (voir - I - 1.2 «Le nombre possible de processus concurrents dans un objet»).

Dans le cas des objets mono-programmés, la synchronisation n'a pas à assurer la protection des données internes qui sont accédées de manière totalement sérialisée du fait du modèle. C'est le cas des langages *Pool, Hybrid, Eiffel Parallèle, Act++.*

Dans le cas des objets multi-programmés, la synchronisation doit aussi assurer ce rôle de protection. On pourrait ici imaginer revenir à des outils tels que les sémaphores ou moniteurs, mais pour les mêmes raisons que précédemment, des outils de plus haut niveau sont utilisés tels que les conditions d'ordonnancement de *Po*, les conditions d'activation de *Guide* ou les classes comportementales de *Dragoon...*

- I - 2.1.5 La synchronisation éclatée ou centralisée

On parle de synchronisation **éclatée** lorsque l'expression de la synchronisation est dispersée dans le texte source de l'objet. On dit par contre qu'elle est **centralisée** lorsque l'expression de la synchronisation n'apparaît que dans une partie bien définie du texte de l'objet.

La synchronisation éclatée pose un problème de conception et de réutilisation du source. En effet, la synchronisation étant répartie sur l'ensemble des méthodes, l'utilisateur doit avoir une vision de l'ensemble des méthodes pour connaître la synchronisation décrite sur l'objet. C'est par exemple le cas des langages *Pool et Hybrid.*

La synchronisation centralisée est utilisée en général pour ordonnancer les activités traversées par l'objet. Nous verrons par la suite que si l'expression de cet ordonnancement est abstraite, la synchronisation peut être réutilisable. La synchronisation centralisée est utilisée dans les langages *Eiffel Parallèle, Po, Guide, Dragoon.*

- I - 2.2 Classification des principaux langages

Ces quatre points nous permettent de présenter un ensemble représentatif de langages à objets introduisant la synchronisation dans les objets.

- I - 2.2.1 Pool

Pool est un langage développé par l'équipe de Pierre America aux laboratoires Phillips.

Expression des rendez-vous: l'instruction «Answer»

Le programmeur utilise l'instruction «*Answer*» pour sérialiser les différentes activités appelantes en définissant l'ordonnancement des exécutions de méthode à l'intérieur d'un objet. L'instruction Answer (liste de méthodes) indique qu'un rendez-vous est attendu avec un appelant d'une des méthodes de la liste.

Type de communication: synchrone

Dans *Pool_I* la communication asynchrone est introduite.

Type d'objet: objet mono-programmés

Le système limite à un le nombre de flots d'exécution dans un objet. Il n'y a donc pas de parallélisme intra-objet. Les méthodes s'exécutent en exclusion mutuelle. Les variables locales sont ainsi protégées des accès concurrents. Le programmeur ne doit pas gérer l'exclusion mutuelle, car elle est garantie par le modèle d'exécution.

Type de synchronisation: éclatée

L'instruction «Answer» peut apparaître à la fois dans les méthodes et dans le «body» qui décrit l'activité lancée à la création de l'objet. Si les «Answers» ne sont utilisés que dans le body, la synchronisation est centralisée.

Exemple

Nous prendrons l'exemple d'un tampon contenant un nombre limité d'éléments. Deux méthodes *put* et *get* sont définies sur l'objet tampon. La méthode *put* dépose un élément dans le tampon alors que la méthode *get* en extrait un. Dans l'implémentation du tampon en *Pool* (figure 2.5 «Le tampon en Pool-T»), une structure de donnée matérialise la structure du tampon. Ici, seule la description de la synchronisation nous intéresse.

```
IMPL UNIT buffer
VAR l:struc
METHOD put(i:item)
    BEGIN
    IF l!full THEN
        ANSWER(get)
    FI
    l!put_ele(i)
    RETURN SELF
END
METHOD get():item
    BEGIN
    IF l!empty THEN
        ANSWER(put)
    FI
    RETURN(l!get_elt())
END
BODY
    l!create
    DO
        ANSWER(put,get)
    OD
END
```

figure 2.5 *Le tampon en Pool-T*

Dans la méthode *put*, si le tampon est plein, la méthode est bloquée et l'objet n'accepte que l'exécution de la méthode *get*. Le body initialise la structure de donnée, puis autorise soit un dépôt, soit un retrait d'un élément.

NB: D'autres descriptions du tampon sont possibles en Pool. Dans l'exemple ci dessus nous avons privilégié l'utilisation des «Answers» dans les méthodes en éclatant la synchronisation. Les autres solutions seront abordées dans d'autres langages.

- I - 2.2.2 Hybrid

Hybrid est un langage de l'université de Genève développé par l'équipe O. Nierstrasz.

Expression des rendez-vous: les «delay-queues»

Type de communication: synchrone

Type d'objet: mono-programmé

Type de synchronisation: éclatée

Commentaires

La synchronisation dans *Hybrid* [Nierstrasz87] s'effectue en introduisant des files d'attente sur chaque méthode. Ces files d'attente sont contrôlées au niveau du langage et leurs programmations sont réparties dans le corps des méthodes. Des files d'attentes, appelées «**delay queues**», sont introduites pour synchroniser les activités. Ce mécanisme

permet de «**retarder**» les exécutions de méthodes sur un objet. La manipulation des *delay queues* permet de programmer explicitement l'exécution d'une méthode ou sa mise en attente.

Les manipulations sur les «delay queues» sont réduites à:

- Ouvrir une «delay queue» (instruction *open*): les appels en attente à la méthode associée à la delay queue sont acceptés et leurs exécutions possibles.

- Fermer une «delay queue» (instruction *close*): les appels à la méthode associée à la delay queue ne sont plus traités. Ils sont mis dans la file d'attente spécifique à la méthode.

Exemple

Nous reprenons l'exemple d'un tampon contenant un nombre limité d'éléments (tampon borné). La méthode *put* dépose un élément dans le tampon et la méthode *get* extrait un élément.

```
type Bounded_buf of (itemType, bounf :< enumType)
    abstract {
        put : item ->;
        get : -> itemType; }
private
    var  notFull <- open , notEmpty <- close: delay;
    var  buffer : array [bound] of itemtype ;
    var  getIndex <- bound.first, putIndex<- bound.first : bound;

put (item : itemType) ->;
    uses notFull ;
    {
    buffer [putIndex] := item;
    if (putIndex =? bound.last) {
                putIndex := bound.first ;
        } else{  putIndex.inc;
    }
    if (getIndex =? putIndex) {      # buffer plein
                notFull.close ;      # interdire le put car le buffer est plein
    }
    notEmpty.Open ;                  # autoriser le get
    return ;
}
get () -> itemType;
    uses notEmpty ;
    {
    var item: ItemType ;
    item := buffer [getIndex] ;
    if (getIndex =? bound.last) {
                getIndex := bound.first ;
        } else {  getIndex.inc;
    }
    if (getIndex =? putIndex) {      #buffer vide
                notEmpty.close;      # interdire le get car le buffer est vide
    }
    notFull.Open ;                   # autoriser le put
    return (item) ;
}
```

figure 2.6 *La classe «Bounded_buffer» écrite en Hybrid*

Lors du retrait du dernier élément du tampon, les exécutions de la méthode *get* doivent être retardées, la «delay queue» *not_empty* est alors fermée. Un dépôt dans le tampon entraîne l'ouverture de la «delay queue» *not_empty* permettant un éventuel retrait. Si le tampon est plein, la «delay queue» *not_full* est fermée.

- I - 2.2.3 Eiffel Parallèle (Caromel)

Eiffel Parallèle est une extension du langage séquentiel *Eiffel* au parallélisme réalisée par D. Caromel.

Expression des rendez-vous: l'instruction «Serve»

La méthode de nom «live» définie dans les classes d'objets actifs décrit le comportement de l'objet en spécifiant les actions à effectuer par l'objet lors de la réception d'une requête. Le «live» d'un objet fonctionne en serveur et définit la synchronisation de l'objet.

L'extension du langage fournit une série de primitives permettant de décrire le corps du serveur, en particulier l'instruction *Serve* qui exprime qu'un rendez-vous est possible avec une requête.

Type de communication: asynchrone (l'attente par nécessité)

Type d'objet: mono-programmé

Type de synchronisation: centralisée (le «live»)

Commentaire

L'attente par nécessité

L'attente par nécessité est introduite dans cette extension d'*Eiffel*. Elle est basée sur la réflexion suivante: L'attente du résultat d'un appel peut être retardée jusqu'à la première utilisation du résultat en lecture.

Par exemple:

v:= p.f(...); <l'appel asynchrone de la méthode *f* sur l'objet *p*>

...

<utilisation de v>

Au moment de l'appel de méthode, la variable *v* est alors considérée comme une variable *attendue* (non-présente). Le processus n'est bloqué que lorsque la variable v est utilisée et qu'elle est attendue (non-présente). Lorsque le résultat sera retourné la variable devient alors présente et le processus est débloqué.

L'attente par nécessité est une solution de re-synchronisation élégante qui évite les demandes explicites de résultat et les attentes non fondées. L'attente par nécessité est un outil évolué de programmation. Mais l'attente par nécessité doit être associée à d'autres mécanismes pour pouvoir préciser toutes les contraintes de synchronisation et définir ainsi le comportement d'un objet. Pour le problème du tampon, l'attente par nécessité comme les "*future*" variables n'apportent pas de solution.

Exemple

```
class UNBOUND_BUFFER[T]
export
    put,get
inherit
    LINKED_QUEUE[T]
    PROCESS
        redefine Live
feature
    Live is
     do
        from
        until I_am_alone
        loop
         Serve put
         if not_empty then
           Serve get
         end
        end
    end
```

figure 2.7 *Le tampon en Eiffel Parallèle*

Dans cet exemple, l'instruction *Serve put* teste si une requête de la méthode *put* est en attente, puis lance éventuellement son exécution (idem pour *Serve get*).

On notera ici l'héritage multiple utilisé à la fois pour hériter d'une réalisation du tampon (LINKED-QUEUE) et pour hériter des caractéristiques des objets actifs (PROCESS).

- I - 2.2.4 Act ++

Act ++ est une extension par Kafura du modèle d'acteur.

Expression des rendez-vous: les comportements abstraits

La synchronisation est définie à la déclaration de la partie «comportementale». Le programmeur définit l'ensemble des comportements abstraits de l'objet, états de l'objet vis à vis de la synchronisation, en indiquant pour chacun d'eux, une liste de méthodes pouvant être servies dans cet état.

Type de communication: synchrone

Type d'objet: mono-programmé

Type de synchronisation: centralisée dans la partie «behavior» mais partiellement éclatée avec l'instruction become

L'instruction de changement d'état des langages d'acteurs permet de spécifier le prochain état de l'objet.

Exemple

Le tampon borné (figure 2.8 «La classe «Bounded Buffer» écrite en Act++») programmé en *Act++* définit trois états pour le tampon. Lorsque le tampon est vide, l'objet n'accepte que les dépôts d'éléments; lorsque le tampon est plein, l'objet n'accepte que les retraits. Dans l'état intermédiaire où le tampon n'est ni plein ni vide, il peut accepter les dépôts et les retraits.

La méthode «*buffer*» initialise le tampon en le mettant dans l'état «empty_buffer» et change d'état à la fin de chaque méthode au fur et à mesure des dépôts et des retraits d'éléments.

```
class bounded_buffer : ACTOR {
    int_array buf[MAX];
    int in,out;
behaviour.
    empty_buffer        = {put()};
    full_buffer         = {get()};
    partial_buffer      = {get(),put()};
public:
 buffer()
    {
    in = 0; out = 0;
    become empty_buffer;
    }
 void put(int item)
    {
    buf[in++] = item;
    in %= MAX;
    if (in == out % MAX)
        become full_buffer;
    else
        become partial_buffer;
    }
 int get()
    {
    reply buf[out++];
    out %= MAX;
    if (in == out)
        become empty_buffer;
    else
        become partial_buffer;
    }
};
```

figure 2.8 *La classe «Bounded Buffer» écrite en Act++*

- I - 2.2.5 Abcl/1

Abcl/1 est le langage d'acteur développé par l'équipe d'Yonezawa

Expression des rendez-vous: l'instruction «Waits For»

L'instruction *«Waits For»* permet de forcer l'attente d'un type de requête avant l'exécution de la requête courante. Reprenons l'exemple du tampon borné, l'exécution de la méthode *put* lorsque le tampon est plein attend une requête *get*, l'exécute puis continue sa propre exécution.

Type de communication: synchrone et asynchrone (les variables "**future**")

Type d'objet: mono-programmé

Type de synchronisation: éclatée

Commentaires

Les variables "*future*"

Un objet peut déléguer la synchronisation à un objet spécialisé. Lors d'un appel de méthode en mode asynchrone, ces objets particuliers se substituent aux objets courants pour attendre les réponses de la requête. L'objet courant est ainsi libéré et pourra à sa volonté communiquer avec l'objet de synchronisation pour récupérer les éventuelles réponses. Ces objets sont des outils de programmation, mais ne peuvent exprimer toutes les contraintes de synchronisation sur les objets.

Les variables "*future*" définies dans *Abcl/1* sont des objets qui remplissent cette fonction. Elles attendent la ou les réponses après un envoi de message asynchrone.

Prenons un exemple:

T <= M $x

L'acteur courant envoie un message *M* à l'acteur *T*. La variable "*future*" de nom *x* recevra les résultats à la place de l'acteur émetteur. Dès l'envoi de message, l'acteur peut continuer son traitement. L'acteur consultera par la suite le contenu de la variable "*future*" x et pourra ainsi:

- demander si le résultat est retourné.
- extraire un ou plusieurs résultats.

NB: Dans Concurrent-Smalltalk, les Cbox/s permettent d'implémenter les mêmes fonctionnalités.

Exemple

```
[object Buffer
    (state declare-the-storage-for buffer)
    (script
        (=> [:put aProduct]
          (when the-storage-is-full
                (wait-for          ;;waits for a [:get] message.
                        (=>[:get]
                        remove-a-product-from-the storage-and-return-it)))
          store aProduct)
        (=>[:get]
          (when the-storage-is-empty
                (wait-for          ;;waits for a [:put ..] message
                        (=>[:put a Product]
                        send-aProduct-to-the-object-which-sent-[:get]-message))
            remove-a-product-from-the-storage-and-return-it)))]
```

figure 2.9 *La classe tampon écrite en Abcl/1*

- I - 2.2.6 Po

Po est un langage parallèle à objets développé à l'université de Bologne (Italie)

Expression des rendez-vous: les conditions d'ordonnancement

Type de communication: synchrone et asynchrone

Type d'objet: multi-programmé

Type de synchronisation: centralisée

Commentaires

Dans le langage Po, les objets sont actifs et le processus de l'objet, appelé «Scheduling Process» contrôle les invocations à l'objet. Les invocations sont traduites par des envois de message qui sont stockés à leur réception dans une file spécifique de l'objet destinataire. Le processus serveur traite ces messages, et peut lancer simultanément plusieurs traitements. Pour exprimer la synchronisation, chaque méthode est munie d'une condition d'ordonnancement dite «scheduling_cond». Le code du serveur évalue pour chaque méthode sa condition d'ordonnancement et lance le cas échéant l'exécution de la méthode invoquée. La condition est constituée d'une expression booléenne construite à partir des messages de requête en attente et des informations locales de l'objet. Par exemple, la condition peut tester l'existence d'un message de requête en attente

Exemple: Expression d'une condition d'ordonnancement d'une méthode de lecture dans un tampon borné:

```
METHOD: scheduling put RETURNS: <Boolean>
    (firstinqueue (msg | op = "put") and dim <= size)
                ^ ifTrue ([DEQUEUE(msg) EXECUTE(msg)])
END_METHOD
```

figure 2.10 *Condition d'ordonnancement de la méthode*

Dans ce même exemple du tampon, la condition d'ordonnancement de la méthode *get* s'exprimerait par (firstinqueue (msg | op = "get") and dim > 0). Elle filtre les messages en testant l'existence d'un message contenant une demande à la méthode de nom «*get*». La fonction *fisrtinqueue*() permet de rechercher le message le plus ancien vérifiant cette condition. La condition d'ordonnancement vérifie qu'il reste au moins un élément dans le tampon.

En fonction du résultat de la condition d'ordonnancement, l'utilisateur programme explicitement l'action devant être effectuée, appelée «consequent_action». Dans l'exemple précédent, si la condition est vérifiée, le serveur doit extraire le message de la file et exécuter la méthode.

Le système offre au programmeur un ensemble de primitives permettant d'exprimer les conditions d'ordonnancement:

ISINQUEUE(filtre)	teste l'existence d'un message vérifiant le filtre
FIRSTINQUEUE(filtre)	recherche le premier message vérifiant le filtre

D'autres primitives permettent d'exprimer les actions à effectuer par le serveur:

DEQUEUE(msg)	retire le message de la file
EXECUTE(msg)	exécute la méthode invoquée dans le message
SUSPEND(event)	met l'activité courante en attente sur un événement
ISWAITING(event)	teste si l'activité est en attente
CONTINUE(msg)	relance l'activité
ABORT(msg)	abandonne le traitement du message

La solution retenue dans le langage *Po* laisse au programmeur l'écriture du serveur. Il doit définir les conditions d'ordonnancement ainsi que les actions à effectuer pour chaque méthode sensible.

- I - 2.2.7 Guide

Guide est un système distribué à objets développé à l'université de Grenoble

Expression des rendez-vous: les conditions d'activation

Type de communication: synchrone

Type d'objet: multi-programmé

Type de synchronisation: centralisée

Commentaires

La synchronisation des activités dans le système s'effectue par un mécanisme de ***conditions d'activation*** des méthodes. La condition d'activation est attachée à une méthode de la classe et doit être satisfaite pour lancer une exécution de cette méthode. Les conditions utilisent l'état local de l'objet ainsi que les paramètres de l'invocation.

Les conditions d'activation sont des expressions booléennes calculées à partir d'un ensemble de compteurs de synchronisation définis pour chaque méthode d'un objet donné. Les compteurs de synchronisation sont similaires aux compteurs de Verjus [Robert77]. En voici la liste:

invoked(m)	nombre d'invocations de la méthode m (demande d'exécution)
started(m)	nombre d'invocations acceptées (non bloquées) de la méthode m
completed(m)	nombre de méthodes totalement exécutées
current(m)	nombre d'activités parallèles exécutant la méthode m
pending(m)	nombre d'activités bloquées sur l'invocation de la méthode m
current(m)	started(m) - completed(m)
pending(m)	invoked(m) - started(m)

Nb: Si une méthode n'a pas de condition d'activation, elle n'est pas contrôlée et tous ses appels sont immédiatement exécutés.

Le modèle défini dans le système *Guide* permet un parallélisme intra-objet. Pour protéger l'objet, les accès aux variables d'instance utilisent des méthodes particulières pouvant être, elles aussi, contrôlées par des conditions d'activation.

Exemple

Nous reprenons l'exemple du tampon borné; mais ici, le producteur peut déposer des éléments en même temps que le consommateur en retire.

```
CLASS Bounded_buffer IS
IMPLEMENTS ProducerConsumer
CONST size = <some constant>
buffer : Array[0..size-1] OF Element
first, last : Integer = 0;
METHOD Put (IN m : Element)
    BEGIN
    buffer[last]:=m
    last:= last +1 MOD size;
    END Put;
METHOD Get(OUT m:Element)
    BEGIN
    m:=buffer[first]
    first := first + 1 MOD size
    END Get
CONTROL
    Put: (completed(Put) - completed(Get) < size)
                AND current(Put) = 0
    Get: (completed(Put) > completed(Get) )
                AND current(Get) = 0
END
```

figure 2.11 *La classe «bounded_buffer» écrite avec Guide*

La partie «control» spécifie les contraintes de synchronisation de la classe. La méthode *put* ne peut être exécutée que s'il reste des cases libres dans le tampon; (quand la différence entre le nombre total de *put* exécutés et le nombre total de *get* exécutés est inférieure à la taille maximum du tableau (completed(Put) - completed(Get) < size)). Un dépôt dans le tampon s'effectue en exclusion mutuelle des autres dépôts (*current(Put) = 0*).

- I - 2.2.8 Dragoon

Dragoon est un langage parallèle à objets développé à Milan (Italie)

Expression des rendez-vous: les classes comportementales

Type de communication: synchrone et asynchrone

Type d'objet: multi-programmé

Type de synchronisation: centralisée

Commentaires

La solution retenue dans *Dragoon* utilise comme *Guide* les compteurs de synchronisation. Elle définit donc pour chaque méthode un ensemble de compteurs qui est associé à une instance de la classe.

Les classes comportementales

L'idée de *Dragoon* est de séparer les aspects fonctionnels d'un objet, du contrôle de son utilisation (synchronisation). Le langage définit des classes «unbehavioured» (séquentielles) et des classes «behavioural» (décrivant le comportement). Par héritage d'une classe «unbehavioured» et par utilisation d'une classe «behavioural», on obtient une classe «behavioured» qui possède les attributs et méthodes de la première et le comportement de la seconde.

Le programmeur définit un ensemble de classes comportementales qui décrivent des comportements. Il se constitue ainsi une bibliothèque d'abstractions qu'il pourra utiliser pour exprimer la synchronisation d'une classe.

Exemple

```
class BOUNDED_BUFFER is
    intoduces
        procedure       Put(ITEM: in SOME_TYPE);
        procedure       Get(ITEM: out SOME_TYPE);
        function        Not_Full()
end BUFFER;

class body BOUNDED_BUFFER is
    STORE:SOME_TYPE_STRUCTURE;
    procedure Put(ITEM: in SOME_TYPE)
        begin   ...     end;
    procedure Get(ITEM: out SOME_TYPE)
        begin   ...     end;
    function Not_Full()
        begin   ...     end;
    begin
        ...
    end;
end BOUNDED_BUFFER;
```

figure 2.12 *Interface d'une classe*

```
behavioural class PRODS_CONS is
    ruled POP, GOP, NOT_FULL;
    where
        per(POP) <=> active(POP) = 0 AND NOT_FULL;
        per(GOP) <=> act(POP) > act(GOP) AND active(GOP)= 0;
end PRODS_CONS;
```

figure 2.13 *La classe comportementale*

```
class PRODS_CONS_BUFFER is
    inherits BUFFER;
    ruled by PRODS_CONS;
    where
                Get:GOP;
                Put:POP;
                Not_Full:NOT_FULL;
end  PRODS_CONS_BUFFER;
```

figure 2.14 *La classe «bounded_buffer» écrite avec Dragoon*

La classe «*Bounded_Buffer*» définit l'interface de la classe du *tampon borné* et son «implémentation» est décrite dans la classe body «Bounded_Buffer». Le comportement de la classe est séparé de son «implémentation». La classe comportementale «*Prods_Cons*» décrit le comportement de type *producteur-consommateur*. Elle définit les contraintes de synchronisation en associant aux deux méthodes «virtuelles» une expression booléenne manipulant les compteurs de synchronisation. La classe «*Prods_Cons_Buffer*» définit la classe tampon régie par un comportement *de type producteur-consommateur* et réalisée par les méthodes de la classe «*Buffer*».

- I - 2.2.9 Tableau récapitulatif

	Expression de la synchronisation	Communication		Objet		Synchronisation	
		synchrone	asynchrone	mono-programmé	multi-programmé	éclatée	centralisée
Pool	l'instruction Answer	X	X	X		X	X
Hybrid	les delay queues	X		X		X	
Eiffel Parallèle	l'instruction Serve et l'attente par nécessité		X	X			X
Act++	les comportements abstraits	X		X		X	X
Abcl/1	l'instruction Waits For et les varainbles *future*	X	X	X		X	
Po	les conditions d'ordonnancement	X	X		X		X
Guide	les conditions d'activation	X			X		X
Dragoon	les conditions d'activation et les classes comportementales	X			X		X

figure 2.15 *Tableau récapitulatif*

- I - 2.3 L'héritage des contraintes de synchronisation

Le mécanisme d'héritage permet la réutilisation du code des sur-classes. Par héritage des méthodes, le programmeur espère récupérer les contraintes de synchronisation des sur-classes. Nous montrerons que l'héritage et la synchronisation sont souvent très délicats à combiner. Il est parfois difficile d'hériter de la synchronisation définie dans les sur-classes.

Nous étudierons l'héritage des contraintes de synchronisation dans les deux types de synchronisation introduits précédemment.

- I - 2.3.1 La synchronisation éclatée

Lorsque la synchronisation est éclatée dans l'ensemble des méthodes, l'héritage d'une classe entraîne celui de l'ensemble de la synchronisation de la sur-classe. Mais la modification des contraintes est alors délicate. Il est souvent difficile d'intégrer de nouvelles méthodes sans redéfinir les méthodes héritées pour intégrer la nouvelle synchronisation.

La synchronisation de bas niveau

La synchronisation par sémaphore ou moniteur peut être réutilisée dans les sous-classes. Mais la définition d'une nouvelle méthode à synchroniser peut modifier la synchronisation définie dans les sur-classes. L'héritage est difficile.

L'héritage d'un ordonnancement réparti

L'article de Kafura [Kafura89] montre que l'ajout d'une méthode dans une sous-classe peut entraîner la définition d'une nouvelle «file d'attente». Les méthodes définies dans une sur-classe ne peuvent contrôler cette «delay queue» puisque celle-ci n'est pas encore connue à la déclaration de cette sur-classe.

Le problème est similaire dans *Pool_T;* les instructions «Answer» nécessitent une liste explicite de méthodes acceptables et ne peuvent utiliser les méthodes des sous-classes non connues de la sur-classe.

L'héritage des contraintes réparties de synchronisation n'a donc pas de solution satisfaisante sans mécanisme de redéfinition de la synchronisation héritée.

- I - 2.3.2 La synchronisation centralisée

La synchronisation centralisée regroupe l'expression de la synchronisation dans une partie spécifique de la description de la classe. A priori, l'héritage des contraintes de synchronisation devrait s'avérer plus facile. Dans le cas le plus défavorable, la partie synchronisation devra être réécrite.

- I - 2.3.2.1 L'héritage d'une tâche de fond synchronisante

Dans les langages utilisant une tâche de fond synchronisante décrite par le programmeur, l'héritage semble difficile. Dans *Pool*, l'héritage et la synchronisation posent des problèmes d'incompatibilité. La tâche de fond «body» définie dans une classe ne peut prendre en compte les nouvelles méthodes introduites dans les sous-classes. Les contraintes de synchronisation établies par les instructions «answer/s» ne peuvent prendre en compte les nouvelles méthodes des sous-classes. P. America, concepteur du langage, développe ce point dans l'article [America87].

Une solution consiste à ne pas hériter du code de la tâche de fond et de redéfinir l'activité de fond de l'objet pour chaque classe; mais il n'y a pas réellement héritage (voir *Eiffel Parallèle*).

Les différentes versions de *Pool* privilégient soit l'héritage, soit le parallélisme et la synchronisation. La coexistence des deux mécanismes n'est pas résolue de façon satisfaisante dans Pool.

- I - 2.3.2.2 L'héritage des comportements abstraits

Le langage *Act++* autorise l'héritage. Les classes des objets héritent de la partie comportementale des sur-classes. Le programmeur a la possibilité de renommer et de redéfinir les comportements hérités. La prise en charge de la synchronisation des nouvelles méthodes définies dans une sous-classe est intégrée dans les redéfinitions des comportements.

Exemple:

```
class extended_buffer : public bounded_buffer {
behaviour:
    extended_empty_buffer = {put}
        renames empty_buffer;
    extended_full_buffer = {get(),get_rear()}
        redefines full_buffer;
    extended_partial_buffer = {get(),get_rear(),put()}
        redefines partial_buffer;
public:
extended_buffer()
    {
    in = 0; out = 0;
    become extended_empty_buffer;
    }
int get_rear()
    {
    reply buf[--in%MAX];
    if (in == out)
        become extended_empty_buffer;
    else
        become extended_partial_buffer;
    }
};
```

figure 2.16 *Extended_Buffer en Act ++*

L'exemple montre l'extension de la classe «bounded_buffer». Une nouvelle méthode *get_rear* extrait l'élément le plus récent déposé dans le tampon. Elle rentre en conflit avec la méthode *put*. La nouvelle classe «extended_buffer» hérite de la classe «bounded_buffer». Elle redéfinit ou renomme les comportements hérités en intégrant la nouvelle méthode *get_rear()*.

L'héritage n'est pas complet puisque les comportements abstraits des sur-classes sont redéfinis dans les sous-classes.

- I - 2.3.2.3 Les «Enabled Sets» de Tomlinson

Le prototype *Rosette* reprend les comportements abstraits définis dans *Act++*. La description d'une classe comprend une partie *comportement* qui définit l'ensemble des états possibles de l'objet avec la liste des méthodes ouvertes.

Manipulation des ensembles de méthodes permises

La solution d'*Act++* est critiquée par Tomlinson. La redéfinition des comportements dans une sous-classe doit explicitement reprendre la liste des méthodes définies dans les sur-classes. La solution proposée dans Rosette regroupe la liste des méthodes permises dans un ensemble appelé «Enabled Set». Des opérations ensemblistes sur les «Enabled Sets» sont définies par le langage. Par exemple, l'opération "+" construit un ensemble par union de deux autres.

Exemple:

```
(defObject FiniteBuf [Buf N]
    extends: Top
    (method (init lim)
        (next (empty) [buf [ ]] [N lim]))
    (local (empty)          (enable [put]))
    (local (full)           (enable [get]))
    (local (partial)        (+ (empty) (full)))
    (method (put item)
        (next
                (if buf full (full) else (partial))
                [buf (add item to_end_of buf)]))
    (method (get)
        (return first_item_in buf)
        (next
                (if buf empty (empty) else (partial))
                [buf (remote first_item_from buf)])))
```

figure 2.17 *La classe «Bounded_Buffer» écrite en Rosette*

Le prototype *Rosette* est développé en *Lisp*. Dans la partie comportement, l'ensemble des méthodes associées à l'état *partiel* est créé par union des ensembles *empty* et *full*.

L'héritage des «Enabled Sets»

L'héritage des classes dans le prototype *Rosette* intègre l'héritage des comportements introduits dans *Act++*. Contrairement à celui-ci, les comportements hérités ne sont pas redéfinis, mais étendus. Les comportements hérités à modifier sont déclarés dans les comportements de la sous-classe comme une extension de la sur-classe. Les nouvelles

méthodes «activables» sont alors ajoutées à l'«Enabled Set» de la sur-classe sans spécifier de nouveau la liste des méthodes activables développées dans la sur-classe. Les opérations d'union définies dans les sur-classes répercutent l'ajout des nouvelles méthodes dans les ensembles. Exemple:

```
(defObjetct LifoBuf []
    extends: [b FiniteBuf]
    (local (full)
        (+ (-> b full) enable [get_rear]))
    (method (get_rear)
    (return last_item_in buf)
        (next
                (if buf empty (empty) else (partial))
                [buf (remote last_item_from buf)])))
```

figure 2.18 *La classe «Lifo_buffer» écrite en Rosette*

Cet exemple montre une sous-classe de «FiniteBuf». La sous classe «LifoBuf» définit une nouvelle méthode *get_rear* qui extrait l'élément le plus récent du tampon. La variable *b* introduit un lien local sur la super-classe. Le comportement *full* est associé grâce à cette variable *b* à l'état *full* de la sur-classe (FiniteBuf). La nouvelle méthode *get_rear* est ajoutée à l'«Enabled Set» *full* de la classe «FiniteBuf». L'état *partiel* défini dans la sur-classe comme union des ensembles de *empty* et *full* récupère indirectement la méthode *get_rear*.

Les ensembles de comportements

Suite aux travaux de Tomlinson, Kafura a introduit cette notion d'ensemble dans le langage Act++[Kafura91]. Les ensembles de comportements bénéficient de la flexibilité de la manipulation d'ensemble et de l'abstraction des comportements.

- I - 2.3.2.4 L'héritage des conditions d'ordonnancement

Dans les langages permettant le parallélisme intra-objet, l'héritage des classes doit intégrer l'héritage des contraintes de synchronisation de l'objet.

Le langage *Po* permet l'héritage multiple des classes. Le code du processus du serveur évalue les conditions d'ordonnancement des méthodes locales puis des méthodes héritées.

Une méthode héritée doit redéfinir sa condition d'ordonnancement. L'introduction d'une nouvelle méthode dans une sous-classe oblige à réécrire les conditions d'ordonnancement des méthodes définies des sur-classes.

- I - 2.3.2.5 L'héritage des conditions d'activation

Les conditions d'activation sont utilisées pour synchroniser les objets multi-programmés. Le système *Guide* permet l'héritage des classes. Les contraintes de synchronisation décrites dans les classes sont par défaut héritées avec les méthodes de la classe. Mais il y a association dynamique de la condition d'activation et de la méthode. Les contraintes héritées peuvent être redéfinies dans les sous-classes indépendamment des méthodes (la condition est alors surchargée).

Dans le langage *Dragoon*, l'héritage des contraintes est inhérent au langage puisqu'une classe comportementalisée doit hériter d'une classe séquentielle «unbehavioured». Une restriction existe dans le langage: une classe ne pouvant en effet pas hériter d'une déjà comportementalisée. Une classe comportementalisée possède déjà ses propres contraintes de synchronisation, son héritage ne pourrait intégrer les contraintes liées aux nouvelles méthodes définies dans la sous-classe. La synchronisation est introduite au niveau bas de la hiérarchie dans les classes comportementalisées. Il n'y a pas à proprement parler d'héritage des contraintes de synchronisation.

- I - 2.3.2.6 Extension des ensembles de Tomlinson

J.P Bahsoun et L. Feraud [Bahsoun91] proposent une extension des ensembles de Tomlinson dans un environnement d'objets multi-programmés.

La synchronisation est exprimée dans une partie spécifique du code. Elle regroupe:

- les contraintes de synchronisation pour chaque méthode,
- un ensemble de variables utilisées par la synchronisation,
- les préconditions pour chaque méthode.

Lorsqu'une condition d'activation est vérifiée, le système exécute la précondition avant d'exécuter la méthode. Le code de la précondition modifie les variables de synchronisation et donc l'état de l'objet. A l'exécution de la méthode, le système peut transmettre à la méthode des paramètres de synchronisation calculés dans les préconditions.

```
buffer class
exports PUT(X:in INTEGER), GET(X: out INTEGER)
feature
    T:Array(0..N-1)
    PUT: procedure(X: in INTEGER) (I: SYNCHO 1..N) is
        T(I):=X;
    end enqueue
    GET:procedure(X: out INTEGER) (I:SYNCHRO 1..N) is
        X:=T(I)
    end dequeue
constraints
    HEAD:0..N-1:=0
    TAIL:0..N-1:=0
    COUNT:0..N:=0;
    for put(I:SYNCHRO 0..N)
        COUNT < N --> HEAD := (HEAD+1) mod N;
                      I:= HEAD
                      COUNT := COUNT +1
    for get(I:SYNCHRO 0..N)
        COUNT > 0 --> TAIL := (TAIL+1) mod N;
                      I:=TAIL
                      COUNT:= COUNT -1
```

figure 2.19 *Tampon borné*

Dans cette description du tampon, un dépôt est associé à la condition d'activation: reste-t- il de la place? (COUNT <N). Dans la précondition de la méthode de dépôt, la nouvelle position dans le tableau est calculée par rapport aux variables de synchronisation (ici la valeur de HEAD).

L'héritage des classes introduit est inspiré de Tomlinson. Le principe est d'ajouter de nouvelles contraintes à celles déjà héritées. Le système utilise la notion de foule «crowd» introduit par [Hewitt79] qui dénombre le nombre d'appels (ici de méthodes en cours d'exécution).

Voici l'extension du tampon avec une nouvelle méthode de retrait. Elle utilise la classe buffer_mutex qui est l'extension de la classe tampon avec exclusion mutuelle des exécution de méthodes.

```
New_Buffer Class
exports PUT(X:in INTEGER), GET(X: out INTEGER),GET_REAR(X:out INTEGER)
inherit buffer_mutex
feature
    GET_REAR:procedure(X: out INTEGER) (I:SYNCHRO 1..N) is
        X:=T(I)
    end dequeue
constraints
    for GET_REAR(I:SYNCHRO 0..N)
        not COUNT = 0 -->       I := HEAD;
                                HEAD:= (HEAD-1) mod N;
                                COUNT := COUNT -1
    for PUT: *empty_crowd (GET_REAR)            -- exclusion with GET_REAR
    for GET: *empty_crowd (GET_REAR)            -- exclusion with GET_REAR
    for POP: empty_crowd (PUT) * empty_crowd(GET)
                                    -- exclusion with PUT and GET
```

figure 2.20 *Tampon étendu*

Dans cet exemple, la primitive booléenne *empty_crowd(M)* teste s'il existe des exécutions en cours de la méthode M.

- I - 2.3.3 Les Abstractions pour la synchronisation

L'architecture logicielle Pvc (**P**rocesseur **V**irtuel de **C**lasse) est un support de langages parallèles à objets [Courtrai91a]. Les objets Pvc sont actifs et multi-programmés; leur programmation nécessite donc la définition d'une synchronisation. Celle ci est basée sur **les conditions d'activation** et **les compteurs de synchronisation** [Courtrai91b,Geib92]. Plutôt que de définir des conditions et compteurs sur chaque méthode, nous définissons des ensembles de méthodes (appelés **abstractions pour la synchronisation**). Cette extension permet une meilleure abstraction de l'expression des contraintes de synchronisation et augmente la réutisabilité des ces contraintes dans un contexte d'héritage des classes. La synchronisation est centralisée pour être «réutilisable».

- I - 2.3.3.1 Les Abstractions

Nous avons choisi d'exprimer la synchronisation par les conditions d'activation avec des compteurs de synchronisation. Une abstraction est constituée d'un ensemble de conditions d'activation et d'un ensembles de méthodes. Cet ensemble de méthodes correspond à un élément de synchronisation de l'application. Les compteurs de synchronisation s'appliquent sur des ensembles de méthodes. Les valeurs des compteurs correspondent au cumul des valeurs des compteurs appliqués à chaque méthode de l'ensemble.

Soit *e* un ensemble de méthodes d'un objet, nous utilisons les compteurs suivants:

act(*e*) : nombre d'activations des méthodes de *e*
(mis à jour au lancement d'une méthode de *e*)

fin(*e*) : nombre de terminaisons des méthodes de *e*
(mis à jour à la fin de l'exécution d'une méthode de *e*)

req(*e*) : nombre de requêtes en attente des méthodes de *e*
(mis à jour à la réception d'un message pour une méthode de *e*)

On peut définir simplement le nombre d'exécutions en cours de l'ensemble *e* par

active(*e*) = act(*e*) - fin(*e*)

Par exemple, une contrainte d'exclusion mutuelle sur une méthode *m* peut s'exprimer de la manière suivante :

mutex.set(m); /*définition d'un ensemble contenant *m*/

mutex.all.set(active(mutex)=0); /*définition d'un ensemble de conditions d'activation*/

L'opération **set** construit un ensemble de méthodes associé à une abstraction. L'opération **all** définit les conditions d'activation qui s'appliqueront sur toutes les méthodes de l'ensemble.

Une méthode *m* peut être incluse dans plusieurs ensembles de conditions d'activation, l'exécution de *m* n'est alors possible qu'après l'examen de toutes les conditions contenues dans les ensembles possédant la méthode *m*.

Nous présentons maintenant la classe Tampon écrite dans notre langage intermédiaire et plus particulièrement la partie *contrôle* où est exprimée la synchronisation.

```
/*                    Classe Buffer                    */
CLASS(BUFFER):T
/*                    Variables d'instances            */
tab          :ARRAY[T];
/******              Methodes                 *******/
/*           dépot d'un élément dans le tampon         */
METHOD(put,elt:T)
... END;/*buffer_put*/
/*           retrait d'un élément du tampon            */
METHOD(get):T
...END;/*buffer_get*/
/******              partie Contrôle          *******/
CONTROL
    Deposit.set (put);
    Withdraw.set (get);
    Mutex1.set (put);
    Mutex2.set (get);
    Deposit.all.set( act(Deposit) - fin(Withdraw) > tab.size );
    Withdraw.all.set(fin(Deposit) > act(Withdraw) );
    Mutex1.all.set(active(Mutex1) = 0);
    Mutex2.all.set(active(Mutex2) = 0)
END
END
```

figure 2.21 *Le Tampon borné*

Quatre abstractions de synchronisation sont utilisées: *Deposit* et *Withdraw* définissent la synchronisation comportementale de la classe et *Mutex1*, *Mutex2* représentent la synchronisation interne.

- I - 2.3.3.2 L'héritage des abstractions

La synchronisation dans Pvc reprend l'idée d'ensembles de Tomlinson (voir la section - I - 2.3.2.3 «Les «Enabled Sets» de Tomlinson»), appliquée aux compteurs de synchronisation et aux conditions d'activation. Deux types d'ensembles sont manipulables par des opérations ensemblistes: les ensembles de méthodes qui représentent des abstractions de synchronisation, et les ensembles de conditions d'activation attachés aux méthodes.

Le cadre de notre étude est restreint à un langage à objets avec parallélisme et synchronisation, mais sans possibilité de redéfinition ni de renommage[1].

Ces ensembles de conditions d'activation vont naturellement pouvoir être modifiés ou étendus lors de l'héritage. Nous étudions maintenant les principales possibilités d'héritage de contraintes de synchronisation.

Spécialisation par héritage

Reprenons l'exemple du buffer ; nous voulons spécialiser ce buffer en ajoutant une priorité des retraits sur les dépôts. Une manière simple de faire cela est de compléter l'ensemble des conditions d'activation associé aux méthodes de dépôt, en indiquant qu'il ne doit pas y avoir de requête de retrait en attente. Pour éviter les interblocages, la condition doit permettre le *dépôt* d'un élément si le tampon est vide et cela même s'il y a des demandes de *retraits* en attente. La description des contraintes se fait par héritage de la manière suivante :

```
CLASS(PRIORITY_BUFFER)
    INHERIT BUFFER
    CONTROL
        Deposit.all.append (req( Withdraw )=0 OR (fin(Deposit) = act(Withdraw))
    END
END
```

figure 2.22 *La classe Tampon avec priorité*

Nous avons donc ajouté une contrainte de synchronisation sur l'abstraction *Deposit* représentant les dépôts. L'instruction **append** ajoute une nouvelle condition d'activation dans un ensemble hérité des conditions d'activation.

L'ajout de méthodes dans une sous-classe

L'ajout d'une méthode lors de l'héritage, entraînera l'intégration de cette nouvelle méthode dans les différents ensembles de méthodes héritées déjà définis et/ou la définition de nouveaux ensembles de conditions d'activation. Cette modification peut être importante ; les abstractions de synchronisation (ensembles de méthodes) aident alors à l'exprimer.

1. Une seconde étude plus complète a été réalisée dans le cadre d'un stage de D.E.S.S. [Namyst92].

Introduisons une méthode *getrec* (retrait de l'élément le plus récent) du tampon initialement défini. Cette méthode entre en compétition avec la méthode *put* (dépôt) puisque les deux opérations s'effectuent sur la même extrémité du tampon.

- Cette méthode doit donc s'intégrer dans *Mutex1* qui représente le besoin d'exclusion mutuelle sur l'extrémité concernée du tampon, et dans *Withdraw* qui représente de façon abstraite les opérations de retraits.

- Puis l'ensemble des conditions d'activation pour la nouvelle méthode peut être défini à l'aide de ces ensembles étendus.

Cela donne:

```
CLASS (EXTENTED_BUFFER)
    INHERIT BUFFER
    METHOD (getrec)
        ...
    END;        /*getrec*/
    CONTROL
        Withdraw.append (getrec);
        Mutex1.append (getrec)
    END
END
```

figure 2.23 *La classe Tampon étendu*

Définition de nouvelles abstractions

L'ajout d'une nouvelle méthode peut entraîner la définition de nouvelles abstractions de synchronisation.

A partir du tampon initial, introduisons maintenant une méthode *size* qui retourne le nombre d'éléments dans le tampon. Cette méthode de "lecture" doit être différenciée de celles d'"écriture" pour un besoin d'accès exclusif au nombre d'éléments. Cela s'exprime de la manière suivante, en introduisant deux nouveaux ensembles *Writer* et *Reader :*

```
CLASS (SIZED_BUFFER)
    INHERIT BUFFER
    METHOD (size)
        ...
    END;        /*size*/
    CONTROL
        Writer.union(Deposit,Withdraw);
        Reader.set(size);
        Writer.all.set(active(Reader)=0);
        Reader.all.set(active(Writer)=0)
    END
END
```

figure 2.24 *La classe Tampon avec comptage*

L'opérateur **union** fusionne plusieurs ensembles déjà définis. L'abstraction *Writer* réutilise les ensembles hérités d'abstraction; elle représente toutes les méthodes déjà définies qui modifient le tampon: les méthodes de *depôt* et de *retrait*. Les deux ensembles *Writer* et *Reader* définissent une synchronisation qui permet l'exclusion mutuelle des exécutions de méthode entre les deux ensembles de méthodes.

L'héritage multiple

L'héritage des classes peut être multiple. Dans ce cas les contraintes sont héritées de différentes branches d'héritage. Nous pouvons ici simplement cumuler les trois extensions précédentes par simple héritage multiple.

```
CLASS(PRIORITY_SIZED_EXTENTED_BUFFER)
    INHERIT PRIORITY_BUFFER
    INHERIT SIZED_BUFFER
    INHERIT EXTENTED_BUFFER
END
```

figure 2.25 *La classe Priority_Sized_Extented_Buffer*

Il n'y a pas lieu ici de modifier les conditions d'activation, le mécanisme d'héritage créera ces conditions à partir de celles héritées. L'héritage répété qui apparaît ici est pris en charge par le mécanisme d'héritage. La synchronisation de la classe *Buffer* héritée par plusieurs chemins d'héritage n'est étendue qu'une seule fois.

- I - 2.4 Conclusion sur «Objets et Synchronisation»

Comme tout langage parallèle, un langage parallèle à objets doit permettre l'expression de la synchronisation voulue. Ici la synchronisation doit être exprimée dans les classes d'objets et être plus particulièrement liée aux invocations de méthodes. Dans les différentes versions de *Parallel Eiffel*, J.F. Colin et C. Gransart ont développé des outils de synchronisation de bas niveau qui leur ont permis de reconstruire les principales synchronisation (éclatée avec des conditions d'activation ou centralisée avec un «body»).

La synchronisation décrit souvent un ordonnancement des méthodes en associant des conditions d'activation aux méthodes. Cette solution permet de contrôler les objets multi-programmés.

Pour des raisons de lisibilité et de réutilisabilité, les contraintes de synchronisation doivent être centralisées dans une partie spécifique, dite de *contrôle*, d'une classe. Le langage *Dragoon* permet même d'extraire cette partie de la classe grâce aux classes comportementales. Les contraintes de synchronisation sont avantageusement appliquées sur un ensemble de méthodes (Tomlinson). De même, les contraintes de synchronisation sont avantageusement réunies en ensemble de contraintes (Feraud). Cela permet une meilleure manipulation de ces contraintes par des opérations ensemblistes lors de l'héritage.

Nous avons introduit un héritage des contraintes de synchronisation basé sur les ensembles et sur les abstractions de synchronisation. Nos abstractions et leur programmation incrémentale par l'héritage expriment naturellement les contraintes de synchronisation comportementale et interne sur les objets. Le mécanisme ensembliste fournit un haut degré de réutilisabilité et résout une grande partie des problèmes d'héritage des contraintes de synchronisation. Nous devons maintenant étudier les problèmes inhérents aux redéfinitions et aux renommages des méthodes et proposer des outils de synchronisation plus évolués que les compteurs de synchronisation.

- I - 3. Objets et Répartition

Dans la partie - I - 1. «Objets et Parallélisme» nous avons présenté différents modèles de langages parallèles à objets. L'objectif de ces langages est de pouvoir décrire facilement les applications naturellement parallèles en incluant des entités actives coopérant à la réalisation de ces applications. Un autre objectif tout aussi naturel et prédominant est l'exploitation des machines parallèles, c'est à dire les machines pouvant relayer de façon physique le parallélisme introduit dans le langage.

Ces machines se caractérisent par une multiplication des ressources calcul (les processeurs), des ressources mémoire et des ressources de communication. Elles nécessitent toutes une distribution des modèles d'exécution sur ces différentes ressources. Nous nous proposons de présenter ici les problèmes rencontrés et les solutions utilisées dans les systèmes existants.

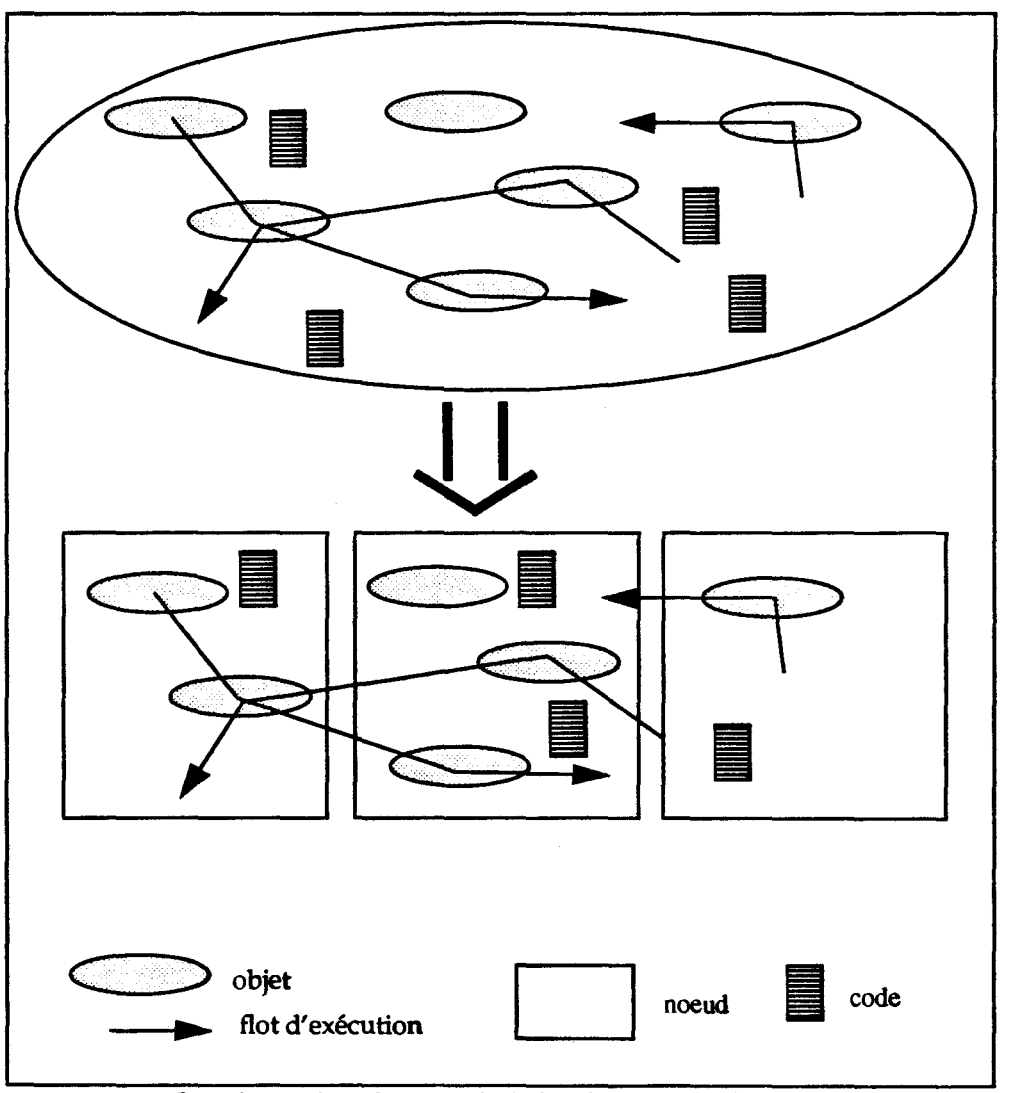


figure 3.0 *La répartition des entités à l'exécution sur la machine cible*

Le modèle d'exécution séquentiel

Pour tous les langages à objets, on trouve à l'exécution deux grandes composantes:

- 1 - Les objets, reflets de l'état de l'application à un moment donné.

- 2 - Une structure de données représentant les classes utilisées dans l'application. Ces classes renferment le code de l'application.

A partir de là, l'exécution se déroule comme un enchaînement d'appels procéduraux de routines (ou méthodes) sur les objets. Un tel appel d'une méthode sur un objet se déroule lui-même en deux phases.

- 1 - Une phase de «**method lookup**» qui recherche dans la structure des classes le code correspondant à l'appel, et cela en fonction du type (ou classe d'instanciation) de l'objet. Le «lookup» utilise le lien existant entre un objet et sa classe d'instanciation pour accéder à la structure des classes.

- 2 - Une phase d'**exécution** du code sélectionné

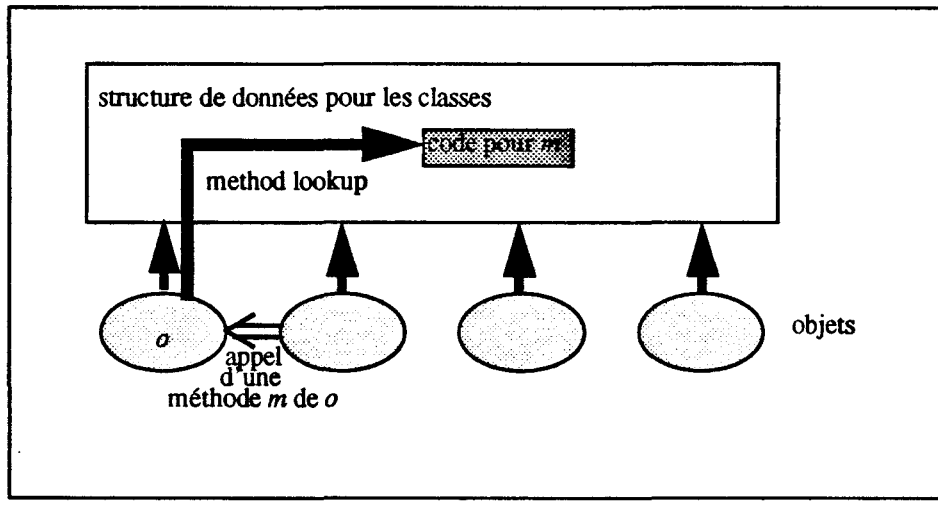


figure 3.1 *Le modèle d'exécution*

Ce schéma caractérise le modèle d'exécution des langages à objets et c'est ce schéma qui doit être distribué sur les architectures parallèles. Nous mettons à part immédiatement le cas des multi-processeurs (ou shared memory parallel architecture) formés de plusieurs processeurs connectés à une unique mémoire commune. Ici le modèle séquentiel ne subit pas de modification importante, seules les exécutions de méthodes peuvent être prises en charge en parallèle par les processeurs. C'est le cas du langage *Presto*, où le problème de répartition des tâches est résolu à l'aide d'objets «processeur» et d'un objet «ordonnanceur».

On remarque ici que la mémoire commune fortement sollicitée reste un goulot d'étranglement limitant le parallélisme réel.

Architectures sans mémoire commune

Si l'on tente d'implanter les objets et les structures des données des classes sur une architecture sans mémoire commune, plusieurs problèmes peuvent se poser:

- 1 - Les pointeurs utilisés sur une architecture à mémoire commune doivent maintenant être manipulés avec précaution. En mémoire distribuée, la portée restreinte d'un pointeur, limitée à la mémoire d'un noeud, nécessite des mécanismes plus évolués de désignation d'objets.

- 2 - La répartition des objets sur le noeud de la machine cible doit garantir le bon fonctionnement du modèle d'exécution. Il faut éviter les cas de «défaut de localité» où l'objet et le code de la méthode ne se trouvent pas sur le même noeud. Si le modèle d'exécution ne garantit pas l'absence de défaut de localité, le système doit mettre en place d'autre mécanismes spécialisés telle la migration.

Il existe deux approches possibles, utilisées dans différentes implantations de langages parallèles à objets:

- 1 - La duplication de la structure de données des classes sur les différents sites.

- 2 - L'éclatement de la structure de données des classes sur les différents sites.

Nous allons détailler ces deux solutions.

- I - 3.1 Duplication sur différents sites

La première approche consiste à dupliquer l'environnement complet des classes (structure de données des classes) sur plusieurs machines. L'ensemble du code des classes est copié sur les différents sites et un réseau de communication est mis en place entre les sites. Le système établit les communications entre environnements de sites différents.

Le code des méthodes est alors dupliqué sur des différents sites du réseau, mais les instanciations s'effectuent sur l'un ou l'autre des sites. La création peut s'effectuer à distance créant ainsi une référence inter-système.

Le problème est d'intégrer dans un système des références sur des objets distants. Ces objets doivent être manipulés à distance.

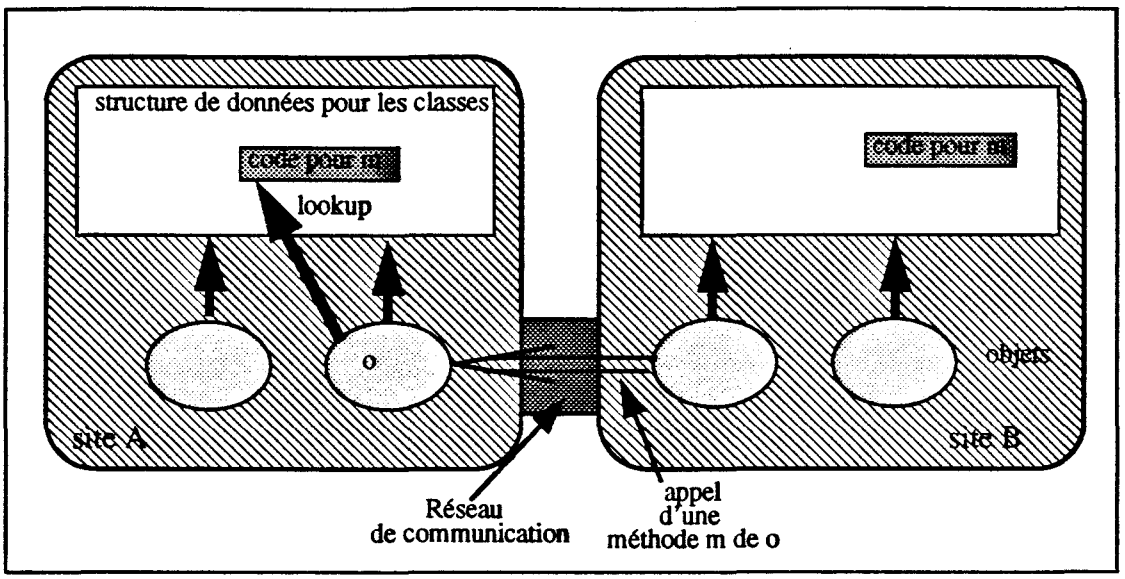


figure 3.2 *Duplication sur plusieurs sites*

Dans une version réseau de *Pool-T* , l'ensemble du code compilé est dupliqué sur les différentes machines du réseau. Le système gère les communications entre objets distants et régule la charge des noeuds lors des créations d'objets.

On trouve différents essais de coopération entre plusieurs machines *Smalltalk* dans un univers réparti (Bennet, Schelvis). L'architecture cible est ici un réseau de stations de travail. Chaque site possède son environnement *Smalltalk* complet. Le but des travaux est de partager entre les sites quelques objets en gardant la cohérence des activités locales (associées aux utilisateurs *Smalltalk* des différentes machines du réseau). *Smalltalk* se divise en deux parties principales: La *machine virtuelle* composée de l'«interprète», des primitives *smalltalk* et du gérant mémoire. L'*image Smalltalk* regroupant les différents objets du système au fur et à mesure de leurs instanciations. Les deux solutions présentées ci-après modifient pour l'une l'*image Smalltalk* sans toucher à la *machine virtuelle* et pour l'autre la machine virtuelle.

- I - 3.1.1 Les objets fantômes

Distributed Smalltalk de Bennet fait coopérer plusieurs *Smalltalk* sur un réseau sans modifier la machine virtuelle. La hiérarchie de classe est propre à un site et à un usager. Les deux *Smalltalk* restent indépendants. Une classe et toutes ses instances sont localisées sur le même site et le «method lookup» est réalisé localement à un site.

Un objet peut désigner un autre qui réside sur un site différent. Les objets distants ne sont connus d'un site qu'après des opérations spécialisées (création d'objets distants, nommage d'un objet distant). Pour manipuler un objet distant, l'objet-client sollicite un «**objet-proxy**»[Decouchant86], représentant l'objet distant sur le site courant. A la création d'un objet distant le système envoie un message de création au site distant. Celui-ci retourne le nom de l'instance créée sur le site distant. Le site local crée un objet proxy associé contenant le nom de l'objet distant ainsi que le nom du site propriétaire de l'objet.

L'objet proxy s'intègre aux autres objets locaux du site et reçoit toutes les invocations de méthode destinées à l'objet distant. Le proxy transforme un message *Smalltalk* en un réel messages qu'il redirige vers le site propriétaire de l'objet (figure 3.3 «Objet proxy dans Distributed Smalltalk»).

Du point de vue de l'objet-client, le proxy masque le réseau. La solution du proxy est courante dans les implantations réparties de *Smalltalk* [McCullough87].

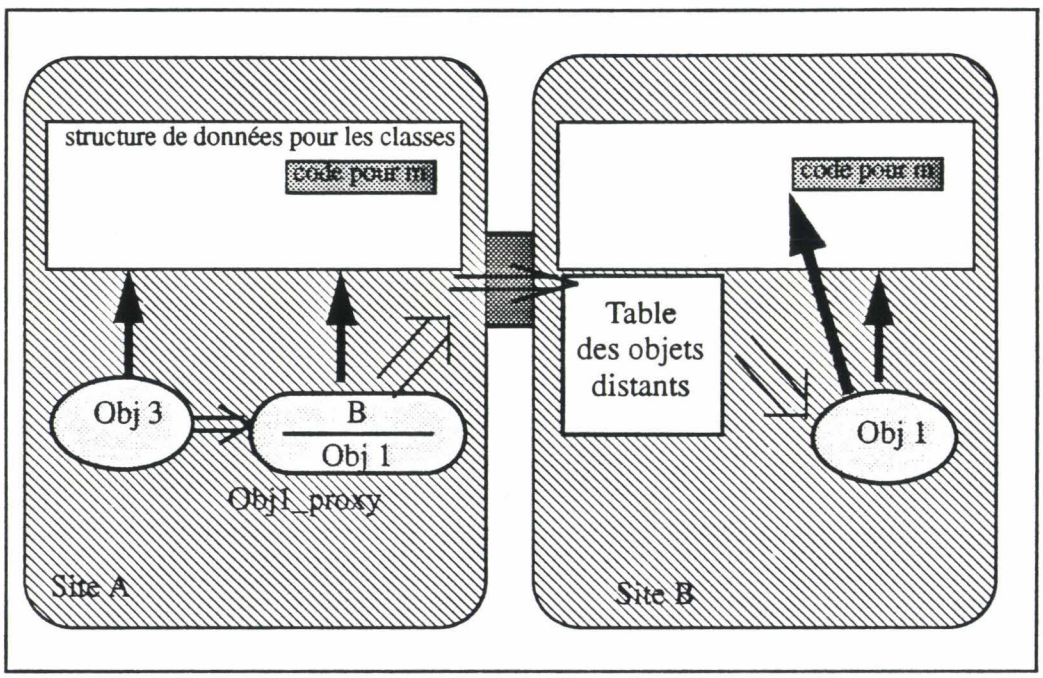


figure 3.3 *Objet proxy dans Distributed Smalltalk*

Dans cet exemple, le site *A* peut accéder à l'objet obj1 du site *B*. Cet objet a un objet proxy associé sur le site *A* contenant le nom et le site propriétaire de l'objet. Les messages arrivant aux objets proxy du site *A*, sont redirigés vers le site *B*. Le système du site *B*, grâce à la table des objets distants, retrouve l'objet invoqué, transforme le message Smalltalk et le lui transmet. La réponse de la méthode invoquée parcourt le chemin inverse.

- I - 3.1.2 Gestion de copies d'objets

Une autre version répartie de *Smalltalk*, appelée elle-aussi *Distributed Smalltalk*, [Schelvis88] modifie la machine virtuelle Smalltalk. L'image reste inchangée et ne perturbe pas les applications futures. La machine virtuelle modifiée est chargée de rendre transparentes les questions de localisation des objets en traitant les accès distants aux objets et en gérant un mécanisme de copies des objets.

Le gérant d'objets du système maintient un espace mémoire spécifique «ReplicaSpace» contenant les objets qui sont copiés à raison d'un exemplaire sur chaque site. La cohérence des différentes versions est maintenue par le gérant d'objets.

Nb: A chaque activité est associé un «host» correspondant au site propriétaire de cette activité (associé à l'utilisateur). Des objets particuliers, les objets «home» sont attachés à un site physique (correspondant aux objets I_O: écran, souris, contrôleur... locaux à chaque site). A la réception d'un message par un «home» objet, celui-ci recherche l'activité associée au message et redirige le cas échéant ce message vers le site propriétaire de l'activité.

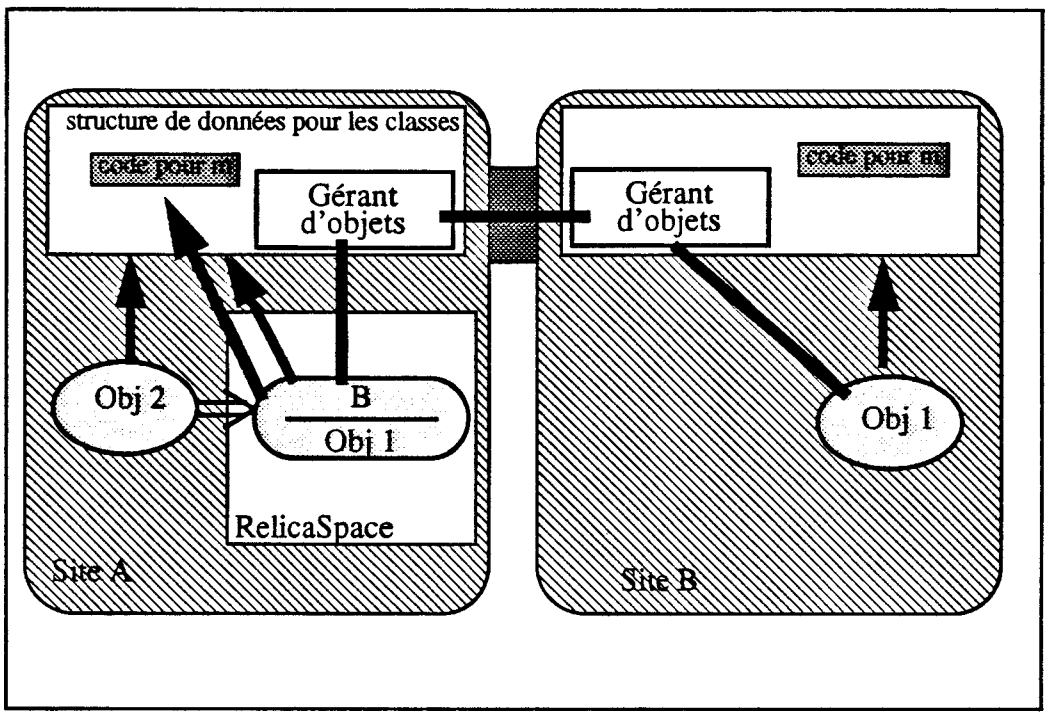


figure 3.4 *Copie d'objet dans Distributed Smalltalk*

Dans cet exemple, l'objet obj1, est dupliqué sur le site distant *A* dans les emplacements spécialisés des différents sites. L'objet dupliqué s'intègre aux objets locaux du site et peut donc communiquer avec eux. Le «method lookup» est réalisé localement et l'exécution de la méthode s'effectue sur la copie de l'objet.

Une modification d'une variable d'instance d'un objet (exemple obj1) est récupérée par le gérant d'objets local (site *A*) qui la communique aux autres gérants d'objets distants (ici le site *A*) par diffusion. Chaque gérant d'objets met à jour la copie de l'objet modifié.

Dans ces deux versions distribuées de *Smalltalk*, le système distribué privilégie les traitements locaux. Les liaisons avec les objets d'un autre site pénalisent le système (gestion d'une table ou gestion des copies). La gestion de copies distribuées oblige des protections supplémentaires et la cohérence des copies doit être conservée. Dans les deux systèmes présentés, l'objectif est de faire coopérer deux environnements complets prévus pour une architecture mono-processeur. Dans un contexte fortement distribué, le modèle d'exécution doit être remis en cause.

- I - 3.2 Eclatement sur différents sites

Une seconde approche du problème consiste à éclater la structure de données des classes sur le réseau. Le code des méthodes est alors dispersé sur l'ensemble du réseau. Cette approche limite les copies de code.

Le problème est de garantir lors de l'exécution d'une méthode que l'objet cible et le code de la méthode sont sur le même noeud.

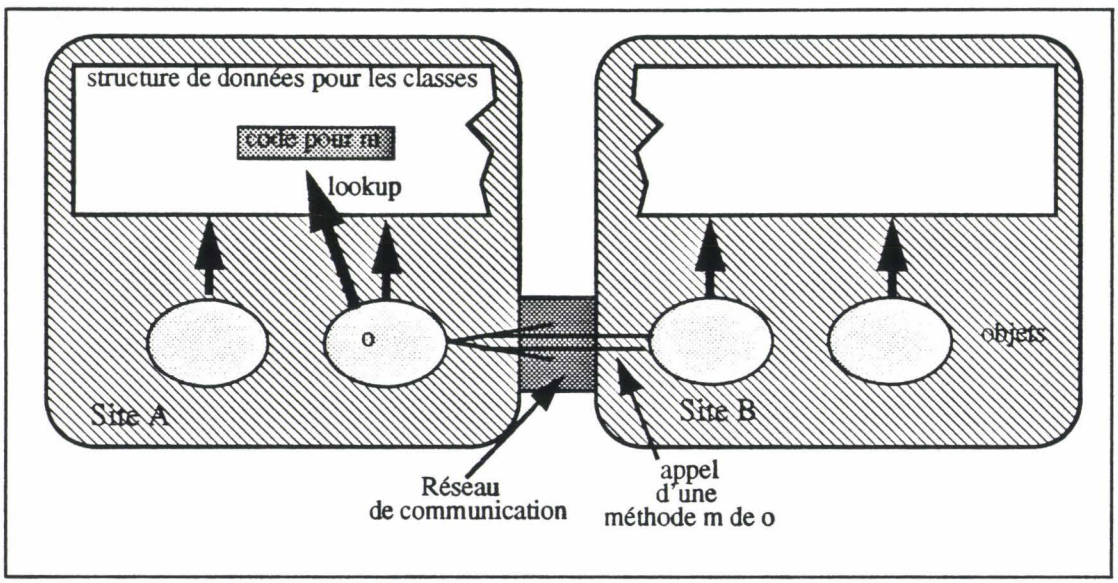


figure 3.5 *Eclatement sur plusieurs sites*

Dans cette approche, nous trouvons deux stratégies de création d'un objet:

- 1 - Eclatement selon les classes. Les instances sont réparties en fonction de leur classe. Un objet est créé sur le site où se trouve sa classe

- 2 - Eclatement en fonction de la charge. On essaye dans cette approche de répartir en fonction de la charge de chaque noeud, les instances sur les différents sites. Le code des méthodes est dispersé et le système utilise des mécanismes particuliers tels que le chargement dynamique de code ou la migration (des objets ou du code) pour résoudre les «défauts de localité»

- I - 3.2.1 Eclatement selon les classes

Les classes sont ici réparties sur l'ensemble des sites du réseau mais les instances sont localisées sur les noeuds associés à leur classe d'instanciation. Une classe à l'exécution comprend ainsi le code de ses méthodes et l'ensemble de ses instances. Le modèle d'exécution peut donc lancer l'exécution d'une méthode sur une instance, la méthode et la classe étant sur le même noeud. Le problème réside sur les communications entre objets de classes différentes.

Dans le système distribué à objets *Edoms*, un site peut accueillir plusieurs classes et un serveur global prend en charge les échanges entre les objets de sites différents.

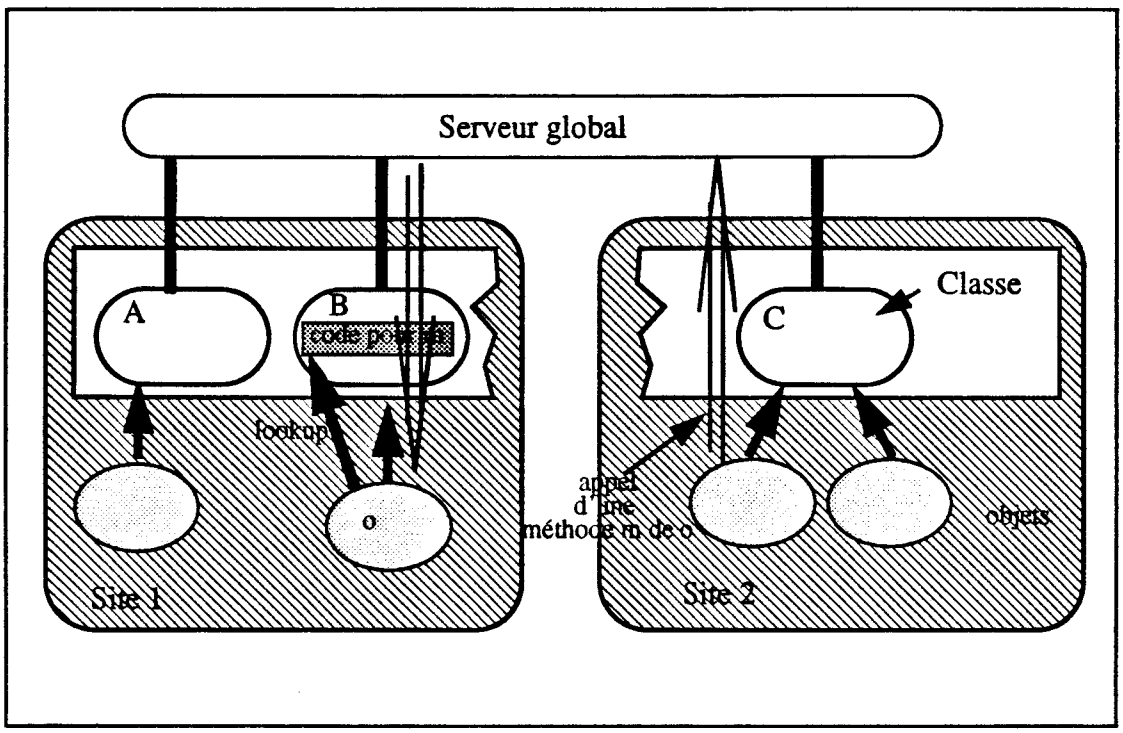


figure 3.6 *Le serveur EDOMS*

Dans le schéma ci-dessus, un ensemble de trois classes sont réparties sur deux noeuds. Les instances sont créées sur le noeud de leur classe d'instanciation. Les communications entre objets de noeuds différents sont prises en charge par un serveur global. Le serveur centralise les échanges et pénalise donc les performances liées au parallélisme.

Dans le système *Amoeba*, les objets sont décrits par des types réalisés par des serveurs de type à l'exécution. Chaque serveur regroupe les objets du type et le code associé. Les traitements peuvent ainsi s'effectuer localement et les communications entre objets de différents types passent par le serveur.

Dans ces propositions, la répartition des instances s'effectue en fonction de la localisation de la classe (ou du type). La solution de répartition en fonction des classes nécessite une adéquation entre le nombre de classes et celui des sites. Les classes fortement sollicitées surchargent certains noeuds alors que d'autres classes peu utilisées sous-exploitent leur noeud d'accueil.

- I - 3.2.2 Eclatement selon la charge

La répartition des objets peut s'effectuer en fonction de la charge de chaque noeud et non en fonction de l'appartenance à une classe. Les objets étant répartis indépendamment des classes, des mécanismes spécifiques doivent être mis en place pour éviter les cas de «défaut de localité», tels que la migration des objets, le chargement dynamique de code...

Pour résoudre le cas de «défaut de localité», une solution vise à pouvoir déplacer un élément de l'application à l'exécution. Cette migration consiste à transporter l'objet vers le noeud contenant le code de la méthode (migration des objets) ou déplacer le code de la méthode vers les objets à manipuler (migration du code). La migration est ici un outil du mécanisme d'exécution.

Le mécanisme de migration des objets utilise les communications via le réseau. Un protocole de migration est établi, permettant ainsi le transport des données locales de l'objet vers le noeud destinataire qui le reconstruira.

Dans le système *Sos*, les objets peuvent être importés et peuvent s'exporter d'eux-mêmes. La migration d'un objet peut également s'effectuer par copie de celui-ci.

La migration du code est en général une «simple copie» de code, on parlera alors d'un chargement dynamique du code. Le système doit connaître lors d'une invocation de méthode le noeud sur lequel elle réside pour demander une copie (principe de «demander la recette» décrit dans l'article de Briot [Briot87].

La migration est utilisée dans la plupart des langages parallèles à objets: *Emerald, Abcl/1, Sos, Guide.* Dans l'implantation de *Abcl/1* [Takada90] sur une architecture distribuée, le code des méthodes est transcrit dans un langage intermédiaire qui sera interprété sur l'environnement local d'un noeud.

Les *RPC* (Remote Procedure Call) sont utilisées pour exécuter le code d'une procédure à distance. Ramenées à un langage à objets, elles permettent de déclencher une méthode distante sur un objet local, les données locales de l'objet sont transmises au moment de l'appel à distance.

La migration des objets permet d'équilibrer la charge des noeuds en déplaçant certains objets des noeuds fortement chargés. Le système les déplace sur les noeuds plus faiblement chargés. Cette régulation de charge peut être contrôlée par le système ou décrit par l'utilisateur.

Exemple: la migration dans Emerald

Emerald fonctionne sur réseau de stations de travail. C'est un langage à objets, sans classe et à typage statique. Un système de désignation des objets est défini sur l'ensemble du réseau. L'invocation d'une méthode (*opération* dans la terminologie *Emerald*) sur un objet distant conduit à la *migration* vers le site distant du processus qui applique la méthode ou de l'objet-client. Le problème de l'accès au code des méthodes est pris en charge par le noyau *Emerald* [Jul88].

Les objets sont répartis sur les noeuds de la machine. Pour chaque groupe d'objets résidant sur un même site, le système leur associe un exemplaire de leurs méthodes. Le noyau peut retrouver et charger librement ces méthodes à la demande.

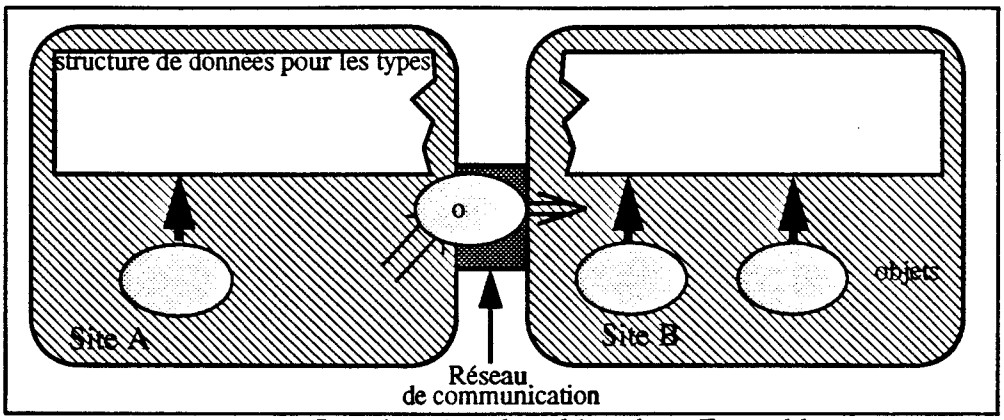


figure 3.7 *La migration des objets dans Emerald*

- I - 3.3 Conclusion sur «Objets et Répartition»

L'implantation d'un langage à objets à la Smalltalk sur un multicomputer pose plusieurs problèmes liés à la mémoire distribuée de l'architecture. L'utilisation de pointeurs en mémoire est limitée à l'intérieur d'un noeud physique de la machine. Le système doit résoudre le cas de «défaut de localité».

Du fait de l'absence de mémoire commune, les interactions entre les noeuds font nécessairement appel à la communication par messages [Andrews83]. Ce niveau de communication est complètement caché dans les solutions présentées.

Les exemples cités portent principalement sur des réseaux de stations de travail. Les multicomputers ne se différencient pas de ces réseaux sur le plan des principes. Les solutions citées peuvent donc être reprises pour résoudre le problème de localité. Par contre, les caractéristiques quantitatives de ces deux classes de machines ne peuvent pas être confondues. Un multicomputer possède plus de processeurs qu'un réseau: le problème de distribution se pose à une échelle plus vaste. Le multicomputer est bien plus performant qu'un réseau local. Si dans un réseau de stations de travail, les communications doivent être limitées, elles sont au contraire peu coûteuses sur un multicomputer. En outre, la mémoire locale d'un noeud d'un multicomputer constitue une ressource à gérer avec économie. Elle n'est pas accompagnée d'une mémoire secondaire. Les techniques de copie systématique de code, ou de gestion de copies d'objets, butent sur la rareté de la ressource mémoire.

Toutes ces remarques nous conduisent aux conclusions suivantes:

- L'implantation de langages parallèles à objets sur des multicomputers nécessite:

  . l'élaboration d'un schéma de désignation des objets étendu à tous les noeuds de la machine.

  . Une approche nouvelle du problème de défaut de localité, sachant que la plupart des techniques utilisées sur les réseaux de stations s'appliquent difficilement aux multicomputers.

- La définition de la place accordée à la communication par messages. Cela constitue le principal apport du reste de cette thèse.

Conclusion

Nous avons présenté un rééxamen des principaux langages parallèles à objets qui porte sur trois points:

- **L'intégration du parallélisme dans le langage.** Cette étude montre que les objets actifs sont particulièrement adaptés aux machines parallèles. Outre l'apport de la programmation objet, le parallélisme intrinsèque du modèle exploite naturellement le parallélisme de la machine cible. Parmi ces objets, les multi-programmés présentent un degré plus important de parallélisme, augmentant ainsi la capacité de traitement de l'objet mais nécessitent la mise en place d'une synchronisation intra-objet. Les objets fragmentés permettent de répartir un objet sur plusieurs noeuds. L'objet est alors multi-programmé, mais ses fragments sont mono-programmés et n'impliquent donc pas de synchronisation intra-fragment.

- **L'expression de la synchronisation** dans les langages à objets est différente selon que l'objet soit mono ou multi-programmé. Les solutions d'héritage des contraintes de synchronisation dans les objets mono-programmés, telle que celle décrite sur le prototype *Rosette*, sont applicables aux objets multi-programmés où la synchronisation intra-objet s'exprime différemment.

- La **répartition du code et des objets sur une mémoire distribuée.** Les différentes solutions sont des compromis entre: 1) La duplication du code sur tous les noeuds (ce qui permet d'accéder au code n'importe où) 2) découper l'application en tronçons de code implanté chacun sur un noeud (cette solution nécessite un modèle d'exécution plus élaboré intégrant par exemple des possibilités de migration du code ou des données).

L'implantation sur une machine parallèle de ces langages parallèles complexes à objets nécessite l'existence d'un support logiciel évolué. On peut citer ici: *Cool* [Habert90, Lea91] couche objet sur le système *Chorus* [Rozier88], la «machine virtuelle» de *Comandos* [Marques89, Boyer90, Comandos91] support du système réparti *Guide* ou encore *Amoeba* et la couche Objet *Orca* [Bal87]. Ces environnements basés sur les objets sont naturellement dédiés aux langages parallèles à objets.

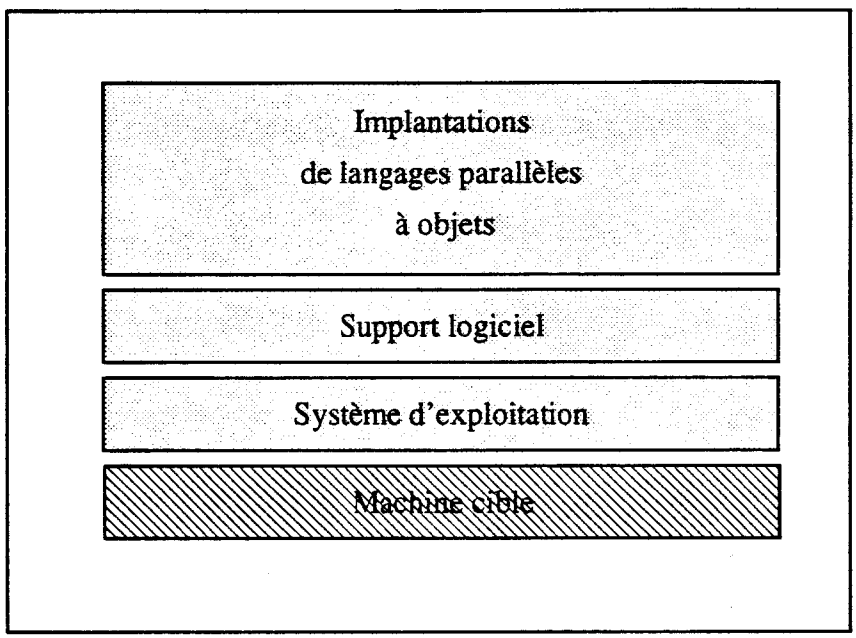


figure 4.0

Notre démarche est semblable : nous proposons un outil évolué pour la conception des applications parallèles, les Composants Actifs de Communication. L'environnement minimal est facilement réalisable au dessus du système d'exploitation de la machine cible et permet ainsi le «portage» aisé de l'environnement et un moindre coût de développement. En outre, nous cherchons à produire un support logiciel suffisamment souple pour être directement utilisable par le programmeur sans l'embarrasser des contraintes matérielles. La couche objet sera développée au dessus de cette plate-forme de développement comme une application parallèle parmi d'autres (Elle est présentée au chapitre III).

Chapitre - II -

Les Composants Actifs de Communication

Ce chapitre présente les **Composants Actifs de Communication** (ou Cac) [Courtrai92a,92b,92d]. Les Cac/s et leur environnement de programmation constituent un outil de conception d'applications parallèles pour machines parallèles dont les multicomputers. Les Cac/s sont nés du constat suivant.

Les machines parallèles tels que les multicomputers sont de plus en plus puissantes et complexes à programmer; ceci d'autant plus que le nombre d'outils de programmation est faible. Les systèmes d'exploitation sur ces multicomputers ne proposent que des outils de bas niveau de programmation du parallélisme. Une application est alors matérialisée par un ensemble de processus communiquants. L'architecture engendre une hétérogénéité des primitives du système au niveau des échanges entre processus, de la synchronisation de ces processus et de la gestion mémoire en fonction de la localisation de chaque processus. Ces contraintes liées à l'architecture distribuée subsistent lors de la programmation de ces machines.

Pour supprimer ces contraintes, nous introduisons les Composants Actifs de Communication. Une notion de Cac est une vision structurante des données d'une application et le Cac est ainsi la seule structure de l'environnement. La proposition est basée sur le concept Objet appliqué aux systèmes. L'outil fournit une interface homogène

qui masque les contraintes matérielles au programmeur et la localisation de chaque entité. L'environnement Cac/s nous servira de plate-forme pour l'implantation de langages parallèles à objets actifs (voir chapitre III).

Le **Cac** est donc une entité de programmation qui regroupe une activité (processus au sens des système), une environnement local et une boîte aux lettres. Cette structure est facilement réalisable au dessus des systèmes d'exploitation. Elle est proche de celle des acteurs telle qu'elle a été définie par G. Agha [Agha86]; mais la programmation et le modèle d'exécution des Acteurs sont différents. Une application est définie par un ensemble de Cac/s communiquants par envoi de message. L'environnement Cac est suffisamment flexible pour reconstruire au dessus d'autres modèles parallèles (système d'Acteurs, ou classes d'Objets Actifs). Nous utilisons les Cac/s pour la conception et la modélisation d'applications parallèles,

Au dessus de la structure des Cac/s et de leur programmation, les **modules** regroupent un ensemble de fonctions comportementales. A l'exécution, le module est un outil de répartition du code et des données. Chaque module regroupe donc une partie du code de l'application.

Nous avons développé un prototype Cac sur une machine parallèle, le MultiCluster II de chez Parsytec. La plate-forme de développement est structurée en couche pour permettre le portage aisé du prototype. Elle permet à faible coût le développement des autres modèles existant (comme celui des Acteurs ou des Objets Actifs).

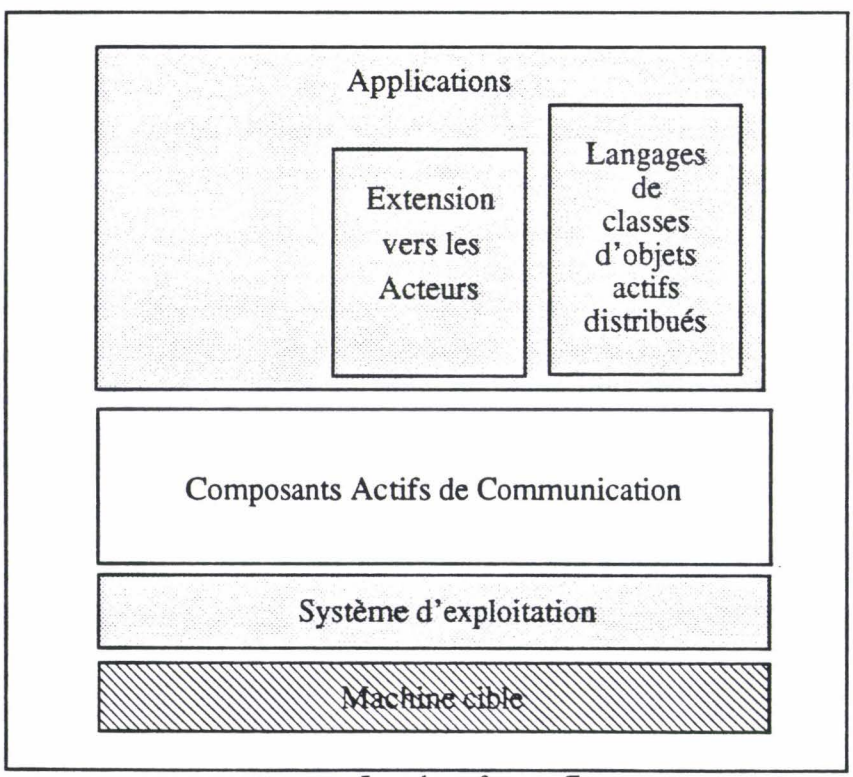


figure 1.0 *La plate-forme Cac*

Autres réalisations [Takada90, Markhoff90, Di Santo91, Glavitto91, Gautron92, Capobianchi92]. Leur objectif est différent: ils implantent directement un langage d'acteurs sur une machine parallèle. Notre solution consiste à d'implanter une couche Acteur minimale sur laquelle nous pourront développer les autres concepts Objets.

- II - 1. Les Cac/s

Nous introduisons le Composant Actif de Communication comme le grain de décomposition d'une application en entités autonomes s'exécutant concurremment sur la machine cible. Une application doit être définie par la spécification du comportement des différents Cac/s nécessaires et de la coopération de ces différents Cac/s. La coopération des Cac/s se fonde sur la communication asynchrone par envois de messages entre Cac/s.

- II - 1.1 La notion de Cac

Nous proposons une structure de base, appelée **Composant Actif de Communication** (Cac), d'un niveau plus évolué que le processus. Cette structure programmable intègre une activité propre (processus), une couche de communication et un environnement privé. Une bibliothèque spécifique permettra la programmation et la manipulation de ces composants actifs de communication. La programmation d'applications parallèles s'effectue en décrivant un assemblage de Composants Actifs de Communication et en programmant leur activité.

L'environnement des Composants Actifs de Communication permet

- la prise en charge d'un parallélisme massif de tâches car ici « tout est Cac», et chaque Cac est une activité autonome.
- d'avoir une vision structurée des données et activités. Chaque Cac encapsule des données privées et une activité locale.
- une description des applications parallèles indépendante des particularités des architectures et systèmes sous-jacents.

Nous détaillons maintenant la structure des Composants Actifs de Communication.

- II - 1.1.1 Structure d'un composant

La structure d'un Composant Actif de Communication est proche de celle des *Acteurs*. Elle se compose d'un **processus** (partie traitement), d'une **boîte aux lettres** (partie communication), et d'un **environnement local** (mémoire locale du composant) (figure 1.0 «Structure d'un Composant Actif de Communication»).

Le processus

Chaque composant possède une activité autonome matérialisée par un processus. Le domaine de ce processus est exclusivement limité au composant. Le processus matérialise le comportement du composant. Cette activité est programmable par l'utilisateur en écrivant une fonction comportementale. Cette activité gère les données locales du composant et effectue les communications avec les autres composants.

La boîte aux lettres

Toutes les communications entre composants utilisent une structure de communication spécialisée. Cette structure, associée à chaque composant, est une boîte aux lettres. Elle stocke tous les messages destinés au composant. Les communications sont asynchrones et s'effectuent en déposant un message dans la boîte aux lettres du composant destinataire. Le processus du composant peut explicitement extraire les messages déposés dans sa boîte pour les traiter. La boîte aux lettres est créée à la création du composant et son nom identifie le composant dans le système.

L'environnement local

L'environnement local d'un Cac est la mémoire locale du composant. Il regroupe toutes les données locales (variables ou fonctions locales) du composant. Cet environnement est géré dynamiquement par le processus du composant. Le run-time fournit deux primitives d'allocation et de libération mémoire. L'allocation mémoire s'effectue dans l'espace du noeud où évolue le Cac.

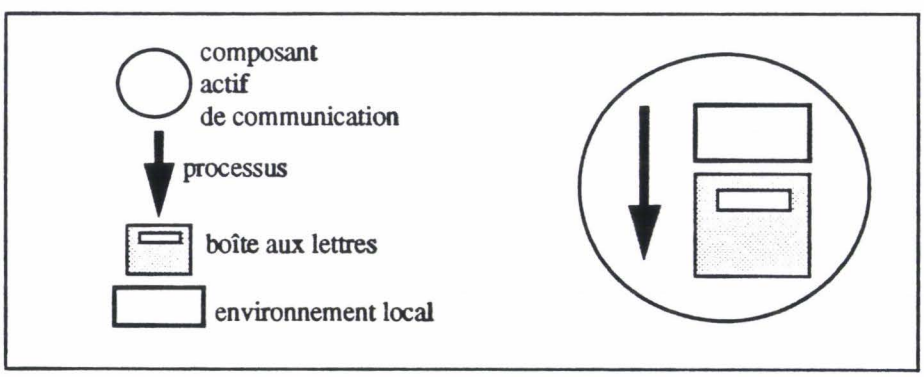


figure 1.0 *Structure d'un Composant Actif de Communication*

A ce niveau, l'outil ne demande rien à l'architecture et système sous-jacent sauf sur les points suivants:

- 1 - Un Cac est entièrement localisé sur un noeud de la machine et ne migre pas.
- 2 - Le noeud en question doit pouvoir accueillir le processus du Cac; il doit donc être multi-tâches pour que plusieurs Cac/s puissent coexister sur un même noeud.
- 3 - Le noeud doit pouvoir accueillir les structures de données du Cac, c'est à dire sa boîte aux lettres et son environnement privé.
- 4 - Le système sous-jacent doit être capable de transporter un message vers un Cac quelconque, quelle que soit sa localisation sur l'ensemble des noeuds. Cela nécessite un système de désignation des boîtes aux lettres sur l'ensemble du système et un mécanisme de routage des messages.

Les trois premiers points concernant l'accueil des Cac/s sur les noeuds sont tout à fait compatibles avec les machines visées: multicomputers. Ces machines MIMD sont effectivement formées de noeuds puissants qui intègrent souvent le multi-tâches au niveau du micro-code et possèdent en général plusieurs Moctets.

Le dernier point (- 4 -) concernant la désignation et le routage est à la charge du logiciel de base.

- II - 1.1.2 La désignation dans l'environnement des Cac/s

Deux systèmes de désignation sont nécessaires:

- 1 - un permettant la désignation de comportement pour la création de Cac,
- 2 - un permettant la désignation des Cac/s. En fait les boîtes aux lettres pour la communication entre Cac/s.

Les comportements

La création d'un Cac nécessite la désignation d'un comportement que le processus du Cac prend en charge. Pour pouvoir créer un Cac sur un noeud quelconque de la machine, un comportement doit être désigné par un nom valide sur l'ensemble du système.

A ce niveau aucune hypothèse n'est faite sur la localisation de la création. Cela dépend de la structure d'accueil des Cac/s. Une contrainte dite de localité est cependant imposée: le code du comportement doit être chargé sur le noeud où se fait la création.

Les Cac/s

La création d'un Cac retourne un nom pourvu d'un sens pour le système afin de permettre aux autres Cac/s de communiquer lui (par envoi de message).

En réalité ce nom n'est autre que celui de la boîte aux lettres du Cac créé: pour le système la communication entre Cac/s se fait uniquement par envois de message qui sont déposés dans la boîte aux lettres des destinataires. La primitive d'envoi de message donne le message au système pour le transporter dans la boîte aux lettres du Cac destinataire quelle que soit sa localisation.

- II - 1.2 L'interface de programmation des Cac/s

Le programmeur décrit ainsi une application parallèle en un ensemble de comportements de Cac/s matérialisant les activités parallèles de son application. Nous décrirons maintenant la programmation des composants en détaillant les primitives de gestion des composants. La bibliothèque comprend les primitives d'accès aux ressources du système et les primitives de communication entre Cac/s. En plus de ces primitives de communication, nous proposons une gestion d'autres boîtes aux lettres locales aux Cac/s qui permet une modélisation et une programmation plus aisées des problèmes parallèles.

La programmation d'une fonction comportementale s'écrit dans le language *C*. Ce langage support nous permet de réutiliser toutes les structures de contrôle du langage. Les primitives du run-time Cac sont accessibles à partir du langage *C* par une bibliothèque de fonctions. Nous présentons ici les principales fonctions. L'ensemble complet des primitives est décrit en Annexe -II- «Les primitives du run-time Cac».

- II - 1.2.1 Primitives de gestion des ressources

Les composants

Une ressource «composant» est gérée par le système. Dans l'environnement de programmation Cac, les références sur les Cac/s sont typées (type *Component).*

La fonction *NewComponent(Behavior b, args...):Component* crée un nouveau Cac de comportement *b*. Le nom de la fonction comportementale (*b*) doit être passé en paramètre à la primitive ainsi que ses éventuels arguments (*args*). La primitive *NewComponent()* est synchrone et retourne le nom de la boîte aux lettres du composant créé.

Cette primitive *NewComponent(Behavior b, args...)* permet la création de Cac à distance. Elle est utilisable de la même manière, que *b* soit implanté sur le noeud local ou sur un noeud distant. C'est donc fondamentalement cette primitive qui permet l'exécution répartie de nos applications.

La fonction *KillComponent(Component c)* détruit à distance le composant de nom (*c*). La destruction s'effectue par l'envoi d'un message particulier au composant *c* à déduire, le composant *c* pouvant être distant. Le processus du composant doit explicitement recevoir le message de destruction. Il détruit alors la boîte aux lettres, libère l'espace réservé à l'environnement local et s'arrête.

La mémoire

La mémoire est gérée par le système lors de la création de composants actifs de communication. L'espace mémoire de l'environnement local du composant est créé dans la phase d'initialisation du composant. Le processus du composant peut augmenter ensuite explicitement la taille de l'environnement local. L'allocation mémoire et la libération s'effectuent par les deux primitives *new()* et *dispose()*. Ces opérations restent locales.

- II - 1.2.2 Les communications entre les composants

Les communications entre les composants se font par dépôt et retrait de messages dans les boîtes aux lettres des Cac/s. Les communications entre composants sont asynchrones. Les messages contenus dans la boîte d'un composant sont stockés dans l'ordre de leur arrivée. Seul le composant propriétaire de la boîte aux lettres peut consulter et extraire les messages.

Le nom d'un Cac est obtenu par le primitive *NewComponent()*, en l'occurrence le nom de la boîte aux lettres. Une fois créée, la référence d'un Cac peut être passée en argument d'un message. Ce sont les deux opérations pour obtenir une référence à un Cac et permettront de lui envoyer des messages.

Les messages entre composants actifs de communication n'ont pas de structure particulière. Le mécanisme de communication transmet directement un bloc de données. Un type pointeur de message *Mess_Ptr* est défini dans l'environnement de programmation des Cac/s pour permettre la manipulation des messages. L'application structure elle-même ses messages.

Voici la liste des primitives de communication:

- La fonction *NewMessage(int s): Mess_Ptr* alloue l'espace mémoire d'un message en réservant *s* mots mémoire (correspondant à la taille du message) et retourne un pointeur sur le message.
- La fonction *FreeMessage(Mess_Ptr m)* libère la place réservée au message *m*.
- La primitive SendaMessage*(Component c, Mess_Ptr m)* dépose un message *m* dans une boîte aux lettres d'un composant de nom *c*. La communication est transparente quelle que soit la localisation du composant destinataire c. La structure du message n'a pas d'importance; la primitive transfère un bloc de données. L'appel est asynchrone, le processus utilisant *SendaMessage()* est libéré dès la prise en charge du message par la couche de communication.
- La fonction *GetMessage(Mess_Ptr * m)* extrait un message de la boîte aux lettres du composant courant. Le message extrait est accessible par la variable *m*. La fonction extrait le plus vieux message de la boîte. Si la boîte est vide, la primitive se bloque jusqu'à la réception d'un message par le composant.

- II - 1.2.3 Les boîtes aux lettres locales

La réalisation des composants actifs de communication est écrite au dessus d'une gestion de boîte aux lettres. Il est intéressant de laisser les primitives de manipulation de boîte aux lettres accessibles au niveau de la programmation. Un composant peut ainsi créer plusieurs boîtes aux lettres locales. Chaque boîte aux lettres permet, par exemple, d'identifier les requêtes, sans gérer un mécanisme de nommage de requêtes et de retour de réponses.

Ces boîtes aux lettres, appelées **Box/s**, sont des Cac/s dépourvus de processus et d'environnement local. Les références sur les *Box/s* sont typées (type Box). Ces structures passives sont locales à l'environnement d'un composant et sont en général temporaires. Le composant qui créé un Box, est le seul à pouvoir la consulter. Le nom de la boîte peut être transmis dans des messages aux autres composants qui pourront alors y déposer des messages par la primitive *SendaMessage()*.

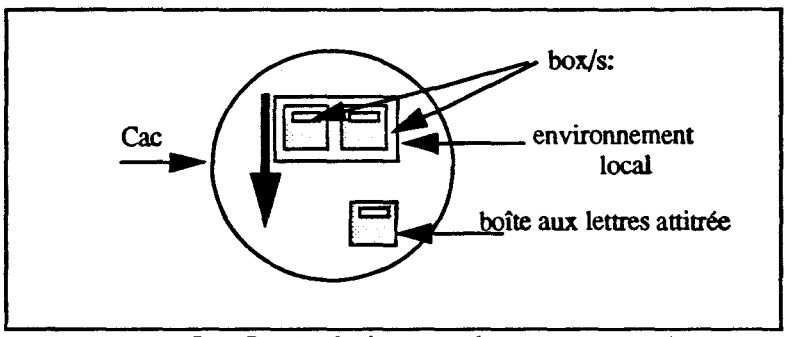


figure 1.1 *Les Box/s: boîtes aux lettres temporaires*

L'interface de programmation des *Box/s* se définit comme suit:

- La fonction *NewBox()*: *Box* crée localement une nouvelle boîte aux lettres et retourne l'adresse de cette *Box*.

- La fonction *FreeBox(Box b)* détruit la boîte aux lettres de nom *b*.

- La fonction *GetMessageInBox(Box b, Mess_Ptr *m)* extrait le plus vieux des messages de la boîte aux lettres *b*. Le message extrait est accessible par la variable *m*. Si la boîte est vide, la primitive se bloque jusqu'à la réception d'un message par le composant.

- Une fonction *NumberMessageInBox(Box b): int* retourne le nombre de messages contenus dans la boîte aux lettres *b*. La primitive n'est pas bloquante.

- La primitive *ReadFromBox(Box b, int i, Mess_Ptr *m)* lit le i ^{ème} message dans la boîte aux lettres *b*. Le message est dupliqué sans être retiré de la boîte. La copie du message est accessible par la variable *m*. La primitive doit être appelée après l'appel de *NumberMessageInBox*() et le programmeur doit garantir la valeur de *i*.

- La primitive *DeleteMessageInBox(Box b, int i)* détruit le i ^{ème} message de la boîte aux lettres *b*.

- Une primitive *WaitOnBox(Box b)*, bloque le processus appelant jusqu'au prochain événement arrivant sur la boîte aux lettres *b* (dépôt de message ou extraction de message). Le processus n'est bloqué que si aucun événement n'est intervenu depuis l'appel précédent à la primitive *WaitOnBox*().

A partir des primitives *NumberMessageInBox()*, *ReadFromBox*(), *DeleteMessageInBox*(), *WaitOnBox*(), l'utilisateur peut filtrer les messages reçus.

Le nom d'une Box est compatible avec le nom d'un composant (type Component). Par exemple, l'élément nommé c de la primitive *SendaMessage(c,m)* est indifféremment un composant ou une boîte aux lettres.

- II - 1.3 Exemples de fonctions comportementales

- II - 1.3.1 Calcul d'une factorielle

Une méthode de calcul récursive et parallèle de la fonction factorielle s'effectue par dichotomie (l'exemple est tiré de l'article de Kafura [Kafura 90]). L'appel de fac(n) lance le calcul sur l'intervalle [1..n]. A chaque pas du calcul, l'intervalle [min..max] est scindé en deux et les calculs des deux parties s'effectuent en parallèle (par l'appel de fac(min,(min+max)/2) et fac((min+max)2)+1, max)). Les résultats retournés par les deux appels sont multipliés et la valeur résultat est retournée. Une branche de calcul s'arrête lorsque les valeurs min et max sont égales. Une des valeurs est alors retournée. (voir figure 1.2 «Calcul fac(1..4)»).

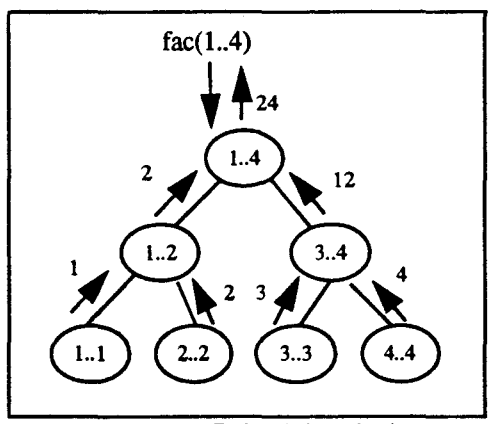


figure 1.2 *Calcul fac(1..4)*

Cette méthode de calcul peut être réalisée par les composants actifs de communication. Les Cac/s de comportement factoriel *B_FAC* sont les entités actives de calcul. La primitive *NewComponent(B_FAC,retour,min,max)* crée un composant factoriel dont le processus exécute la fonction comportmentale *fac* décrite dans le langage C. L'intervalle du calcul est transmis à la primitive ainsi que l'adresse *retour* qui est ici une continuation explicite du calcul de *fac* sur l'intervalle [min..max]. Un composant client doit lancer l'application, il crée le premier composant de type *fac*, attend un message de ce composant puis affiche le résultat.

Programmation

L'exemple utilise une primitive *ArgMess(Mess_Ptr m, int i)* qui permet l'accès direct à l'$i^{ème}$ argument du message *m*. La variable *my* mémorise le nom du composant courant qui exécute la fonction comportementale. La primitive *ConstructMess(Mess_Ptr m, args)* alloue un bloc mémoire pour le message *m* et le remplit avec une suite de valeurs *args*.

```
/*                                    B_FAC                                    */
#include<Prim_Cac.h>

void fac(int *arg)
    /*comportement d'un cac fac, arg paramètres de création du Cac */
    /* arg[1] == adresse de retour arg[2].. arg[3] intervalle de calcul*/
    {
    Mess_Ptr        mes1,mes2,res;
    if (arg[2] == arg[3]) {          /*min = max fin d'une branche de calcul*/
        ConstructMess(&res,arg[2]);  /*construction d'un message garni avec arg[2]*/
        SendaMessage(arg[1],res)
    ) else { /* création des deux Cac/s fac, attente des réponses et retour du résultat*/
        NewComponent(B_FAC,my,arg[2],(arg[2] + arg[3]) / 2);
        NewComponent(B_FAC,my,((arg[2] + arg[3]) / 2)+1,arg[3]);
        GetMessage(&mes1);           /*attente d'un message de réponse*/
        GetMessage(&mes2);           /*attente de l'autre message de réponse*/
        ConstructMess(&res,ArgMess(mes1,1)* ArgMess(mes2,1));
        SendaMessage(arg[1],res);
    };
}
```

figure 1.3 *Comportement d'un Cac factoriel*

```
/*                                    B_START                                  */
#include<Prim_Cac.h>

void start(int *arg)
    {
    Mess_Ptr    mes;
    int         n = 20 ;                    /* factorielle à calculer */
    NewComponent(B_FAC,my,1,n);             /* 1..n => intervalle de départ du calcul*/
    GetMessage(&mes);                       /*attente d'une réponse*/
    printf("resultat fac (%d) = %d\n",n,ArgMess(mes,1));
}
```

figure 1.4 *Appel de la fonction factorielle*

Les comportements sont nommés globalement dans l'environnement de programmation. Un fichier contient la liste des noms de comportements déjà développés. Chaque comportement est associé à un numéro qui l'identifie (ici XXX).

```
/*              Behavior.h              */

#define B_FAC           XXX
```

figure 1.5 *Le nommage des comportements*

- II - 1.3.2 L'Arbre binaire de recherche

Nous avons repris d'autres exemples d'applications parallèles dont l'arbre binaire de recherche (*Abcl/1, Pool/T et Extented-Eiffel*). Contrairement à l'exemple des factorielles où le parallélisme est lié à l'algorithme de calcul, le parallélisme est ici lié aux noeuds de l'arbre qui sont des objets actifs et surtout aux traitements simultanés d'insertion et de recherche dans la totalité de l'arbre (sur des noeuds différents de la machine cible).

L'arbre binaire de recherche est une structure de données contenant un ensemble de clés. Chaque noeud de l'arbre contient 3 informations: une valeur et deux pointeurs sur le fils droit et fils gauche du noeud. L'arbre est trié suivant les valeurs contenues dans les noeuds: (La valeur contenue dans le fils droit est toujours inférieure à celle du noeud courant, elle-même inférieure à la valeur contenue dans le fils droit. Au départ, l'arbre ne possède qu'un noeud racine vide (sans sous-arbre et sans valeur).

L'arbre peut recevoir deux types de requêtes:

- 1 - L'insertion: l'insertion d'une valeur dans l'arbre commence par rechercher à partir de la racine une feuille vide. Lorsque la feuille est trouvée, le système insère l'information et crée deux feuilles vides qu'il relie à la feuille courante. Il n'y a pas d'accusé de réception.

- 2 - La recherche: la recherche d'une valeur s'effectue par comparaison de celles des noeuds, ceci à partir de la racine. Lorsque la valeur du noeud est différente de celle recherchée, la recherche est propagée vers l'un des deux fils. La recherche s'arrête lorsque la valeur est trouvée ou si le traitement aboutit sur une feuille. L'opération retourne toujours un résultat booléen correspondant au succès ou non de la recherche.

La structure d'un arbre de 4 éléments peut se schématiser comme suit:

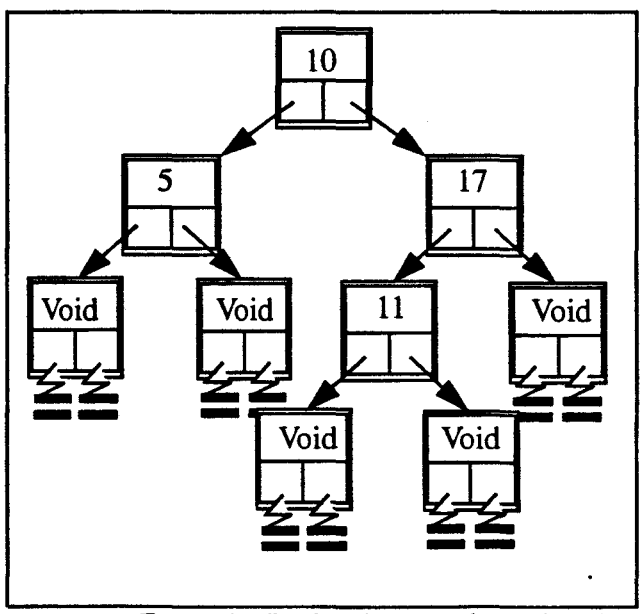


figure 1.6 *Exemple d'arbre binaire de recherche*

Programmation

```
/*              fonction comportementale d'un noeud de l'arbre                */
#include<Prim_Cac.h>

void node(int *arg)
    {
    MessPtr         mes, reply;
    Component       left, right, sender;
    int             value, mes_value;
    left = VOID; right = VOID; value = VOID;
    while (1) {
        GetMessage(&mes);                       /* lecture d'un message */
        switch (ArgMess(mes,1) ) {              /* nom de la requete*/
            case 1:                             /* insertion */
                mes_value = ArgMess(mes,2);     /* valeur à inserer*/
                if (value == VOID) {            /* feuille */
                        value   = mes_value;
                        left    = NewComponent(NODE); /*création du fils gauche*/
                        right   = NewComponent(NODE); /*création du fils droit*/
                }else if (value > mes_value)
                        SendaMessage(left,mes);
                else
                        SendaMessage(right,mes);
            break;
            case 2:                             /* recherche */
                mes_value = ArgMess(mes,2);     /* valeur à rechercher*/
                sender = ArgMess(mes,3);        /* le mandataire*/
                if (value == VOID) {            /* feuille*/
                        ConstructMess(&reply,FALSE);
                        SendaMessage(sender,reply);
                } else if (value == mes_value) {
                        ConstructMess(&reply,TRUE);
                        SendaMessage(sender,reply);
                } else if (value > mes_value)
                        SendaMessage(left,mes);
                else
                        SendaMessage(right,mes);
            break;
        }
    }
}
```

figure 1.7 *La fonction comportementale d'un noeud de l'arbre*

Chaque noeud de l'arbre est matérialisé par un Cac. Un tableau de nom *data* mémorise les variables locales de chaque noeud de l'arbre. La fonction comportementale initialise les données locales et se met en attente de message «de requête». Lors de la réception d'un message, le noeud extrait du message le type de la requête et la traite.

La fonction *rand*() retourne un nombre aléatoire.

```
/*                                  START                                  */

#include<Prim_Cac.h>

#define MAX 500

void start(int *arg)
    {
    Component       root;
    Mess_Ptr        mes,req;
    int             i;

    root = NewComponent(NODE,); /*création du noeud racine*/

    for (i=0 ; i < MAX; i++) {
        /*insertion de Max valeurs produites aléatoirement */
        ConstructMess(&req,rand());
        SendaMessage(root,req);
    }
    for (i=0 ; i < MAX; i++) {
        /*recherche de Max valeurs produites aléatoirement */
        ConstructMess(&req,rand(),my);
        SendaMessage(root,req);
    }
    for (i=0 ; i < MAX; i++)
        /* récupération des messages de réponse de recherche (sans distinction)*/
        GetMessage(&mes);
}
```

figure 1.8 *Utilisation de l'arbre*

Le comportement START utilise l'arbre binaire. Il crée le noeud racine puis génére un nombre prédéfini de messages d'insertion en mode asynchrone vers le noeud racine. La valeur à insérer est calculée aléatoirement en fonction du nombre d'insertions. (pour avoir une change de succès lors des recherches). Puis il génére un nombre égal de messages asynchrones de recherche de valeurs vers le noeud racine (les valeurs sont aussi calculées aléatoirement). Il se met alors en attente de toutes les réponses.

- II - 2. Les modules

Nous avons introduit dans la première partie les Cac/s comme structure programmable pour la construction d'applications parallèles. Nous présentons maintenant le **Module** comme le grain logique de répartition du code et des Cac/s. Primitivement, un Cac est créé en désignant le code décrivant son comportement. Ce sont les «fonctions comportementales» des Cac/s utilisés dans l'application. Ces fonctions sont regroupées dans les différents modules utilisés dans l'application. Ainsi donc un Cac sera créé «dans» un module. Le code d'une application est ainsi partitionné en modules contenant chacun un sous-ensemble de fonctions comportementales. Cette distribution des comportements en modules peut être relayée juste avant la phase d'exécution de l'application. Durant cette phase d'exécution, un module est implanté sur un des noeuds de la machine cible; il regroupe alors le code des comportements du module ainsi qu'un ensemble de Cac/s ayant ces comportements[1].

- II - 2.1 La notion de Module

Les modules ont deux fonctions.

- 1 - Un module est une entité de partage de code. Il regroupe un ensemble de fonctions comportementales désignées globalement (voir - II - 1.1.2 «La désignation dans l'environnement des Cac/s»). Ces noms globaux sont utilisés par un autre module pour la création à distance de composant pour nommer un comportement. L'environnement interdit la migration du code à l'exécution et un module réside entièrement sur un noeud de la machine.

- 2 - Le module est aussi la structure d'accueil des Cac/s sur les différents noeuds de la machine cible. Chaque module regroupe un ensemble de Cac/s ayant chacun un comportement parmi ceux contenus dans le module. Un composant vit sur un module et ne peut migrer sur aucun autre.

Chaque module est nommé par un nom logique qui identifie l'ensemble des fonctions comportementales contenues dans le module. Ce nom logique permettra de manipuler le module lors de la phase d'implantation sur un noeud.

Les critères de regroupement des fonctions comportementales sont libres:

- 1 - La phase de répartition des fonctions comportementale peut être réalisée juste avant l'exécution de l'application pour prendre en compte les contraintes de l'architecture cible. Cette distribution peut être un résultat d'un outil de répartition de code et d'équilibrage de charge ayant le but de répartir au mieux les Cac/s. Cette opération peut être cachée au programmeur. Le module est ici un outil de

1. Nous verrons plus loin que les Cac/s ayant le même comportement peuvent être répartis sur plusieurs noeuds.

répartition

- 2 - Le groupement des fonctions peut être lié à leur programmation. Les modules peuvent explicitement être le résultat d'une programmation modulaire comme par exemple une bibliothèque de fonctions comportementales livrée sous forme d'un module compilé. Le module est considéré ici comme un outil de développement.

La distribution des fonctions comportementales dans les modules permet de multiples possibilités. Si un module ne contient qu'un et un seul comportement, il peut être associé à la notion de «classe» de Cac. Mais la notion de module est plus souple que la notion de classe: un module peut contenir plusieurs fonctions comportementales et plusieurs modules peuvent contenir la même fonction comportementale parmi leur ensemble de fonctions. Le module est une entité de distribution de code et des objets à l'exécution. La distribution peut être facilement modifiée par le programmeur sans changer le source de l'application.

- II - 2.1.1 L'intérieur d'un module.

La mémoire d'un module est structurée en deux espaces:

- 1 - Une table des fonctions comportementales du module. Cet espace contient plus exactement le code des comportements du module.

- 2 - Un espace mémoire de Cac. Chaque module définit une zone mémoire réservée aux Cac/s du module à l'exécution de l'application. La structure d'un Cac est entièrement allouée dans cette zone.

Chaque module possède une activité spécifique gérant les créations de composants dans le module. Cet élément, appelé **gestionnaire de module**, est réalisé par un Cac spécialisé. Il reçoit les "requêtes au module" et en particulier les demandes de création de composant.

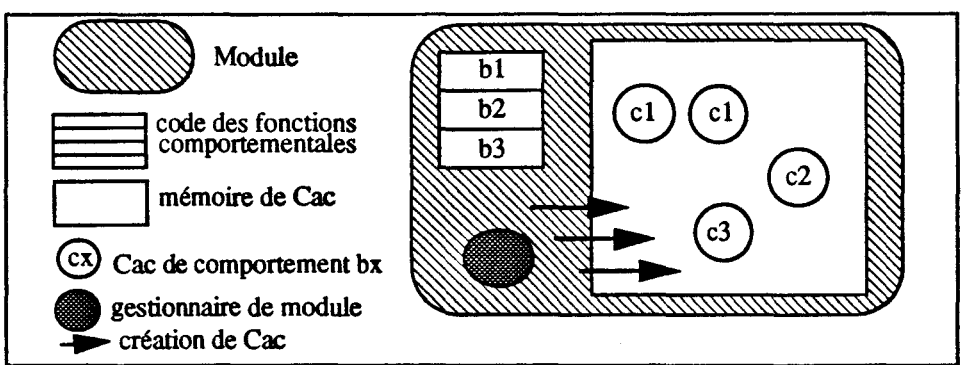


figure 2.0 *Structure d'un module*

- II - 2.1.2 L'interface de programmation d'un module

Un module se décrit dans un fichier C. Le nom du fichier C désigne le nom logique du module. Ce fichier contient la liste des fonctions comportementales du module. La fonction *main()* du fichier C initialise le module, crée l'environnement du module et déclare les fonctions comportementales du module et diffuse les noms des comportements aux autres modules de l'application.

La primitive *InsertBehavior(function F, Behavior B)* ajoute un nouveau comportement au module. Cette fonction est une instruction de la fonction principale du module (la fonction *main()*) et intervient dans la phase d'initialisation des modules. Durant l'exécution de l'application, un module ne peut accueillir que des Cac/s de comportements introduits par l'instruction *InsertBehavior()*. Cette fonction associe un comportement de nom *B* à une fonction *F* écrite en C. Cette fonction *F* doit être inclue dans le fichier avant l'instruction *InsertBehavior()*.

Le programmeur spécifie un module particulier comprenant le comportement du Cac «root». Ce Cac lancé après l'initialisation de tous les modules par le système, lance l'application. La fonction comportementale de ce Cac est programmable comme toutes autres fonctions comportementales.

La fonction *InitModule()* initialise les structures de données internes du système. La fonction *StartModule()* diffuse les noms des comportements du module et lance le Cac gestionnaire de module.

Exemples

Calcul des factorielles

Nous reprenons l'exemple du calcul des factorielles. Nous décrivons ici un module d'accueil pour ces Cac/s factoriels.

```
/*                          M_FAC                          */

#include<Prim_Cac.h>
extern void fac(int *arg)

int main()
    {
    InitModule(); /*initilisation des structures internes*/
    InsertBehavior(fac,FAC); /*insertion d'un comportement*/
    StartModule(); /*lancement du module*/
}
```

figure 2.1 *Le module M_Fac*

NB: Ici le module Factoriel peut être assimilé à une classe de Cac factorielle.

Le module *M_Start* (figure 2.2 «Le module M_Start») démarre l'application. Il comprend la fonction comportementale du «root» Cac créé automatiquement par le système (dans la fonction *StartModule()*). A la fin de son existence ce composant retourne un message de fin d'application au système qui produira l'arrêt total de l'ensemble de l'application.

```
/*                M_START                */

#include<Prim_Cac.h>
extern void start( int *arg);

int main()
     {
     InitModule();
     InsertBehavior(start,START);
     StartModule();
}
```

figure 2.2 *Le module M_Start*

NB: Les deux modules seront compilés séparément. Une fois compilé un module peut être utilisé dans plusieurs applications.

Reprenons maintenant l'exemple de l'arbre binaire de recherche. Nous pouvons définir un seul module contenant les deux fonctions comportementales.

```
/*                M_NODE                */

#include<Prim_Cac.h>
extern void node(int *arg);
extern void start(int *arg);

int main()
     {
     InitModule();
     InsertBehavior(node,NODE);
     InsertBehavior(start,START);
     StartModule();
}
```

figure 2.3 *Le module M_Node*

Nous verrons dans les sections suivantes comment créer un réel parallélisme entre les Cac/s dont les comportements sont regroupés dans un seul module (Cas de l'exemple ci-dessus).

- II - 2.2 Les modules à l'exécution

Après la compilation des modules d'une application, le programmeur doit décrire un réseau de modules pour les répartir sur les noeuds de la machine cible. Cette opération indépendante de la phase de programmation peut ainsi prendre en compte les caractéristiques matérielles de l'architecture cible tels que le nombre de noeuds de la machine et le réseau de connection des noeuds.

Chaque module compilé est implanté sur un noeud de la machine cible; un noeud de la machine pouvant accueillir plusieurs modules. Le programmeur peut définir lui-même la localisation de chaque module sur les noeuds, mais peut aussi laisser cette tâche au système sous-jacent. Les composants actifs de communication sont créés en fonction de la localisation de leur fonction comportementale. Chaque composant est attaché à un module et réside toute sa vie sur le noeud de la machine où est localisé ce module.

- II - 2.2.1 La duplication de module

L'environnement des Cac/s propose un outil supplémentaire de répartition du code, la duplication des modules. Un même module peut, à l'exécution, être dupliqué sur plusieurs noeuds. Le programmeur décrit dans un premier temps ses modules de composants et lors d'une phase de préparation de l'application, un réseau de modules. Il définit pour chaque module le nombre de copies.

La duplication présente deux avantages:

- 1- Décharger les noeuds en mémoire. Un module est dupliqué en plusieurs exemplaires implantés sur des noeuds différents de la machine. Le système répartit alors la création des composants sur les modules dupliqués de même nom. La mémoire de chaque noeud est donc allouée uniformément.

- 2 - Augmenter le parallélisme de l'application. Sans duplication de module, les Cac/s de même comportement sont a priori[1] dans le même module. Il ne peut donc y avoir de réel parallélisme d'exécution entre ces Cac/s. Si la duplication de module est permise, l'outil permet un réel parallélisme entre les Cac/s des modules dupliqués.

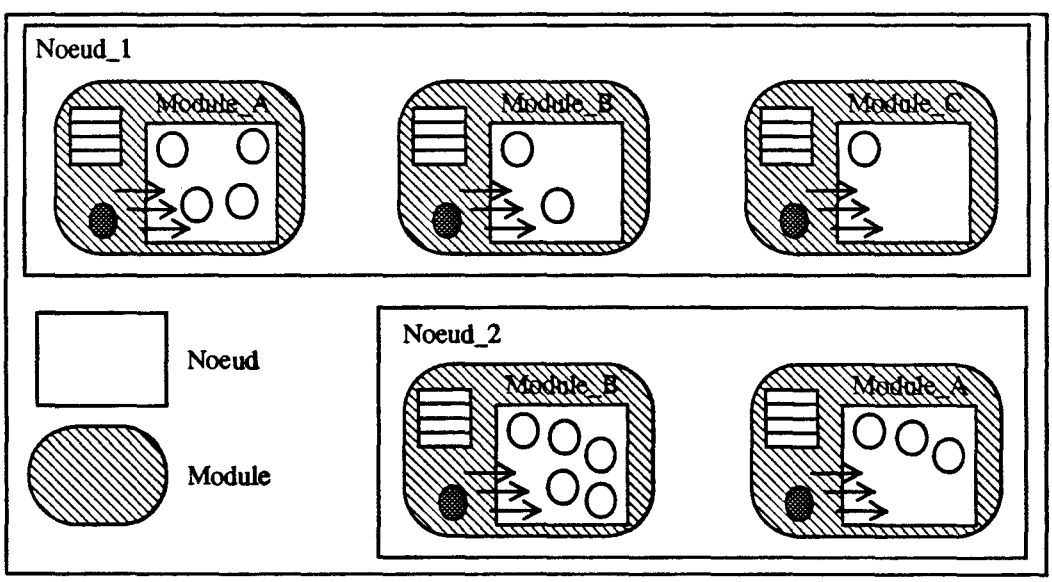


figure 2.4 *Les modules sur les noeuds*

1. Si le code du comportement n'est pas inséré dans des modules de types différents (non dupliqués)

Les composants de même comportement doivent être répartis sur l'ensemble des modules ayant ce comportement. La distribution est basée sur la coopération des différents gestionnaires de module notamment de ceux dupliqués. Un Cac est créé sur le noeud le moins chargé, où est implanté un module ayant la fonction comportementale du Cac.

Le programmeur décompose son application en un ensemble de modules répartissant à un premier niveau le code de son application. Puis, avant l'exécution, il définit le nombre de duplications de chacun de ces modules. Cette phase est indépendante de la conception proprement dite, c'est un paramètre de l'exécution. Il peut ainsi, en fonction de la machine cible, adapter l'exécution de son application sans en modifier le source.

- II - 2.2.2 Construction du réseau de modules

A l'exécution une application est représentée par un réseau de modules. Le réseau est décrit dans un fichier spécifique qui reprend la liste des modules compilés de l'application et pour chacun d'eux, le nombre de duplications.

La commande *ConstructModulesNet ...>Fichier* construit le fichier représentant le réseau de modules. Les paramètres de la commande sont une liste de couples (nom du module compilé, nombre de copies). Cette commande génére un fichier, repris par le logiciel de base lors du chargement de l'application.

exemples

Nous reprenons l'exemple des factorielles. Nous avons précédemment décrit deux modules *M_Fac* et *M_Start*. Un réseau de modules contenant, 1 module *M_Start* et 10 modules *M_Fac* dupliqués, est produit par la commande suivante.

```
/*        Génération du réseau de module pour l'application factorielle  */

      ConstructModulesNet 1 M_Start, 10 M_Fac > Factoriel10
```

figure 2.5 *Construction d'un réseau pour l'application factorielle*

NB: La description du réseau de modules est contenue dans le fichier Factorielle10.

La création d'un réseau de 20 modules *M_Node* pour l'application de l'arbre binaire de recherche s'effectue somme suit:

```
/*        Génération du réseau de module pour l'application Arbre        */

      ConstructModulesNet 20 M_Node > Arbre20
```

figure 2.6 *Construction d'un réseau pour l'application de l'arbre*

- II - 2.2.3 Le réseau de modules sur les noeuds

Le réseau de modules doit alors être implanté («mappé») sur les noeuds de la machine cible. Cette répartition peut s'effectuer de deux manières:

- 1 - Le programmeur de l'application peut répartir lui-même les modules sur les noeuds. Il reprend le fichier généré par la commande *ConstructModulesNet()* et précise pour chaque module, un noeud de la machine cible. Il décrit ainsi la répartition du code et des données sur la machine.

- 2 - La répartition des modules peut aussi être à la charge du logiciel de base. Cette option est mise par défaut par la commande *ConstructModulesNet()*. Selon la charge de chaque noeud du réseau, le système distribue les modules sur les noeuds.

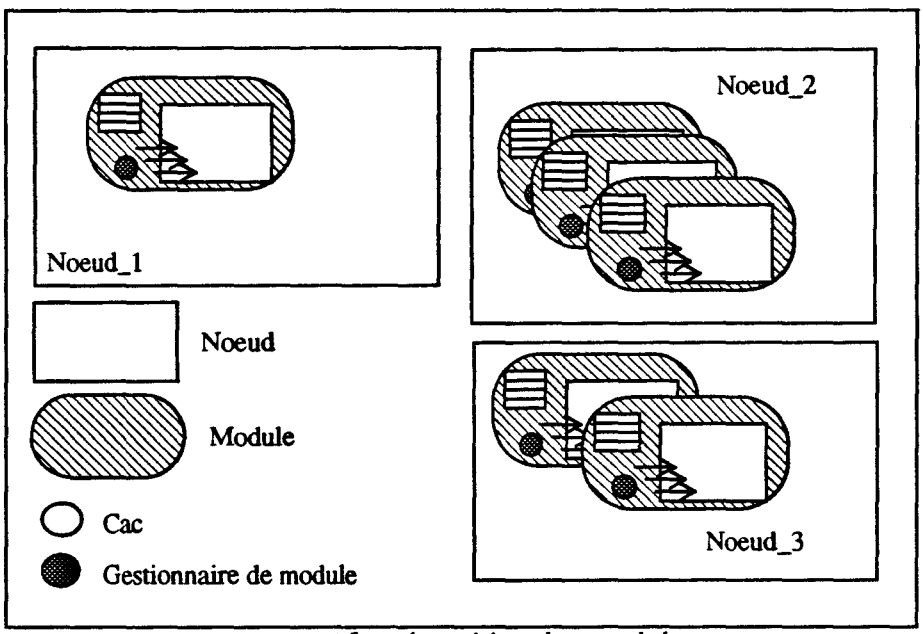


figure 2.7 *La répartition des modules*

NB: Lors de la phase d'initialisation, les modules se communiquent entre eux leur nom logique ainsi que les noms des comportements de Cac/s qu'ils contiennent.

La primitive *InsertBehavior()* et la duplication de module (dans la construction d'un réseau) sont deux outils de répartition du code et des Cac/s sur les noeuds de la machine cible. Sans duplication de module et avec un seul comportement par module, la notion de «classe» est aussi une structure de répartition à l'exécution. Au contraire, si tout les fonctions comportementales sont regroupées dans un même module qui est dupliqué sur tous les noeuds, l'ensemble du code est alors chargé sur tous les noeuds et un Cac peut ainsi être créé sur n'importe quel noeud. D'autres solutions intermédiaires peuvent être réalisées grâce à ces deux outils.

- II - 3. Implantation et Mesures

Un prototype Cac est implanté sur un multicomputer: le MultiCluster II [Parsytec90] de chez *Parsytec*. Chaque noeud est constitué d'un T800, Transputer (25MHZ, 10 MIPS) et 2 MOctets de mémoire locale. Il communique avec ses voisins à travers quatre liens avec une vitesse de 20 Mbits/s. La topologie du réseau est reconfigurable.

- II - 3.1 Le système Hélios

Nous avons choisi le système d'exploitation Hélios disponible sur le Multicluster II pour les raisons suivantes: Il permet la programmation en *C*, gère le «routage» des messages sur les noeuds et propose un outil de «mapping».

Hélios [Perihelion 89] est un système d'exploitation UNIX-like [Ritchie74] réparti; c'est à dire qu'il permet d'utiliser le parallélisme de la machine sous-jacente mais aussi ses possibilités de communication. Nous avons utilisé le compilateur C-ANSI du système Hélios. Nous détaillerons ici les outils pour l'expression sous Hélios du parallélisme et la communication.

- II - 3.1.1 Expression du parallélisme sous Hélios

La notion de tâche Hélios est proche de celle de processus UNIX. A une tâche Hélios est attaché un certain nombre de ressources systèmes. Une tâche Hélios comprend du code (ensemble de fonctions comprenant la fonction *main()*) et des données (pile ou données locales à la tâche).

Une application parallèle Hélios se constitue d'un ensemble de tâches s'exécutant en parallèle (ou pseudo) sur les différents noeuds de la machine. Le CDL (*Component Distribution Language*) permet de créer des tâches Hélios. Les tâches Hélios sont créées sur un Transputer et ne peuvent migrer au cours de leur exécution. Le programmeur a la possibilité de laisser au CDL le soin de placer au mieux les tâches Hélios sur l'ensemble des Transputers et ceci en fonction de la charge des Transputers au moment du lancement du programme. Le programmeur peut forcer l'exécution d'une tâche sur un Transputer donné en vue de minimiser essentiellement les coûts de communication.

Le Transputer dispose d'un micro-code implémentant un ordonnanceur de processus élémentaire. Ce micro-code est accessible par un jeu réduit d'instruction. Chaque processus élémentaire dispose d'une pile d'évaluation des appels procéduraux (au moins 1 KOctets). Ils sont créés à l'intérieur d'une tâche et se partagent l'espace mémoire de la tâche où ils sont localisés. Ces processus sont légers et économiques : ils ne nécessitent pas de ressource mémoire importante et sont ordonnancés par le «hardware».

- II - 3.1.2 Les communications sous Hélios

Le CDL permet de répartir un ensemble de tâches sur l'ensemble des noeuds de la machine, mais aussi d'établir des liens de communication entre tâches. Les communications de bas niveau du système Hélios, communication par «Port» ne sont pas fiabilisées. Nous avons donc choisi les communications par tubes UNIX. Les deux primitives *write* (ie *read*) UNIX sont utilisées pour émettre (ie recevoir) des données à une autre tâche.

Les processus d'une tâche peuvent communiquer au travers de la mémoire de la tâche. Des sémaphores permettent de protéger l'accès à ces données et d'en garantir ainsi l'intégrité. Pour communiquer avec des processus d'autres tâches, les processus utiliseront les tubes.

- II - 3.1.3 Mesures d'Hélios

Avant de présenter les mesures du run-time des Cac, nous présenterons les performances du système d'exploitation Hélios support du run-time. Une précédente étude [Hemery91, Lazure90] a testé les performances du système Hélios.

Nous reprenons, dans la figure 3.0 «Mesures d'Hélios», les principaux résultats de cette évaluation. Certaines mesures ont été réévaluées dans les conditions optimales. Les temps obtenus sont donnés en micro-seconde.

Fonctions	Temps en µs
affectation	0.6
appel de fonction C	2.7
allocation mémoire (64 - 1024o)	70.0 - 70.0
libération mémoire (64 - 1024o)	70.0 - 70.0
création de processus	150.0
memcpy(64 - 1024o)	9.0 - 67.0
écriture dans un pipe (64 - 1024o)	1080.0 - 1850.0

figure 3.0 *Mesures d'Hélios*

L'étude montre que l'allocation mémoire et la création de processus sont coûteuses sur Hélios par rapport à un appel de fonction. La couche de communication par tube Hélios est fiabilisée, mais les performances par rapport aux communications par «Port» sont moindres (voir l'étude de D. Lazure [Lazure90]).

NB: Sur notre version d'Hélios, le rapport entre le temps d'une communication et une affectation est de l'ordre de 2000. Ce rapport n'est pas encourageant sur une telle architecture. W. Athas [Athas88] envisage des rapports de 50.

- II - 3.2 Le prototype

Le prototype Cac est développé au dessus d'une interface système. Elle a pour rôle de cacher au run-time Cac les détails matériels. L'interface système propose une gestion des boîtes aux lettres, appelées *Boxes* et une gestion mémoire au dessus du système d'exploitation. Nous décrirons ici la réalisation des *Boxes*,

Dans une seconde section, nous aborderons le run-time Cac/s en détaillant l'implantation des structures Modules et Composants.

- II - 3.2.1 L'interface système

Une application parallèle Hélios est constituée d'un ensemble de tâches (Processus UNIX) communiquant par des tubes (pipes UNIX). Nous avons choisi de connecter complètement les différentes tâches de l'application. Pour tout couple de tâches (T_i, T_j), elles sont connectées par deux tubes: Un pour les communications de T_i vers T_j et un autre pour les communications de T_j vers T_i.

Des processus sont créés à l'intérieur des tâches. Les processus d'une tâche communiquent par la mémoire locale de la tâche, globale à l'ensemble des processus de la tâche. Il n'y a pas de mémoire commune et partagée entre tâches. Pour communiquer avec les processus d'une autre tâche, les processus doivent accéder aux tubes. Nous nous proposons de définir un mécanisme évolué et uniforme de communication inter-processus. La structure de boite aux lettres (*Box*) est évoluée car elle permet à la fois la synchronisation et la communication. Dans un premier temps, nous décrirons la notion de *Box* qui a été définie; nous détaillerons ensuite les manipulations que nous proposons pour les *Boxes*. Nous donnerons enfin des éléments sur la réalisation de ces *Boxes*.

- II - 3.2.1.1 Structure d'une *Box*

Une *Box* est une structure de données qui contient des messages. Un message est une zone mémoire non structurée de longueur fixe définie par l'utilisateur. Il n'y a pas de limitation à la longueur des messages, ni sur le nombre de messages dans une *Box*. (mis à part la capacité mémoire des Transputers!)

Un message est émis par une primitive de dépôt (*PutToBox*) qui le place dans une *Box*. Un processus peut attendre des messages dans une *Box* (*WaitOnBox*) et accéder aux contenus de ces messages (*ReadFromBox*).

Une *Box* est créée sur une tâche où elle résidera tout au long de son existence. Un mécanisme de migration de *Box* n'a pas été étudié.

Les *Boxes* sont identifiées par un mot de 4 Octets. Les deux premiers Octets sont utilisés pour identifier la tâche où se trouve la *Box*. Deux autres Octets servent à identifier la *Box* dans la tâche.

Au cours de l'exécution d'un programme, les numéros de *Box* sont uniques et ne sont pas réutilisés.

Les primitives de manipulation de *Box* sont transparentes à la localisation des *Boxes*. Elles fonctionnent de la même manière pour une *Box* locale (processus appelant la primitive appartenant à la même tâche que la *Box*) que pour une *Box* distante (processus appelant référençant une *Box* d'une autre tâche).

Les messages d'une *Box* sont identifiés par un numéro. Exemple: un processus peut lire le 3ème message d'une *Box* et détruire ce 3ème message. (*ReadFromBox* et *DeleteMessageFromBox*). Si deux processus exécutent ces mêmes actions, le résultat est indéterminé. Pour éviter le problème de l'accès en consultation par plusieurs processus, nous avons introduit la notion de verrou sur les *Boxes*. Si une *Box* est partagée entre plusieurs processus pour y extraire des messages, la *Box* peut être protégée par les primitives *AcquireBox* et *ReleaseBox*.

- II - 3.2.1.2 Structure de la tâche

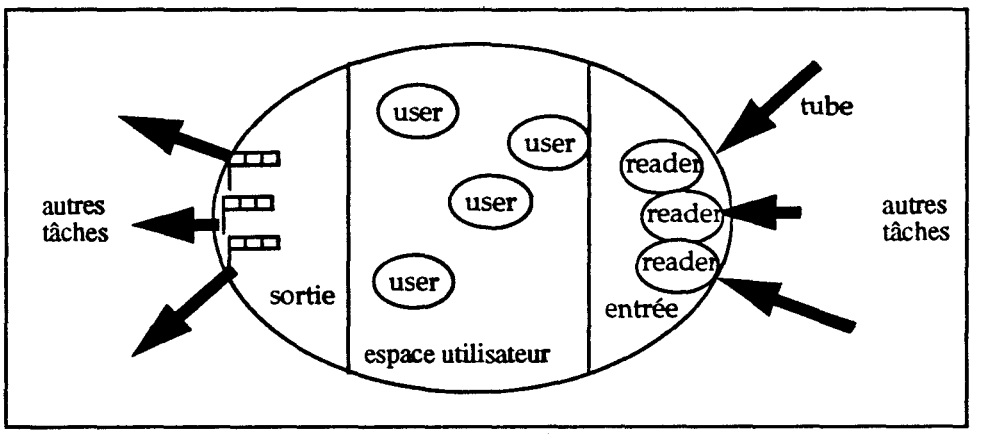


figure 3.1 *Structure d'une Tâche*

La tâche Hélios se structure en 3 zones: Une **zone d'entrée** comprenant des tubes de communication ouverts en lecture vers les autres tâches de l'application. Les messages provenant des autres tâches arriveront sur ces tubes. Un processus reader est chargé de lire les messages arrivant sur chacun de ces tubes. Ces processus reader sont créés par la primitive *InitCommunication*. Une deuxième zone de traitement (**espace utilisateur**) contiendra le code et les données des processus utilisateurs. Nous y retrouvons tous les processus utilisateurs créés par la primitive Hélios *Fork*. Une troisième **zone de sortie** contiendra les tubes de communication vers les autres tâches. Lorsqu'un processus utilisateur émet un message vers une *Box* distante (*PutToBox* distant), son message sera transmis sur le tube de communication vers la tâche où réside la *Box*. Un sémaphore d'exclusion mutuelle protège le tube dans le cas où plusieurs processus envoient des messages vers la même tâche.

- II - 3.2.1.3 Réalisation des *Boxes*

Une *Box* contient les éléments suivants: Une liste de message *ListMessage*, le nombre de messages de la *Box* (NbMessage), un sémaphore d'exclusion mutuelle *Mutex* pour garantir l'accès exclusif à la liste par plusieurs processus, et un sémaphore *Attente* qui mémorise le nombre de nouveaux messages arrivés sur la *Box*.

```
typedef struct {
        List ListMessage;
        int NbMessage;
        Semaphore Attente;
        Semaphore Mutex;
        } Box, *BoxPtr;
```

L'envoi de message (*PutToBox*) se déroule de la manière suivante: Si la *Box* appartient à la même tâche que le processus émetteur, le message s'ajoute à la liste de ceux de la *Box*. Dans le cas contraire, le message est émis vers la tâche où réside la *Box*. (Ecriture sur le tube en sortie associé à la tâche)

La lecture de messages s'effectue par la primitive *ReadFromBox*. Cette primitive renvoie la copie d'un message de la *Box*. Si cette *Box* est locale, l'accès est direct à la structure *Box*. Si la *Box* est distante, la requête est émise vers la tâche distante. Une Box *Reply* est créée pour l'attente de la réponse. Cette requête est exécutée par le processus *reader* distant qui exécutera un *ReadFromBox* local et transmettra la réponse au processus demandeur. Le processus demandeur récupère le résultat et détruit la *Box Reply*.

Les autres primitives sont réalisées sur le même schéma.

- II - 3.2.1.4 Détails d'utilisation des primitives

Les utilisateurs des *Boxes* ont utilisé facilement ces primitives. Deux primitives ont cependant posé quelques problèmes. Le *WaitOnBox* et le *NumberMessageInBox*:

La primitive *NumberMessageInBox* est une primitive non bloquante pour le processus appelant. Elle renvoie le nombre de messages de la *Box* au moment de l'appel.

La primitive *WaitOnBox* est une primitive qui peut être bloquante pour le processus appelant. Le processus appelant est bloqué si aucun nouveau message est arrivé depuis le dernier *WaitOnBox* ou *NumberMessageInBox*. Le *WaitOnBox* renvoie le nombre de messages de la *Box*.

InitCommunication (int nbfd); Cette primitive initialise, au niveau de chaque tâche, les tubes de communication avec les autres tâches. Vous devez donc lui passer en paramètre le nombre *nbfd* de tâches de l'application.

EndCommunication (int nbfd); Cette primitive ferme les *nbfd* tubes de communication avec les autres tâches. Cette primitive doit être appelée à la fin de l'application et permet une fin propre du programme.

- II - 3.2.2 Gestion mémoire

Des primitives de gestion mémoire ont été développées pour fournir des solutions au problème de la gestion mémoire aux processus de la tâche. Les primitives de gestion mémoire (Malloc et Free) sont protégées par un sémaphore d'exclusion mutuelle.

void InitMemory ()

Cette primitive initialise le sémaphore d'exclusion mutuelle d'accès au tas. *InitMemory* doit être appelée au tout début du programme (avant le *InitCommunication*).

*void *new (int size)*

C'est la primitive d'allocation mémoire de *size* Octets qui renvoie un pointeur sur la zone allouée.

*void dispose (void *p)*

C'est la primitive de libération mémoire. La zone mémoire d'adresse *p* a été alloué par la primitive *new*.

- II - 3.2.3 Le prototype Cac

En utilisant les fonctionnalités du run-time système, nous avons développé un prototype implantant les composants actifs de communication. Nous décrivons dans cette section le prototype Cac. Il se présente sous la forme d'une bibliothèques de primitives Cac qui ont été développées en C et bien sûr utilisables à partir du C. Le choix du langage C rend aisé le portage du prototype sur d'autres plates-formes.

- II - 3.2.3.1 Structure du module

Le programmeur décrit les modules de son application dans des modules source C. (Un fichier C). Un module source est constitué d'un ensemble de fonctions décrivant les comportements des composants du module. Chaque module source est ensuite compilé et il y a génération d'un **module exécutable** (fichier contenant du code machine). Au lancement de l'application, le module exécutable est chargé en mémoire et nous l'appellerons, module en cours d'exécution ou plus simplement module. Sur le système Hélios, un module en cours d'exécution s'appelle une tâche. Un module exécutable peut être lancé plus d'une fois et nous pouvons donc en avoir plusieurs en cours d'exécution construits chacun à partir du même module exécutable (duplication de module).

Pour décrire la répartition de son application en terme de module, le programmeur utilisera un outil de description de son application: le *CDL*. Le développeur décrit dans un fichier *CDL*, les différents modules à l'exécution (nom du module exécutable, liens de communications avec les autres modules et le numéro du Transputer d'exécution).

C'est de l'interprétation et l'évaluation de ce fichier *CDL* que va dépendre le lancement effectif de l'application. Les différentes tâches sont créées ainsi que les tubes de communication entre les tâches. La figure suivante décrit un fichier *CDL* avec 3 modules: deux modules *Fac* et un module *Start*:

```
#! /helios/bin/cdl

component Start {
      code Start;
      streams ,,,,<|Start_Start,<|Start_Fac0,<|Start_Fac1,
                  >|Start_Start,>|Fac0_Start,>|Fac1_Start;
       puid /Cluster/01;
}

component Fac0 {
      code Fac;
      streams ,,,,<|Fac0_Start,<|Fac0_Fac0,<|Fac0_Fac1,
                >|Start_Fac0,>|Fac0_Fac0,>|Fac1_Fac0;
      puid /Cluster/02;
}

component Fac1 {
      code Fac;
      streams ,,,,<|Fac1_Start,<|Fac1_Fac0,<|Fac1_Fac1,
                  >|Start_Fac1,>|Fac0_Fac1,>|Fac1_Fac1;
      puid /Cluster/03;
}

Start 3 0 1 ^^ Fac0 3 1 2 ^^ Fac1 3 2 2
```

figure 3.2 *Exemple de CDL décrivant trois modules*

La dernière ligne du fichier *CDL* décrit le lancement des modules. Chaque module est lancé avec 3 paramètres qui sont: le nombre total de modules, le numéro du module à l'exécution et le nombre de modules dupliqués du même type.

Un utilitaire développé avec l'interface système permet la génération automatique d'un *CDL*. Exemple, la commande

gene 3 1 Start 2 Fac > module3

génére un fichier *CDL* de nom *module3* contenant 1 module *Start* et 2 modules *Fac*. Le fichier module3 peut ensuite être exécuté. Cette commande réalise l'instruction *ConstructModulesNet()*.

A l'exécution, le module comprend les éléments suivants:

- **Un gestionnaire de module.** C'est un Cac spécialisé qui est chargé de créer les Cac/s à l'intérieur du module. Il en existe un par module. La création d'un Cac est dynamique et se fait à la demande d'un autre Cac qui s'adresse au Gestionnaire de Module.

- **Une table des fonctions.** Les Cac/s sont créés à partir de fonctions contenues dans la table des fonctions du module. La création de Cac dans un module n'est possible que si le comportement est défini au niveau de ce module.

- II - 3.2.3.2 Structure du Cac

Le prototype Cac implante la structure Cac par assemblage des éléments suivants:

- Une structure de *Box* qui a été détaillée au niveau de l'interface système. Cette boîte est créée par le gestionnaire de module en utilisant la primitive *NewBox*().
- Un processus Hélios pour l'activité du Cac. Il est créé par la primitive *Fork* Hélios. Ce processus est créé par le gestionnaire de module à partir d'une fonction de la table des fonctions du module. Le nom de la boîte aux lettres est passé en paramètre à la fonction.
- L'environnement local du Cac est initialement constitué par la pile d'exécution du processus créé. Cet environnement peut être étendu par le Cac en utilisant la primitive d'allocation mémoire *new*.

La figure ci-dessous illustre la structure du Cac telle qu'elle a été réalisée:

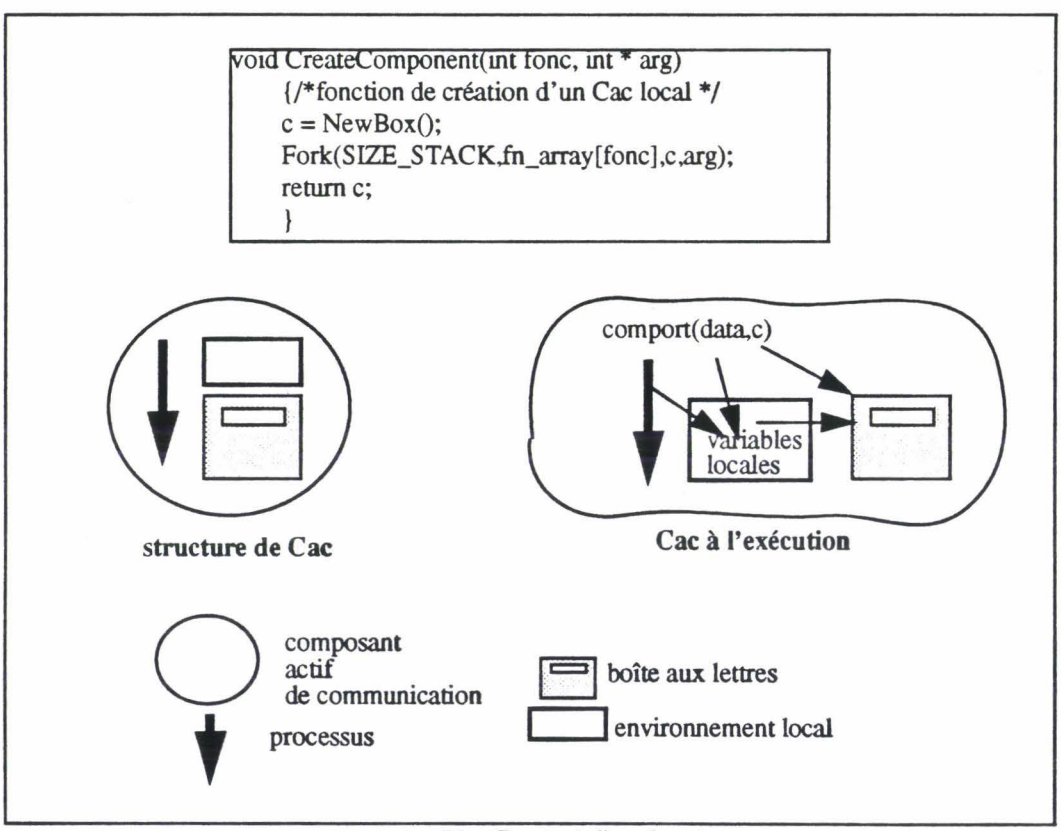


figure 3.3 *Un Cac/s à l'exécution*

- II - 3.2.3.3 Les communications entre Cac/s

Les communications entre Cac/s utilisent directement les communications par Box/s. Le détail de ces communications est déjà décrit dans la section - II - 3.2.1 «L'interface système».

Chaque composant possède dans son environnement local le nom de sa boite aux lettres qui l'identifie.

- II - 3.2.3.4 Stratégie de répartition des Cac/s sur les modules

Pour créer un composant, le Cac "créateur" doit appeler la primitive *NewComponent*(). Cette primitive est basée sur la coopération entre les modules. Nous avons développé une stratégie«naïve» de répartition pour réaliser nos premiers prototypes. La répartition des entités actives est primordiale dans le projet. Elle est succinctement abordée ici ; une autre thèse lui est consacrée.

Répartition en fonction des sites.

Les messages de création sont envoyés à un module spécifique (*module de distribution*) qui centralise la répartition des créations des Cac/s. Le module choisit le site le moins chargé contenant un module ayant la fonction comportementale. Le message de création est réémis vers le gestionnaire de ce module. Le module doit contenir un historique de la charge de chaque noeud. Le module de distribution est chargé sur un noeud qui lui est dédié.

La solution prend en compte la charge de chaque noeud ; mais la décision de création est centralisée dans un module spécifique.

- II - 3.2.4 Mesures

A partir du prototype réalisé sur le multicomputer, nous avons effectué une série de mesures pour évaluer les différents niveaux de développement de l'environnement des Cac/s.

- II - 3.2.4.1 Mesures du run-time Cac/s

Voici les mesures effectuées des principales primitives de l'interface de programmation des Cac/s.

Primitives	Temps en ms
NewComponnent (local - distant)	0.280 - 3.750
SendaMessage --- local (64 -1024o)	0.210 - 0.210
SendaMessage --- distant (64 -1024o)	3.460 - 4.250
GetMessage(ReadFromBox + DeleteInBox avec un message présent)	0.245
NewMessage	0.080
FreeMessage	0.205

figure 3.4 *Mesures du run-time, temps en milli-secondes*

Les mesures des primitives peuvent se calculer en cumulant le temps des différents appels aux primitives Hélios que comporte la primitive. Les primitives du prototype sont pénalisées par la gestion mémoire coûteuse d'Hélios.

Ces mesures doivent être remises dans le contexte parallèle de la machine cible. Les mauvaises performances de certaines primitives sont largement compensées par le parallélisme d'exécution des Cac/s.

- II - 3.2.4.2 Mesures de la répartition

Nous recherchons à mesurer les performances de la stratégie de répartition. Nous avons donc développé une application test.

Application utilisant un nombre important de Cac/s différents.

L'application test met en oeuvre un nombre important de Cac/s (800) de types différents:

- Les *Cac/s de type Job* effectuent un traitement. Il peut y avoir 8 types différents de *Job*. Ces *Jobs* matérialisent le parallélisme inter-objet.
- Un ou plusieurs Cac/*s Producteurs*. Ils demandent un nombre important de créations de *Job* et permettent ainsi de réaliser des demandes parallèles de création.

Nous mesurons le durée de création des Cac/s Job/s et non le temps total de l'application. Les *Producteurs* sont créés au départ de l'application (par le Cac root). Les *Jobs* de 8 types différents sont créés dynamiquement par les *producteurs*. L'application crée 800 Job/s.

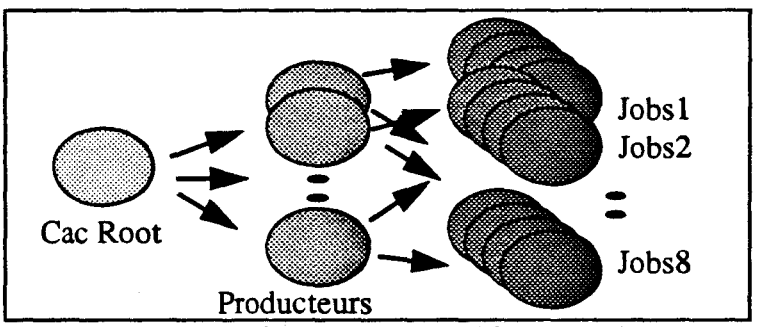


figure 3.5 *1..8 Producteurs, 800 Jobs(1..8)*

Tous les comportements sont localisés dans des modules distincts nommés: *Producteur, Job1, .., Job8*). Les modules *Producteur* ou *Job* peuvent être dupliqués.

Dans toutes les mesures suivantes, le nombre de transputers est toujours supérieur à celui des modules de l'application.

Demandes séquentielles de créations de Cac/s

Dans une première étape l'application ne crée qu'un seul *Producteur*. Les demandes de créations de Cac/s sont donc sérialisées par ce Cac demandeur. On mesure ici un nombre important de demandes de créations de Cac/s différents.

Nous faisons varier le nombre de types de modules Job. Le Producteur demande circulairement la création d'un Job de type différent. Les types des 800 Cac/s Job/s sont donc répartis entre les types de Job. Par exemple, pour la première mesure, il n'y a qu'un type de Job, l'application crée donc 800 Cac/s Job1).

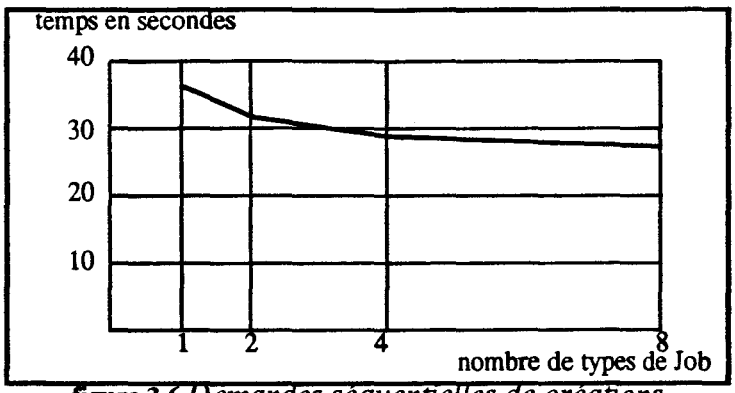


figure 3.6 *Demandes séquentielles de créations*

La stratégie centralise cette distribution dans le module de répartition qui redirige alors chaque demande vers l'un des modules *Job*.

Demandes parallèles de créations de Cac/s

Pour obtenir un parallélisme des demandes de création, nous augmentons le nombre de *Producteurs* et dupliquons le nombre de modules *Producteur*. Les demandes de créations de Job ne sont donc plus sérialisées. Le nombre total de *Job*/s reste égal à 800, et nous ne dupliquons toujours pas le module *Job*.

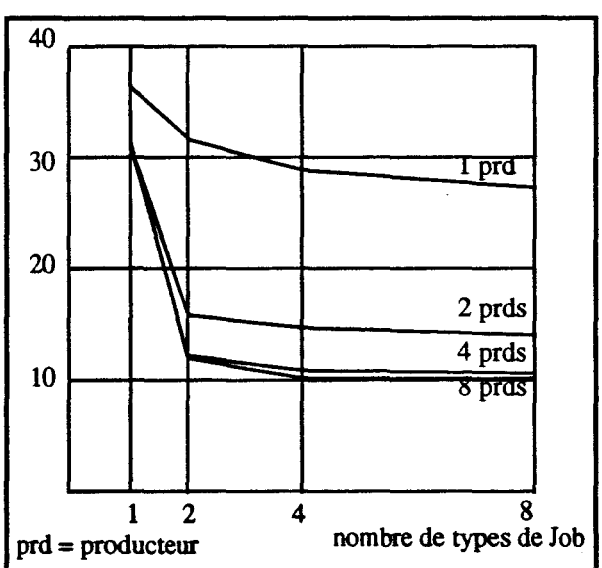


figure 3.7 *Demandes parallèles de créations de Cac/s*

Lorsque les demandes de création de Cac sont parallèles le nombre de messages supplémentaires engendrés par l'algorithme de répartition augmentent.

Créations parallèles de Cac/s avec des modules dupliqués

Nous reprenons les conditions précédentes, et dupliquons les modules Job. Les 800 Cac/s Jobs sont répartis sur les différents modules dupliqués. Par exemple, pour un coefficient de 4 et avec 2 types de modules Job nous avons 4*2 modules Job, 4 de chaque type).

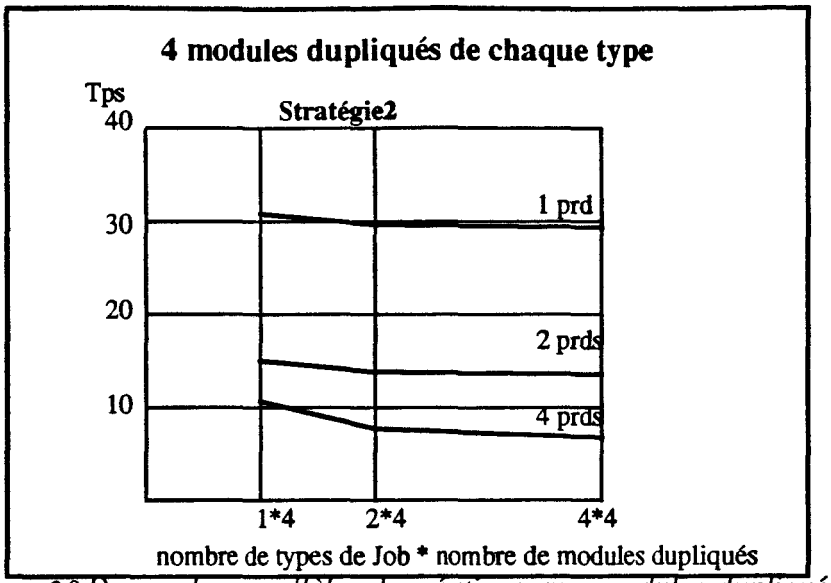


figure 3.8 *Demandes parallèles de créations avec modules dupliqués*

La stratégie produit des résultats performants lorsque les modules dupliqués est plus nombreux que les Producteurs.

- II - 3.2.4.3 Mesures des applications Cac/s

Nous cherchons à mesurer les applications Cac. Les mesures ont été effectuées sur 10 Transputers.

Mesures de l'application factorielle

Nous avons mesuré fac(500)[1].

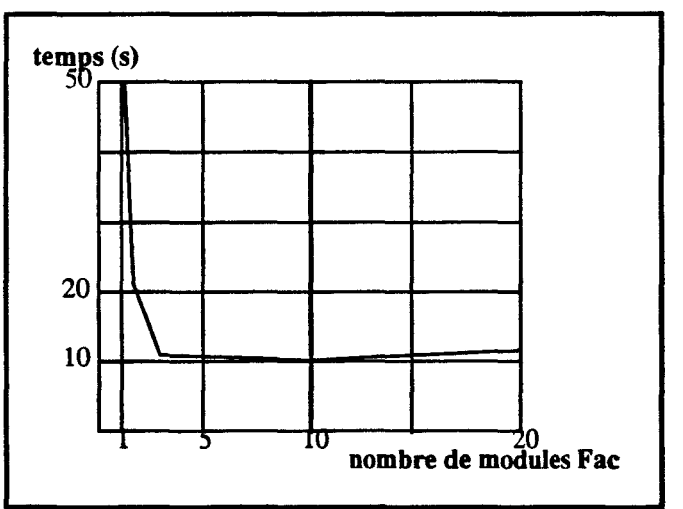


figure 3.9 *Mesures de l'application Fac*

Les mesures montrent l'intérêt de dupliquer les modules. Le coût des communications entre Cac/s des modules dupliqués est compensé par le parallélisme réel des Cac/s des modules dupliqués.

1. Pour éviter un dépassement de capacité dans les entiers (fac(500)), nous avons remplacé la multiplication par une addition. Nous obtenons alors la somme des n premiers nombres.

Mesures sur l'arbre binaire

Nous avons mesuré le temps d'exécution d'insertions et de recherches dans un arbre binaire (voir section - II - 1.3.2 «L'Arbre binaire de recherche»). Pour cette série de mesures, Max (nombre d'insertions et de recherches) = 500 avec des clés calculées aléatoirement. Le nombre de transputers est toujours égal à 10.

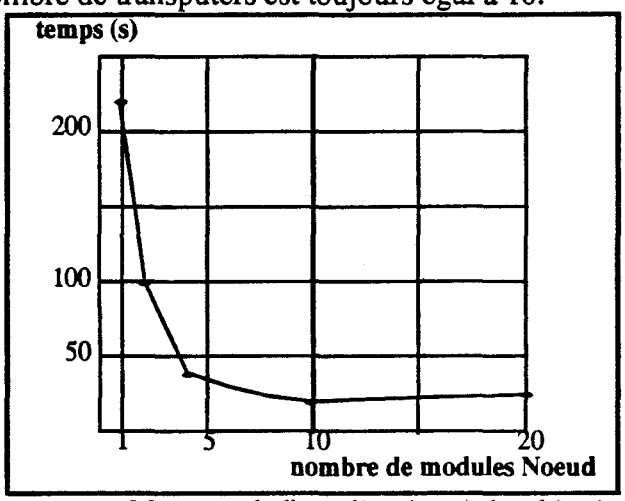


figure 3.10 *Mesures de l'application Arbre binaire*

Les performances liées à la duplication de module se confirment ici. L'application factorielle génére un nombre important de Cac/s durant l'ensemble du calcul. La programmation de l'arbre binaire de recherche utilise un parallélisme important lors des recherches, où il n'y a plus de création de Cac/s.

Les résultats se décomposent en deux parties:

- 1 - Le nombre de modules est inférieur au nombre de Transputers : La courbe est proche de la courbe idéale (t (n)= t(1) / n); mais elle s'éloigne au fur et à mesure que le nombre de messages augmente.
- 2 - Les Transputers sont moins nombreux que les modules de l'application : Certains modules dupliqués s'exécutent sur le même Transputer et les performances décroissent fortement. Dans cette partie de la courbe, les meilleurs résultats sont obtenus lorsque le nombre de modules dupliqués est un multiple du nombre de Transputers, les Cac/s sont alors répartis uniformément sur les Transputers.

Nous pouvons tirer plusieurs remarques de cet ensemble de mesures:

- 1- La duplication de module augmente les performances des applications possédant du parallélisme.
- 2 - Le nombre optimal de modules est directement lié au nombre de Transputers disponibles.
- 3 - Il faut éviter d'avoir plus de modules dupliqués que de Transputers
- 4 - La duplication de module doit se faire en phase terminale pour intégrer les contraintes matérielles telles que le nombre de Transputers ou la topologie du réseau de Transputers.

Conclusion

Nous avons défini un environnement pour la modèlisation et la conception des applications parallèles. L'environnement est basé sur une structure portable qu'est le Composant Actif de Communication. Cette abstraction du parallélisme est définie par une fonction comportementale et la structure d'un Cac est proche de celle d'un Acteur. Les composants sont regroupés dans des modules, entités de partage de code, qui sont distribués sur les différents noeuds de la machine cible. Les modules coopèrent entre eux pour prendre en charge la répartition de Cac/s sur les modules. La duplication de module constitue un outil de répartition permettant de distribuer au mieux le code sur les noeuds. Le programmeur peut ainsi influencer sur cette répartition et prend en compte, en dehors de la phase de programmation, les contraintes matérielles de la machine cible.

Les développements actuels portent sur le portage du run-time composant sur d'autres plates-formes matérielles. Une version est en cours de réalisation sur un réseau de station SUN en utilisant les processus UNIX, «sockets» et «threads» [Sun88] au dessus du système d'exploitation SUN-OS. Pour améliorer l'efficacité de l'environnement Cac, nous envisageons le «portage» du run-time Cac sur un micro-kernel *(Mach* [Acceta86] ou *Chorus* [Rozier88]).

Nous devons, pour poursuivre le prototype, développer un ramasse-miettes de composants. Le ramasse-miettes pourrait être réalisé de la manière suivante. Sur chaque module, une version du ramasse-miettes local traiterait tous les composants non référencés à l'extérieur du module. Les autres références de composants seraient pris en charge par un ramasse-miettes distribué utilisant le réseau de communication.

Nous devons également perfectionner l'algorithme de répartition des Cac/s sur les différents modules. Il faut en effet tenir compte qu'un module peut contenir plusieurs comportements et qu'un comportement peut être défini dans plusieurs modules de types différents. L'algorithme doit prendre en compte les Cac/s récupérés par le ramasse-miettes ainsi que la consommation CPU des Cac/s. La distribution des Cac devrait s'accompagner d'une primitive de création de Cac dirigée par le programmeur de l'application. Elle nécessitera une notion de noeud logique visible à l'utilisateur.

Les Composants Actifs de Communication vont nous servir de plate-forme aux développements d'autres applications qui sont:

- 1 - Une extension des Cac/s vers un modèle d'acteur, les Cac/s ayant une structure interne proche de celle des acteurs.
- 2 - L'objectif du projet Pvc reste l'implantation d'un langage parallèle à objets actifs sur un Multicomputer. Les Cac/s serviront de support à l'implantation de plusieurs modèles d'exécution de langages parallèles à objets actifs.

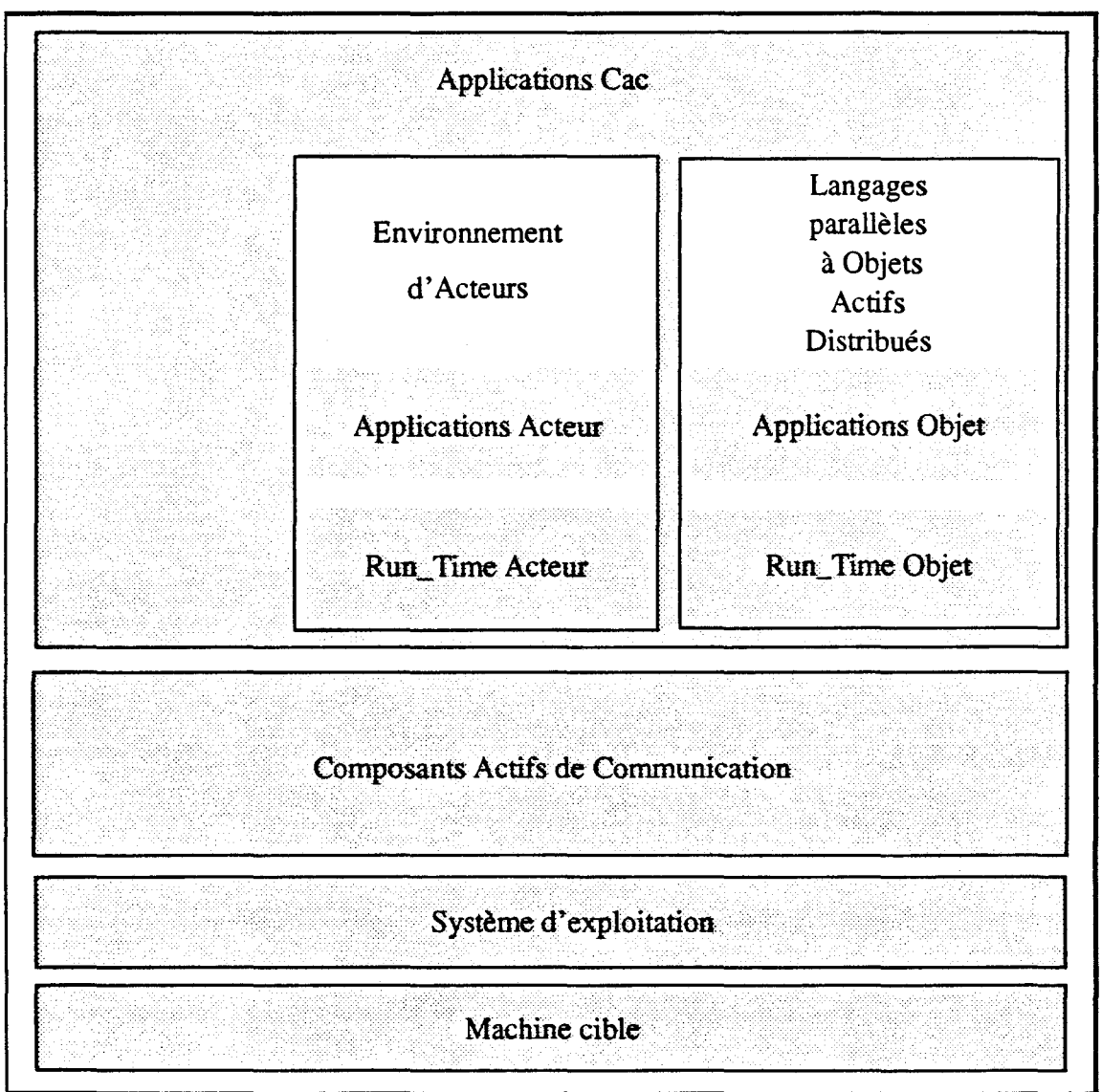


figure 4.0 *Applications des Cac/s*

Chapitre - III -

Implantations d'environnements d'Objets Actifs

Ce chapitre présente plusieurs implantations de langages d'objets actifs au dessus des Composants Actifs de Communication (décrits au chapitre précédent) sur une machine parallèle.

Le modèle des Objets Actifs permet de modèliser les applications fortement parallèles. Le parallélisme est engendré par la création d'objet et par un parallélisme intra-objet où plusieurs activités coexistent au sein d'un même objet (voir le chapitre I, section - I - 1.6 «Mode de création du parallélisme»). Ces langages offrent des outils évolués de description des objets et de leur comportement, tels que les classes, l'héritage, le mécanisme d'exécution de méthode, le polymorphisme...

Nous cherchons à reconstruire au dessus des Composants Actifs de Communication un environnement Objets de haut niveau. Les Cac/s sont considérés comme des Objets Actifs systèmes:

- Ils sont mono-programmés alors que les Objets Actifs (ou les Acteurs non-sérialisés) peuvent être multi-programmés.
- Chaque Cac est décrit par une fonction comportementale qui n'est pas structurée en méthodes (des classes d'objets actifs) ou en comportements (des scripts des Acteurs).
- L'environnement local d'un Cac est un ensemble de données locales d'une fonction et son caractère dynamique le différencie d'un ensemble de variables d'instance ou accointances.

- Les échanges entre composants ne sont structurés que par le programmeur de l'application. Il n'y a pas de mécanisme prédéfini d'invocation de méthode tel qu'on le trouve dans les modèles d'exécution objet.

Dans une première phase, nous avons reconstruit au dessus des Cac/s un environnement évolué d'Acteurs [Courtrai92f]. Cette extension est une première étape vers les objets multi-programmés. Cette évolution vers les acteurs est facilitée par le fait que la structure d'un Cac est proche de celle des Acteurs. Les Cac/s sont des objets actifs simples. Ils sont mono-programmés et possèdent un jeu d'instructions réduit. Nous cherchons maintenant à étoffer les possibilités des Cac/s. Pour permettre un plus grand degré de parallélisme, nos nouveaux Acteurs sont non-sérialisés (multi-programmés). De plus nous avons développé la primitive *become* de changement d'état d'un Acteur, celle ci pouvant entraîner la migration d'acteur. Dans cette même phase, et pour contrôler les flots d'exécution d'un même acteur, nous reprenons un mécanisme évolué de synchronisation proposé dans le langage *Act++* [Kafura89,90].

Dans une seconde phase, nous nous sommes attachés à rechercher un haut degré de parallélisme dans les Objets Actifs [Courtrai92c]. Nous proposons plusieurs représentations d'objets actifs distribués intrinsecquement parallèles pour machine parallèle. Ce parallélisme intra-objet de données et d'exécution est implicite. C'est l'originalité de la proposition qui s'oppose aux autres propositions rencontrées dans les langages d'objets fragmentés et qui se justifie du fait des machines visées. Cette réalisation est accompagnée d'un outil de répartition du code basé sur l'existence des classes à l'exécution. L'outil permet de nombreuses possibilités.

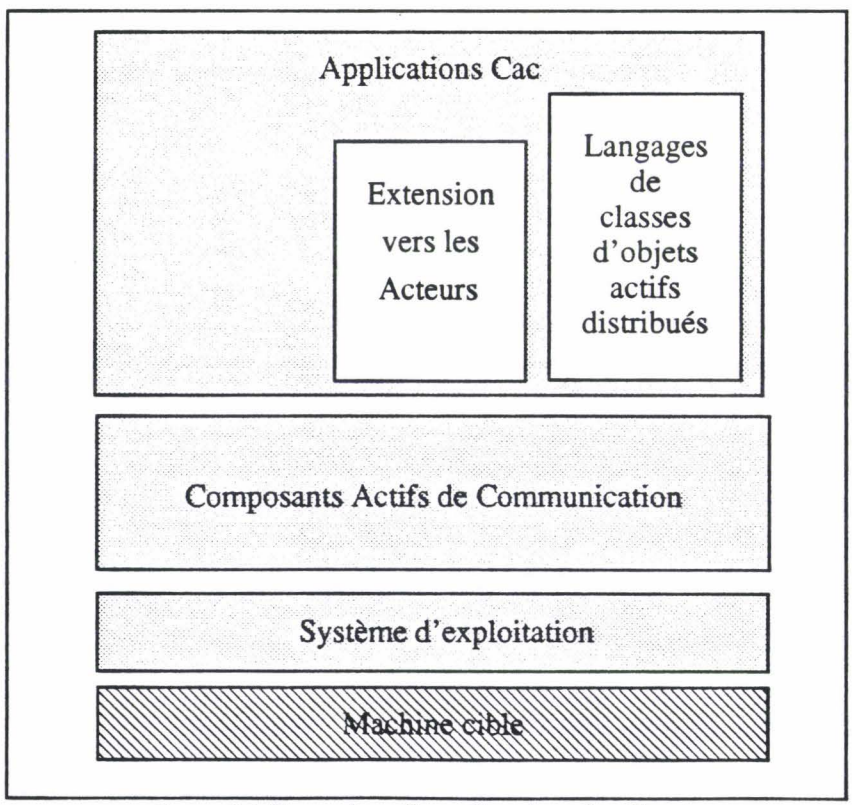


figure 1.0 *Domaine d'application des Cac/s*

- III - 1. Une extension vers les Acteurs d'Agha

Le modèle Acteur a été introduit par C. Hewitt [Hewitt73,77] au début des années 70 dans le domaine de l'intelligence artificielle pour la représentation des connaissances de manière distribuée. Le modèle a été repris, puis formalisé par G. Agha [Agha86,87] qui le présenta comme un modèle d'exécution parallèle dans les systèmes distribués. Les années de recherche sur le modèle Acteur ont fortement contribué à l'évolution de la programmation parallèle distribuée (sans mémoire commune). Le modèle Acteur est basé sur la programmation de petits objets possédant leur propre autonomie.

«Les acteurs sont des agents experts vivant en société et communiquant entre eux»
[Hewit73]

Le **script** d'un acteur décrit les différents **comportements** de l'acteur lors de la réception d'un message. L'acteur analyse le message et exécute la partie de code déterminée (code d'un comportement). Durant le traitement d'un message, l'acteur peut spécifier (par l'instruction **Become**) son état de remplacement (comportement) qui sera utilisé pour le traitement du message suivant. Le nouvel état peut utiliser un autre script. Un acteur non sérialisé peut traiter le message suivant avant la fin du traitement du message courant.

Le modèle d'exécution présente naturellement un haut degré de parallélisme du fait de l'activité des acteurs. Le parallélisme est engendré par la création d'acteur ou part les acteurs non-séralisés.

L'implantation d'un environnement d'acteurs sur machine parallèle semble intéressante et permettrait d'utiliser intrinsèquement le parallélisme de ces machines. Il existe actuellement peu de réalisations du modèle d'acteur sur de telles machines [Athas88, Markhoff90, Di Santo91].

Les Composants Actifs de Communication (décrits au chapitre II) ont une structure d'un objet. L'extension du modèle des Cac/s vers un modèle d'exécution Acteur semble aisée et peu coûteuse en développement. L'extension bénéficierait des propriétés du run-time des composants (notion de module et leur duplication). Néanmoins on peut citer les différences entre les deux modèles.

- 1 - Si un Cac est par définition séquentiel, le modèle Acteur possède, avec les acteurs non sérialisés, une ouverture sur un parallélisme intra-objet et la protection des données internes d'un acteur est garantie par son modèle d'exécution.
- 2 - Les extensions du modèle Acteur, telles que *Abcl/1*[Yonezawa86,86b,87] ou *Act++*[Kafura89,90], proposent des outils évolués permettant d'exprimer une synchronisation dans les applications, contrairement au modèle Cac où la synchronisation s'exprime par les opérations d'envoi et de réception de messages.

La réalisation du modèle d'exécution Acteur est une première étape vers les langages à objets actifs multi-programmés.

L'idée de cette extension est de proposer une structure du modèle Acteur pour la répartition du code. Si la répartition des comportements des Cac/s en modules doit être décrite dans l'application, le modèle Acteur propose une structure de script (correspondant aux classes dans les langages orientés objets [Wegner87]), pouvant être naturellement conservée à l'exécution (idée des classes de Cac).

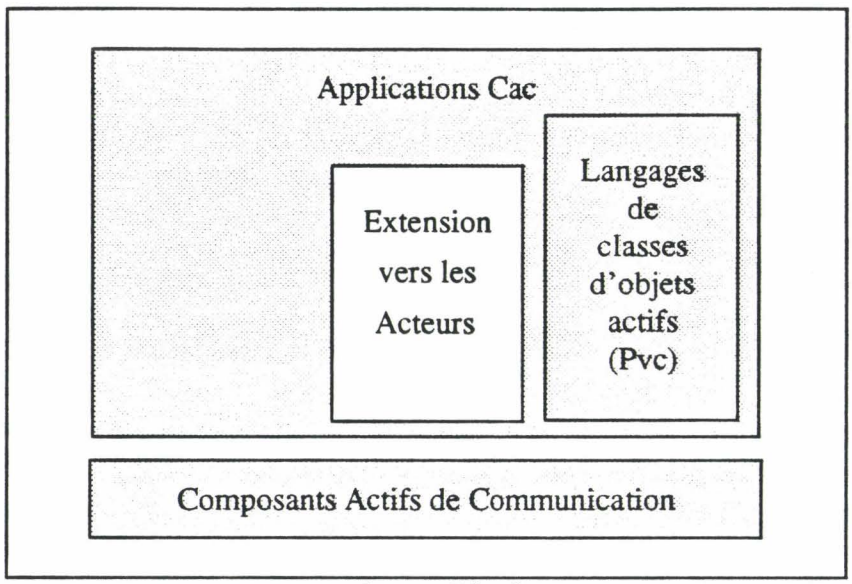


figure 1.1 *Le modèle d'acteur*

- III - 1.1 Implantation des acteurs

La structure d'un composant actif de communication est proche de celle des acteurs. Comme ceux-ci, un composant possède une boîte aux lettres qui l'identifie et dans laquelle il reçoit les messages des autres composants. Le processus du composant matérialise une activité autonome et l'environnement local contient des données locales. Les primitives de communication permettent les envois asynchrones de message entre composants.

Pour implanter le modèle acteur au dessus des Cac/s, une structure d'accueil matérialisant les acteurs doit être définie. Nous devons choisir un modèle de répartition des acteurs sur les noeuds de la machine et mettre en place un mécanisme d'envoi et de traitement de messages. La principale difficulté est la réalisation de la primitive *Become* qui permet de changer l'état d'un acteur sans en modifier son nom.

- III - 1.1.1 Le prototype Acteur

Un premier prototype implantant les acteurs au dessus des composants actifs de communication a été développé. Son objectif est de montrer la faisabilité de l'extension du run-time des composants en un modèle d'acteur. Les fonctionnalités de la programmation sur le prototype sont réduites au minimum: une fonction de création d'acteur, la fonction *Become* et deux primitives, l'une d'envoi et l'autre de réception de messages. Les seuls types de données utilisés sont les références sur les acteurs et les valeurs entières. Les valeurs entières sont des objets primitifs manipulés par valeur.

- III - 1.1.1.1 Répartition des acteurs

Un script est implanté dans un module Cac, entité de partage de code des composants actifs de communication. Nous utilisons la notion de *script* comme entité de répartition. Le module regroupe le code des comportements décrits dans un *script* et accueille les acteurs créés avec ce script. Chaque nom de *script* est confondu avec celui du module qui l'implante. Les modules pourront être dupliqués à l'exécution, ce qui permettra la répartition sur plusieurs noeuds des acteurs de même script.

Le code contenu dans un module se décompose en :

- une fonction *NewActor* de création d'acteur, prédéfinie dans tous les modules.
- une fonction *Become* correspondant au changement d'état, prédéfinie dans tous les modules.
- une fonction *Script* décrivant les comportements de l'acteur, à définir pour chaque module.

Le gestionnaire d'un module est considéré ici comme un serveur d'acteur (figure 1.2 «Un module = un script (ensemble de comportements)»); il traite et centralise toutes les créations d'acteurs dans le module. Il intervient également lors des changements d'état des acteurs (become). Le serveur contient la table de correspondance entre les noms des différents scripts et leurs modules associés. Celle-ci permet la création d'acteurs à distance (dans un autre module).

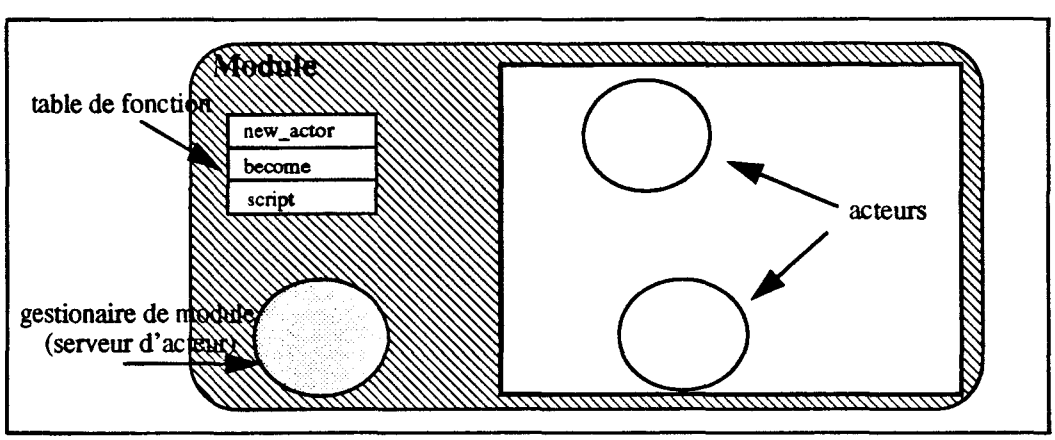


figure 1.2 *Un module = un script (ensemble de comportements)*

- III - 1.1.1.2 Structure des acteurs

Chaque acteur est matérialisé par un Cac localisé sur le module du script de l'acteur. Le nom du composant désigne l'acteur. Le processus d'un composant exécute le script local du module. Les accointances d'un acteur (données locales) sont stockées dans une table de l'environnement local du composant. La boîte aux lettres du composant sert de structure de communication de l'acteur

Lors d'une création d'acteur, le gestionnaire de module exécute la primitive de nom *NewActor()* de la table des fonctions du module; cela crée un Cac lancé avec la fonction *script* locale au module.

- III - 1.1.2 Les primitives Acteurs

La réalisation du modèle d'acteur s'accompagne du développement de primitives de gestion des acteurs. Nous avons défini un nouveau type, *Actor*, au dessus du type *Component*. La bibliothèque de programmation acteur contient une primitive, *NewActor*, de création d'acteur, une primitive *Become* ainsi que deux primitives d'envoi *Send* et de réception de messages *GetMessage*.

La programmation des acteurs sur le prototype s'écrit dans le langage C (langage de programmation des Cac/s) en utilisant cette bibliothèque des acteurs. Un acteur est décrit dans un fichier C qui contient son script (la fonction C de nom *script()*) décrivant les comportements de l'acteur.

- III - 1.1.2.1 Création d'un acteur

La création d'un acteur s'exprime par un appel à la fonction *NewActor(int S, int* accoin): Actor* où *S* est un nom de *script*. Le tableau *accoin*[1] contient les valeurs initiales des accointances pour le nouvel acteur. L'appel de *NewActor*() est synchrone et la fonction retourne le nom du nouvel acteur.

Réalisation

La création d'un acteur est une simple création de composant. Le script de l'acteur est directement associé à un nom de module.

```
Actor NewActor(int script, int * accoin)
    {
    return(NewComponent(script,SCRIPT,accoin));
}
```

figure 1.3 *La primitive NewActor*

La demande de création d'un composant s'effectue par l'envoi d'un message au gestionnaire du module courant. Si ce module est différent de celui où est localisé l'acteur, le gestionnaire de module détermine une adresse du module et lui redirige le message de création. Dans les deux cas, le gestionnaire concerné lance alors la fonction locale de création d'acteur qui retournera l'adresse du nouvel acteur.

- III - 1.1.2.2 Opération Become

Le changement d'état s'effectue par la primitive *Become(Actor my, int S, int *accoin)* où *S* est le nom du nouveau script. Les valeurs des accointances sont transmises par l'intermédiaire du tableau *accoin*. Le nom de l'acteur courant(*my*) est passé en paramètre pour conserver le même nom lors du mécanisme de changement d'état. Les accointances sont dupliquées et le nouvel état exécute le script *S*. Deux cas peuvent se présenter:

---

1. Les tableaux d'entiers du prototype permettent de représenter les accointances d'un acteur et les arguments des fonctions, une référence étant traitée par le système comme une valeur entière.

- Lorsque l'opération **Become** reste locale, elle est matérialisée par une création partielle de composant actif de communication, l'adresse devant être conservée. Pour permettre le parallélisme intra-acteur (acteur non-sérialisé), la réexécution du script engendre la création d'un nouveau processus (instruction *Fork*). Les deux processus exécutent le script sur des copies des accointances.

- Si l'acteur change de *script*, (**Become** distant), il change de module, mais son adresse doit rester inchangée. Un Cac matérialisant le nouvel état est créé sur le module du nouveau script. L'ancien composant n'est pas détruit (ou récupéré) et sert de relais; tous les messages reçus par ce composant sont alors redirigés vers le nouveau composant.

La fonction *CopyArray(tab)* du run-time Cac retourne une copie du tableau *tab* passé en paramètre.

```
void Become(Actor my, int script, int *accoin)
    {
    Actor r;
    Mess_Ptr m;
    if (script == script_self)
        Fork(fn[comportement],my,CopyArray(accoin));
    else {
        r = NewComponent(script,SCRIPT,CopyArray(accoin));
        while(1) {
                GetMessage(&m);
                SendaMessage(r,m);
        }
    }
}
```

figure 1.4 *La primitive Become*

NB: Le code de la primitive est ici simplifié. La boucle qui redirige les messages peut en réalité être stoppée lors de la destruction de l'acteur.

Si un acteur «*Become*» plusieurs fois, on constate alors la création d'une chaîne de composants relais. Celle-ci pénalise le système en multipliant les communications.Chaque message destiné à un acteur doit suivre la chaîne des composants relais. Pour éviter ce phénomène, la chaîne peut toujours être réduite à deux composants: celui de départ de l'acteur (qui représente le nom de l'acteur) et le composant courant matérialisant l'acteur.

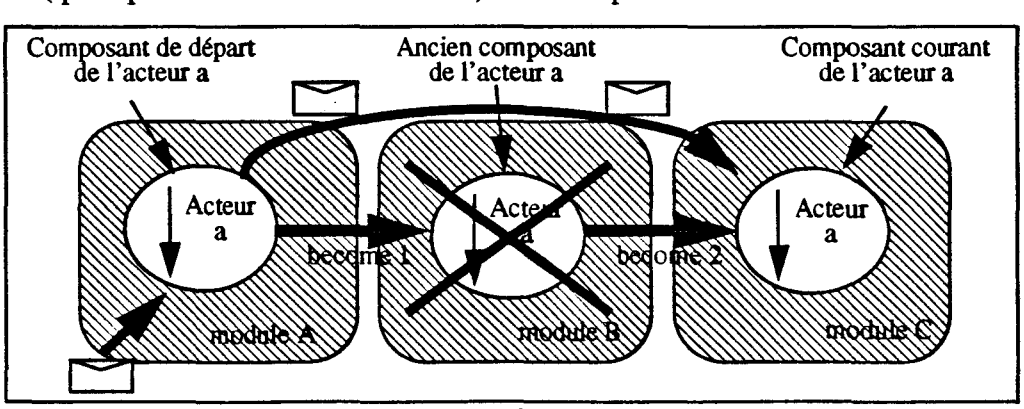


figure 1.5 *Elimination des composants relais.*

- III - 1.1.2.3 Primitives d'envoi de message

Le prototype Acteur redéfinit deux primitives de transfert d'information au dessus des primitives de communication des composants. Un protocole de communication est mis en place au dessus de la couche composant[1].

La fonction *Send(Actor act, int *arg)* transfère à l'acteur de nom *act* une liste de valeurs sous la forme d'un tableau d'entier (*arg*). Si le script d'un acteur possède plusieurs comportements, le premier argument (*arg[0]*) du message identifie le comportement.

La fonction *GetMessage(Mess_Ptr * m)* se met en attente d'un message déposé dans la boîte aux lettres de l'acteur courant. Le message est accessible par la variable *m*.

- III - 1.1.3 Exemples

- III - 1.1.3.1 L'acteur factoriel

Nous illustrons l'extension vers les Acteurs en reprenant le calcul récursif des factorielles tiré du livre d'Agha [Agha86].

Le calcul récursif de la fonction factorielle s'exprime avec deux «types» d'acteur. L'acteur *Rec_Factorial* décompose le calcul et l'acteur *Customer* effectue les multiplications du calcul. La figure 1.6 «Exécution de la fonction factorielle(3)», tirée du livre de [Masini90], montre le calcul de la fonction factorielle(3):

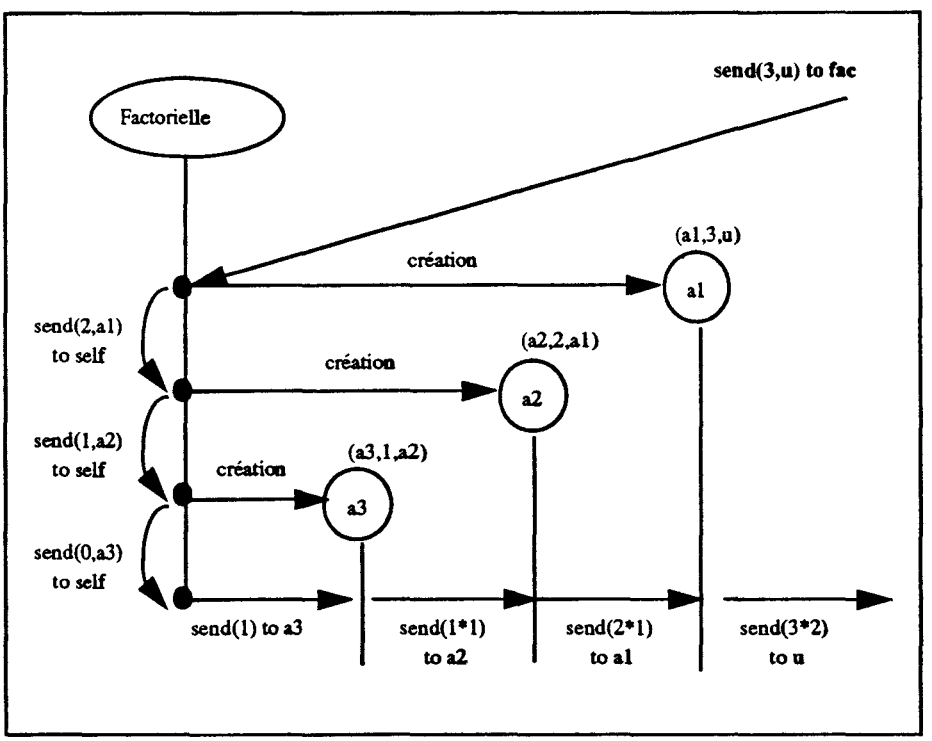


figure 1.6 *Exécution de la fonction factorielle(3)*

1. Les informations transférées dans le messages, sont soit des références sur d'autres acteurs ou des valeurs entières.

Un acteur *Rec_Factorial* accepte les messages contenant un entier *n* et une adresse de mandataire *u*. L'acteur spécifie alors son état de remplacement (*Rec_Factorial*) qui effectuera le pas suivant du calcul. Si l'entier contenu dans le message est nul, l'acteur retourne directement la valeur 1 à l'adresse *u*. Dans les autres cas, l'acteur crée un acteur *Customer* avec comme accointances, les deux paramètres du message (n et u). Pour effectuer le pas suivant de calcul, l'acteur *Rec_Factorial* s'envoie un message contenant la valeur entière décrémentée de 1 ainsi que l'adresse de l'acteur consommateur créé. Le message sera traité par l'état suivant de l'acteur.

L'acteur *Customer* accepte les messages contenant un entier *k*, multiplie cet entier par la valeur entière locale *n* et retourne la valeur calculée au mandataire *u*.

Dans notre environnement, la fonction script est lancée dès la création de l'acteur et le système copie directement les arguments de l'appel de la fonction NewActor() pour générer la table des d'accointances.

Une variable *m_name* définit le nom du Script de l'acteur. Elle est associée au nom du module Cac.

```
/*                                  M_FAC                                  */
#include<Prim_Act.h>
int m_name =     FAC;
void fac(Actor my, int *arg, int *accoin)
     /* arg[0] =taille, arg[1] = adresse de retour, arg[2] = min, arg[3] = max */
     /* accoin[ ] vide */
     {
     Actor                c1;
     Become(my,M_FAC,accoin);
     if (arg[2] == 0)
          Send(arg[1],1);
      else{
          c1 = NewActor(M_CUS,arg[1],arg[2]);
          Send(my,c1,arg[2]-1);
     };
     dispose(arg);
}
void script(Actor my,int *accoin)
     {
     Mess_Ptr mes;
     GetMessage(&mes);
     fac(my,MessArgToArray(mes),accoin);
     FreeMessage(mes);
     EndActor();
}
```

figure 1.7 *Le module M_FAC*

La primitive *MessArgToArray(m)* retourne une copie des arguments du message *m* sous la forme d'un tableau. Tous les tableaux de l'environnement ont leur première case qui mémorise la taille du tableau.

```
/*                                  M_CUS                                  */
#include<Prim_Act.h>
int  m_name = CUS;
void consomme(Actor my, int *arg, int * accoin)
     /*arg[1] = valeur entière reçue (arg tableau des message)*/
     /* accoin tableau des accointances */
     /*accoin[0] = 2, accoin[1] adresse de retour, accoin[2] valeur entiere */
     {
     Send(accoin[1],accoin[2] * arg[1]);
     dispose(arg);
}
void script(Actor my,int *accoin)
     {
     Mess_Ptr mes
     GetMessage(&mes);
     consomme(my,MessArgToArray(mes),accoin);
     FreeMess(mes);
     EndActor();
}
```

figure 1.8 *Le module M_CUS*

Le module ci-après M_DIA montre une utilisation des factorielles

```
/*                                  M_DIA                                  */
#include<Prim_Act.h>
int m_name=      DIA
void script(Actor my,int *accoin)
     {
     Actor          c;
     int n = 20; /* factorielle à calculer */
     int *mes;
     c = NewActor(FAC);
     Send(c,my,n);
     GetMessage(&mes);
     printf("resultat fac (%d) = %d\n",n,*ArgMess(mes,1));
     EndActor();
}
```

figure 1.9 *Le module M_DIA*

Après compilation des trois fichiers de noms Dia, Fac et Cus, l'utilisateur doit créer un fichier *CDL* pour configurer son réseau de module. Pour cela, il utilise la commande *ConstructModulesNet()* de l'interface système (voir - II - 3.2.3 «Le prototype Cac». Exemple:

```
ConstructModulesNet  1 Dia  1 Fac  1 Cus
```

figure 1.10 *création d'un réseau de modules*

Cette commande définit un réseau de trois modules : *Dia, Fac, Cus.*

Le run-time des composants permet la duplication de modules; le programmeur peut ainsi décrire un réseau de plusieurs modules *Fac* ou *Cus* sans en modifier le source. Le modèle répartit alors les acteurs de même type sur les modules de ce type. L'opération *Become* sans changement de script reste locale au module.

Par exemple, la configuration utilisant deux modules *Fac* et deux modules *Cus* se décrit comme suit:

```
ConstructModulesNet 1 Dia 2 Fac 2 Cus
```

figure 1.11 *Configuration avec deux modules fac*

NB: Le calcul d'une factorielle est peu parallèle, les traitements se séralisent sur les envois de message. La duplication de module n'offre donc que peu d'intérêt dans cet exemple. Elle peut par contre être utilisée pour calculer plusieurs factorielles en parallèle.

- III - 1.1.3.2 Le compte bancaire

Nous prenons un second exemple pour illustrer le rôle de la fonction *script*. Nous décrivons un acteur *compte_bancaire* contenant plusieurs comportements répondant au trois types de requête: *dépôt, retrait* et *solde*.

Le tableau des accointances est constitué d'une seule valeur: le solde du compte.

L'acteur *compte_bancaire* est défini à l'aide de trois comportements *dépôt, retrait* et *solde*. Les messages envoyés à l'acteur *compte_ bancaire* contiennent une zone nommant le comportement désiré. Le script du *compte_bancaire* gère la réception de message et lance un des trois comportements.

Pour faciliter le nommage des comportements définis dans un script, nous utilisons un fichier *interface* au script (fichier d'inclusion.h). L'interface de l'acteur *compte_bancaire* est définie dans un fichier Interface (*I_C_B.h*). Ce fichier contient la liste des noms des comportements exportés (*DEPOT, RETRAIT, SOLDE*). Le fichier est intégré dans le source des modules des acteurs clients.

```
/*                    I_C_B.h                */
#define RETRAIT          1
#define DEPOT            2
#define SOLDE            3
```

```
/*************************************************************************/
/*                                  M_C_B                                */
/*************************************************************************/
#include<Prim_Act.h>
#include <I_C_B.h>
int   m_name = M_C_B;

void depot (Actor my, int *arg, int *accoin)
     /* arg[0] =taille du tableau, arg[1] = type du message, arg[2] = somme*/
     /* accoin[0] = 1 , accoin[1] = solde*/
     {
     Become(my,M_C_B,accoin[1] + arg[2]);
     dispose(arg);
}
void retrait (Actor my, int *arg, int *accoin)
     /* arg[0] =taille, arg[1] = type du message, arg[2] = somme*/
     /* accoin[0] = 1 , accoin[1] = solde*/
     {
     Become(my,M_C_B,accoin[1] - arg[2]);
     dispose(arg);
}
void solde (Actor my, int *arg, int *accoin)
     /* arg[0] =taille, arg[1] = type du message, arg[2] = mandataire*/
     /* accoin[0] = 1 , accoin[1] = solde*/
     {
     Send(arg[2], accoin[1]);
     Become(my,M_C_B,accoin[1]);
     dispose(arg);
}
void script(Actor my,int *accoin)
     {
     Mess_Ptr     m;
     int *arg;
     GetMessage(&m); /* arg[1] type de message RETRAIT/DEPOT/SOLDE*/
     switch (*ArgMess(m,1)]) {
          case DEPOT:                depot(my,arg,accoin);break;
          case RETRAIT:              retrait(my,arg,accoin); break;
          case SOLDE:                solde(my,arg,accoin); break;
     };
     FreeMessage(m);
     EndActor();
}
```

figure 1.12 *L'acteur Compte_Bancaire*

```
/*                Dialogue                */
#include<Prim_Act.h>
#include <I_C_B.h>
void script(Actor my)
    {
    Actor              c;
    int                *mes;
    c = NewActor(M_C_B);
    Send(c,DEPOT,4000);
    Send(c,RETRAIT,1000);
    Send(c,SOLDE,my);
    GetMessage(mes);
    printf("solde = %d\n",*ArgMess(mes,1));
    EndActor()
}
```

figure 1.13 *Utilisation de l'acteur Compte_Bancaire*

- III - 1.1.4 Mesures

Le prototype des acteurs est développé sur le run-time des composants actifs de communication évoluant sur le Multicluster II. Voici les mesures des principales primitives acteur

Primitives	Temps en ms
NewActor (local - distant)	0.30 - 3.90
Become (locale -distant)	0.25 - 3,55
Send (local - distant)	0.23 - 3.50
GetMessage(extraction d'un message present)	0.24
Accès aux accointances	0.01

figure 1.14 *Mesures du run-time Acteurs, temps en milli-secondes*

Les performances (figure 1.14 «Mesures du run-time Acteurs, temps en milli-secondes») du prototype acteur sont très proches des performances du run-time Cac sous-jacent. Les légères différences sont liées au transport des accointances.

Nous avons mesuré l'application factorielle. Nous obtenons pour Fac(500)[1] un temps de l'ordre d'une vingtaine de secondes:

Fac(500)	Temps en s
1 module Dia, 1 module Fac, 1 module Cus	21.9
1 module Dia, 2 module Fac, 2 module Cus	22.4

figure 1.15 *Mesures du factoriel*

Le calcul de la fonction factorielle est peu parallèle et s'apparente davantage à une fonction récursive. La duplication de module n'apporte ici aucun intérêt.

1. Pour éviter un dépassement de capacité dans les entiers (fac(500)), nous avons remplacé la multiplication par une addition. Nous obtenons alors la somme des n (500) premiers nombres.

Pour tester les performances du modèle acteur dans un contexte parallèle, nous avons développé en termes d'acteurs, le calcul des factorielles dichotomiques déjà réalisé sur les Cac/s (voir - II - 3.2.4.3 «Mesures des applications Cac/s»). Les mesures sont effectuées dans la même configuration matérielle. On teste le gain apporté par la duplication de module et la perte liée au run-time acteur au dessus du run-time Cac.

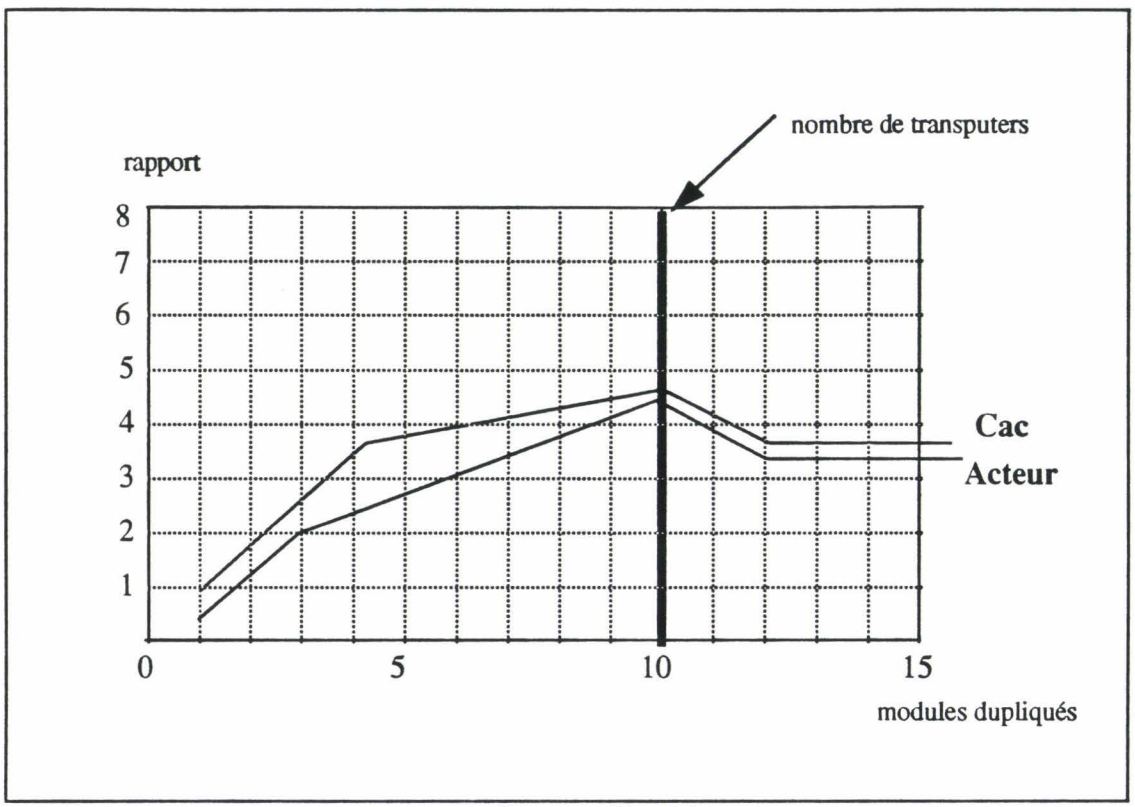


figure 1.16 *Rapport Cac / Acteur*

Le point de coordonnées (1,1) est le point de référence. Il correspond au temps réalisé par l'application avec un seul module *Fac* (pour la version Cac).

Les différences entre les deux run-times sont liées aux deux envois de message du modèle acteur. En effet pour créer un acteur, la primitive *New_Actor()* produit un premier message, l'acteur exécute alors son script. Il faut ensuite lui envoyer explicitement un second message pour le brancher sur un comportement. Dans le modèle Cac, la fonction comportementale est directement lancée à sa création.

Lorsque le nombre de modules dupliqués augmente, la différence entre les deux modèles se réduit en valeur absolue. Pour un seul module *Fac*, l'application Acteur est deux fois plus lente que la même application Cac; alors que pour dix modules *Fac*, l'application *Fac* est presque aussi performante.

- III - 1.2 Implantation d'une synchronisation

Dans le prototype décrit dans la partie - III - 1.1 «Implantation des acteurs», la synchronisation d'un acteur doit être décrite explicitement dans le code du script. Le programmeur doit écrire la synchronisation dans l'acteur en testant les messages reçus et autoriser le lancement d'un comportement en fonction de l'état courant de l'acteur et de la requête. Nous proposons une extension du Prototype Acteur comprenant un mécanisme de synchronisation plus évolué. Nous avons donc choisi d'intégrer, la synchronisation décrite dans *Act++* [Kafura89, 90] parce qu'elle nous paraît bien adaptée à la répartition de code du modèle Cac.

La synchronisation dans les acteurs s'exprime dans *Act ++* en définissant, en fonction de l'état de l'acteur, les comportements autorisés. Prenons l'exemple d'un acteur tampon, le comportement *get* ne sera autorisé que si le tampon n'est pas vide. Dans le langage *Act++*, un objet acteur est défini dans une classe d'acteur et l'expression de la synchronisation est répartie dans les sous-classes décrivant chacune un comportement de l'acteur (voir chapitre I «Les classes comportementales de *Act++*»). Chaque classe définit un sous-ensemble de fonctions autorisées parmi les fonctions de la sur-classe. Dans notre exemple, l'acteur tampon serait défini dans trois classes: *buffer_vide, buffer_plein et buffer_partiel* et l'objet évoluerait entre ces trois classes.

Nous introduisons dans le prototype la notion de type; un type étant l'ensemble des scripts que peut prendre un acteur durant sa vie. Un acteur se décrit alors en termes de scripts correspondant chacun à une synchronisation particulière de l'acteur. Pour reprendre l'exemple du buffer, l'acteur se décrit par trois scripts: tampon plein, tampon vide et tampon partiel et le type *t_tampon* regroupe les trois scripts.

- III - 1.2.1 Structure de type d'acteur à l'exécution

La répartition des acteurs et du code s'effectue en regroupant les scripts d'un même **type** d'acteur dans une même entité à l'exécution. La notion de *type* est ici associée à un ensemble de scripts.

Un acteur est décrit dans un module regroupant le code des *script*s définissant le type de l'acteur. Un acteur sera créé dans un module, en disposant ainsi du code de tous ses scripts. L'utilisation du concept de module pour le partage de code, est pleinement ici justifiée puisqu'un acteur regroupe plusieurs scripts qui utilisent un même ensemble de comportements.

Chaque type d'acteur est confondu avec le nom du module qui implante l'ensemble de ses scripts. Le code d'un module comprend, outre les deux fonctions *NewActor* et *Become*, les *Script*s de l'acteur.

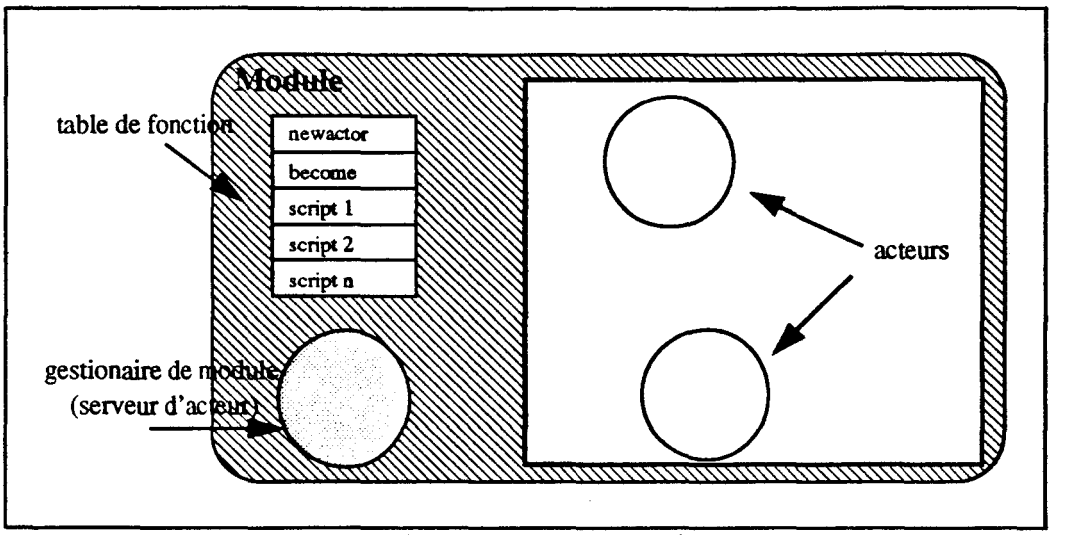


figure 1.17 *Un module = groupe de scripts*

- III - 1.2.2 La programmation d'acteurs synchronisés

L'extension du premier prototype modifie les primitives, *NewActor* et *Become* déjà développées. Un comportement est maintenant nommé par le couple (type et nom du script dans le type).

Un type est décrit dans un fichier C où une fonction *script()* doit être spécifiée (une ou plus).

- III - 1.2.2.1 L'opération Become

Le changement d'état s'effectue par la fonction *Become(Actor my, int S, int *accoin)* où *S* est le nom du nouveau script. Les accointances sont dupliquées et le traitement du nouvel état réexécute un script toujours local au module. L'acteur ne migre donc jamais.

```
void Become(Actor my, int script, int *accoin)
    {
        Fork(fn[script],my,CopyArray(accoin));
}
```

figure 1.18 *La primitive Become*

- III - 1.2.2.2 Le filtrage des messages

Le mécanisme de synchronisation dans *Act ++*, permet à chaque script d'un acteur de ne traiter qu'un sous ensemble de requêtes, parmi les requêtes du type. Dans notre implantation, cette opération est réalisée en filtrant les messages de requête lors de l'opération de retrait de message. Par exemple, le script d'un tampon dans un état *buffer_vide* utilise un filtre qui ne sélectionne que les messages de type *put*.

La primitive *GetMessageFiltrer(Mess_Ptr *m, Filtre f)* se met en attente d'un message vérifiant le filtre *f*. La liste *f* contient les noms des requêtes «permises». Le message sélectionné est accessible par la variable *m*. Cette instruction est réalisée avec les primitives d'accès aux *Box/s* (voir chapitre II)

```
/*                fonction de filtrage des message sur le requêtes                */
void GetMessageFiltrer(Acteur my, Mess_Ptr *m, int *filtre_list)
    {/*my acteur courant, m pointeur sur message recherché, filtre_list liste des requêtes*/
        int n = NumberMessageInBox(my);
        do {
                i = 1;
                do { /*consultation des messages de la boîte*/
                        ReadFromBox(my,i,m);
                        if ( IsInSet(*ArgMess(*m,1),filtre_list)) {
                                DeleteMessageInBox(my,i);
                                return;
                        };
                        i ++;
                } while (i <= n);
                n = WaitOnBox(my); /*attente d'un événement*/
        } while (1);
}
```

figure 1.19 *La primitive GetMessageFiltrer*

La technique de filtrage permet de ne pas extraire de la boîte aux lettres d'un acteur les messages indésirables tout en gardant leur ordre d'arrivée.

- III - 1.2.3 Exemple

Prenons l'exemple d'une pile d'entiers considérée illimitée. Elle se décrit par deux scripts:

- *pile_vide* (n'accepte d'exécuter que les demandes de dépôt)
- *pile_partielle* (accepte que les demandes de dépôt et de retrait)

Dans cet exemple, la pile est matérialisée par un tableau en mémoire que l'on initialise à la création de l'acteur. Deux primitives *GetAt() et SetAt()* permettent d'accéder en lecture et en écriture à un élément du tableau.

NB: Les tableaux (type de données) sont manipulés par référence afin d'éviter les copies d'objet de grosse taille. L'instruction Become est toujours locale et chaque comportement travaille sur le même tableau.

Une pile d'éléments est créée par l'instruction *NewActor()*:

pile = NewActor(M_PILE,SCRIPT_VIDE)

```
/*                          I_PILE                          */
/*          Comportements                                   */
#define     PUSH            1
#define     POP             2
/*          Scripts                                         */
#define     SCRIPT_VIDE     1
#define     SCRIPT_PARTIEL  2
```

```
/*                          M_PILE                          */
#include <I_PILE>

void push (Actor my, int *arg, int *accoin)
    /* arg[0] =2, arg[1] = type du message, arg[2] = élément*/
    /* accoin[0] = 3 , accoin[1] = position courante dans la pile
                accoin[2] = taille de la pile , accoin[3] = tableau d'éléments*/
    {
    SetAt(accoin[3],accoin[1],arg[2]); /*Pile[pos] = élément*/
    accoin[1] ++; /*pos ++*/
    Become(my,SCRIPT_PARTIEL, accoin);
    dispose(arg);
}
void pop (Actor my, int *arg, int *accoin)
    /* arg[0] =2, arg[1] = type du message, arg[2] = addresse de retour (mandataire)*/
    {
    accoin[1] --; /*pos -- */
    Send(arg[2],GetAt(accoin[3],accoin[1])); /*send(mandataire,Pile[pos])*/
    if (accoin[1] > 0) /*pos > 0 */
        Become(my,SCRIPT_PARTIEL,accoin);
    else
        Become(my,SCRIPT_VIDE,accoin);
    dispose(arg);
}
void script_Partiel(Actor my,int *accoin)
    {
    Mess_Ptr    mes;
    GetMessageFiltrer(&mes,Filtrer(PUSH,POP));
    switch (*ArgMess(mes,1)) {/*nom de la requête PUSH/POP*/
        case PUSH:          push(my,MessArgToArray(mes),accoin);break;
        case POP:           pop(my,MessArgToArray(mes),accoin);break;
    };
    FreeMessage(mes);
    EndActor();
}
void script_Vide(Actor my,int *accoin)
    {
    Mess_Ptr    mes;
    GetMessageFiltrer(&mes,Filtrer(PUSH));
    push(my,MessArgToArray(mes),accoin);
    FreeMessage(mes);
    EndActor();
}
void InitActor(Actor my,int *arg,int **accoin)
    {int *tmp;                              /*variable temporaire*/
    accoin = New(4);
    SetAt(accoin,0, 3);                     /*taille du tableau accoin */
    SetAt(accoin,1, 3);                     /* position courante dans la pile */
    SetAt(accoin,2, MAX];                   /* taille maximale de la pile */
    tmp = New(MAX);                         /* pile = new(taille)*/
    SetAt(accoin,3, tmp);
}
```

figure 1.20 *La pile*

NB: La fonction InitActor() permet d'initialiser un acteur. Elle est appelée dès la création de l'acteur avant l'exécution du premier script. La fonction peut être paramétrée par les arguments de l'instruction NEW_ACTOR(). Dans l'exemple, la fonction alloue le tableau d'entier matérialisant la pile, puis elle initialise les variables de contrôle délimitant l'intervalle utilisé du tableau.

```
/*                    Script d'un client de l'actor Pile                    */
#include <I_PILE>

void script(Actor my,int *accoin)
    {
    Actor           c1;
    Mess_Ptr        mes;
    /*création d'un acteur pile*/
    c1 = NewActor(M_PILE,SCRIPT_VIDE);

    Send(c1,PUSH,5);                          /*dépot de l'entier 5*/
    Send(c1,PUSH,10);                         /*dépot de l'entier 10*/

    Send(c1,POP,my);                          /*demande de retrait*/
    GetMessage(&mes);                         /*attente d'une valeur*/
    printf("valeur extraite= %d\n",*ArgMess(mes,1));

    Send(c1,POP,my);                          /*demande de retrait*/
    GetMessage(&mes);                         /*attente d'une valeur*/
    printf("valeur extraite= %d\n",*ArgMess(mes,1));

    Send(c1,PUSH,3);                          /*dépot de l'entier 3*/

    Send(c1,POP,my);                          /*demande de retrait*/
    GetMessage(&mes);                         /*attente d'une valeur*/
    printf("valeur extraite= %d\n",*ArgMess(mes,1));

    EndActor();
}
```

figure 1.21 *Utilisation de l'acteur PILE*

Conclusion

Le prototype Acteur est une application des Composants Actifs de Communication. Il a montré les bonnes propriétés du modèle Cac, telles que la vision structurante des données, les structures naturellement parallèles et l'indépendance vis à vis de l'architecture cible. De plus, le développement du prototype Acteur a directement bénéficié de la gestion dynamique des processus du modèle Cac et de la couche communication fiabilisée.

La réalisation de deux extensions Acteurs justifie l'étude d'une implantation de langages d'objets actifs sur les Composants Actifs de Communication par les points suivants:

- 1 - Le prototype Acteur montre que le modèle Cac peut supporter plusieurs niveaux de regroupement d'objets tels que les ensembles de comportements de Cac, le script d'un acteur, les types (ensemble de scripts d'acteur).

- 2 - Les Cac/s permettent de réaliser des entités multi-programmées en réalisant une entité avec plusieurs Cac/s.

- 3 - La partie - III - 1.2 «Implantation d'une synchronisation» montre que le modèle d'exécution des Cac/s n'empêche pas la conception et la réalisation d'une synchronisation plus évoluée des entités de l'application.

Les étapes suivantes vers les objets actifs multi-programmés sont: i) la définition d'une structure plus évoluée et surtout plus parallèles pour accueillir un objet permettant un partage des variables d'instances. ii) Intégration d'un mécanisme d'héritage comme moyen de réutilisation du code développé. iii) Intégration d'un mécanisme de synchronisation intra-objet pour les objets multi-programmés

Extension du prototype Acteur

Plusieurs travaux sur le prototype actuel seraient nécessaires:

- 1 - Une version intégrerait un typage des données (accointances, variables locales d'un comportement et arguments des messages) ainsi qu'un nombre plus important de types de base: caractères, réels, chaînes de caractères, tableaux...

- 2 - Pour compléter le modèle Acteur, nous devrions introduire un mécanisme de partage des connaissances. Dans *Act++* ou *Actra*[Thomas89, McAffer91], les acteurs sont décrits en classe d'acteur, et le partage des connaissances s'effectue par *héritage des classes*. Lieberman [Lieberman86] a introduit la notion de *délégation* comme autre mécanisme de partage. L'article de J.P. Briot [Briot87] montre que le mécanisme de délégation est plus flexible que l'héritage.

Ces améliorations ne font pas l'objet de cette thèse.

- III - 2. Implantation d'Objets Actifs Distribués

Nous avons proposé au chapitre I «Les Langages Parallèles à Objets» une classification des principaux langages à objets parallèles. Une catégorie d'objets actifs, notamment celle des objets actifs multi-programmés, nous semble particulièrement intéressante car elle génère un degré de parallélisme important (grâce au parallélisme inter et intra-objet). Ces objets actifs multi-programmés permettent de programmer naturellement des applications hautement parallèles et facilitent l'expression du **parallélisme dès la conception.**

Les multicomputers ont été conçus pour accroître la puissance de calcul des ordinateurs. L'idée consistait à multiplier les unités de traitement (processeurs); c'est le **parallélisme architectural**. Pour exploiter pleinement de telles machines, il est indispensable que les programmes soient aussi parallèles que possible. Les multicomputers, et plus généralement les architectures parallèles, permettent d'exprimer un parallélisme qui demeure à l'exécution. C'est le **parallélisme d'exécution.**

Notre étude se consacre donc à l'implantation des langages parallèles à objets sur les multicomputers. Nous avons porté une attention toute particulière à privilégier le parallélisme, et à conserver au mieux le parallélisme de conception pendant l'exécution. Point original de notre démarche, nous avons tenu à générer un **parallélisme réel** pour le parallélisme intra-objet. Pour cela, nous avons travaillé sur la représentation et l'allocation des objets actifs en mémoire. Là encore, nous avons tenu à utiliser les spécificités de l'architecture matérielle. Pour matérialiser réellement le parallélisme de conception, nous proposons d'éclater la représentation des objets. Ce travail est une première phase de l'implantation d'un langage parallèle à objets actifs sur multicomputers.

L'implantation d'un langage sur une machine parallèle est aussi une répartition du code sur les noeuds du multicomputer. Dans un langage à objets, le code (les méthodes) est contenu dans une structure de données des classes à l'exécution (voir chapitre I, partie - I - 3. «Objets et Répartition»). Nous proposons une répartition de cette structure qui supporte nos représentations éclatées d'objets actifs. Nous cherchons en particulier une représentation en partie contrôlée par le programmeur, lequel pourra ainsi agir sur la distribution des objets sur les noeuds.

Nous avons développé un prototype qui reprend les représentations distribuées d'objets actifs sur le run-time des Composant Actifs de Communication (décrits aux chapitre II). Les objets sont matérialisés à l'exécution par un ensemble de Cac/s dont les processus réalisent les activités intra-objet. Nous réalisons les objets multi-programmés avec un petit nombre de types de Cac/s, un Cac étant mono-programmé. Le code des méthodes est distribué sur les modules, entités de partage de code du système Cac.

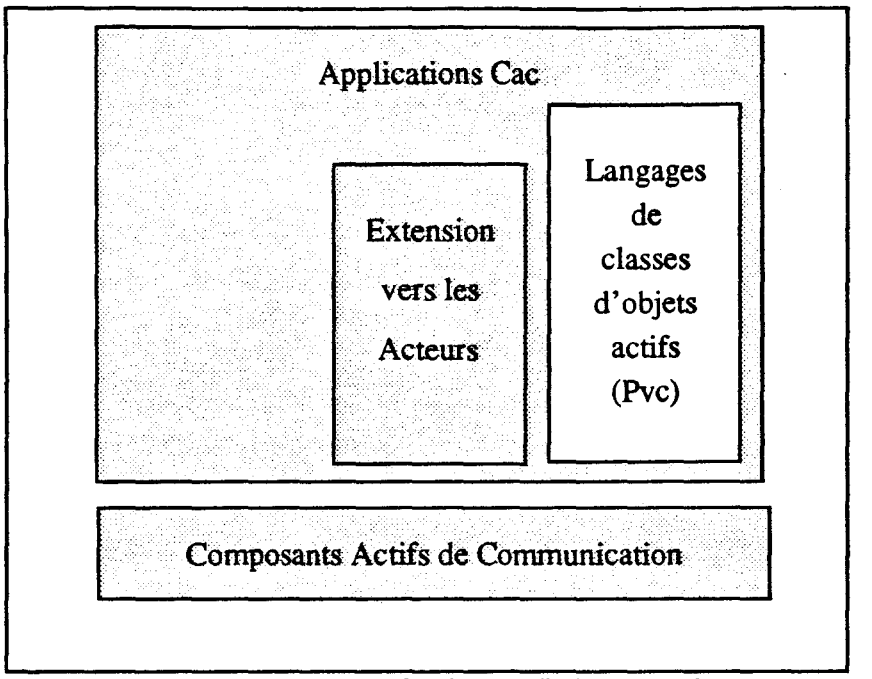


figure 2.0 *Langages de classes d'objets actifs*

- III - 2.1 Les objets actifs distribués

Les objets actifs permettent d'intégrer une entité de parallélisme dans la structure même de l'objet (- I - 1.1.2 «Objets Actifs (les objets serveurs et les Acteurs)»). Cette encapsulation du parallélisme dans chaque objet protège naturellement l'environnement local de l'objet des autres flots d'exécution de l'application. Parmi ces objets actifs, les objets actifs multi-programmés autorisent un parallélisme intra-objet où plusieurs activités coexistent à l'intérieur même de l'objet. Ces activités intra-objet sont en général contrôlées par une activité particulière de l'objet.

D'un point de vue fonctionnel, un objet actif mono ou multi-programmé peut être représenté par:

- **Un nom** global fournit par un mécanisme de désignation. Il identifie l'objet et permet les échanges entre les objets de l'application dont les messages de requête à l'objet.
- **Une mémoire locale** renferme les valeurs des variables d'instance.
- **Une ou plusieurs activités locales** (une seule dans le cas des objets mono-programmés) qui traitent les messages de requête à l'objet.
- **Un lien vers le code de ses méthodes d'instance** permettant à l'objet de lancer les exécutions de méthode.

- III - 2.1.1 Représentation des objets

Contrairement aux objets passifs, la structure d'un objet actif ne peut pas être qu'un simple bloc mémoire contenant les variables d'instance. Nous proposons plusieurs représentations des objets actifs qui reprennent la structure fonctionnelle d'un objet. Notre objectif est de paralléliser au mieux les activités internes d'un objet. De plus, nous cherchons à produire une interface homogène à l'objet, permettant, lors de l'exécution, la cohabition d'objets de représentations différentes.

Pour modéliser et réaliser les différentes représentations d'objets actifs, nous utilisons les Composants Actifs de Communication décrits au chapitre II. Les Cac/s ont de bonnes propriétés pour concevoir des applications parallèles dont font parties les modèles d'exécution de langages parallèles à objets. En effet, un Cac encapsule dans sa structure, tel un objet actif, un processus et un environnement local. Le modèle Cac fournit de surcroît un mécanisme de désignation fonctionnant en contexte réparti. L'utilisation des composants dans la réalisation permet de détacher la réalisation d'objets actifs, des problèmes matériels.

Dans cette section, nous ne nous intéresserons pas au problème de localité du code des méthodes. Celui-ci sera abordé et traité dans la section suivante(-III-2.2). Nous considérerons dans ce qui suit, que l'ensemble du code est accessible sur tous les noeuds.

- III - 2.1.1.1 L'interface d'un objet actif

Notre modèle permet aux objets, quelle qu'en soit leur représentation, de coexister et de communiquer entre eux. Les objets présentent alors une interface homogène permettant aux autres objets de dialoguer avec eux. Cette interface est composée du nom de l'objet et d'un protocole inter-objet d'invocation de méthode.

Le nommage d'un objet

Chaque objet est associé à un Cac spécialisé appelé CRO (Composant Représentant d'Objet). Ce Cac identifie et désigne l'objet; un Cac ayant un nom unique dans le système, Le nom du CRO est utilisé par les autres objets pour communiquer avec lui. Le CRO reçoit toutes les requêtes externes à l'objet. Le CRO d'un objet existe durant toute la vie de l'objet.

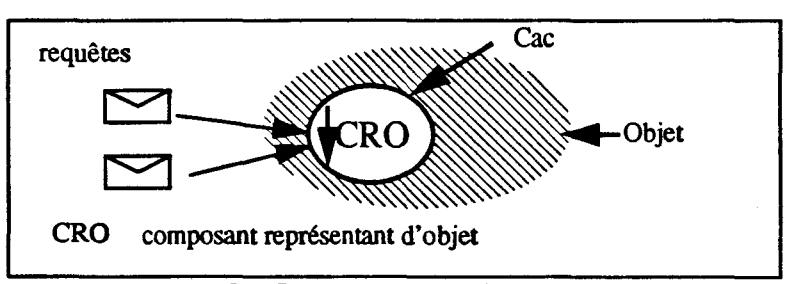


figure 2.1 *Le Composant représentant d'objet*

La fonctionnalité de désignation de l'objet par le CRO sera conservée dans toutes les représentations d'objets que nous proposerons.

L'appel de méthode

Chaque requête est matérialisée par un envoi de message qui représente l'appel de la méthode sur une instance donnée. Il se compose logiquement d'une référence sur la méthode invoquée et d'un nombre quelconque d'arguments. Le message peut aussi contenir une référence sur la méthode mandataire (méthode appelante ou par continuation, une référence sur une autre méthode en exécution). Le nom du mandataire est mis dans la structure de message. Le message, une fois construit, est envoyé à l'objet cible. Les messages de requête sont identiques quelque soit le type de la représentation de l'objet. Voici, schématisée, la structure d'un message de requête:

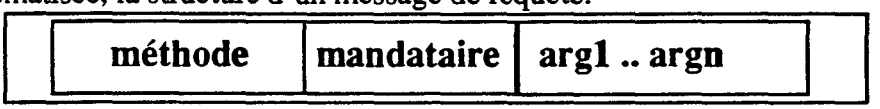


figure 2.2 *Structure d'un message de requête*

Le mécanisme d'invocation de méthode est réalisé au dessus des représentations des objets actifs. Nous présentons deux cas d'invocations: l'appel synchrone et l'appel asynchrone. L'invocation en mode synchrone bloque la méthode appelante jusqu'au retour d'une réponse qui la libérera. L'appel asynchrone est non bloquant.

L'appel de méthode synchrone

Prenons l'exemple d'un objet **o1** effectuant une invocation de la méthode **m** sur un objet distant **o2**. Pour indentifier le mandataire de la requête, une boîte aux lettres locale à **o1** est créée et transmise dans le message. Elle indentifie le mandataire de la requête et reçoit la réponse. L'opération se décompose en trois phases.

- 1 - Envoi d'un message représentant **o2.m(o1,args)** vers le CRO de l'objet **o2**.
- 2 - Réception du message de requête par l'object **o2** et exécution de la méthode **m** invoquée avec les paramètres **(o1,args)**.
- 3 - Après traitement, un message de réponse est retourné directement à la boîte aux lettres locale à **o1**(mandataire).

Nous montrons maintenant la faisabilité du modèle d'exécution sur le prototype Cac existant:

Les objets sont constitués par des Cac/s et chaque objet est associé à un Cac CRO qui le représente. La primitive d'invocation synchrone de méthode est matérialisée par un envoi de message au CRO de l'objet cible.

Le message de requête est construit par la primitive Cac *SendRequest()* qui prépare un message et l'envoie au destinataire. Le nom de la méthode invoquée ainsi que le paramètre d'appel sont copiés dans le message. La primitive crée la boîte aux lettres locale pour recevoir la réponse de l'invocation; le nom de cette boîte est inséré dans les arguments du message de requête. Après l'envoi du message, la primitive se met en attente d'un message de réponse, qui sera déposé dans la boîte aux lettres locale. L'attente s'effectue par primitive Cac: *GetMessage()*). Lorsque le message de réponse est retiré de la boite, la primitive en extrait la réponse et la retourne à l'appelant de la fonction.

Dans notre prototype actuel, les méthodes d'instances sont regroupées dans un tableau. Elles sont alors nommées par des entiers qui correspondent à leur position dans le tableau des méthodes d'une instance. La liste des arguments d'appel est matérialisée par un tableau de valeurs entières; le type Component étant compatible avec un entier (voir chapitre II).

```
/***********          Appel de fonction en mode synchrone (Call)          ******/
int  Call(Component cro, int request, int * args)
     {
     Mess_Ptr m;
     int res;
     Box b = NewBox(my);                    /*création d'une boîte aux lettres locale*/
     SendRequest(cro,request,b,args);       /*envoi du message de requête*/
     GetMessage(b,&m);                      /*attente du message de réponse*/
     res = ArgMess(m,1);                    /*extraction de la réponse du message*/
     FreeMessage(m);                        /*destruction du message*/
     return(res);                           /*retourne le résultat */
}
```

figure 2.3 *L'appel synchrone*

L'appel de méthode asynchrone

Une invocation de méthode asynchrone sans resynchronisation sur un résultat constitue un simple envoi de message à l'instance cible. Si la méthode appelante veut se resynchroniser sur une réponse de l'appel asynchrone, elle doit transmettre une adresse de retour (nom d'une boîte aux lettres mandataire) lors de l'envoi de message de requête.

La réalisation de l'appel asynchrone s'effectue en découpant la primitive d'appel synchrone en deux phases: (l'envoi du message de requête et l'attente d'un message de réponse). Si la méthode appelante désire récupérer un éventuel résultat, elle crée une boîte aux lettres spécifique qu'elle passe en paramètre de la primitive: le programmeur spécifie ainsi si il désire ou non récupérer une réponse ou traduire une continuation.

La resynchronisation sur le résultat est une simple attente sur la boîte aux lettres locale

```
/***********          Appel de fonction en mode asynchrone (CallAsynchrone)   ******/
void     CallAsynchrone(Component cro, int request, Box b,int * args)
     {
     SendRequest(cro,request,b,args);       /*envoi du message de requête*/
}
/***********          resynchronisation sur une réponse                       ******/
int      GetReply(Box b)
     {
     Mess_Ptr m;
     int res;
     GetMessage(b,&m);                      /*attente du message de réponse*/
     res = ArgMess(m,1);                    /*extraction de la réponse du message*/
     FreeMessage(m);                        /*destruction du message*/
     return(res);                           /*retourne le résultat */
}
```

figure 2.4 *L'appel asynchrone et la resynchronisation sur une réponse*

- III - 2.1.1.2 Les objets mono-programmés

Les objets mono-programmés sont sérialisés et traitent alors de façon séquentielle les requêtes à l'objet. Une synchronisation dans l'objet n'est donc pas nécessaire. Chaque objet mono-programmé peut être représenté par un seul Composant Actif de Communication, «objet actif mono-programmé» du run-time Cac.

Pour représenter ces objets, nous reprenons le CRO représentant l'objet auquel nous lui ajoutons les autres fonctionnalités d'un objet. Il contient maintenant les variables locales de l'objet et traite directement les requêtes externes et internes (self-invocations) à l'objet. Le processus du CRO sérialise naturellement les exécutions de méthode dans l'objet.

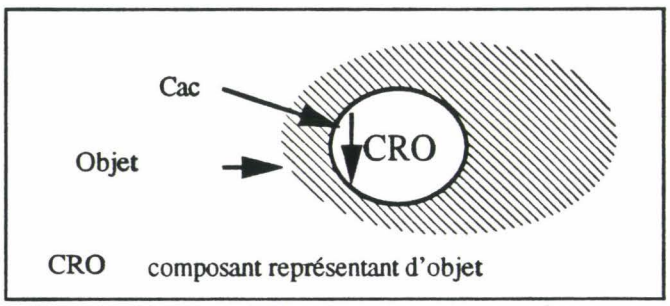


figure 2.5 *Objet mono-programmé*

Le composant CRO est une spécialisation d'un Cac. Pour réaliser l'objet mono-programmé, la structure du CRO se découpe ainsi:

- 1- une **boîte aux lettres** qui est le seul point d'accès à l'objet qu'il représente. Elle reçoit (et contient) toutes les requêtes (invocations de méthode) à l'objet.
- 2 - un **environnement local** contenant toutes les variables d'instance de l'objet.
- 3 - un **processus**, activité de l'objet. Il initialise l'objet, extrait les messages de requête de la boîte aux lettres de l'objet et exécute les méthodes. Il manipule directement les variables d'instance de l'objet contenues dans l'environnement local du composant.

Nous schématisons les événements intervenant lors des principales opérations sur un objet (accès local ou distant à un attribut, invocation de méthode et self-invocation). Les événements nous intéressant sont : les traitements locaux, les envois de messages et les créations de Cac.

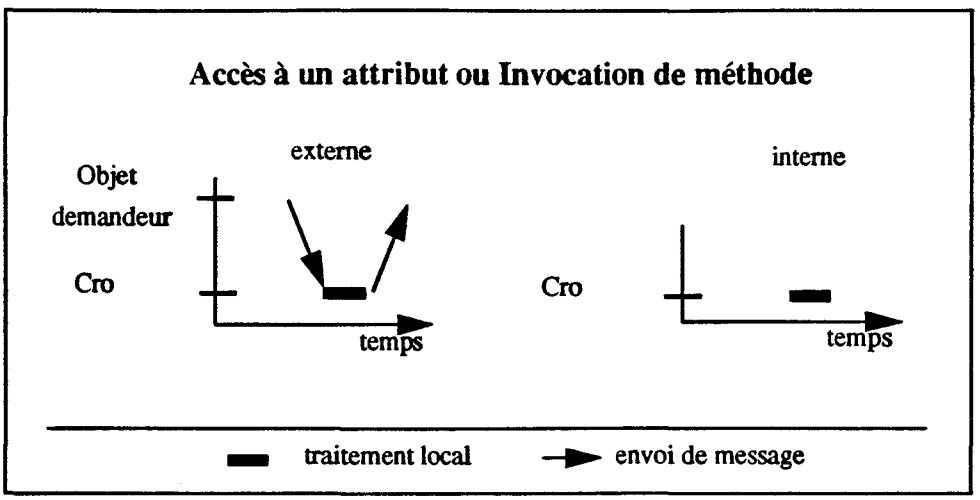


figure 2.6 *Trace d'événements*

Lors d'une invocation de méthode, l'objet demandeur envoie un message de requête au CRO de l'objet cible. Ce CRO exécute alors directement la méthode invoquée et retourne éventuellement une réponse dans un message à l'objet demandeur.

réalisation

La fonction comportementale du CRO d'un objet mono-programmé est présentée ci-après.

- Les variables d'instance de l'objet sont stockées dans un tableau alloué dans l'environnement local du CRO. A son initialisation, le CRO exécute la primitive *InitObject()* qui initialise les variables d'instance de l'objet. Cette fonction est programmable par l'utilisateur.

- Le processus du CRO se met ensuite en attente d'un message de requête qu'il traitera par une exécution de la méthode invoquée. Le code des méthodes d'instance est stocké dans un tableau de méthode. Les arguments d'appel à la méthode sont transmis à la fonction lors de son appel. A la fin de l'exécution d'une méthode, le CRO détruit le message de requête et se met en attente d'un autre message.

Le nombre d'attributs, le tableau des méthodes et la fonction *InitObject()* sont des paramètres de la classe et doivent être décrits par le programmeur de la classe d'objets actifs.

La programmation de nos objets actifs autorise les demandes externes d'accès aux variables d'instance de l'objet. Les messages de requêtes aux attributs sont constitués de deux ou trois paramètres qui sont dans l'ordre: le type de demande (WRITE, READ), le nom de l'attribut (numéro d'ordre dans le tableau des attributs), la nouvelle valeur de l'attribut (pour les écritures); ce dernier paramètre est optionnel.

NB: Une exécution de méthode peut aussi accéder aux variables d'instance de l'objet. Ces accès se font directement en mémoire dans le tableau des attributs.

```
/*******                    fonction comportementale d'un CRO                    *******/
void Cro(Component my,int *arg) /*my objet courant*/
    {
    Mess_Ptr m;
    int *attr = new(nb_attr);                    /*tableau des attributs*/
    InitObject(my,attr,arg);                     /*initilisation programmable de l'objet*/
    do {
      GetMessage(&m);                            /*extraction d'un message*/
      if (m->request == ACCESS_ATTR)             /*accès aux attributs*/
        switch (ArgMess(m,1)) {                  /*type d'accès*/
          case WRITE :                           /*accès en écriture*/
                attr[ArgMess(m,2)] = ArgMess(m,3); /*affectation*/
                SendReply(m->mandataire,attr[ArgMess(m,2)]);
                break;
          case READ :                            /*accès en lecture*/
                SendReply(m->mandataire,attr[ArgMess(m,2)]);
                break;
        }
      else                                       /* invocation de méhtode*/
         methode[m->request](my,attr,ArgMessToArray(m)); /*exécution de la méthode*/
    FreeMessage(m);
    } while (! KillComponent());
    dispose(attr);
}
```

figure 2.7 *La fonction comportementale du CRO*

La fonction *SendReply(des,args)* construit et envoie un message de réponse à l'objet *des* avec les paramètres *args* passés à la fonction. La fonction *KillComponent()* permet au système d'arrêter un composant.

NB: Une maquette de classe d' objet mono-programmé est présentée en Annexe -III-.

- III - 2.1.1.3 Les objets multi-programmés

La structure d'un objet multi-programmé est plus complexe que celle d'un Composant Actif de Communication; l'objet pouvant posséder plusieurs activités. Chaque objet actif est donc matérialisé par un assemblage dynamique de Cac/s dont chacun représente une activité de l'objet. Les Cac/s ne se partageant pas de mémoire commune, la représentation des objets peut être aisément répartie. (La distribution de chaque Cac se fera en fonction de la répartition des fonctions comportementales des Cac/s sur les noeuds. (voir partie - III - 2.2 «Répartition des classes»)

L'exécution décentralisée des méthodes

Nous cherchons à séparer, l'entité représentant de l'objet, des exécutions des méthodes sur cet objet et à exploiter réellement le parallélisme intra-objet. La représentation pourra alors être répartie.

Un composant CRO est toujours créé à l'instanciation de chaque instance et sert de base au développement d'autres composants matérialisant les exécutions de méthode de l'objet. Ces exécutions de méthode ne sont donc plus effectuées par le CRO de l'objet, mais sont déléguées à des Composants d'Exécution de Méthode (CEM). Pour chaque requête, le composant CRO crée un CEM qui exécutera la méthode invoquée. Le composant CEM

n'existe que durant l'exécution de la méthode. La dissociation de l'exécution de méthode et de la tâche de réception des requêtes permet un parallélisme intra-objet.

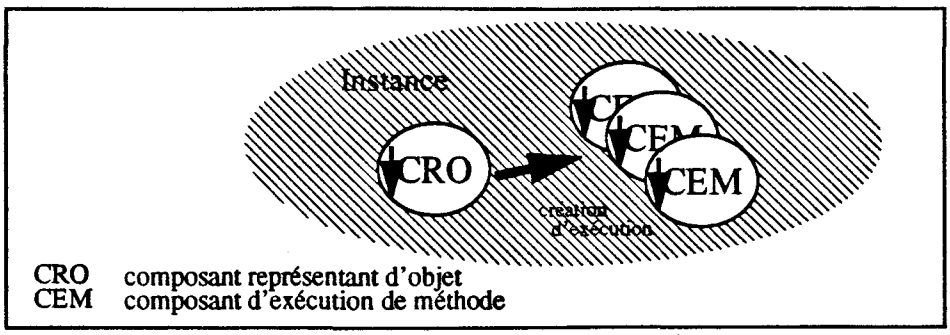


figure 2.8 *Objet multi-programmés*

Les différents CEM/s sont répartis sur les noeuds de la machine cible où est chargé le code des méthodes invoquées. La représention de l'objet est distribuée à l'exécution; elle évolue dynamiquement en fonction des exécutions de méthode sur l'objet. L'objet n'est plus une entité à part entière à l'exécution mais un agrégat de composants (CRO et CEM).

Les variables d'instance de l'objet sont stockées dans l'environnement local du CRO. Lorsqu'une méthode veut accéder à un attribut de l'objet, le composant CEM envoie un message au CRO qui traitera la requête. Cette coopération entre les CEM/s et le CRO semble nuire aux performances du modèle d'exécution en produisant des communications supplémentaires. Nous verrons dans la partie - III - 2.1.4 «Mesures des différents modèles d'objet» que cet «overhead» est largement compensé par le réel parallélisme intra-objet engendré par les CEM/s.

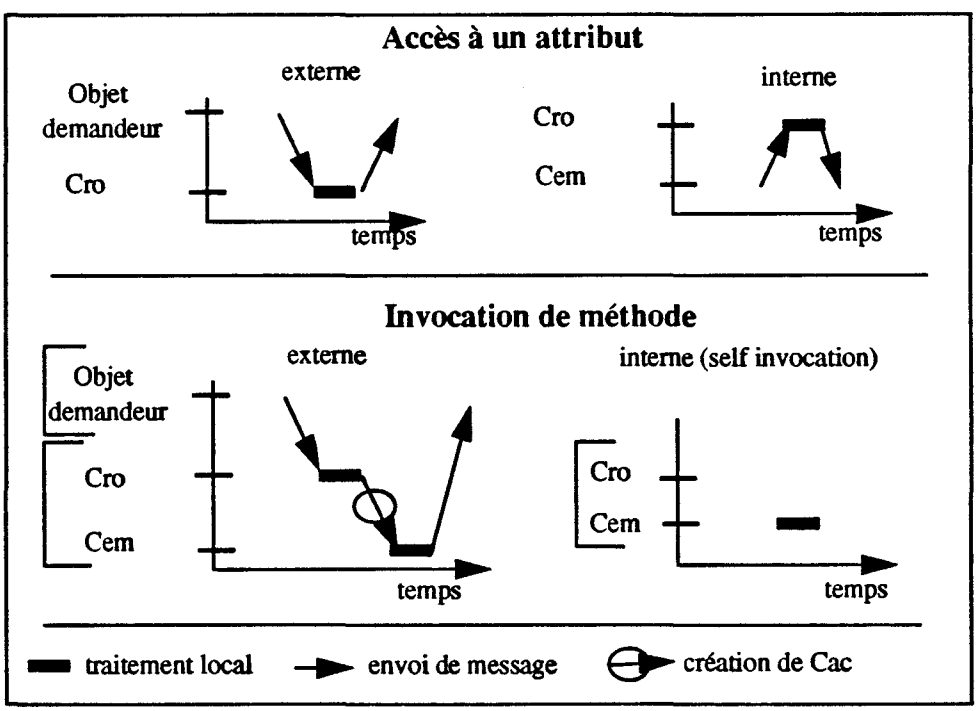


figure 2.9 *Trace d'événements*

Pour montrer le parallélisme potentiel, nous montrons les traces possibles de deux invocations simultanées sur le même objet.

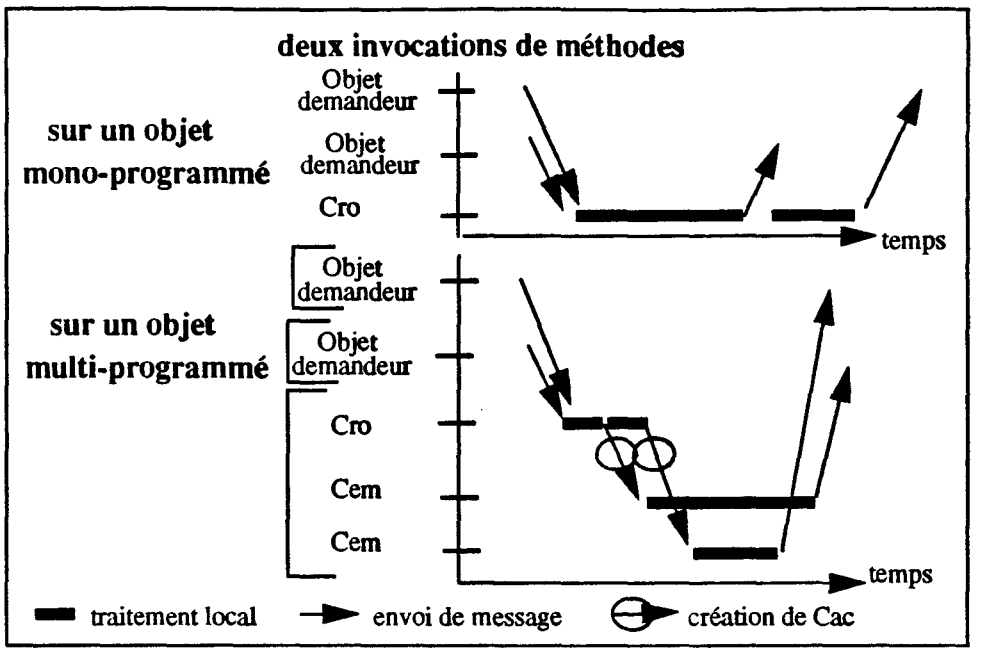


figure 2.10 *Le parallélisme dans l'objet*

réalisation

La fonction comportementale d'un CRO d'un objet multi-programmé est similaire à celle du CRO d'un objet mono-programmé. Les invocations de méthode sont traitées par une création de Cac de type CEM. Les arguments d'appel à la méthode sont transmis lors de la création de chaque CEM. Les demandes aux attributs sont traitées directement dans le CRO par des accès au tableau des variables d'instance.

```
/*****          fonction comportementale d'un CRO multiprogrammé          ******/
void CroMultiProg(Component my,int *arg)
    {
    Mess_Ptr m;
    int *attr = new(nb_attr);                    /*création du tableau des attributs*/
    InitObject(my,attr,arg);                     /*initialisation des attributs*/
    do {
      GetMessage(&m);                            /*extraction d'une requête*/
      if (m->request == ACCESS_ATTR)
        switch (ArgMess(m,1)) {                  /*type d'accès*/
          case WRITE :                           /*ecriture*/
                attr[ArgMess(m,2)] = ArgMess(m,3);
                SendReply(m->mandataire,attr[ArgMess(m,2)]);
                break;
          case READ:                             /*lecture*/
                SendReply(m->mandataire,attr[ArgMess(m,2)]);
                break;
        }
      else {                        /*invocation de méthode -> création d'un CEM*/
        NewComponent(CEM, m->request,my,ArgMessToArray(m));
      FreeMessage(m);
    } while (! KillComponent());
    dispose(attr);
}
```

figure 2.11 *La fonction comportementale du CRO d'un objet multi-programmé*

La fonction comportementale d'un CEM est une simple exécution de la méthode invoquée.

```
/*****  fonction comportementale d'un CEM          ******/
void Cem(Component my, int request, Component obj, int *arg)
     {
     methode[request](obj,arg);/*exécution de la fonction*/
}
```

figure 2.12 *La fonction comportementale du CEM*

Accès décentralisé aux variables d'instances

Dans la représentation précédente d'un objet, le CRO doit traiter à la fois les messages de requêtes à l'instance et les accès aux attributs. Le CRO peut s'avérer être alors un goulot d'étranglement dans l'objet. Si l'objet reçoit un nombre important de requêtes, les accès internes aux attributs sont ralentis puisqu'ils sont traités par le même composant. Pour résoudre ce problème, nous suggérons de séparer les accès aux variables d'instance des invocations de méthode.

Nous proposons donc la mise en place d'un Composant de Gestion des Attributs (CGA) qui regroupent les variables d'instance de l'objet. Toutes les modifications d'attributs sont alors effectuées exclusivement par ce CGA créé à l'instanciation de l'objet par le CRO.

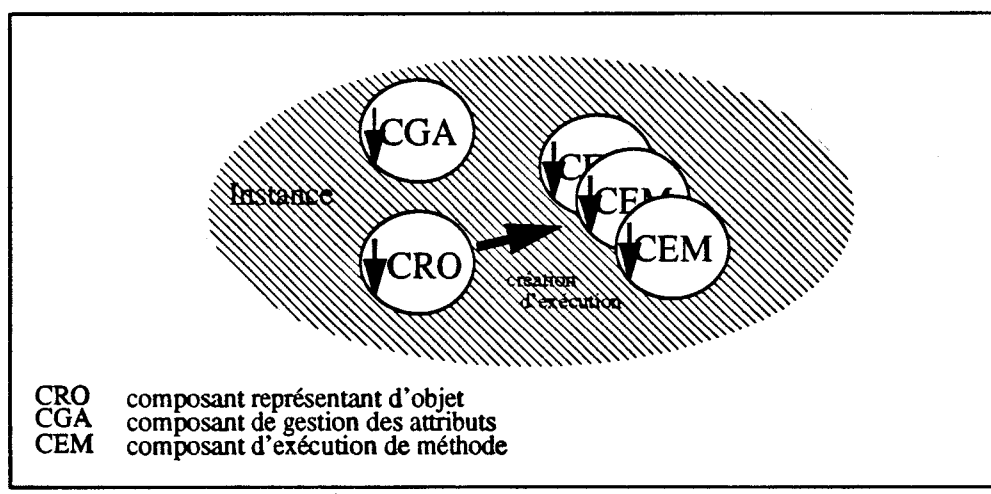


figure 2.13 *Le Composant de Gestion des Attributs*

Le processus du CGA sérialise les accès aux attributs requis par les CEM/s. Il évite la mise en place d'une synchronisation de bas niveau dans les CEM/s nécessaire si les variables étaient directement accessibles par ceux-ci. Le CGA ne contenant que des données, il peut donc être localisé sur n'importe quel noeud de la machine cible. Ceci, d'autant plus que les CEM/s sont déjà répartis.

Le modèle d'un objet actif est ici conservé à l'exécution. Les trois composantes d'un objet actif que sont le nom, les variables d'instance et les activités, sont matérialisées dans des entités indépendantes.

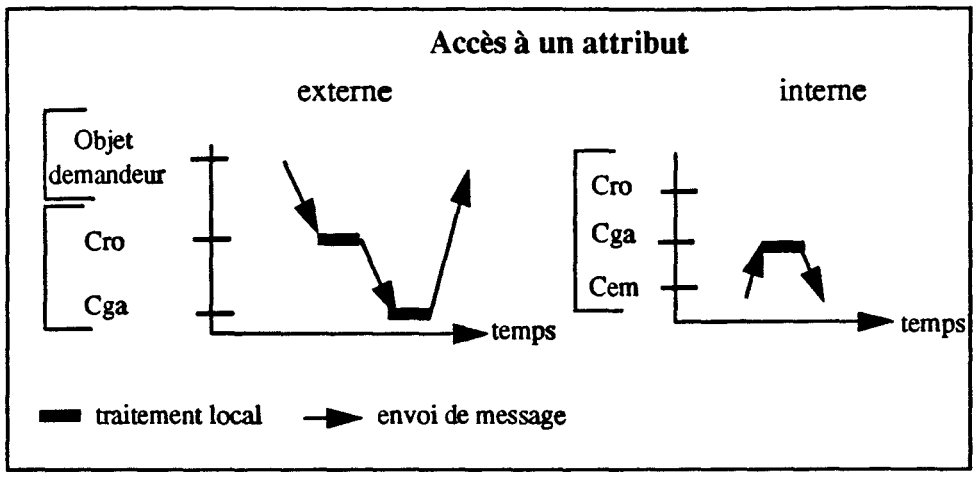


figure 2.14 *Trace d'événements*

réalisation

La fonction comportementale du CRO d'un objet multi-programmé crée le composant CGA et lui redirige le message d'accès aux attributs. Le nom du CGA est passé en paramètre de lancement de tous les CEM/s créés par le CRO. Le CEM peut alors communiquer directement avec le CGA sans perturber le CRO de l'objet.

```
/****            fonction comportementale d'un CRO multiprogrammé avec CGA****/
void CroMultiProgCga(Component my,int *arg)
    {
    Mess_Ptr m;
    Component cga;                          /*déclaration du Cac CGA*/
    if (nb_attr>0)                          /*teste l'existence d'au moins un attribut*/
      cga = NewComponent(CGA,my,nb_attr);   /*création du cga*/
    InitObject(my,cga,arg);                 /*initialisation des attributs*/
    do {
     GetMessage(&m);
     if (m->request == ACCESS_ATTR)
        SendaMessage(cga,m);                /*redirection des accès vers le cga*/
    else {                                  /*invocation de méthode*/
        NewComponent(CEM,m->request,my,cga,ArgMessToArray(m)));
        FreeMessage(m);
     }
    } while (! KillComponent());
}
```

figure 2.15 *Code d'un CRO multi-programmé avec un CGA*

La fonction comportementale du CGA alloue l'espace mémoire du tableau des variables d'instance. Le nom de l'objet et le nombre d'attributs sont passés au CGA lors de sa création. Il extrait une à une les requêtes aux attributs et les traite par des accès au tableau des attributs.

```
/***********          fonction comportementale d'un CGA                ******/
void Cga(Component my,int *arg)
    {/*arg[1] objet courant, arg[2] nombre d'attributs*/
    Mess_Ptr m;
    int *attr = new(arg[2]);              /* nombre d'attributs*/
    Component obj = arg[1];               /*nom de l'objet*/
    do {
     GetMessage(&m);                      /*extraction d'un requête*/
     if (m->request == ACCESS_ATTR)
        switch (ArgMess(m,1)) {           /*type d'accès*/
                case WRITE :              /*ecriture*/
                                attr[ArgMess(m,2)] =ArgMess(m,3);
                                SendReply(m->mandataire,attr[ArgMess(m,2)]);
                                break;
                case READ :               /*lecture */
                                SendReply(m->mandataire,attr[ArgMess(m,2)]);
                                break;
        }
     FreeMessage(m);
    } while (! KillComponent());
    dispose(attr);
}
```

figure 2.16 *La fonction comportementale d'un CGA*

Eclatement des variables d'instances

L'idée de dissocier les variables d'instance de l'objet peut être développée. Il est en effet intéressant d'éclater le CGA en plusieurs Cac/s pour:

- 1 - des raisons matérielles: les objets physiquement éclatés comportant des attributs qui sont localisés sur des noeuds spécifiques.
- 2 - éviter les goulots d'étranglement sur certains attributs et permettre des accès parallèles aux variables d'instance. Les méthodes d'une instance peuvent accéder continuellement aux mêmes attributs; un CGA unique parviendrait vite à saturation. En éclatant le CGA en plusieurs CGA/s, la représentation permettrait des accès parallèles à des attributs distincts.
- 3 - rapprocher certains attributs des méthodes qui les utilisent. On constate dans la programmation objet que certaines méthodes ou groupe de méthodes accèdent fréquemment à une ou un sous-ensemble de variables d'instance. Prenons l'exemple d'un objet ayant plusieurs attributs de type structure de donnée (tableau...). Chaque structure de données et ses variables de contrôle forme un sous ensemble de variables d'instance rattaché à un sous-groupe de méthodes manipulant la structure. Ces relations particulières entre les méthodes et les variables d'instances peuvent être matérialisées à l'exécution en rapprochant une partie des attributs de l'exécution de ces méthodes. Les attributs seront alors sur le même noeud que les exécutions de méthodes qui les utilisent fréquemment privilégiant ainsi les performances de l'application.

Le CGA est alors éclaté en plusieurs CGA/s gérant chacun une ou un ensemble de variables d'instance. Les CGA/s sont créés à l'instanciation de l'objet et sont connus par les CEM/s qui peuvent alors communiquer directement avec eux.

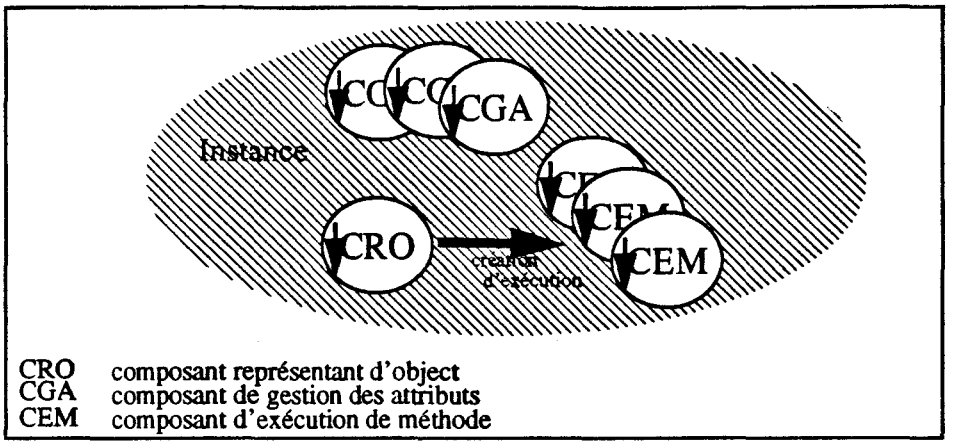


figure 2.17 *Variables d'instance réparties*

Nous proposons en Annexe I, une fragmentation des instances en fonction de l'héritage des classes.

réalisation

La fonction comportementale du CRO doit créer et initialiser les CGA/s de l'objet. Elle crée un tableau contenant les adresses des différents CGA/s qui sera passé aux différents CEM/s à leur création. Le nombre d'attributs par fragment d'objet et la localisation de chaque attribut dans les fragments sont calculés par deux fonctions *NbAttrFrag()* et *SearchAttrFrag()*. Ces deux fonctions sont initialisées par le programmeur.

```
/**      fonction comportementale d'un CRO multiprogrammé avec plusieurs Cga/s ****/
void CroMultiProgCgas(Component my,int *arg)
    {
    Mess_Ptr m;
    Component *cga;                                  /*tableau des Cga/s*/
    int i;
    if (nb_frag > 0) {
      cga = new(nb_frag );                           /*allocation du tableau des Cga*/
      for (i = 1 ; i <= nb_frag; i ++)
        cga[i] = NewComponent(CGA,my,NbAttrFrag(i)); /*création d'un Cga*/
    };
    InitObjectFrag(my,cga,arg);
    do {
        GetMessage(&m);
        if (m->request == ACCESS_ATTR) /*rederige les messages vers les Cga/s*/
                SendaMessage(cga[SearchAttrFrag(ArgMess(m,2))],m);
        else {
                NewComponent(CEM,m->request,my,cga,ArgMessToArray(m));
                FreeMessage(m);
        }
    } while (! KillComponent());
    dispose(cga);
}
```

figure 2.18 *La fonction comportementale du CRO d'un objet éclaté*

L'éclatement d'un CGA ne peut être automatique pour des raisons d'efficacité et ne doit concerner qu'un nombre restreint d'objets. Il est donc préférable de laisser cette tâche au programmeur de l'objet qui prendra la décision de répartir ou non les variables d'instance de l'objet.

- III - 2.1.2 La synchronisation dans l'objet

Nous avons précédemment décrit plusieurs représentations d'objets multi-programmés sans nous soucier des problèmes de synchronisation dans l'objet. Mais les différentes activités concurrentes dans ces objets nécessitent l'utilisation d'une synchronisation. Nous proposons des extensions à nos représentations objets intégrant deux types de synchronisation: la synchronisation des accès aux variables d'instance; la synchronisation a priori des exécutions de méthode dans l'objet.

- III - 2.1.2.1 La synchronisation des accès aux variables d'instance

Pour garantir l'unicité de certaines manipulations d'attributs, telles que *attr = attr* +1. Un CEM peut se réserver le CGA. Pour cela, il envoie un message LOCK au CGA. A sa réception, le CGA retourne l'adresse d'une nouvelle boîte aux lettres locale créée spécialement par les communications entre ce CEM et le CGA. Le CGA ne lit alors que les messages déposés dans cette boîte mettant ainsi en attente toutes les autres demandes venant des autres CEM/s. A la fin de l'opération, le CEM doit envoyer un message UNLOCK qui débloque le CGA qui lit alors les messages en attente dans sa propre boîte.

réalisation

```
/***********          fonction comportementale d'un CGA protégé          ********/
void CgaLock(Component my,int *arg)
    {/*arg[1] objet courant, arg[2] nombre d'attributs pour le cga*/
    Mess_Ptr m;
    int *attr;
    Component obj = arg[1];        /*nom de l'objet*/
    Box box = my;                  /*boite aux lettres de travail inialisée avec la boîte de l'objet*/
    attr = new(arg[2]);            /*création du tableau des attributs*/
    do {
      GetMessage(box,&m);                    /*extraction d'un message de la boite de travail*/
      if (m->request == ACCESS_ATTR)
         switch (ArgMess(m,1)) {
                case LOCK :    box = NewBox();          /*création d'une boîte privée*/
                               SendReply(m->mandataire,box);
                               break;
                case UNLOCK : FreeBox(box);             /*libération du Cga*/
                               box = my; /*
                               SendReply(m->mandatire,box);
                               break;
                case WRITE :   attr[ArgMess(m,2)] = ArgMess(m,3);          /*ecriture*/
                               SendReply(m->mandataire,attr[ArgMess(m,2)];
                               break;
                case READ :    SendReply(m->mandataire,attr[ArgMess(m,2)]);  /*lecture*/
                               break;
         };
      FreeMessage(m);
    } while (! KillComponent());
    dispose(attr));
}
```

figure 2.19 *La fonction comportementale d'un CGA protégé*

- III - 2.1.2.2 La synchronisation a priori de méthodes

Les objets multi-programmés permettent les exécutions concurrentes de méthodes qui nécessitent une synchronisation a priori dans l'objet. Cette synchronisation intra-objet peut s'exprimer par des conditions d'activation sur les méthodes (voir partie - I - 2. «Objets et Synchronisation» du chapitre I). Pour exécuter une méthode, il faut que sa condition d'activation associée soit vérifiée.

Cette tâche de synchronisation est logiquement effectuée par le CRO qui reçoit les requêtes à l'objet. Ce traitement se décompose comme suit: la lecture des requêtes, l'évaluation des conditions d'activation des méthodes, création d'un CEM pour les méthodes dont la condition d'activation est vérifiée et la prise en charge des requêtes en attente dont la condition n'est pas encore vérifiée.

Nous avons choisi de réaliser les conditions d'activation en utilisant des compteurs de synchronisation sur les événements intervenant dans l'objet et des fonctions de garde sur les méthodes à synchroniser.

Pour une méthode nous utilisons les compteurs suivants:

- le nombre d'activations de la méthode,
- le nombre de terminaisons des méthodes
- le nombre de requêtes en attente de la méthode
- le nombre de méthodes actives

Ces quatre compteurs sont utilisables par le programmeur de l'objet pour décrire les conditions d'activation sur les méthodes de l'objet.

Toute la synchronisation est centralisée dans la fonction comportementale du CRO. Les compteurs de synchronisation sont stockés dans un tableau interne à l'objet. Chaque élément du tableau contient les valeurs des compteurs pour une méthode de l'instance.

Les conditions d'activation sont contenues dans un tableau spécifique initialisé par le programmeur. L'indice dans le tableau correspond à celui de la méthode gardée dans le tableau des méthodes. Par défaut une méthode n'est pas contrôlée.

réalisation

Les messages de requête, dont la condition n'est pas vérifiée, doivent être conservée dans le CRO. Pour cela, le CRO n'extrait de sa boîte que les demandes pouvant être lancées. Il conserve ainsi les requêtes dans leur ordre d'arrivée. Le CRO utilise les primitives de bas niveau de manipulation des Box/s voir (- II - 1.2.3 «Les boîtes aux lettres locales»). (Rappel: la primitive *NumberMessageInBox()* retourne le nombre de messages d'une boite et la primitive *WaitOnBox()* attend un nouvel événement depuis le dernier appel de *WaitOnBox()*. Un événement est un dépôt de message ou une destruction de message dans la boite (modification en écriture de la boîte). Ces deux primitives permettent de n'extraire de la boîte que les invocations pouvant être traitées, ceci afin d'éviter d'effectuer de l'attente active.

La fin de l'exécution d'une méthode provoque un envoi de message au CRO de l'objet de type *END_METHOD*. De plus, nous avons ajouté un champ à la structure des messages de requête permettant de contrôler leur première évaluation. Les messages ne sont retirés de la boîte aux lettres de l'objet que si leur condition d'activation est vérifiée. La boîte contient donc les nouvelles requêtes ainsi que celles en attente. Le nouveau champs *m->control* permet d'identifier la première lecture d'une requête de ses relectures.

```
/****                    fonction comportementale d'un CRO synchronisé                    ****/
void CroMultiProgSynchroCga(Component my,int *arg)
    {
    Mess_Ptr m;
    t_cmp * cmp          = new(nb_methode);          /*création du tableau des compteurs*/
    Component cga        = NewComponent(CGA,my,nb_attr); /*Cac cga*/
    int num, i, treat ;  /*variables de controle de l'algorithme de synchronisation*/
    InitCmp(cmp);                                    /*Raz des compteurs*/
    InitObject(my,cga,arg,);                         /*initialisation des attributs*/
    do {
      if ((num = NumberMessageInBox(my)) > 0);  /*compte le nombre de messages*/
        i = 1; treat = 0;
        do {
           ReadFromBox(my,i,m);                  /*lecture d'un message*/
           switch(m->request) {
                case ACCESS_ATTR :               /*accès aux attributs*/
                   SendaMessage(cga,m);          /*redirection du message vers le Cga*/
                   DeleteMessageInBox(my,i);     /*destruction du message dans la boîte*/
                   treat =1;
                   break;
                case END_METHOD :                /*message de fin de méthode*/
                   cmp[ArgMess(m,1)].in ++;      /*nombre de terminaisons*/
                   cmp[ArgMess(m,1)].active --;  /*nombre de méthodes en cours*/
                   DeleteMessageInBox(my,i);     /*destruction du message dans la boîte*/
                   treat =1;
                   break;
                default :                        /*invocation de méthode*/
                   if (m->control) { /* Première évaluation de la condition sur la requête */
                         cmp[m->request].req ++; /*nombre de requêtes en attente*/
                         m->control = 0;
                   };
                   if (cond_activation[m->request](cmp,attr)){
                         cmp[m->request].req --;  /*nombre de requêtes en atetnte*/
                         cmp[m->request].act ++;  /*nombre d'activations */
                         cmp[m->request].active++;/*nombre de méthodes en cours */
                         NewComponent(CEM,m->request,my,cga,ArgMessToArray(m));
                         DeleteMessageInBox(my,i);/*destruction du message dans la boîte*/
                         treat =1;
                   }
           }
        } while ( (i++ < num ) && (! treat) );
      }
      if (!treat) WaitOnBox(my);                 /*attente passive d'un nouvel événement*/
    } while (! KillComponent());
    dispose(cmp);
}
```

figure 2.20 *La fonction comportementale d'un CRO d'un objet synchronisé*

- III - 2.1.3 Exemple: la Collection d'éléments

Pour illustrer ces représentations réparties d'objets multi-programmés, nous présentons ici l'exemple d'un objet *collection*. L'objet est constitué des caractéristiques suivantes: la collection contient un ensemble d'objets de même type. Deux actions sur cet objet sont possibles: l'ajout d'un élément et la recherche dans la collection. L'utilisateur de la collection ne voit que l'interface de la collection (méthode ajout et recherche) et ne connaît donc pas la réalisation de la collection. Voici un exemple d'objet *collection* d'entier:

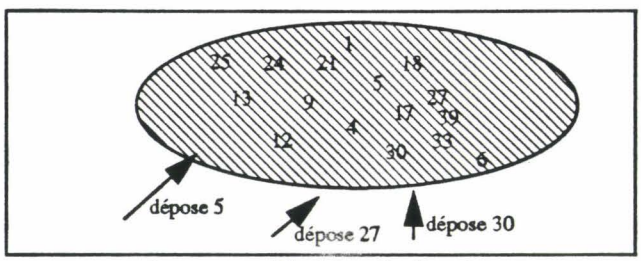


figure 2.21 *L'objet Collection*

- III - 2.1.3.1 Réalisation

L'objet, *collection*, est réalisé pour être réparti sur plusieurs noeuds. Il est découpé en plusieurs tables qui contiennent chacune une partie des éléments de la collection. Les éléments sont distribués sur les tables par une fonction de hachage (de type modulo). Une zone de débordement, représentée par une des tables, reçoit les éléments n'ayant pas pu s'insérer dans les autres tables (faute de place).

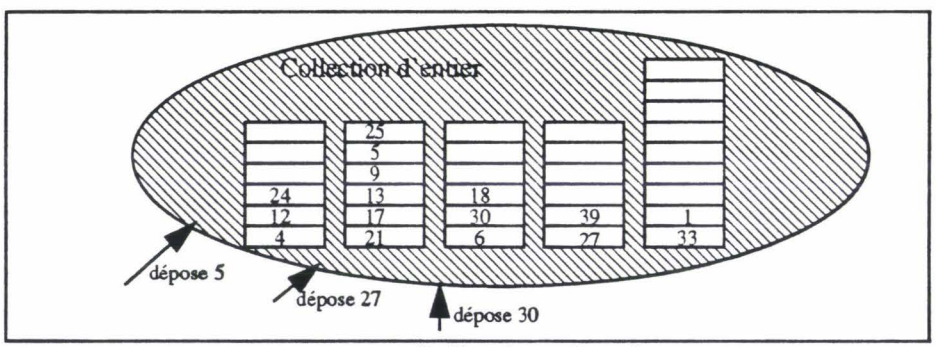


figure 2.22 *Collection répartie*

Pour représenter cette structure, nous utilisons une représentation multi-programmée contenant plusieurs CGA/s. Chaque CGA est associé à une table de la collection et possède: un attribut de type tableau, qui matérialise la table, les variables de contrôle du tableau (taille de la tableau, position pour l'insertion d'un élément). Le CGA est protégé par les méthodes *LOCK* et *UNLOCK* qui interdisent les accès concurrents dans une même table.

Cette représentation de l'objet permet de paralléliser les traitements sur les différentes tables. Elle autorise, en effet, d'insérer simultanément plusieurs éléments dans des tables différentes et de rechercher plusieurs éléments en parallèle. De plus, les éléments de la collection sont répartis sur les noeuds contenant les CGA/s de la collection. Ces propriétés sont directement héritées du modèle de représentation choisi.

- III - 2.1.3.2 Programmation

La code de l'objet se présente en deux sous-ensembles de méthodes: 1) les méthodes qui constituent l'interface (exportées de l'objet) et qui s'appliquent sur l'objet entier. 2) les méthodes de manipulation de table (ajouter un élément à la table, recherche d'un élément). Ces deux méthodes s'appliquent à un CGA. Les CGA/s de l'objet étant construits sur la même base (nom et place des variables d'instance), les méthodes peuvent s'appliquer indifféremment sur n'importe quel CGA.

Dans cette réalisation, la collection est constituée de quatre tables, plus une zone de débordement.

```
/*****************************************************************************/
/*                        COLLECTION REPARTIE D'ELEMENTS                     */
/*****************************************************************************/
#include <Prim_Obj.h>
/*****************************************************************************/
/*                              Methodes sur les CGA/s                       */
/*****************************************************************************/
void initTable(Component my,tobject obj, Component *cga,int *arg)
    {                           /*arg[2] numéro de cga, arg[3] taille du table*/
    Call(cga[arg[2]],ACCESS_ATTR,3,A_WRITE,1,arg[3]);          /*taille*/
    Call(cga[arg[2]],ACCESS_ATTR,3,A_TAB_CREATE,2,arg[3]);     /*tableau*/
    Call(cga[arg[2]],ACCESS_ATTR,3,A_WRITE,3,0);               /*position*/
    ReplySelf();
}
void putInTable(Component my, tobject obj, Component *cga, int *arg)
    {                           /*arg[2] numéro de cga, arg[3] élément à insérer*/
    Box b;
    int pos,fin;
    b = Call(cag[arg[2]],ACCESS_ATTR,A_LOCK);          /*réservation du CGA*/
    fin = Call(b,ACCESS_ATTR,A_READ,1);                /*lecture de la taille*/
    pos = Call(b,ACCESS_ATTR,A_READ,3);                /*lecture de la position*/
    if (pos >= fin)
        ReplyValue(0);                                 /*tableau plein*/
     else {
        Call(b,ACCESS_ATTR,A_TAB_SET,2,pos,arg[3]);    /*ecriture dans le tableau*/
        Call(b,ACCESS_ATTR,A_INC,3);                   /*incrémente pos*/
        ReplySelf();
    }
    b = Call(b,ACCESS_ATTR,A_UNLOCK);                  /*débloque le CGA*/
}
void chercheInTable(Component my, tobject obj, Component *cga, int *arg)
    {                           /*arg[2] numéro de cga, arg[3] élément recherché*/
    Box b;
    int pos, res,i=0,trouve = 0;
    b = Call(cag[arg[2]],ACCESS_ATTR,A_LOCK);          /*réservation du CGA*/
    pos = Call(b,ACCESS_ATTR,A_READ,3);                /*lecture de la taille*/
    if (pos != 0)
      do {                                             /*tableau non vide*/
        res=Call(b,ACCESS_ATTR,A_TAB_GET,2,i);         /*lecture du ième élément*/
        if (res == arg[3]) trouve =1;                  /*teste avec l'élement recherché*/
      } while ((++i< pos) && (! trouve));
    b = Call(b,ACCESS_ATTR,A_UNLOCK);                  /*débloque le CGA*/
    ReplyValue(trouve);
}
```

```
/*****************************************************************************/
/*                          Methodes sur l'objet                             */
/*****************************************************************************/
void init(Component my, tobject obj, Component *cga, int *arg)
    {                                   /*arg[2] taille des table*/
    int i;
    for (i=1; i<=4; i++)                /*création des 4 tables */
        Call(obj,InitTable,i,arg[2]);   /*initialisation des tables*/
    Call(obj,InitTable,5,arg[2]);       /*initilisation de la zone de débordement*/
    ReplySelf();
}
void depose(Component my, tobject obj, Component *cga, int *arg)
    {                                   /*arg[1] mandataire, arg[2] élément à insérer*/
    int res;
    res = Call(obj,putTable,arg[2] % 4 + 1,arg[2]);   /*essai de dépôt dans un cga*/
    if (res == 0) {                     /*tableau plein */
        res = Call(obj,putTable,5,arg[2]);            /*dépôt dans la zone de débordement*/
        if (res == 0)       printf("collection pleine elt %d \n",arg[2]);
    }
    ReplyValue(res);
}
void cherche(Component my, tobject obj, Component *cga, int *arg)
    {                                   /*arg[1] mandataire, arg[2] élément recherché*/
    int res;
    res = Call(obj,chercheTable,arg[2] %4 + 1,arg[2]);/*recherche dans un cga*/
    if (res == 0) {                     /*élément non trouvé*/
        res = Call(obj,chercheTable,5,arg[2]);        /*recherche dans la zone de débordement*/
    }
    ReplyValue(res);
}
void InitType(void)
    {
    type_objet = MULTI_CGAS;            /*représentation de l'objet*/
    name = COLLECTION;                  /*nom de type*/
    InitMethode(6);                     /*nombre de méthodes*/
    methode[0] = init;                  /*association de chaque méthode */
    methode[1] = initTable;             /*avec sa fonction C */
    methode[2] = putInTable;
    methode[3] = chercheInTable;
    methode[4] = depose;
    methode[5] = cherche;
    nb_attr = 15;                       /*nombre de fragments*/
    InitCga(5,3,3,3,3,3);               /*nombre de fragments et répartition des attributs*/
}
```

figure 2.23 *Programmation de la Collection répartie*

Une fonction *InitType()* décrit le type des objets, le nom du type, le nombre de CGA/s, et initialise le tableau des méthodes. Dans cette primitive, la fonction *InitCga()* détermine le nombre de CGA/s (ici 5) et pour chacun d'eux le nombre d'attributs (ici 3 par CGA).

- III - 2.1.4 Mesures des différents modèles d'objet

Nous avons élaboré plusieurs représentations distribuées d'objets mono et multi-programmés. Un premier prototype est réalisé sur le run-time Cac sur lequel sont développées les différentes fonctions comportementales représentant les CRO, CGA et CEM des objets actifs. Le prototype objet fonctionne au dessus du run-time Cac sur le MultiCluster II.

Nous cherchons maintenant à mesurer les différentes représentations d'objet et le gain apporté par nos représentations. Dans une première phase, nous présenterons les temps d'exécution des primitives objet. Dans une seconde phase, nous mesurerons les temps globaux de deux applications tests.

- III - 2.1.4.1 Mesures des primitives

Nous mesurons la durée d'exécution moyenne des primitives développées sur le prototype (création d'une instance, invocation de méthode, self-invocation et accès à un attribut). Tous les temps sont donnés en milli-seconde.

Les objets mono-programmés

La représentation d'une instance est centralisée dans le composant représentant d'objet. L'accès aux attributs s'effectue directement par un accès mémoire dans l'environnement local du composant. L'exécution des méthodes dans un objet est réalisée comme un appel procédural.

Primitives	Temps ms
création d'une instance (locale - distante)	0.30 - 4.00
self-invocation	0.05
invocation synchrone (locale - distante)	1.00 - 7.00
accès à un attribut	0.01

figure 2.24 *Les objets mono-programmés*

La différence entre l'invocation locale et celle distante s'explique par les deux envois de messages distants dans le réseau (voir les mesures du run-time Cac - II - 3.2.4.1 «Mesures du run-time Cac/s»). Les deux messages matérialisent la requête vers l'objet cible et la réponse vers l'objet mandataire.

Les objets multi-programmés

La représentation de nos objets multi-programmés est répartie. Les opérations intra-objet effectuées localement dans un objet mono-programmé sont maintenant réparties sur plusieurs Cac/s et donc éventuellement sur plusieurs noeuds.

Exécution décentralisée des méthodes, les CEM/s

Dans les objets multi-programmés, l'exécution d'une méthode s'effectue dans un composant spécifique; le CEM.

Primitives	Temps ms
création d'une instance (locale - distante)	0.30 - 4.0
self-invocation (locale - distant)	1.45 - 7.0
invocation synchrone (locale - distante)	1.45 - 7.5
accès à un attribut (local - distant)	0.90 - 7.0

figure 2.25 *Les objets multi-programmés*

Les différences obtenues par rapport à la version mono-programmée sont liées à la création dynamique d'un composant d'exécution de méthode lors d'une invocation locale ou distante. L'accès aux attributs, lors de l'exécution de méthode, s'effectue par deux communications entre le composant CRO contenant les variables et le composant CEM. Cette opération peut se révéler coûteuse si le CRO est distant. Le degré de parallélisme des exécutions de méthodes intra-objet est important et compense le temps d'accès aux attributs. Si l'objet utilise peu le parallélisme intra-objet, il est préférable de créer les CEM/s sur le même noeud que le CRO de l'objet afin de limiter les communications distantes.

Le CGA

Les attributs de l'objet sont séparés du Cac qui le représente. La représention permet de privilégier les accès internes aux attributs.

Primitives	Temps ms
création d'une instance (locale - distante)	0.60 - 4.3
self-invocation (locale - distant)	1.45 - 7.0
invocation synchrone (locale - distante)	1.45 - 7.5
accès à un attribut (cga local - distant)	0.90 - 7.0

figure 2.26 *Les objets multi-programmés avec un CGA*

Les différences obtenues avec la version précédente sont liées au temps de création du Cac CGA pendant l'instanciation.

Le CGA éclaté

Les attributs sont découpés en sous-ensembles pris en charge par plusieurs CGA/s.

Primitives	Temps ms
création d'une instance (locale - distante) (1,2,3 ensembles)	0.6, 0.9,1.2 - 4.3, 4.6,4.9
self-invocation (locale - distant)	1.45 - 7.0
invocation synchrone (locale - distante)	1.45 - 7.5
accès à un attribut (cga local - distant)	0.90 - 7.0

figure 2.27 *Les objets multi-programmés avec plusieurs CGA/s*

Le temps de création d'un objet est maintenant linéairement proportionnel au nombre de CGA/s qu'il possède.

Les mesures sont étroitement liées aux performances du run-time Cac. Chaque mesure peut se calculer en cumulant les temps de chaque appel aux fonctions du run-time Cac. (Toute amélioration du run-time Cac est directement répercutée sur les performances du run-time.)

- III - 2.1.4.2 Mesures en contexte parallèle

Ces mesures des primitives n'ont qu'une signification relative; en contexte parallèle seul le temps d'exécution de l'application est important. Pour comparer ces différentes représentations dans leur environnement d'exécution parallèle, nous avons développé deux applications tests provoquant un grand nombre d'appels de méthode. Nous essayons de mesurer le temps gagné par le parallélisme intra-objet ainsi que le coût des accès aux attributs.

Mesures du parallélisme intra-objet

La première application crée un seul objet sur lequel sont envoyées 500 invocations de méthodes en mode asynchrone. Nous mesurons ici le parallélisme intra-objet. Le temps est mesuré après les 500 exécutions complètes de méthodes.

Nous avons ensuite modifié le code de la méthode pour intégrer des accès aux variables d'instance dans le source des méthodes. Nous mesurons la même application avec 0, 1, 2et 3 accès aux attributs par méthode.

Nous ne cherchons pas à mesurer une stratégie de répartition du code sur les noeuds. Les mesures ont été effectuées en dupliquant l'ensemble du code sur tous les noeuds. (Nous utilisons 5 noeuds pour répartir l'ensemble des Cac/s des objets).

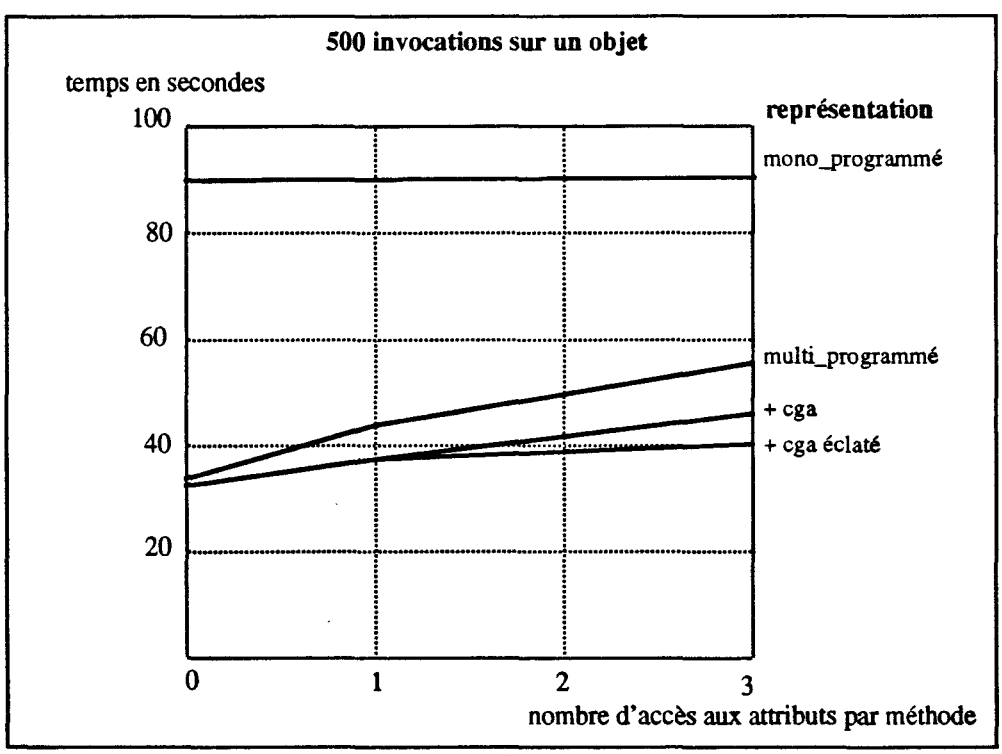


figure 2.28 *Invocations de méthode et accès aux attributs*

La représentation mono-programmée de l'objet sérialise les invocations asynchrones. Il n'y a donc pas de parallélisme. L'accès direct aux attributs n'influe que très peu sur les performances globales de l'application.

Dans les versions multi-programmées, l'objet effectue le traitement des invocations en parallèle; les performances sont directement liées au nombre de noeuds (ici pour 5 noeuds un facteur de 2 à 3.5). Lorsqu'on accroît le nombre d'accès aux attributs, les performances décroissent franchement. L'ajout d'un CGA libère alors le CRO et améliore les performances; celles ci sont encore meilleures en éclatant le CGA.

Mesures du parallélisme intra-objet et inter-objet

Nous avons développé une seconde application reprenant les mêmes descriptions que les précédentes mais le nombre d'objets sollicités passe à trois. Les 500 invocations sont réparties sur alors ces 3 objets. Les mesures sont toujours effectuées sur 5 noeuds. On pourra ainsi comparer le temps global de traitement par rapport aux mesures précédentes). Nous avons donc introduit un parallélisme inter-objet. On peut ainsi mesurer le parallélisme intra-objet en présence d'un parallélisme inter-objet. Par rapport aux mesures précédentes, les conditions sont différentes et nous comparons maintenant: 1) la version mono-programmée avec 3 objets qui peuvent se répartir sur 3 noeuds, parmi les 5 disponibles, 2) les versions multi-programmées où les 3 objets sont réellement répartis sur 5 noeuds. Le gain maximun théorique entre les deux versions est de 1.66 (5 noeuds/ 3 noeuds). L'échelle des temps est ici différente.

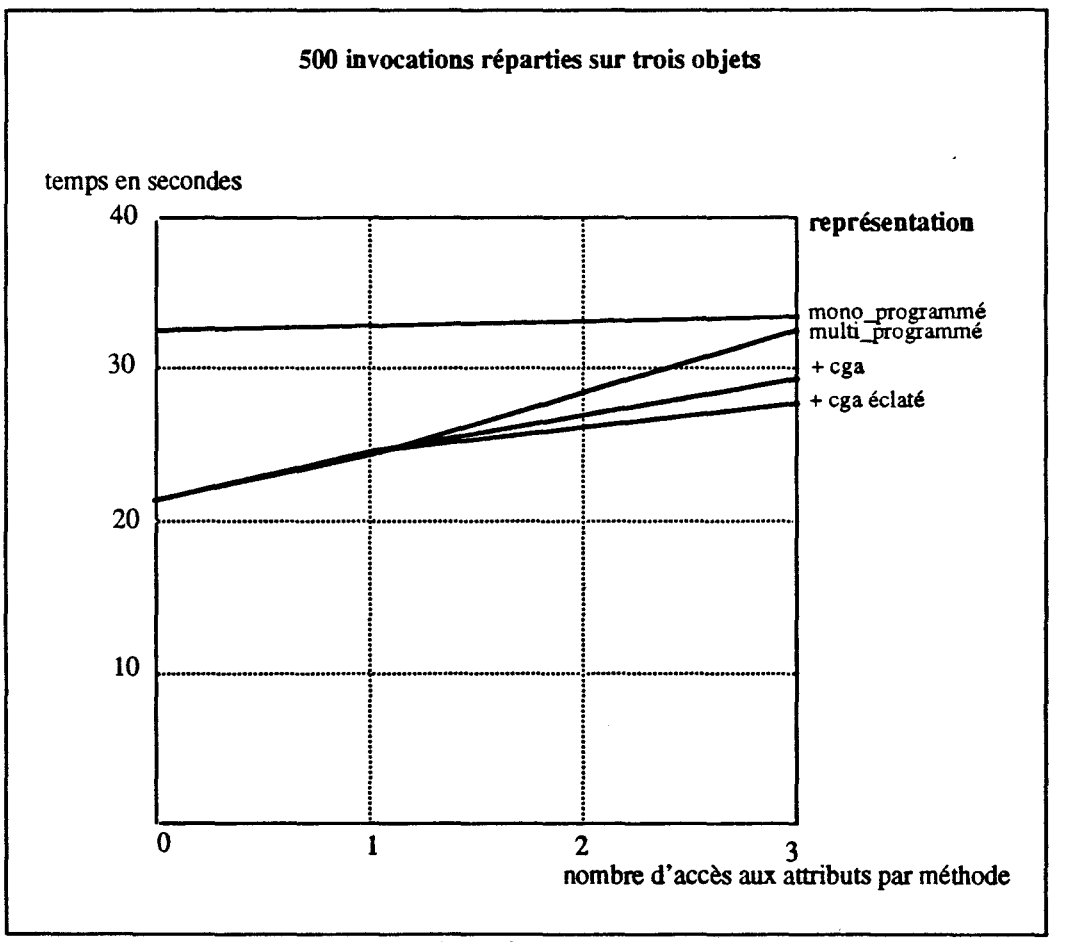


figure 2.29 *Invocations de méthode sur trois objets*

Comparaison des deux séries de mesures (1 et 3 objets):

- La création de plusieurs objets distribue les invocations de méthode sur les trois objets. Dans la version mono-programmée, le gain (2.72) est linéairement proportionnel au nombre d'objets et au nombre de noeuds (3). (la mesure de référence est effectuée sur un objet, la page précédente).

- En versions multi-programmées, la création de plusieurs objets apporte du parallélisme de créations des CEM/s. Le gain varie de 1.3 à 1.8 selon la représentation de l'objet et le nombre d'accès aux attributs.

Comparaison des différents types d'objets (avec 3 objets)

- Les versions multi-programmées restent donc plus efficaces que celle mono-programmée malgré le parallélisme inter-objet (coefficient de 1.5, rappel de la valeur théorique 1.66).

- Les conclusions de la section «Mesures du parallélisme intra-objet» se confirment ici, et, les différentes versions multi-programmées restent dans l'ensemble plus performantes.

- III - 2.1.4.3 Conclusion

Les premières mesures montrent que si l'objet est complexe, les appels aux primitives du run-time sur cet objet sont coûteux. Ces différences doivent être compensées par le réel parallélisme intra-objet. Mais celui-ci n'est effectif que si l'objet est réellement utilisé en tant qu'objet multi-programmé. Il est donc préférable de choisir la bonne représentation de l'objet à sa définition pour optimiser l'application. De plus, tous les objets d'une application ne sont pas toujours fragmentés, synchronisés et multi-programmés. La coexistence des différents types d'objets dans une même application s'avère indispensable.

Les mesures mettent en évidence que le parallélisme intra-objet lié aux exécutions parallèles des méthodes ou à l'éclatement des CGA/s, peut très bien être associé à un parallélisme inter-objet. Les deux parallélismes se complètent alors sans diminuer pour autant les performances globales de l'application.

- III - 2.2 Répartition des classes

Dans la première partie nous avons proposé différentes représentations réparties d'objets actifs. Nous avions supposé que le code était dupliqué sur tous les noeuds de la machine cible. Or cette duplication est coûteuse et d'autres solutions doivent donc être envisagées. Dans la partie (- I - 3. «Objets et Répartition») du chapitre I, nous avons vu les techniques de répartition du code sur les noeuds de la machine cible. Le code d'une application objet ainsi que les éléments propres à la classe sont contenus dans une structure de données à l'exécution représentant les classes de l'application. Deux approches permettant d'implanter un langage à objets sur une machine parallèle sont alors utilisées:

- la duplication de la structure de données des classes sur les différents sites,
- l'éclatement de la structure de données des classes sur les différents sites.

Nous allons reprendre ces deux approches et les adapter aux représentations réparties d'objet décrites dans la partie précédente. La structure de donnée des classes doit être décrite et réalisée dans l'environnement des Cac/s.

- III - 2.2.1 Les classes à l'exécution

Une classe est un outil de programmation qui définit les caractéristiques d'un ensemble d'objets. Elle décrit la composition de ses instances, (le nombre et le type de chaque variable d'instance) et les méthodes pouvant s'appliquer sur les instances.

A l'exécution, ces caractéristiques de la classe doivent être conservées et réparties sur les noeuds de la machine cible. Dans les implantations de langages à objets, les classes en tant qu'objet n'existent pas toujours à l'exécution. Une structure de données regroupe alors une représentation de toutes les caractéristiques des classes nécessaires au fonctionnement de l'application: Ces éléments à l'exécution sont:

- 1 - une maquette d'instance permettant lors de l'instanciation de créer et d'initialiser les variables d'instance d'un objet.
- 2 - le code des méthodes d'instance.

Tous ces éléments composants des différentes classes d'une application sont donc concaténés dans une structure globale des classes. Celle ci est alors implantée sur les noeuds de la machine cible. Elle est utilisée:

- lors de la création des instances pour obtenir la maquette des instances.
- lors des invocations de méthode pour exécuter le code de la méthode invoquée.

- III - 2.2.2 Duplication d'une structure globale des classes

Une première approche consiste à dupliquer la structure globale des classes sur tous les sites de la machine cible. La structure regroupe alors le code de toutes les méthodes définies dans les classes de l'application et l'ensemble des maquettes d'instances des objets de cette application.

Pour illustrer les différentes possibilités de répartition des classes à l'exécution, nous allons travailler sur un exemple abstrait. Une application définit cinq classes, nommées respectivement *A, B, C, D, E*. A un instant donné de l'application, quatre objets ont été créés; une instance de *A*, une instance de *B* et deux instances de *C*. La machine cible possède trois noeuds numérotés de 1 à 3. Le problème consiste à répartir les caractéristiques des classes (les maquettes d'instances et le code des méthodes) sur les noeuds de la machines.

L'application peut alors se schématiser à l'exécution comme suit.

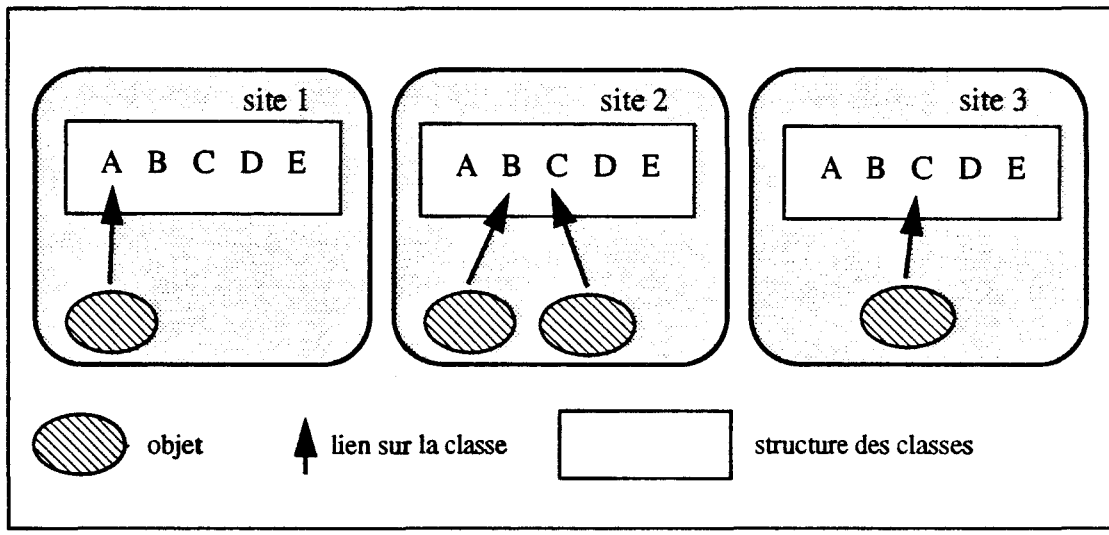


figure 2.30 *Duplication des structures de données des classes*

La duplication du code sur tous les noeuds présente plusieurs avantages:

- Les problèmes de défaut de localité du code, où le code d'une méthode invoquée n'est pas sur le même noeud que l'instance cible (voir chapitre I) sont impossibles. (puisque le code est dupliqué sur tous les noeuds).

- Il n'y a pas de contraintes sur la localisation des objets: ceux-ci peuvent être créés sur n'importe quel noeud de la machine.

- Les objets de l'application sont réalisés par des Cac/s. Le code étant dupliqué sur tous les noeuds, les Cac/s de l'objet peuvent être répartis sur plusieurs noeuds. Il y a donc un réel parallélisme intra-objet.

- Suivant la représentation de l'objet, la distribution des objets sur les noeuds est une distribution de Cac/s. Celle-ci est d'ailleurs laissée à la charge du système Cac sous-jacent.

En termes de Cac/s, l'application ci-dessus peut se matérialiser comme suit

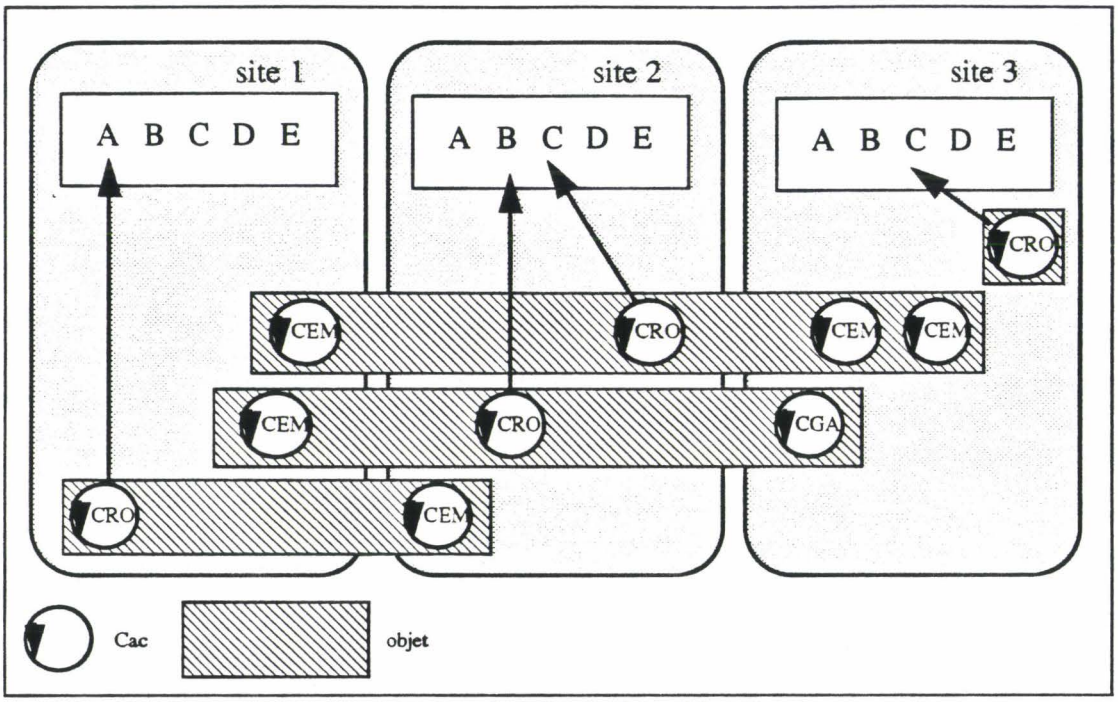


figure 2.31 *Distribution des Cac/s des objets sur les noeuds*

Les traitements des invocation de méthode sur une instance peuvent ainsi se répartir sur plusieurs noeuds de la machine et matérialiser un réel parallélisme intra-objet.

La duplication d'une structure globale des classes présente quelques inconvénients:

- Elle entraîne en effet un chargement important de code sur les noeuds de la machine cible. Elle nécessite pour une application de grosse taille des noeuds ayant une mémoire importante.

- La distribution automatique des Cac/s des objets faite par le système manque de souplesse. En effet, cette stratégie apporte peu de contrôle sur la répartition des objets sur les noeuds. Elle ne laisse aucune possibilité de directive de répartition des objets au programmeur, si aucun outil spécialisé n'est fourni.

Réalisation

Nous avons réalisé la structure de classe avec les **modules**, entités de partage de code du modèle Cac. Les fonctions comportementales des Cac/s constituant les objets sont réparties dans ces modules. La duplication d'une structure globale des classes est réalisée comme suit:

Un seul module contient l'ensemble des méthodes des classes de l'application ainsi que la maquette des instances contenues dans la méthode d'instanciation. Il contient aussi les fonctions comportementales des différents types de Cac/s de l'objet. Le module est ensuite chargé sur tous les noeuds.

- III - 2.2.3 Eclatement de la structure globale des classes

Cette seconde approche du problème consiste à éclater la structure de données des classes sur le réseau. Le code des méthodes est alors dispersé sur l'ensemble du réseau pour limiter les copies de code. Nous proposons différentes variantes qui modifient le nombre de copies de la structure de données des classes.

- III - 2.2.3.1 Une structure de données par classe

L'idée est de conserver la structure de classe à l'exécution. Chaque classe d'objet est alors matérialisée par une structure à l'exécution qui est le support logiciel pour les instances. Elle renferme le code de toutes les méthodes définies dans la classe ainsi que la maquette d'instance des objets de la classe.

La structure de la classe est à l'exécution implantée sur un des noeuds de la machine cible, chaque noeud pouvant accueillir une ou plusieurs classes. La répartition des classes sur les noeuds du multicomputer est prise en charge par le système, mais peut être laissée à la charge du programmeur.

Le traitement d'une invocation de méthode se déroule localement sur un noeud puisque la méthode et l'instance sont implantées dans la même structure à l'exécution, la classe.

Nous reprenons l'exemple précédent, utilisant cinq classes et quatre instances. Voici, une des possibilités de répartition des classes sur les trois noeuds de la machine cible.

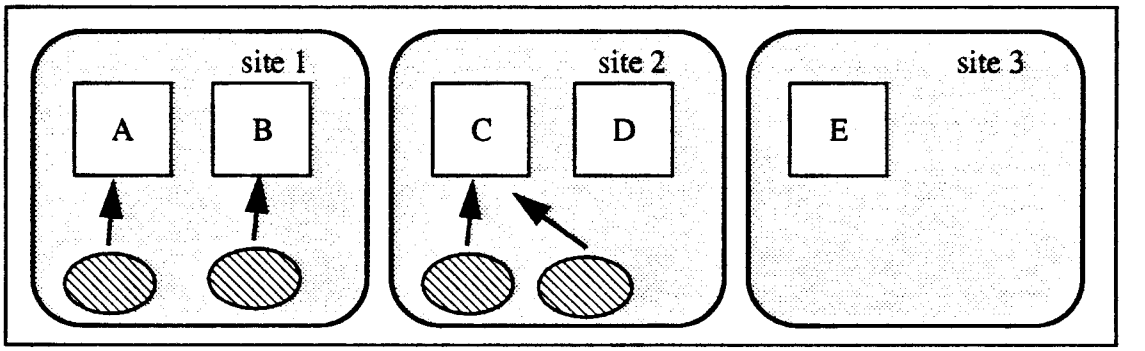


figure 2.32 *Distribution en fonction des classes*

Les avantages de cette stratégie sont:

- La possibilité de répartir explicitement les classes (donc les objets) sur les noeuds de la machine. Ceci d'autant plus que les classes sont des entités du modèle objet et que la notion de classe est connue par le programmeur.
- Cette structure de classe peut être un résultat de compilation et permettre ainsi une réutilisation de classes déjà compilées.

Les inconvénients de cette solution sont liés à la répartition du code en structure de classes:

- Les objets d'une même classe ne peuvent pas être répartis sur plusieurs noeuds.
- Les Cac/s (CRO et CEM) constituant un même objet sont forcément sur le même

noeud et il n'y a donc plus de parallélisme intra-objet et de réelle répartition de représentation des objets.

La solution nécessite:

- 1) la connaissance du réseau de noeuds avant l'exécution de l'application. Ceci pour savoir où implanter chaque classe en fonction de ses relations avec ses voisines et du nombre de noeuds dont l'application dispose.
- 2) une adéquation entre le nombre de noeuds et celui des classes. Il ne faut pas que le nombre de classes soit inférieur au nombre de noeuds. De plus, si les classes sont plus nombreuses que les noeuds, il faut répartir la charge sur les modules. (par exemple en ayant une idée du nombre d'objets créés dans chaque classe). Dans l'exemple ci-dessus, le site 3 contenant la classe *E* est sous-utilisé.

Réalisation

Un module est associé à chaque classe. Il regroupe toutes les méthodes de la classe et les fonctions comportementales des Cac/s constituant la représentation des objets de la classe. Chaque module est alors implanté sur un noeud. La répartition est effectuée explicitement par le programmeur ou automatisée par le système Cac (voir chapitre II)

- III - 2.2.3.2 Duplication de la structure de données d'une classe

Pour atténuer les inconvénients de la solution précédente, nous proposons une nouvelle extension. Pour cela, nous permettons la duplication d'une même structure de classe sur plusieurs noeuds. Nous gardons aussi la structure de classe tout en permettant une répartition plus fine des objets (et des Cac/s).

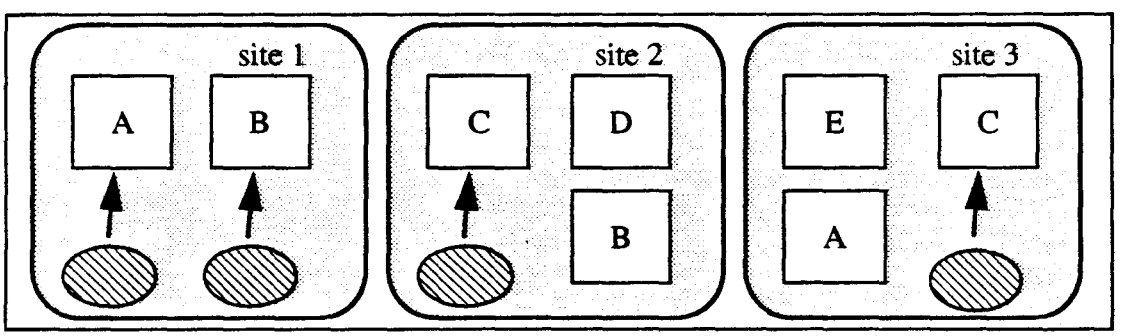


figure 2.33 *Duplication des structures de données des classes*

Les nouveaux avantages de cette extension sont:

- Le programmeur répartit plus finement les classes de son application en fonction des contraintes matérielles de sa machine cible (noeuds et topologie du réseau).
- Les objets d'une même classe sont répartis sur les noeuds d'accueil de cette classe réduisant ainsi la charge des classes fortement sollicitées.
- Les Cac/s (CEM et CRO) d'un même objet sont aussi répartis sur les noeuds d'accueil de la classe. Il y a donc un réel parallélisme intra-objet dans les objets multi-programmés.

L'inconvénient principal de cette solution, tient au fait que le nombre de classes peut être très important. La répartition des classes est alors délicate (voir incontrôlable); mais cette phase peut être prise en charge par le système sous-jacent ou par un outil de «mapping»

NB: Si on copie toutes les classes sur tous les noeuds, cela revient à dupliquer tout le code; la solution est alors identique à celle définissant une structure globale des classes.

Réalisation

La réalisation reprend les modules associés aux classes définis dans la section précédente. Un même module est ensuite chargé sur plusieurs noeuds. Cette duplication doit être contrôlée par le programmeur.

- III - 2.2.4 Evaluation des différentes répartitions

Nous mesurons les performances des différentes répartitions proposées. Nous reprenons pour cela les deux applications décrites dans la section - III - 2.1.4.2 «Mesures en contexte parallèle» de la partie - III - 2.1 «Les objets actifs distribués». Nous étudions l'influence de la répartition de la structure sur le temps total d'exécution de ces deux applications.

Les mesures sont étroitement liées à la répartition des Cac/s décrite dans le chapitre II (voir - II - 3.2.3.4 «Stratégie de répartition des Cac/s sur les modules»). Les deux stratégies, l'une centralisée, l'autre décentralisée, répartissent les Cac/s en fonction du nombre de Cac/s vivants. L'importance d'un Cac en fonction de sa consommation CPU n'est pas prise en compte dans ces deux stratégies.

- III - 2.2.4.1 Mesures

Chaque graphique reprend les différentes répartitions du code (structure globale dupliquée (code dupliqué), une structure par classe (classe), duplication des structures par classe (copie de classe)). Pour chacune d'elles, nous mesurons les temps pour les différentes représentations des objets. Les mesures sont effectuées pour les deux applications. Les noeuds sont toujours plus nombreux que les modules

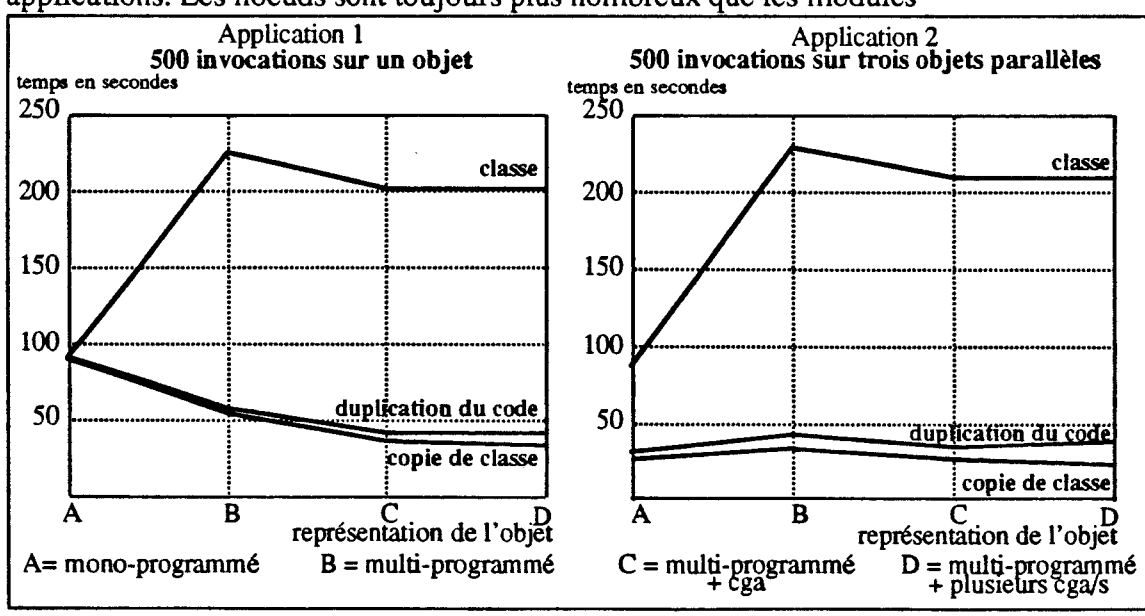


figure 2.34 *Mesures des deux applications*

Lorsqu'il n'y a pas de parallélisme dans l'application (cas de la version mono-programmée avec un objet) les performances des trois stratégies sont identiques.

La stratégie de répartition par classe est à l'évidence moins performante dans le contexte d'objets parallèles. Elle ne répartit pas les Cac/s d'un même objet sur plusieurs noeuds et le noeud contenant la classe de l'objet est surchargé.

La duplication du code et la copie des classes sont nettement plus performantes. Les légères différences entre les deux stratégies sont liées à la possibilité de répartir sélectivement les classes fortement sollicitées dans la stratégie de répartition par copie de classe. Cette remarque est fortement liée à la stratégie de répartition des Cac/s sur les noeuds qui prend en compte le nombre de Cac/s et non leur importance. Si la stratégie de répartition utilisait l'importance du Cac, les deux stratégies, duplication du code et la copie des classes donneraient des résultats similaires (voir supérieurs dans le cas de la duplication du code).

- III - 2.2.4.2 Evolution des travaux et conclusions

Les résultats montrent les bonnes performances de la répartition par copie de certaines structures de classe à l'exécution. Le grain de découpage en classe est suffisamment fin pour bien répartir le code sur les noeuds de la machine cible tout en étant assez gros pour être contrôlable par l'utilisateur.

D'autres solutions sont envisageables. Nous pouvons par exemple éclater la structure d'une classe à l'exécution. L'idée consisterait à séparer le code des méthodes de la classe. Nous pouvons ainsi: 1) regrouper certaines méthodes qui ne nécessitent pas d'être exécutées en parallèle, 2) dupliquer le code d'autres méthodes qui sont utilisées en parallèle. Les structures de classe sont alors éclatées sur les différents sites. L'exemple précédent avec 5 classes et quatre objets pourrait être matérialisé comme suit:

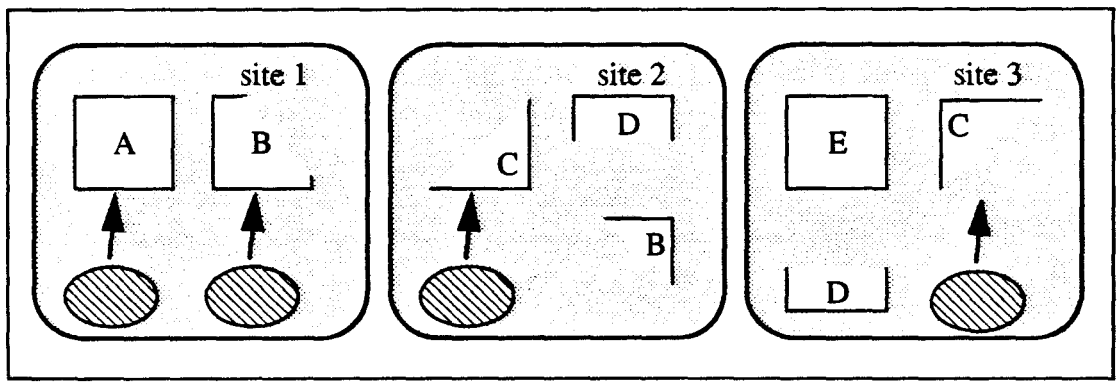


figure 2.35 *Eclatement des classes*

Nous avons donc diminué le grain de répartition du code sur les noeuds pour coller au mieux au parallélisme de l'application. Cet éclatement ne peut être laissé à la seule charge du programmeur. Il nécessite l'utilisation et la création d'un outil de «mapping» spécialisé dans la répartition du code.

Conclusion

Nous avons défini plusieurs représentations distribuées d'objets actifs multi-programmés au dessus du modèle Cac, permettant un réel parallélisme intra-objet. Les représentations reprennent le parallélisme de conception de l'objet en distinguant les fonctionnalités d'un objet que sont: le nommage de l'objet, l'exécution des méthodes et l'accès à ces attributs. Ces différentes représentations créent un parallélisme important dans l'objet qui compense la mise en place d'un modèle d'exécution réparti. Les objets de représentations différentes peuvent coexister à l'exécution. Nous avons montré que ces représentations donnent des performances optimales lorsqu'on utilise la représentation minimale pour un type d'objet donné.

L'éclatement du bloc des variables d'instance et l'exécution distante des méthodes créent le parallélisme dans le modèle. La problématique de ces objets est proche de la fragmentation introduite dans *Gothic* ou *Sos*. Pour faire évoluer nos objets actifs vers ces objets fragmentés, il faudrait associer une entité de traitement (tâche de fond) au CGA pour lui permettre d'exécuter certaines méthodes. Le CGA serait un CRO du type de celui des objets mono-programmés. Il serait alors un sous-objet mono-programmé dans un objet multi-programmé.

Nous avons étudié plusieurs répartitions du code (méthode d'instances) sur une machine sans mémoire commune. L'objectif était d'optimiser la gestion des structures des classes à l'exécution tout en permettant de réaliser les différentes représentations distribuées d'objets définies précédemment. Les meilleurs résultats sont obtenus en conservant une structure de classe à l'exécution. Dans cette solution, la notion de classe devient une entité de répartition. La souplesse de cette solution permet de reconstruire les autres répartitions proposées.

Ces représentations distribuées d'objets sont directement décrites en terme de Composants Actifs de Communication. Un prototype a été développé sur une machine parallèle (le MultiCluster II). Les classes et leurs instances sont à l'exécution, une construction de Cac/s et de modules de Cac.

Nous avons donc montré la faisabilité d'implanter un langage parallèle à objets actifs. Le continuation du projet nécessite le développement d'un compilateur et d'un langage pour simplifier la programmation des classes d'objets actifs. Il permettrait de *«typer»* les éléments de l'application et de cacher complètement le run-time Cac au programmeur. L'environnement de programmation devrait être associé à un outil de réutilisation des classes tel que l'héritage.

NB: Un exemple utilisant les différents types d'objet est décrit en Annexe -III-

Conclusion

Les travaux réalisés dans le cadre de cette thèse ont porté sur trois points: les Composants Actifs de Communication, les objets actifs distribués, l'héritage des contraintes de synchronisation dans un langage parallèle à objets[1]. Dans cette conclusion, nous résumerons nos propositions; puis nous décrirons les perspectives et orientations du projet.

Objectifs atteints

Les Composants Actifs de Communication

Nous avons développé un environnement pour la modélisation et la construction d'applications parallèles. Cet environnement est basé sur les Composants Actifs de Communication, entités structurantes de programmation. Les Cac/s sont des abstractions de parallélisme pour les machines parallèles. Ils se décrivent par une fonction comportementale et leur structure est proche de celle d'un acteur sérialisé. L'approche objet apporte une modélisation aisée des problèmes parallèles et une entité unique de programmation. Le modèle Cac engendre un important parallélisme implicite directement

1. L'héritage des contraintes de synchronisation est décrit en Annexe I

exploitable sur le type de machines visées.

Le modèle Cac définit les modules comme entités de partage de code; ceux-ci regroupent un ensemble de fonctions comportementales de Cac/s. Le module est aussi un composant logiciel et peut être réutilisable pour de nouvelles applications. Celles-ci sont construites en termes de modules. La phase de répartition des fonctions comportementales dans le module est indépendante de la phase de programmation. Le système permet la duplication des modules sur les noeuds lors de la phase de description du réseau de modules. Le mécanisme de duplication de module et la phase de répartition des fonctions comportementales forment l'outil de distribution du code sur les noeuds de la machine cible. L'outil programmable est suffisamment souple pour permettre de multiples solutions de répartition du code sur les noeuds. Le programmeur peut ainsi adapter son application aux contraintes matérielles de sa machine.

La structure du Cac est facilement réalisable sur le système d'exploitation des machines visées (il nécessite l'existence de la notion de processus, d'un gérant mémoire et d'un système de communication). Cette construction proche du système conserve de bonnes performances. Cependant le modèle Cac reste indépendant des mécanismes de base de la machine cible. Le modèle possède un grain fin de parallélisme qui permet au programmeur d'utiliser pleinement les potentiels de la machine cible.

Un prototype Cac a été réalisé sur un multicomputer. Il nous a servi de plate-forme aux développements de plusieurs applications, telles qu'une extension vers les acteurs et la construction d'un modèle d'exécution d'objets actifs multi-programmés.

Les Objets Actifs

A partir des Composants Actifs de Communication, nous avons construit une couche objet plus évoluée. Nous avons donc introduit plusieurs représentations distribuées d'objets actifs multi-programmés permettant un parallélisme intra-objet important. Le parallélisme est créé par l'exécution concurrente des méthodes dans l'objet et par la représentation distribuée des variables d'instance. Les représentations reprennent ainsi le parallélisme de l'objet en distinguant ses différentes fonctionnalités.

Chaque objet multi-programmé est matérialisé par un ensemble de Cac/s à l'exécution. Les Cac/s apportent un parallélisme implicite dans la structure de l'objet et permettent la création dynamique d'activités intra-objet lors du traitement des requêtes à l'objet. L'objet peut ainsi être éclaté sur plusieurs noeuds en fonction des types de Cac/s qu'il contient.

Nous proposons plusieurs types de représentations des objets actifs ayant chacune un degré de parallélisme différent. Tous les objets ont une interface identique au mécanisme d'invocation de méthode. Elle permet la coexistence d'objets de différentes représentations au sein d'une même application. Le programmeur peut ainsi définir pour chaque objet le type de parallélisme.

Nous avons, à partir des modules de Cac/s, réalisé différentes répartitions du code des méthodes à l'exécution. Les méthodes d'instance sont alors réparties sur les modules qui accueilleront les Cac/s des objets. Le programmeur d'une application peut alors orienter la répartition du code des méthodes sur les noeuds. La souplesse du système permet des solutions de répartition les plus variées, telles que la duplication du code sur tous les noeuds ou la conservation d'une structure de classe à l'exécution qui regroupe l'ensemble du code des instances de la classe.

Plusieurs prototypes réalisant une couche objet au dessus des Cac/s ont été développés. Ils ont montré la faisabilité des différentes représentations et surtout la possible coexistence d'objets de différentes représentations.

Nous avons eu une approche ascendante de l'éclatement des objets. Le phénomène d'éclatement est directement lié à la représentation de l'objet et au mécanisme d'invocation de méthodes construits au dessus des Cac/s. Contrairement aux autres approches où la fragmentation est explicite, le modèle proposé permet en plus, une fragmentation implicite et dynamique au fur et à mesure des invocations de méthode.

Perspectives et problèmes restants

Les Composants Actifs de Communication

Le modèle Cac doit être accompagné d'un environnement évolué de programmation.

- 1) Il nécessite en premier lieu un langage de programmation de haut niveau de manipulation de composants. Il permettrait de cacher le run-time à l'utilisation et de «typer» les variables de l'application.

- 2) La répartition des fonctions comportementales des Cac/s dans les modules est actuellement effectuée manuellement. Elle devrait être gérée par un outil de «mapping» contrôlé partiellement par des directives issues du langage.

- 3) L'environnement de programmation doit être associé à un outil de «déverminage» de programme Cac. Une maquette de l'outil est en cours de développement au sein de l'équipe [Roos92].

Nous devons améliorer les prototypes existant sur les points suivants:

- perfectionner l'algorithme de répartition des Cac/s sur les différents modules; ceci afin de tenir compte de la consommation de chaque Cac de l'application.

- connecter le run-time à un ramasse miettes Cac. (l'outil est en cours de réalisation).

- porter le run-time sur d'autres systèmes.

Les objets actifs

Nous avons montré, au travers des différents prototypes, la faisabilité des représentations distribuées d'objets multi-programmés. Nous devons maintenant définir un environnement de programmation. Ce langage devra être compilé et basé sur un programmation incrémentale (par classe d'objet). La compilation produira un source Cac composé des fonctions comportementales et des modules. Il bénéficiera ainsi des expériences sur les prototypes Cac existants et réutilisera des outils Cac en cours de développement.

Un tel langage nécessite de fixer certaines contraintes tels que:

- le nombre de représentations d' objets (mono-programmé, multi-programmé et multi-programmé avec éclatement des variables d'instance ...).
- la répartition du code des classes sur les noeuds (elle est difficilement contrôlable manuellement dans le cas d'un grand nombre de types d'objet, un outil de «mapping» s'avère ici indispensable).

Une critique importante aux prototypes objet développés est l'absence d'objet passif dans le modèle; ceci mis à part les objets primitifs. L'équipe travaille sur un langage mixte comprenant des objets passifs et des fragments (entités actives qui peuvent être associées à des Cac/s).

L'avenir du projet se situe donc au niveau langage; nous avons construit un support pour la construction d'applications parallèles pour multicomputer, validé une représentation distribuée d'objets actifs, il nous reste maintenant à réaliser un environnement de programmation pour valoriser et exporter le projet Pvc.

Annexe - I -

Une fragmentation des objets par héritage

Cette Annexe présente une **fragmentation** des instances en fonction de l'héritage des classes du système Pvc [Courtrai91a]. Ce mécanisme est conçu de manière telle que la localité de l'architecture Pvc reste la plus grande possible. Il utilise les **liens d'héritage entre Pvc**. Cela signifie qu'à un graphe de classes (cas général en héritage multiple) correspond un ensemble de Pvc connectés par des liens d'héritage. En conséquence, la gestion de l'héritage (recherche dynamique de méthodes et de variables d'instance) est réalisée par la coopération des Pvc concernés. Cela introduit deux nouvelles notions.

- D'une part, il apparaît un phénomène de **fragmentation des instances**. Le fait que seules les variables d'instance déclarées dans une classe restent locales au Pvc correspondant, provoque la fragmentation.

- D'autre part, l'utilisation de l'héritage nécessite une stratégie de **redirection des requêtes à instances** pour accéder au bon Pvc parmi ceux intervenant dans l'héritage. Notre implantation de ces mécanismes, que nous avons réalisée, montre la faisabilité d'un tel mécanisme. De plus, la stratégie de redirection des requêtes est tout à fait compatible avec les très bonnes performances des communications dans un multicomputer.

- I - 1. La fragmentation des instances

Dans un langage à objets comme Smalltalk, l'instanciation est une allocation d'un bloc mémoire contenant toutes les variables d'instance définies dans la classe et ses sur-classes. Les objets du système sont reliés entre eux dans une mémoire commune. Et chaque objet possède un pointeur référençant sa classe d'instanciation.

Les fragments

Dans Pvc, les instances sont mises dans une mémoire locale au Pvc. Lors d'une instanciation dans une classe héritant d'autres classes, l'opération d'instanciation est répartie dans les classes du graphe d'héritage. Chaque classe alloue alors un espace mémoire correspondant au nombre des variables d'instance définies localement. L'instance est alors distribuée dans le graphe. Chaque bloc mémoire pour une instance contenant ses variables d'instance définies dans une même classe est appelé **fragment** (repris de la terminologie *Gothic* [Banâtre91]).

Chaque Pvc dispose d'une maquette d'instance représentative des variables d'instance définies dans ce Pvc. Ce moule définit le fragment type à allouer pour le Pvc courant lors d'une instanciation. Prenons l'exemple d'une demande de création d'instance pour une classe donnée. Le mécanisme d'instanciation alloue grâce à sa maquette représentative de la classe un bloc mémoire. Puis l'instanciation est propagée par des envois de messages aux sur-classes directes qui exécuteront le même traitement et propageront à leur tour l'instanciation.

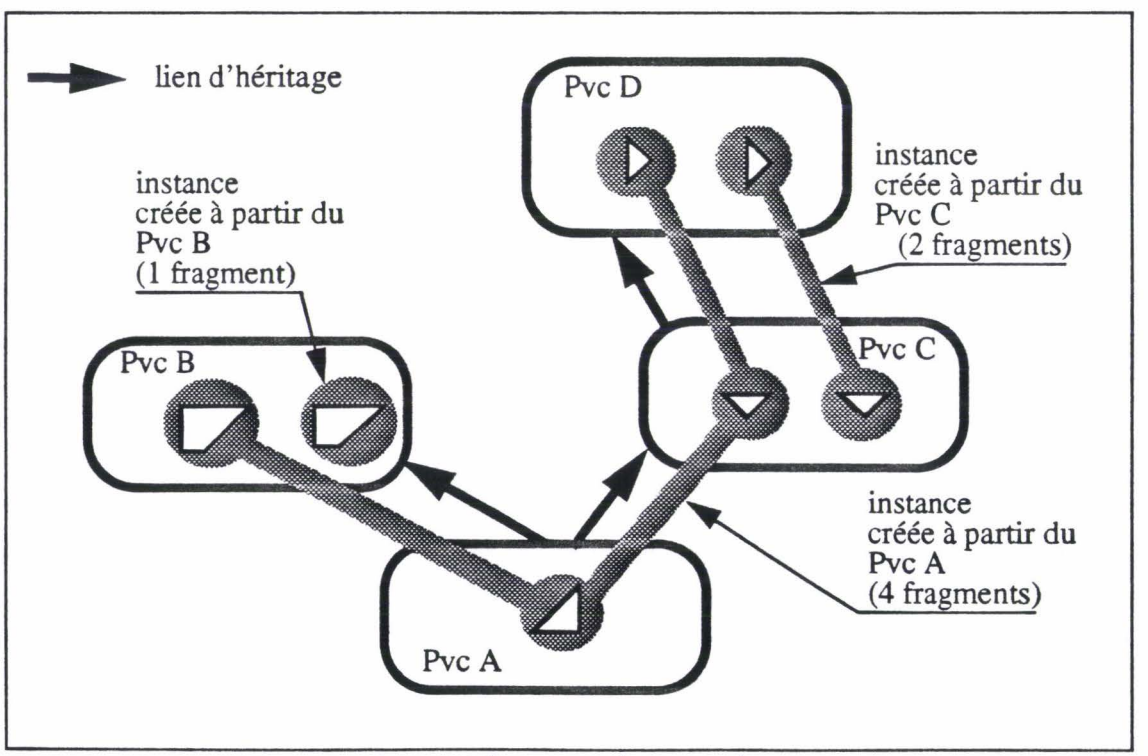


figure 1.0 *Instances fragmentées*

Ce phénomène de fragmentation a l'avantage de conserver la localité des traitements. A l'exécution d'une méthode sur une instance dans un Pvc du graphe d'héritage, il existe toujours un fragment de cette instance. Cette propriété est d'autant plus forte que les méthodes définies dans une classe utilisent principalement les variables d'instance définies localement dans cette classe. Pour accéder aux autres variables d'instances, le système doit faire coopérer les Pvc/s du graphe d'héritage. Les Pvc/s communiquent alors par envois de messages de requête suivis d'un retour d'une réponse (voir la section sur les accès aux variables d'instance).

L'instanciation distribuée

En l'absence d'héritage, la représentation d'une instance d'une classe occupe un morceau de la mémoire d'instance dans le Pvc correspondant à la classe. Ce morceau est alloué pendant le traitement d'une requête de création d'instance, sur la base d'une maquette d'instance représentative de la classe. Lorsque l'héritage est utilisé, on se trouve en présence d'un ensemble de Pvc reliés par des liens d'héritage. L'instanciation se trouve alors répartie de la manière suivante. Supposons qu'une requête de création d'instance arrive à un Pvc. Celui-ci alloue un morceau de sa mémoire en utilisant la maquette dont il dispose. Il **propage**, ensuite, la requête à tous les Pvc qu'il référence par des liens d'héritage. Chacun de ces Pvc alloue à son tour un morceau de sa mémoire en utilisant sa propre maquette puis propage la requête. Le mécanisme s'arrête lorsque chacun des Pvc intervenant dans l'héritage a traité une seule fois la requête de création. C'est le phénomène de **fragmentation:** une instance n'est plus localisée dans un seul Pvc, mais est répartie dans les Pvc impliqués par l'héritage. Chaque morceau, appelé **fragment**, ne contient que les variables d'instance déclarées dans la classe correspondant au Pvc associé.

Ce phénomène été exploité par d'autres implantations de langages parallèles à objets. Il présente un avantage important: la préservation de la propriété de localité. Par contre, il impose des contraintes de coopération entre les Pvc/s contenant les différents fragments qui constituent une instance. Reprenons ces deux points. Premièrement, la propriété de localité est conservée dans la plupart des cas, car on retrouve dans le même Pvc les variables d'instance et les méthodes déclarées dans une classe, comme dans le modèle sans héritage. Or les méthodes d'une classe accèdent essentiellement aux variables d'instance de cette classe, ce qui peut se faire ici localement. Deuxièmement, les accès aux variables d'instance et aux méthodes d'autres Pvc font l'objet d'une coopération entre ces Pvc. Cette coopération passe par la communication par messages, qui est d'un coût acceptable sur un multicomputer. Ce qui minimise le désavantage d'un accès à distance.

Précisons dans un premier temps quelques détails utiles à la compréhension de l'implantation effectuée.

Une instance est toujours nommée par un nom unique dans le système. Le nom d'une instance contient un nom de Pvc. Il s'agit ici du Pvc d'instanciation, c'est-à-dire le Pvc qui reçoit initialement la requête de création. Le nom d'instance contient aussi un nom local, qui repère un fragment d'instance dans la mémoire du Pvc d'instanciation. Le **même** nom d'instance permet de retrouver chaque fragment dans la mémoire de chaque Pvc concerné.

Chaque Pvc connaît un ensemble de noms de méthodes, correspondant aux traitements qui peuvent être demandés à ce Pvc. Cet ensemble de noms comprend les méthodes définies localement et aussi les méthodes héritées. Supposons que le Pvc A hérite du Pvc B. Alors pour une méthode définie dans B, il existe un nom local à A et un nom local à B. La possibilité d'héritage multiple de notre système ne permet pas d'assurer que de tels noms locaux, utilisés dans différents Pvc, aient des valeurs toujours identiques pour une même méthode.

Exemple de fragmentation

Pour illustrer le phénomène de fragmentation, nous prenons l'exemple de deux classes *Personne* et *Etudiant* liées par un lien de type «sous classe de» (tirée de la thèse de B. Carré [Carre89]).

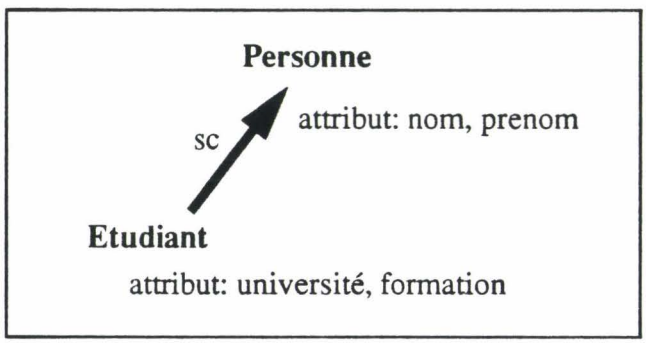


figure 1.1 *Les classes Personne et Etudiant*

Dans la proposition Pvc chaque classe est matérialisée par un Pvc. Les variables d'instance, nom et prénom restent localisés dans la classe *Personne*. Les objets de la classe *Etudiant* (x,y) sont fragmentés sur les deux classes Pvc/s.

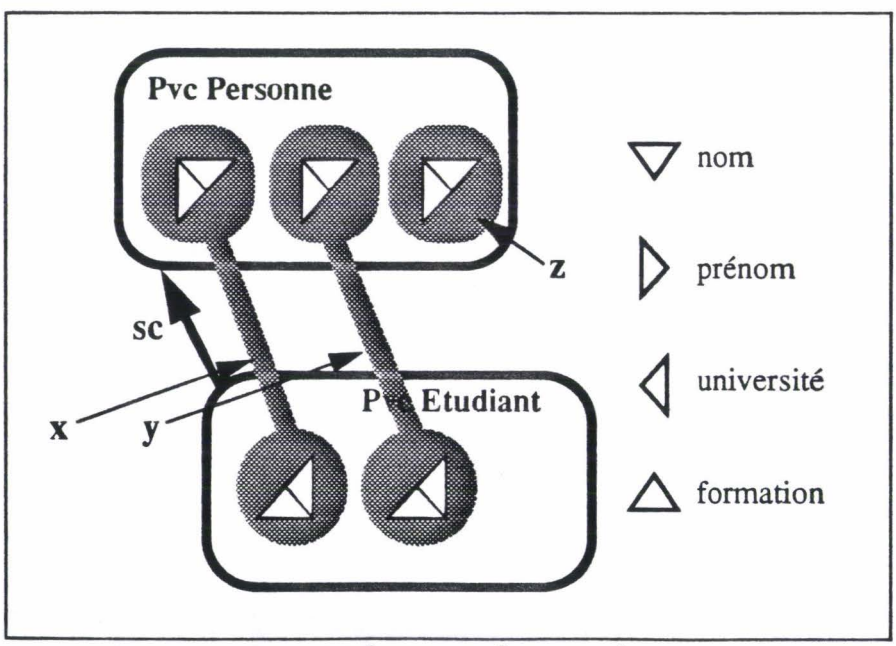


figure 1.2 *Instances fragmentées*

- I - 2. Mécanisme d'invocation de méthodes

Les méthodes applicables sur une instance sont distribuées sur le graphe d'héritage de la classe de déclaration de l'objet puisque le code des méthodes est localisé dans le Pvc représentant la classe. Le code d'une méthode n'est donc pas toujours dans le Pvc de l'objet cible, et peut être localisé dans un Pvc d'une sur-classe. La proposition Pvc consiste à rediriger les messages de requêtes vers le Pvc du code de la méthode invoquée. La méthode pourra alors s'exécuter sur le fragment de l'objet contenu dans le pvc de la méthode.

Le format des Requêtes à Instances

En l'absence d'héritage, un message de requête est directement envoyé au Pvc contenant l'instance. Ce Pvc traite localement la requête. Cela est rendu possible par les caractéristiques générales de notre système:

Une instance est désignée par un O.O.P. (Object-Oriented Pointer) qui contient le nom du Pvc qui renferme l'instance (voir la section «L'instanciation distribuée»), en notant que l'O.O.P. désigne tous les fragments, en présence d'héritage).

Le code d'un Pvc est obtenu par un mécanisme de compilation qui

- demande que les références aux instances soient typées[1],
- impose qu'un Pvc ne soit compilé qu'après les spécifications des Pvc/s qu'il utilise.

Intéressons-nous à une invocation de méthode $o.m$[2], dans laquelle o représente une référence à une instance et m un nom symbolique de méthode. L'invocation est traduite par l'envoi d'un message vers le Pvc contenant l'instance référencée par o; soit ***Pvc(o)*** le nom de ce Pvc qui est retrouvé à l'exécution dans l'O.O.P. implantant o. Le message indique une requête sous la forme d'un nom de méthode local au Pvc de déclaration de o, nom désignant m pour ce Pvc. On utilise ici le Pvc de déclaration de o (***D_Pvc(o)***), seul connu à la compilation.

Le message envoyé est finalement de la forme.

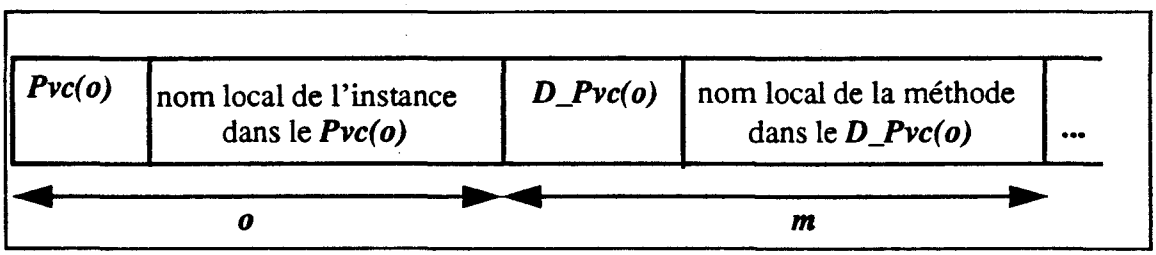


figure 2.0 *format des messages de requête*

1. Le cadre est ici celui de l'équation un type = une classe = un Pvc
2. Nous ne tenons pas compte, ici, du passage d'arguments à la méthode qui ne pose pas de problème particulier.

Le method-lookup calculé

En l'absence d'héritage, on a toujours

$$Pvc(o) = D\_Pvc(o) \quad (1)$$

et donc, le message peut être traité localement dans le Pvc destinataire.

En présence d'héritage, nous gardons le même format de message, facile à traiter à la compilation. Néanmoins, deux problèmes se posent, nous les examinerons par la suite.

Polymorphisme

Un premier problème provient de ce qui s'appelle le **polymorphisme des références aux objets**. Une référence déclarée du type T peut référencer un objet instance d'un sous-type de T. Cela se traduit ici par le fait que l'on n'a plus systématiquement l'égalité (1) ci-dessus mais plutôt

$$Pvc(o) \leq D\_Pvc(o)^1 \quad (2)$$

En conséquence, il se peut que le nom de méthode désignant ***m*** dans le message ne soit pas valide pour le Pvc qui reçoit le message. Ce problème est réglé entièrement par le Pvc destinataire de la manière suivante. L'objet utilise une fonction de traduction $\mathcal{T}$ qui transforme le nom de méthode reçu en un nom local significatif pour le Pvc destinataire (voir la figure 2.1 «fonction de translation T»). Cette fonction se fonde sur les liens d'héritage entre Pvc. Les liens d'héritage sont définis statiquement. La fonction peut donc être préparée pour chaque Pvc à la compilation.

$\forall i$ Pvc_i utilisant la fonction de translation suivante $\mathcal{T}_i$:
- $\forall j$ / $Pvc_i \leq Pvc_j$
- et $\forall k$, nom local de la méthode ***m*** pour le Pvc_j

$$\mathcal{T}_i(k,j) = k'$$

où ***k'*** est le nom de la méthode ***m*** pour le Pvc_i.

figure 2.1 *fonction de translation* $\mathcal{T}$

Héritage

On se trouve donc dans la situation où l'objet dispose toujours d'un nom local valide pour désigner la méthode demandée. A ce niveau, un deuxième problème se présente. Le nom local retourné par la fonction de traduction $\mathcal{T}$peut correspondre à une méthode héritée. Dans cette éventualité, le code de la méthode ne se trouve pas dans le Pvc courant. C'est le cas de **défaut de localité**. Le message doit être redirigé vers le Pvc contenant la méthode. Ce problème est résolu de manière identique au précédent en utilisant une fonction de localisation $\mathcal{L}$, qui fournit pour un nom local de méthode, le nom du Pvc qui contient la méthode (figure 2.2 «la fonction de localisation L»).

1. $\leq$ dénote la relation d'ordre partiel entre classes ou Pvc sur le graphe d'héritage. A<B indique que A hérite de B.

$\forall i$ Pvc_i utilisant la fonction de localisation $\mathcal{L}_i$:
$\forall k$, nom local de la méthode m pour le Pvc_i

$$\mathcal{L}_i(k) = j$$

où
- $Pvc_i \leq Pvc_j$
- et Pvc_j contient le code de la méthode m.

figure 2.2 *la fonction de localisation* $\mathcal{L}$

On notera que cette fonction $\mathcal{L}$ traite les cas de redéfinition et de renommage des méthodes. Ces opérations étant définies statiquement, elles sont prises en charge par le compilateur. La fonction $\mathcal{L}$ de chaque Pvc peut être préparée à la compilation.

Un objet Pvc ayant reçu le message détermine donc si le code de la méthode demandée peut être trouvé localement. Si oui, le traitement est lancé. Si non, le **message inchangé** est simplement **redirigé** vers le fragment dont le nom est retourné par la fonction $\mathcal{L}$. Ce dernier Pvc pourra appliquer localement la méthode. Le message peut rester inchangé car, rappelons que tous les fragments d'une instance ont le même nom pour tous les Pvc concernés par l'héritage.

Les self-invocations

Les self invocations sont des cas particuliers d'invocation de méthode. Une self-invocation est matérialisée par une message envoyé au fragment courant. Celui-ci recherche le Pvc de localisation de la méthode invoquée grâce à la fonction $\mathcal{L}$ et lui redirige la requête.

L'accès aux variables d'instance se traite comme des self-invocations de méthodes. Les self-invocations exigent des précautions particulières et se prêtent à quelques optimisations.

- I - 3. Réalisation des fonctions $\mathcal{T}()$ et $\mathcal{L}()$

L'héritage des classes Pvc est développé au dessus du modèle d'exécution Pvc. Nous détaillons dans cette partie la réalisation des deux fonctions $\mathcal{T}()$ **et** $\mathcal{L}()$ intervenant dans le mécanisme d'invocation de méthodes.

Les appels aux fonctions $\mathcal{T}()$ et $\mathcal{L}()$ sont réalisés par des accès à des tables locales. Deux tables définissant les caractéristiques définies localement ou héritées des sur-classes, sont introduites dans chaque Pvc. Une première table **Def_rout** mémorise toutes les méthodes accessibles par le Pvc dans l'ordre d'héritage des sur-classes et par branche (pour l'héritage multiple). Les méthodes d'une sur-classe sont insérées dans cette table avant les méthodes de ses sous-classes. Les méthodes définies localement sont ajoutées en fin de table. Une deuxième table **Base** mémorise les points d'entrée des Pvc/s sur-classes (position de la première méthode héritée de ce Pvc) dans la table Def_rout du Pvc courant.

-1 - Def_rout

Chaque ligne de la table *Def_rout* se décompose en deux champs:

- Pvc

Le champ Pvc mémorise le numéro du Pvc propriétaire de la méthode dans le graphe d'héritage: le Pvc de dernière définition de la méthode.

- fonction

La zone fonction est un point d'entrée sur le corps de la méthode.

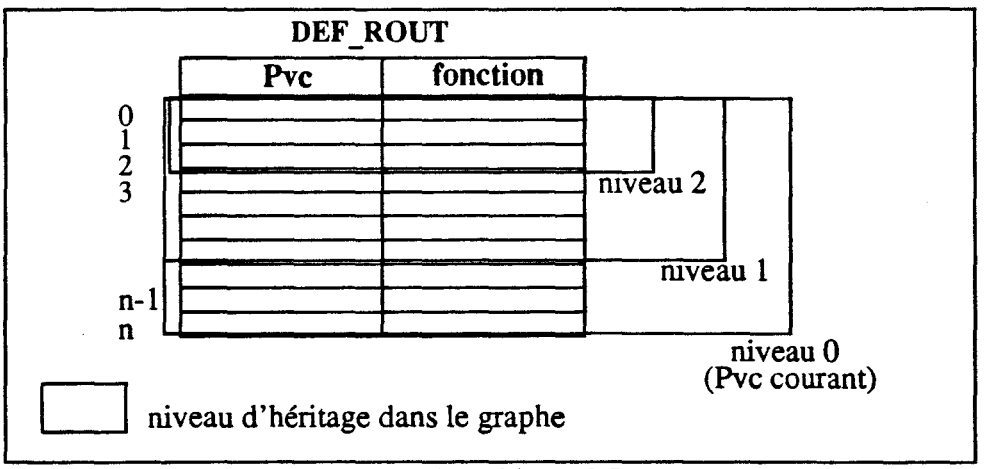


figure 3.0 *La table Def_Rout*

Dans ce shéma, les méthodes regroupées dans le niveau 1 sont héritées d'une sur-classe directe. Les méthodes de niveau 2 sont héritées d'une sur-classe par l'intermédiaire de la sur-classe directe (de niveau1). Les méthodes de cet ancêtre (niveau 2) sont accessibles par ces deux niveaux. Une méthode de niveau 2 appartient par héritage à une classe de niveau 1, mais le Pvc indiqué comme contenant la méthode est de niveau 2.

- 2 -Base

La table **Base** donne l'indice dans le tableau *Def_rout* où commence les définitions des méthodes héritées d'un Pvc donné. L'indice d'entrée dans cette table Base est le nom local de l'ancêtre dans le Pvc courant (nom donné dans la phase d'initialisation).

Les indices dans la table *Base* représentent pour le shéma précédent les niveaux d'héritage.

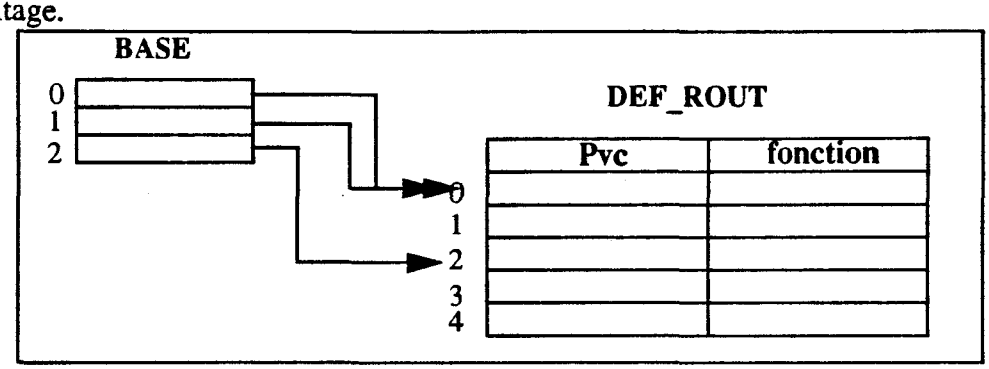


figure 3.1 *La table Base*

- 3- Calcul de $\mathcal{T}$() et $\mathcal{L}$()

Le résultat des fonctions $\mathcal{T}$() et $\mathcal{L}$() sont des calculs et des accès aux deux tables Def_rout et Base du Pvc courant.

La fonction $\mathcal{T}$ traduit un nom de méthode de la branche d'héritage en un nom local au Pvc courant. Le nom de la méthode passé en paramètre de la fonction se décompose en un nom de Pvc (**Pvc_d**) et un numéro local (**Num_d**) de la méthode dans ce Pvc.

$$\mathcal{T}(\textbf{Pvc\_d,Num\_d}) <=> \textbf{Base[Pvc\_d]} + \textbf{Num\_d}$$

Le calcul de la fonction $\mathcal{T}$() est une somme de type base + déplacement. La base est l'indice dans le tableau Def_rout de la première méthode héritée du Pvc passé en paramètre. Cet indice est la valeur contenue dans la case de la table Base indicée par le numéro de Pvc. Le déplacement est le nom local passé en paramètre de la méthode, numéro local de la méthode dans ce Pvc de base. Le calcul donne l'indice dans le tableau Def_rout correspondant à la méthode.

La fonction $\mathcal{L}$ localise d'après un nom local d'une méthode donnée pour le Pvc courant (dans l'exemple **Num_d'**), le Pvc propriétaire du code de cette méthode invoquée. Le nom de la méthode (**Num_d'**) doit être local au Pvc courant. Les autres noms des méthodes héritées des sur-classes doivent être traduits préalablement par la fonction $\mathcal{T}$()

$$\mathcal{L}(\textbf{Num\_d'}) <=> \textbf{Def\_rout[Num\_d'].Pvc}$$

Exemple complet de résolution de (o.m).

L'exemple suivant met en jeu trois Pvc/s nommés A,B et C. Le Pvc C hérite directement des deux autres Pvc/s A et B. Chaque Pvc définit deux méthodes: ma0, ma1 pour Pvc A, mb0, mb1 pour le Pvc B et mc0, mc1 pour le Pvc C. Les méthodes des deux Pvc/s parents A et B peuvent donc logiquement être invoquées sur les instances du Pvc C.

On veut ici résoudre l'invocation **o.mb0** avec o déclaré de type C, référence sur l'objet **o'** instance du Pvc C. A l'instanciation de l'objet de type C, le Pvc C lui a associé un nom **o'** qui l'identifie.

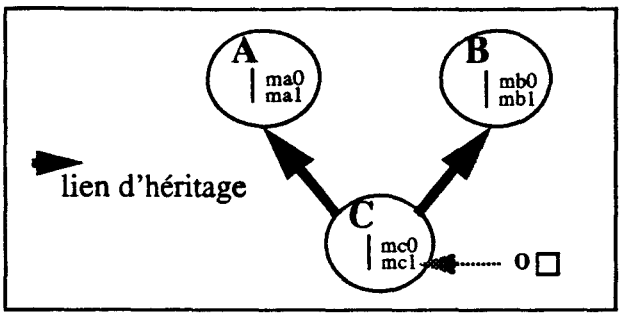


figure 3.2 *Lien d' héritage des trois Pvc/s*

Mise en place des tables Def_rout et Base.

Après la compilation des deux Pvc A et B, le système peut alors compiler le Pvc C. Avant l'exécution, dans une phase d'initialisation des trois Pvc A, B et C, le système génére les tables *Def_rout* et *Base* de chaque Pvc, grâce aux résultats des compilations précédentes.

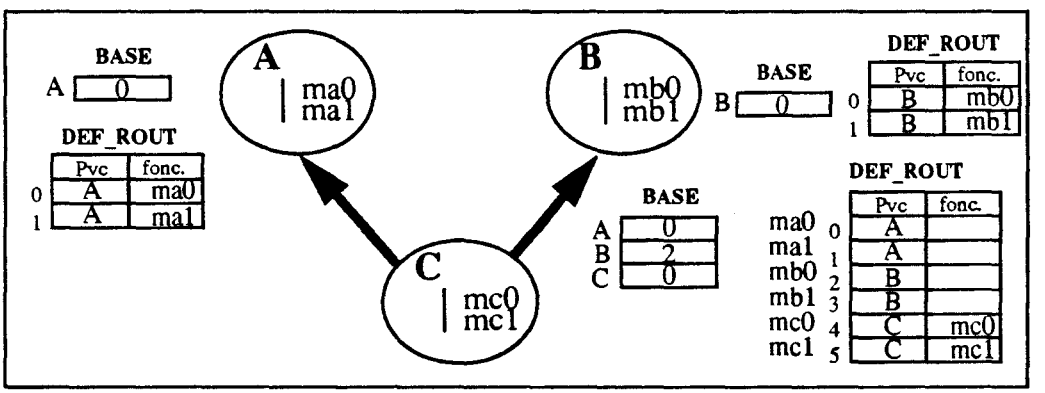


figure 3.3 *Les tables Def_rout et Base*

Dans l'ordre d'héritage (niveau langage), le PVC C hérite du Pvc A puis du PVC B. Dans la table *Def_rout* du Pvc C les deux méthodes héritées du Pvc A sont donc placées avant celles héritées du Pvc B. La première méthode héritée du Pvc B est placée en numéro 2 et cet indice est mémorisé dans la table *Base* du Pvc C. Les deux méthodes locales au Pvc C sont placées en fin de table *Def_rout.* Lorsque le champs fonction de la table Def_rout existe, le code de la méthode correspondante est local au Pvc. Toutes les méthode héritées sont renommées localement: 0,1 pour ma0, ma1, 2, 3 pour mb0, mb1 et 4,5 pour la méthode mc0,mc1. La méthode ma0 du Pvc A et mb0 du Pvc B ont toutes deux dans leurs Pvc/s respectifs le même nom. Le conflit de nom lors de l'héritage dans la Pvc C est résolu par la nouvelle numérotation: 0 pour ma0 et 2 pour mb0.

résolution de o.m

L'appel de la méthode **m** sur l'instance **o** est traduit par l'envoi du message:

(<C,o'> <B,0>)

L'object **o** est référencé par **<C,o'>** : C Pvc d'instanciation de l'objet **o** et **o'** nom de

l'instance dans le Pvc C. La méthode **m** possède un nom symbolique dans le Pvc C <**B,0**>: B est le nom du Pvc de **m** connu à la compilation (type statique). C'est forcèment un Pvc d'un rang au moins égal au Pvc de déclaration de l'objet **o**. L'indice **0** est le nom local de **m** dans Pvc B, numéro d'ordre dans les déclarations des méthodes (**0** première méthode de B).

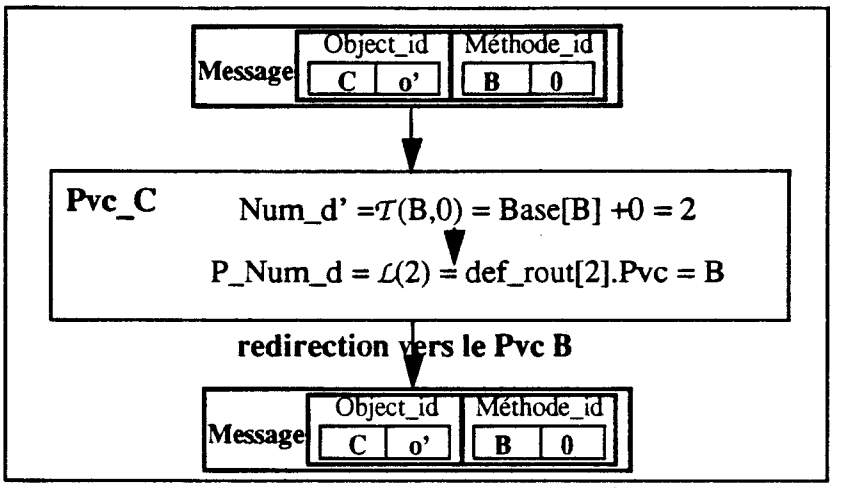


figure 3.4 *Le traitement de o.mb0*

Dans le traitement de **o.m**, la fonction $\mathcal{L}(2)$ retourne le nom du Pvc B comme Pvc propriétaire de la méthode. Le message est alors redirigé par le Pvc courant vers ce Pvc B. Le Pvc B traite alors ce message de la même façon. Mais le calcul $\mathcal{L}(0)$ donne le Pvc B, nom du Pvc courant et provoque le lancement local de la méthode **m** sur l'objet **o**.

La redéfinition de méthode

Les redéfinitions sont traitées avec les invocations des méthodes. La fonction $\mathcal{L}()$ donne pour un nom de méthode donnée, le Pvc de sa dernière définition par rapport au niveau courant du graphe d'héritage. Le traitement des redéfinitions est donc intégré dans les invocations de méthodes. Une redéfinition d'une méthode dans un Pvc est une simple modification de la zone Pvc de la table *Def_rout* du Pvc qui redéfinit cette méthode. La valeur du champ Pvc d'une méthode redéfinie localement est positionnée au nom du Pvc courant. L'appel de cette méthode provoque le lancement direct de la méthode redéfinie.

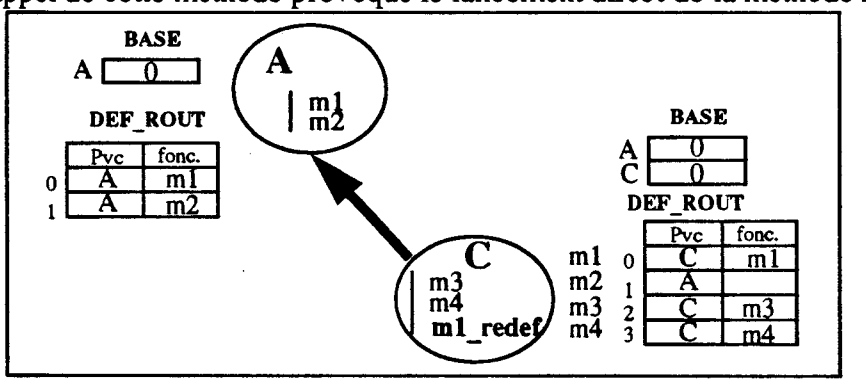


figure 3.5 *La redéfinition*

Dans cet exemple, un Pvc nommé C hérite du Pvc A. La méthode m1 du Pvc A est redéfinie dans la Classe C. La ligne de la table *Def_rout* du Pvc C, correspondant à la méthode m1 héritée et redéfinie. Son champ Pvc est rempli avec le nom du Pvc courant Pvc C au lieu du Pvc A. La fonction $\mathcal{L}()$ du Pvc C donnera pour cette méthode m1 le Pvc C comme Pvc Propriétaire.

La renommage de méthode

Les renommages de méthode sont résolus par le modèle en dédoublant les lignes des méthodes renommées dans la table *Def_rout*. Cela permet une éventuelle redéfinition de la même méthode. La duplication des informations est réalisée par une copie de la ligne de définition dans la table *Def_rout*.

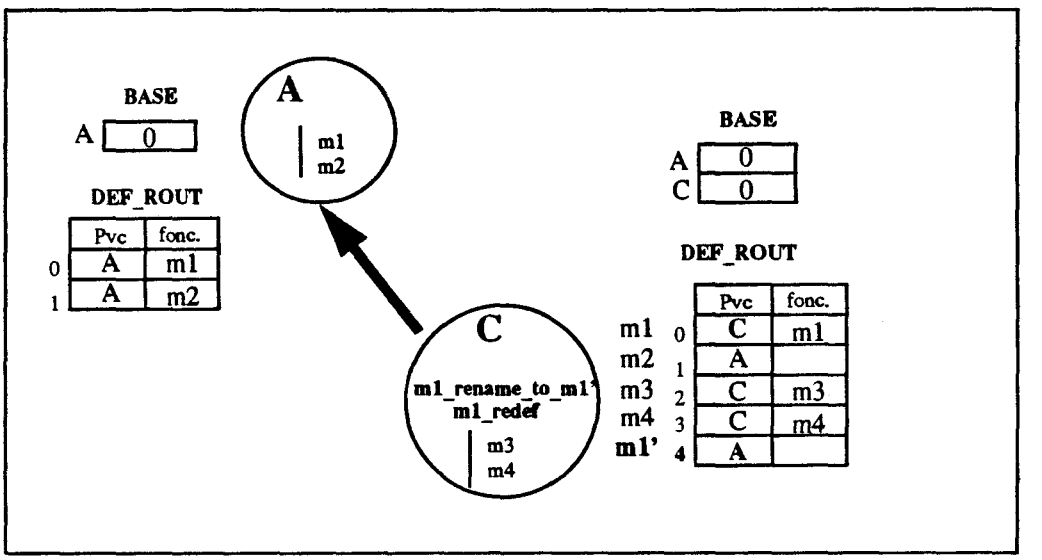


figure 3.6 *Le renommage des méthodes*

Ici, le Pvc C hérite du Pvc A. La méthode m1 de Pvc A est renommée puis redéfinie dans le Pvc C. Dans la table *Def_rout* la ligne correspondant est dédoublée. La première ligne (0) correspondant à la méthode m1 héritée redéfinit cette méthode. Le Pvc propriétaire de m1 est donc le Pvc C. La deuxième ligne (ligne 4) de cette méthode m1 renomme la méthode m1 en m1' associée au Pvc A. La méthode m1 du Pvc A reste donc accessible par le Pvc C, malgré sa redéfinition.

Les self-invocations

Les self invocations doivent être directement envoyées au Pvc de dernière définition de la méthode invoquée. Une redéfinition d'une méthode sous-classe peut modifier le choix du Pvc et celle ci n'est connue qu'à la compilation de la classe de redéfinition.

Le mécanisme d'héritage traite le problème dans une phase d'initialisation. Un booléen (**redef**) ajouté à chaque ligne du tableau *Def_rout*, mémorise dans la phase d'initialisation les éventuelles redéfinitions des méthodes dans les sous-classes par des envois de messages système. Lors d'une self-invocation, avant l'envoi normal du message de requête vers le Pvc d'instanciation, le run-time consulte la table *Def_rout*, et détermine si la méthode est redéfinie ou non. Puis, il envoie le message au Pvc d'instanciation si la méthode est redéfinie au dessous du Pvc courant. Sinon, il envoie le message sur le Pvc qui a défini la méthode grâce aux deux fonctions $\mathcal{L}()$ et $\mathcal{T}()$.

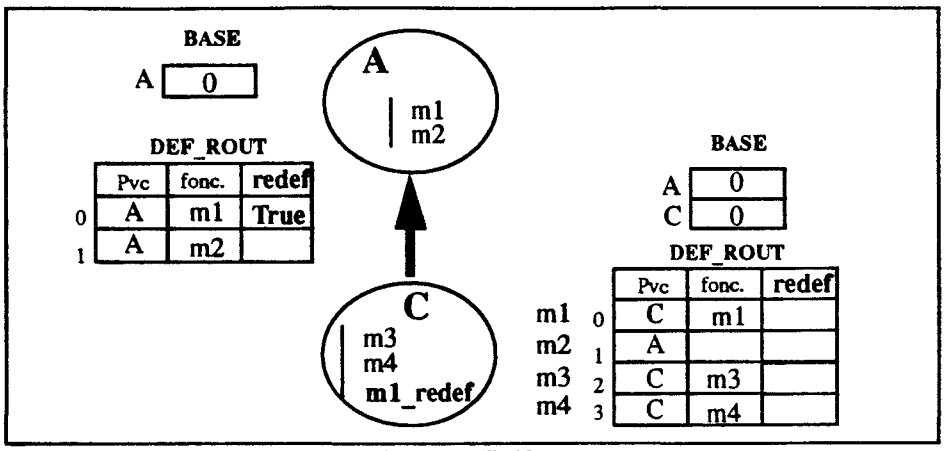


figure 3.7 *Cas des Self-invocations*

Le Pvc C hérite du Pvc A. Dans la phase d'initialisation du Pvc C qui a redéfini la méthode m1, un message signalant cette redéfinition est envoyé au Pvc A. La réception de ce message par le Pvc A positionne la zone *redef* à «Vrai».

Le Pvc A connaît alors, la redéfinition de la méthode m1 dans le Pvc C. Selon le type dynamique de l'objet, deux cas se présentent: Si l'objet est instance de la classe C, le Pvc A redirige les messages de requête de la méthode m1 vers le Pvc C. Si l'objet est une instance du Pvc A et malgré la redéfinition de la méthode m1 dans le Pvc C, la méthode est exécutée dans le Pvc A.

Mais tout appel de m2 lancera son exécution dans le Pvc A quelque soit le type dynamique de l'objet puisque m2 n'est pas signalée comme redéfinie dans une sous-classe.

Conclusion

On a donc décrit un mécanisme de «method lookup» qui fonctionne en contexte réparti, indépendamment de la localisation des Pvc sur les noeuds du multicomputer. Il s'agit d'un lookup calculé réalisé dans le contexte d'un mécanisme d'héritage proche de celui d'Eiffel [Meyer88]. Il fonctionne en un temps borné d'une durée de l'ordre de la transmission de deux messages (la requête d'origine et la requête redirigée). La fragmentation des instances ne nuit pas à la localité des traitements malgré la répartition des méthodes et des fragments dans le graphe d'héritage. Dans la plupart des cas, l'exécution d'une méthode est locale au Pvc: elle ne manipule qu'un fragment d'instance présent dans le Pvc. Les cas de défaut de localité sont résolus de manière originale, sans employer d'objets-proxy, sans reposer sur une facilité de migration des objets, sans nécessiter une gestion de copies d'objets.

Le prototype réalisé montre que la fragmentation par l'héritage pose des problèmes d'efficacité. Cette fragmentation dirigée par l'héritage est liée à la programmation modulaire par classes et la méthodologie de développement des classes est souvent éloignée d'une fragmentation optimale. Le mécanisme doit donc être accompagné d'un outil de fragmentation plus efficace.

Annexe - II -

Les Primitives du run-time Cac

Cette Annexe présente l'interface de programmation des Cac/s du chapitre II. Cet ensemble de primitives constitue la bibliothèque de fonction C du run-time.

- II - 1. Les Cac/s et leur programmation

Les références sur les Composants Actifs de Communication sont de type ***Component.***

Les noms de comportements sont stockés dans un fichier **Behavior.h** commun à l'ensemble des applications. Les comportements sont indentifiés par un numéro unique (type **int**) contenu dans ce fichier.

Les paramètres d'une fonction comportementale d'un Cac sont construits sous la forme d'un tableau d'entiers. Le type entier étant compatible avec le type *Component*, les arguments peuvent être indifféremment une valeur entière ou une référence sur un Cac.

```
/*************************************************************************/
/*        fonctions de gestion des composants:                           */
/*                NewComponent()                                         */
/*                KillComponent()                                        */
/*                EndComponent()                                         */
/*************************************************************************/
```

```
/*****************************************************************/
/*                        NewComponent                           */
/*****************************************************************/

Component NewComponent(Component my, int beh, int * args)
/*      demande la création d'un composant de comportement beh par le composant de
box my. Les arguments d'appel de la fonction comportementale sont sous la forme d'un
tableau d'entiers */

/*****************************************************************/
/*                        KillComponent                          */
/*****************************************************************/

void KillComponent(Component my, Component c)
/*      envoi d'un message d'arrêt au composant c par le composant my */

/*****************************************************************/
/*                        EndComponent                           */
/*****************************************************************/

void EndComponent(Component my)
/*      fin d'un cac: libération mémoire et destruction de la boîte aux lettres */
```

- II - 2. Les primitives de communication

La structure prédéfinie des messages

```
/*                      structure des messages                   */
typedef struct {
 Component sender, recipient;      /*   envoyeur et destinataire   */
 int type;                         /*   type du message            */
 int function;                     /*   fonction demandée          */
 int request;                      /*   requête demandée           */
 int nb;                           /*   nombre d'arguments         */
 void *arg;                        /*   arguments                  */
} Mess, * Mess_Ptr ;
```

Les différents champs vont permettre de *typer* les messages.

Manipulations des messages

```
/*************************************************************************/
/*      fonctions diverses pour la manipulation de messages et d'arguments   */
/*                                                                       */
/*              ArgMess()                                                */
/*              ArgToArray()                                             */
/*              ArgMessToArray()                                         */
/*              NewMessage()                                             */
/*              FreeMessage()                                            */
/*              SetMessage()                                             */
/*              SendaMessage()                                           */
/*              GetMessage()                                             */
/*************************************************************************/
/*************************************************************************/
/*                          ArgMess                                      */
/*************************************************************************/

int * ArgMess(Mess_Ptr mes, int n)
/*      retourne un pointeur sur le n^ieme argument du message mes */

/*************************************************************************/
/*                          ArgToArray                                   */
/*************************************************************************/

int *ArgToArray(int n,...)
{
/*      construit et renvoie un tableau d'entiers à partir d'une liste d'entiers. Le premier
entier de la liste donne le nombre d'éléments de la liste. */

/*************************************************************************/
/*                          ArgMessToArray                               */
/*************************************************************************/

int *ArgMessToArray(Mess_Ptr mes)
/*      renvoie les arguments du message mes sous forme d'un tableau au même format
que celui renvoyé par la fonction précédente */

/*************************************************************************/
/*                          NewMessage                                   */
/*************************************************************************/

void NewMessage(Mess_Ptr *mes, int s)
/*      allocation mémoire d'un message contenant s arguments */
```

```
/*****************************************************************/
/*                          FreeMessage                          */
/*****************************************************************/

void FreeMessage(Mess_Ptr mes)
/*      libération mémoire d'un message */

/*****************************************************************/
/*                          SetMessage                           */
/*****************************************************************/

void SetMessage(Mess_Ptr mes, Component s, Component d, int t, int f, int r, int * arg)
/*      préparation d'un message; garnit les zones: envoyeur, destinataire, type du
message, function, nom de la requête ainsi que la liste des arguments (arg)*/

/*****************************************************************/
/*                          SendaMessage                         */
/*****************************************************************/

void SendaMessage(Component my, Component c, Mess_Ptr m)
/*      envoi du message pointe par m vers le composant c */

/*****************************************************************/
/*                          GetMessage                           */
/*****************************************************************/

void GetMessage(Component my, Component c, Mess_Ptr *m)
/*      prise du premier message de la box c (bloquant). Le message est alors accessible
par la variable m */
```

- II - 3. Les modules et leur environnement

```
/*****************************************************************/
/*      Initialisation gestion et terminaison des modules :      */
/*              InitModule()                                     */
/*              DiffuseName()                                    */
/*              InsertBehavior()                                 */
/*              EndModule()                                      */
/*              StopModules()                                    */
/*****************************************************************/
```

```
/*****************************************************************/
/*                          InitModule                           */
/*****************************************************************/

void InitModule(int argc, char **argv, int nb_f, int nb_b)
/*      initialise les structures internes du module ainsi que le nombre de fonctions, de
comportements */

/*****************************************************************/
/*                          DiffuseName                          */
/*****************************************************************/

void DiffuseName(void)
/*      diffuse à tous les autres modules, les noms des fonctions comportementales
du module */

/*****************************************************************/
/*                          InsertBehavior                       */
/*****************************************************************/

void InsertBehavior(VoidFnPtr f , int bev)
{
/*      ajoute un nouveau comportement de nom bev dans le module en l'associant à
une fonction C de nom f. Cette fonction InsertBehavior() ne peut être utilisée qu'à
l'initialisation du module */

/*****************************************************************/
/*                          EndModule                            */
/*****************************************************************/

void EndModule(void)
/*      fin d'un module (Attend la fin de l'application pour détruire les structures mises
en place) */

/*****************************************************************/
/*                          StopModules                          */
/*****************************************************************/

void StopModules(Component my)
/*      provoque l'arrêt de tous les modules (cette fonction ne doit être appelée que
dans le module racine (premier module lors du lancement dans le cdl de l'application))*/
```

Annexe - III -

Exemples de classes d'objets Actifs

Cette annexe présente deux maquettes de classes d'objets actifs correspondant aux objets décrits dans la partie III.2 du chapitre III. Dans une seconde phase, nous présenterons un exemple complet utilisant les différents types d'objet (simulation d'une station de lavage).

- III - 1. Maquettes de Classes

Nous présentons dans cette annexe, deux maquettes de classe d'objets. La programmation des classes d'objet s'effectue en reprenant l'ossature de ces maquettes. Nous détaillerons ici la maquette d'une classe d'objets mono-programmés et la maquette d'une classe d'objets multi-programmés avec plusieurs CGA/s, qui sont les deux extrêmes.

Maquette d'une classe d'objets actifs mono-programmés.

```
/*                    Maquette de la Classe XXX                    */
#include <Prim_Obj.h>
InitObject( attr, arg)
    {/*méthode d'initilisation des objets de la classe*/
    attr[1] = arg[...];        /*attr, tableau d'attributs*/
    ...                        /*arg, paramètres d'appel de NewObjet()*/
    attr[nb] = ...;
}
void m1(Objet my, int * attr, int *args...)
    {/*my objet courant, attr attributs, arg arguments de la méthodes*/
    /*déclaration des variables locales */
    Objet o;
    int res;
    /*création d'une instance de la classe YYY*/
    o = NewObject(YYY);
    /*Appel synchrone de m_o sur un objet o */
    res = Call(o,m_o,arg1 ...argn);
    /*self invocation*/
    m2(arg1..argn);
    /*accès au xième attribut*/
    res = attr[x];
    /*retour d'une réponse */
    ReplyValue(res);
}
void m2(Objet my, int * attr, int *args...)
    {
    /*corps de la méthode m2*/
}
InitClass()
    {/*initilisation de la Classe*/
    type_class = MONO;          /*type de la classe*/
    class = XXX;                /*nom de la classe*/
    InitMethode(2);             /*nombre de méthodes*/
    methode[0] = m1;            /*initialisation des méthodes*/
    methode[1] = m2;            /*avec leur fonction C*/
    nb_attr = nb;               /*nombre d'attributs*/
}
```

figure 1.0 *Classe d'objets mono-programmés*

Maquette d'une classe d'objets actifs multi-programmés avec plusieurs Cga/s

```
*                    Maquette de la Classe XXX                    */
/#include <Prim_Obj.h>
InitObject(cga,arg)
    {/*cga tableau des cga, arg arguments d'appel de NewObjet()*/
    /*accès à un attribut (le cga n'est pas connu)*/
    AccesAttr(cga,1,A_WRITE, ...);
    /*acces au nième attribut du aième cga en ecriture*/
    AccesCga(cga,n,A_WRITE, ...);
}
void m1(Objet my, Compoent * cga, int *args...)
    {/*my objet courant, cga tableau des cga/s, arg arguments de la méthode*/
    /*déclaration des variables locales */
    Objet o;
    int res;
    /*création d'une instance de la classe YYY*/
    o = NewObject(YYY);
    /*Appel synchrone de m_o sur un objet o */
    res = Call(o,m_o,arg1 ...argn);
    /*self invocation*/
    Call(my,m2,arg1 ...argn); /*self invocation*/
    /*accès au xieme attribut*/
    res = AccesAttr(cga,x,A_READ);
    /*retour d'une réponse */
    ReplyValue(res)
}
void m2(Objet my, Compoent * cga, int *args...)
    {
    /*corps de la méthode m2*/
}
InitClass()
    {
    type_class = MULTI_CGAS;
    class = XXX;
    InitMethode(2);
    methode[0] = m1;
    methode[1] = m2;
    nb_attr = nb_attr;
    InitCga(nb_cga,nb1..nbn); /*nombre de Cga/s
}
```

figure 1.1 *Classe d'objets multi-programmés avec plusieurs Cga/s*

- III - 2. Exemple

La station de lavage

Nous avons développé un exemple montrant la coexistence de plusieurs types de représentations d'objets actifs. Nous simulons une station comprenant deux postes de lavage de voiture. Les voitures se déplacent sur une route formée de tronçons jusqu'à la station. Certains tronçons (parking) peuvent contenir plusieurs voitures. Le contrôleur de la station distribue alors les voitures sur l'un des deux postes de lavage de la station. Une fois lavées, les voitures vont sur un parking final. L'application peut être schématisée comme suit.

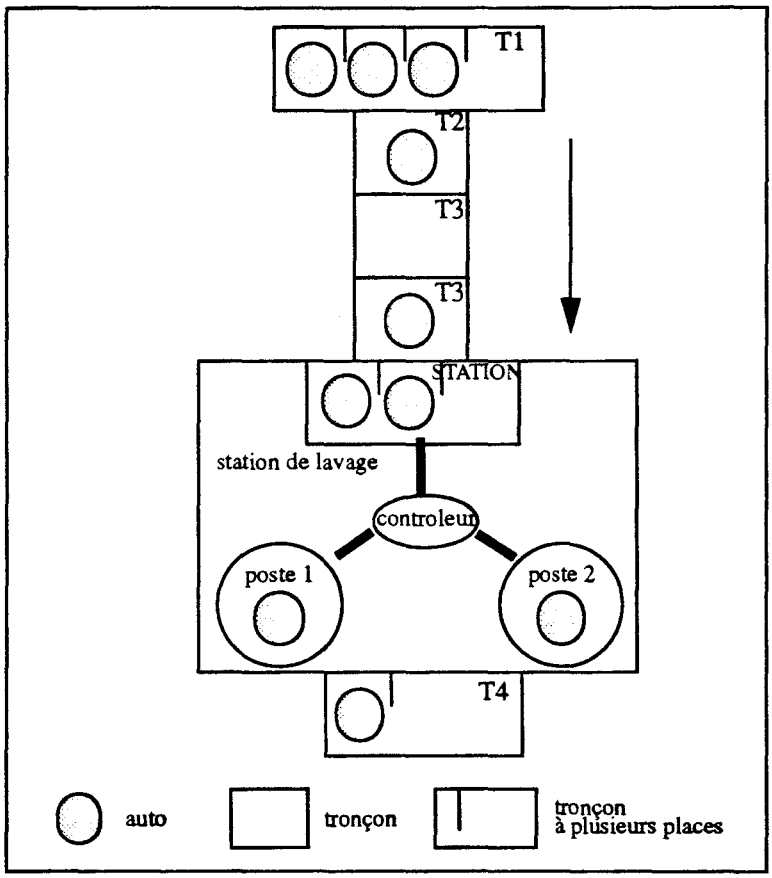


figure 1.2 *La station de lavage à plusieurs postes*

Réalisation

Voici les objets de l'application:

- Les voitures. Ce sont des objets mono-programmés. Elles sont au départ positionnées sur un tronçon (tronçon de départ T1). Elles répondent au seul message *AVANCE* qui provoque leur mise en route de la voiture. Les voitures ne s'arrêtent définitivement qu'à la fin du parcours (ici, le tronçon final T4). Chaque voiture est identifiée par un numéro.

- Les tronçons. Ce sont des objets mono-programmés. Ils représentent les parties du parcours à effectuer. Il existe deux types de tronçons: les tronçons pouvant contenir une seule voiture et les tronçons (parking) pouvant contenir un nombre illimité de voitures. Les tronçons sont numérotés et liés entre eux. Les tronçons reçoivent trois types de requête: les demandes d'entrée dans les tronçons, l'information de sortie d'une voiture du tronçon et les demandes pour connaître le tronçon suivant.

- La station de lavage. Elle est multi-programmée et fragmentée. Elle est constituée d'un contrôleur et de deux postes de travail qui travaillent en tâche de fond. La station est aussi considérée comme un tronçon à plusieurs places et répond donc aux mêmes requêtes. Le contrôleur extrait, en fonction des demandes venant des postes de travail, les voitures du tronçon (station) et les aiguille sur l'un des deux postes.

Voici les trois classes d'objet:

```
/*************************************************************************/
/*                              Class_Auto.c                             */
/*************************************************************************/

#include <Prim_Obj.h>

void InitObject(tobject my,int * attr,int *arg)
     {
     attr[1] = arg[1];              /*numéro de voiture*/
     attr[2] = arg[2];              /*nom du troncon courant*/
}

void avance(tobject my,int *attr,int *arg)
     {
     int res;
     do {
      res = Call(attr[2],SUIVANT);          /*demande le nom du troncon suivant*/
      if (res != VOID) {                    /*teste la fin du parcours*/
         Call(res,DEMANDE,attr[1],attr[2]);/*demande le troncon*/
         attr[2] = res;                     /*changement de tronçon*/
      }
     } while(res != VOID);
     printf("SORTIE VOITURE %d \n",attr[1]);
     ReplySelf();
}

void InitClass(void)
     {
     type_class = MONO;
     class = AUTO;
     InitMethode(1);
     methode[0] = avance;
     nb_attr = 2;
}
```

Les classes tronçon et station_de_lavage utilisent des boîtes aux lettres locales. La fonction *ExtractValues()* retire un message de la boîte et en extrait les valeurs.

```
/*************************************************************************/
/*                              Class_Troncon.c                          */
/*************************************************************************/
#include <Prim_Obj.h>

void InitObject(tobject my,int * attr,int *arg)
    {/* arg[1] nom du troncon, arg[2] troncon suivant */
    attr[1] = arg[1];                       /*nom*/
    attr[2] = arg[2];                       /*suivant*/
    attr[3] = VOID;                         /*libre*/
    attr[4] = NewBox(my);                   /*tampon*/
}
void suivant(tobject my,int *attr,int *arg)
    {
    ReplyValue(attr[2]);
}
void demande(tobject my,int *attr,int *arg)
    {/*arg[1] mandataire, arg[2] nom de la voiture, arg[3] tronçon précédent*/
    if (attr[3] == VOID) {                  /*teste si le tronçon est vide*/
        printf("ENTRER TRONC %d VOITURE %d \n",attr[1],arg[2]);
        attr[3] = arg[2];                   /*mise de la voiture dans le tronçon*/
        if (arg[3] != VOID)                 /*libération du tronçon précédent
            CallAsynchrone(arg[3],SORTIE,VOID,arg[2]); /*libère le précédent*/
        ReplySelf();
    } else {
        printf("EMPILER TRONC %d VOITURE %d \n",attr[1],arg[2]);
        CallAsynchrone(attr[4],VOID,arg[1],arg[2],arg[3]); /*envoi dans le tampon*/
    }
}
void sortie(tobject my,int *attr,int *arg)
    { /*arg[1] mandataire , arg[2] voiture*/
    int *voiture;
    if (NumberMessageInBox(my,attr[4]) > 0) {   /*teste existence de demande en attente*/
        voiture = ExtractValues(attr[4]);       /*extraction d'une demande*.
        printf("DEPILER VOITURE %d TRONCON %d \n",voiture[2],attr[1]);
        attr[3]=voiture[2];                     /*mise de la voiture dans le tronçon*/
        SendReply(my,voiture[1],VOID);          /*liberation de la demande*/
        if (voiture[1] != VOID)
            CallAsynchrone(voiture[3],SORTIE,VOID,voiture[2]);/*libere le precedent*/
     } else {                                   /*pas de demande en attente*/
        attr[3]=VOID;                           /*tronçon vide*/
    }
}
void InitClass(void)
    {
    type_class = MONO;
    class = TRONCON;
    InitMethode(3);
    methode[0] = suivant;
    methode[1] = demande;
    methode[2] = sortie;
    nb_attr = 4;
}
```

La station est multi-programmée et peut traiter deux voitures à la fois. Elle est éclatée en trois segments: les deux postes de lavage et la station en tant que tronçon. Cette répartition en trois parties montre deux intérêts à l'éclatement des variables d'instance: 1) créer un réel parallélisme des activités dans l'objet (les deux postes). 2) représenter l'objet sous deux angles différents (la station et un tronçon).

```
/*****************************************************************************/
/*                                  Station de lavage                        */
/*****************************************************************************/

#include <Prim_Obj.h>

/*                                  partie troncon                           */
void suivant(Component my,tobject obj, Component *cga,int *arg)
    {
    ReplyValue( Call(cga[3],ACCESS_ATTR,A_READ,2))
}

void demande(Component my,tobject obj, Component *cga,int *arg)
    {/*arg[1] mandataire, arg[2] la voiture */
    int box;
    printf("ENTRER STATION VOITURE %d \n",arg[2]);
    box = Call(cga[3],ACCESS_ATTR,A_READ,4);          /*demande l'adresse du controleur */
    CallA(box,VOID,arg[1],1,arg[2]);                  /*envoi de la voiture au controleur */
    if (arg[3] != VOID)                               /*liberation du troncon précedent*/
        CallA(arg[3],SORTIE,VOID,1,arg[2]);           /*libere le troncon precedent*/
}

void sortie(Component my,tobject obj, Component *cga,int *arg)
    {
}
/*                                  partie contrôleur                        */
void controleur(Component my, tobject obj, Component *cga, int *arg)
    {/*initilisation de l'objet Carwash*/
    Box b_interface,b_privee;
    int laveur,*demande;
    b_interface = NewBox(my);                         /*boîte entre le controleur et le troncon
    Call(cga[3],ACCESS_ATTR,A_WRITE,4,b_interface);
    b_privee = NewBox(my);                            /*boîte entre le controleur er les deux postes*/
    CallAsynchrone(obj,lance_poste,VOID,1,b_privee);/*lance le poste 1*/
    CallAsynchrone(obj,lance_poste,VOID,2,b_privee);/*lance le poste 2*/
    ReplySelf();                                      /*libère le mandataire objet root */
    while (1) {                                       /*mise en route du controleur*/
        demande = ExtractValues(my,b_interface);      /*extrait une voiture du troncon */
        laveur = ExtractValue(my,b_privee);           /*extrait une demande d'un des postes*/
        CallAsynchrone(laveur,LAVE,demande[1],demande[2]); /*envoie la voiture -> le poste */
    }
}
```

```
/*                                  poste de lavage                                  */
void lance_poste(Component my, tobject obj, Component *cga, int *arg)
    {/*arg[2] numero de cga ,arg[3] boîte aux lettres vers le contrôleur
    int *voiture,i;
    while(1) {
        CallAsynchrone(arg[3],DEMANDE_TRAVAIL,my,0); /*demande un traitement */
        voiture = ExtractValues(my,my);              /*mise en attente d'un travail */
        printf("LAVE VOITURE %d POSTE %d\n",voiture[2],arg[2]);
        for(i=1;i<500000;i++);                       /*boucle simulant un traitement de 1 s*/
        SendReply(my,voiture[1],voiture[2]);         /*libère la voiture */
        Call(cga[arg[2]],ACCESS_ATTR,A_INC,1);       /*inc le nombre de voitures traitées*/
    }
}

void compteur(Component my, tobject obj, Component *cga, int *arg)
    {
    int res1,res2;
    res1 = Call(cga[1],ACCESS_ATTR,A_READ,1);    /*demande du nbre de voitures traitée*/
    res2 = Call(cga[2],ACCESS_ATTR,A_READ,1);    /*demande du nbre de voitures traitées*/
    ReplyValue(res1+res2);                       /*retour la somme des deux nombres*/
}

/*                              Initialisation d'une station                         */
void InitObject(tobject my,int * cga,int *arg)
    {/* arg[2] tronçon suivant */
    /* poste 1 cga1*/
    Call(cga[1],ACCESS_ATTR,A_WRITE,1,0);        /*nombre de voitures traitées */
    /* poste 2 cga2*/
    Call(cga[2],ACCESS_ATTR,A_WRITE,1,0);        /*nombre de voitures traitées */
    /* partie troncon cga3 */
    Call(cga[3],ACCESS_ATTR,A_WRITE,1,0);        /*nom du troncon
    Call(cga[3],ACCESS_ATTR,A_WRITE,2,arg[2]);   /*troncon suivant */
    Call(cga[3],ACCESS_ATTR,A_WRITE,3,VOID);     /*libre*/
}   Call(cga[3],ACCESS_ATTR,A_WRITE,4,VOID);     /*tampon*/

void InitClass(void)
    {
    type_class = MULTI_CGAS;
    class = CAR_WASH;
    InitMethode(6);
    methode[0] = suivant;
    methode[1] = demande;
    methode[2] = sortie;
    methode[3] = controleur;
    methode[4] = lance_poste;
    methode[5] = compteur;
    nb_attr = 6;
    InitFrag(3,1,1,4); /*trois fragments dont 2 premiers ont un attriobut*/
}
```

Le *root* objet permet d'initialiser l'application. Il crée un objet *STATION_LAVAGE* et lance la méthode *CONTROLEUR*. Il crée aussi les différents tronçons qu'il chaîne entre eux en transférant, lors de la création, la référence du tronçon suivant. Il crée enfin les objets voitures et les lance. Il se met ensuite en attente de message venant de ces voitures (message envoyé à la fin du parcours de la voiture).

```
/*****************************************************************************/
/*                                  ROOT OBJET                               */
/*****************************************************************************/
#include <Prim_Obj.h>

void root(tobject my,int *attr,int *arg)
    {
    tobject     station,autom,tro;
    int         i,res,nb_voiture = 10,nb_troncon =5 ;
    Box         b;
    tro = NewObj(TRONCON,nb,VOID)                   /*dernier troncon */
    station = NewObj(STATION_LAVAGE,tro);           /*le carwash*/
    Call(station,CONTROLEUR);                       /*libéré par une réponse anticipée */
    tro = station;
    for (i=2; i <=nb_troncon; i ++){
        tro = NewObj(TRONCON,nb - i +1,tro);        /*création des troncons et chaînage*/
    }                                               /*tro premier troncon */
    b = NewBox(my);                                 /*box pour les appels asynchrones*/
    for (i=1; i <=nb_voiture; i ++){
        autom = NewObj(AUTO,i,tro);                 /*création d'une auto */
        CallAsynchrone(autom,AVANCE,b,i);           /*avance d'une auto */
    }
    for (i=1; i <=nb_voiture; i ++){
        res = ExtractValue(b);                      /*information de fin de parcours*/
    }
    res = CallAsynchrone(station,COMPTEUR);         /*nombre de voitures traitées*/
    printf("total de voitures traitees %d \n",res);
    ReplySelf();
}
void InitClass(void)
    {
    type_class = MONO;
    class = ROOT;
    InitMethode(1);
    methode[0] = root;
    nb_attr =0;
}
```

Pour illustrer cette application nous présentons une trace d'exécution. Nous avons volontairement créé un nombre faible de voitures.

NB: La configuration du parcours est celle décrite dans la figure (figure 1.2 «La station de lavage à plusieurs postes»)

```
/*Trace d'execution avec 2 voitures 4 troncons */
ENTRER TRONC 2 VOITURE 1
EMPILER TRONC 2 VOITURE 2
ENTRER TRONC 3 VOITURE 1
DEPILER VOITURE 2 TRONCON 2
EMPILER TRONC 3 VOITURE 2
ENTRER STATION VOITURE 1
DEPILER VOITURE 2 TRONCON 3
LAVE VOITURE 1 POSTE 1
ENTRER STATION VOITURE 2
LAVE VOITURE 2 POSTE 2
ENTRER TRONC 4 VOITURE 1
SORTIE VOITURE 1
ENTRER TRONC 4 VOITURE 2
SORTIE VOITURE 2
TOTAL VOITURES TRAITEES 2
```

Dans cette trace, on constate que les deux voitures sont lavées en parallèle.

Nous présentons maintenant quelques mesures sur cette application.

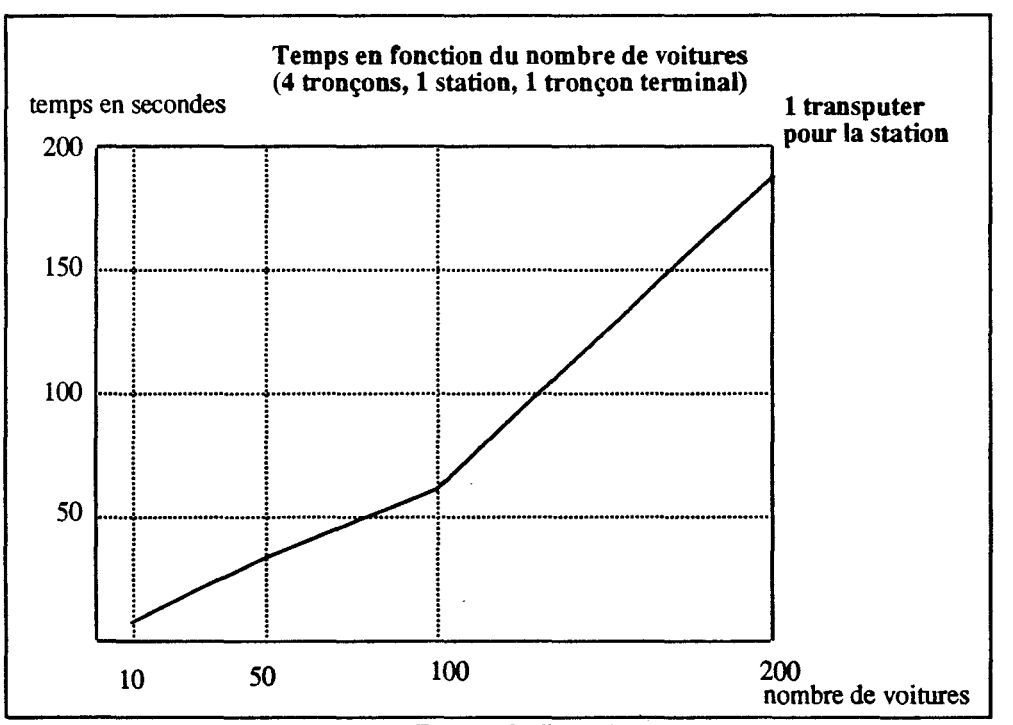


figure 1.3 *Temps de l'application*

Lorsque le nombre de voitures est important, les tronçons sérialisent fortement l'application. Il provoque un bouchon sur le premier tronçon et les voitures s'écoulent alors au fur et à mesures des lavages effectués.

Nous faisons maintenant varier le temps de traitement d'une voiture pour évaluer le parallélisme intra-objet de la station de lavage. De plus nous contrôlons la localisation des 3 Cac/s de l'objet station. Ils sont implantés sur 1 *transputer* puis sur 3 *transputers*.

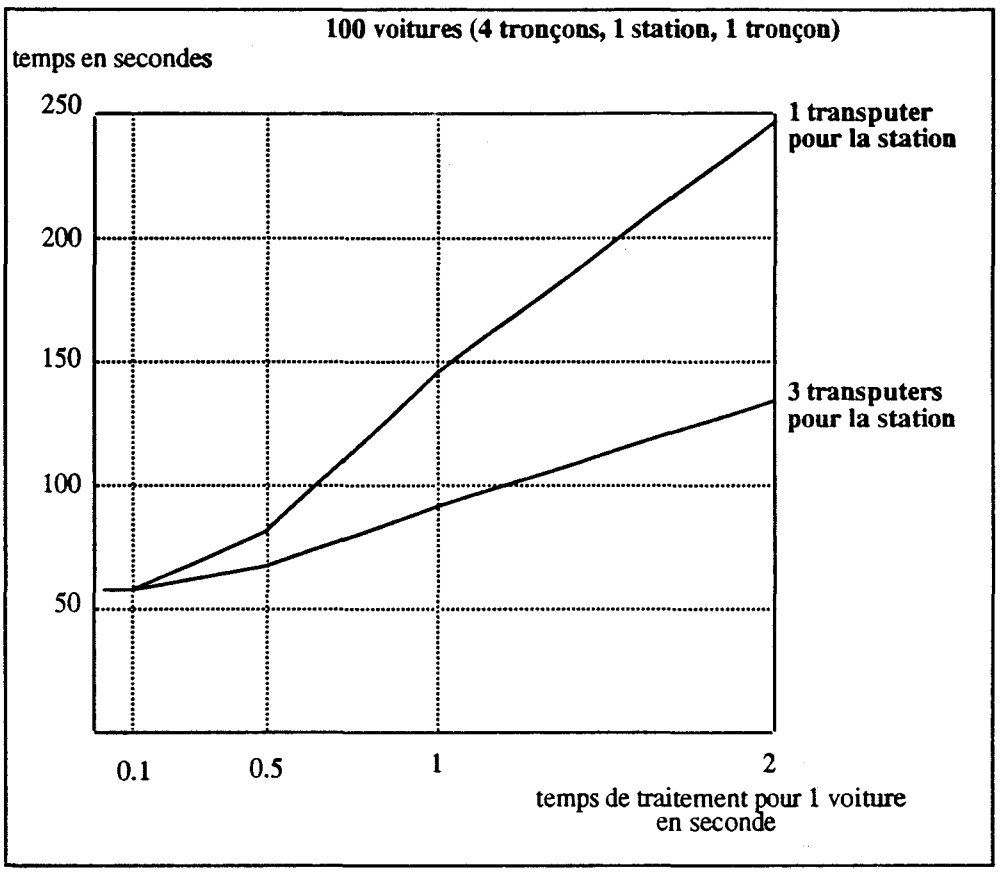


figure 1.4 *Mesure du parallélisme de l'application*

Lorsque le temps de traitement est trop faible, les voitures entrant dans la station sont directement lavées. L'avancée dans les tronçons freine l'application. A l'opposée, si le temps de traitement est trop important, les voitures bouchonnent à l'entrée de la station qui ralentit alors l'ensemble de l'application.

Le gain liée au parallélisme est important lorsque le temps de traitement s'accroît, les stations qui ralentissaient l'application fonctionnent maintenant en parallèle et le temps de passage dans les tronçons reste lui constant.Cette Annexe présente l'interface de programmation des Cac/s du chapitre II. Cet ensemble de primitives constitue la bibliothèque de fonction C du run-time.

Bibliographie

[Acceta86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, M. Young
Mach: A new kernel foundation for Unix development.
Proc. Summer Usenix Conference, july 1986, pp. 93-112

[Ada83]
Reference manual for the Ada programming language
ANSI / MIL-STD 1815 A

[Agha86] G. Agha
Actors. A Model of Concurrent Computation in Distributed Systems
The MIT Press, 1986, 144 Pages

[Agha87] G. Agha, C. Hewitt
Actors. A Conceptual Foundation for Concurrent Object-Oriented Programming
in "Research Directions in Object-Oriented Programming" edited by B. Shriver and P. Wegner, 1987

[America87] P. America
POOL-T : A Parallel Object-Oriented Language
[Yonezawa87], pp. 199-220

[America87b] P. America
Inheritance and Subtyping in a Parallel Object-Oriented Language
ECOOP'87, European Conf. on Object-Oriented Programming, Paris, France, June 1987, Special issue of BIGRE, no. 54, June 1987, pp. 281-289

[America90] P.America
A Parallel Object-Oriented Language with Inheritance and Subtyping
OOPSLA/ECOOP'90 Conf. on Object-Oriented Programming, European Conf. on Object-Oriented Programming, Ottawa, Canada, October 1990, pp 161-168

[Andrews83] G.R. Andrews, F.B. Schneider
Concepts and Notations for Concurrent Programming
Computing Surveys, Vol. 15, No. 1, March 1983, pp. 3-43

[Athas88] W.C. Athas, C.L. Seitz
Multicomputers: Message-Passing Concurrent Computers
Computer, August 1988, pp. 9-24

[Atkinson91] C. Atkinson, S. Goldsack, A. Di Maio, R. Bayan
Object-Oriented Concurrency and Distribution in DRAGOON
JOOP Journal of Object-Oriented Programming, Mars/April 1991, pp. 11-19

[Bahsoun91] J.P. Bahsoun, L. Feraud
Programming Synchronization with Reusability
EastEurOOPe'91 Proceedings, pp. 61-64, September 1991

[Bal87] H.E. Bal, A.S. Tanenbaum, M.F. Kaashoek
Orca: A Language *for Distributed Programming, Report IR-140, Dept. of mathematics and Computer Science, Vrije universiteit, Dec 1987.*

[Balter90] R. Balter, J. Bernadat. D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme
Design and Implementation of Guide, an object-oriented distributed system
Rapport techinque 1-90, Bull-IMAG, 16 novembre 1990

[Banâtre91] J.P. Banâtre, M. Banâtre
Les systèmes distribués - Expérience du Projet GOTHIC
InterEditions 1991

[Bennet87] J.K. Bennet
The design and implementation of Distributed Smalltalk
OOPSLA'87 Proc. of the Second ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Orlando, Florida, October 1987, Special Issue of SIGPLAN Notices, Vol. 22, No. 12, 1987, pp. 318-330

[Bennet90] J.K. Bennet
Experience With Distributed Smalltalk
Software - Practice and Experience, Vol. 20, No. 2, February 1990, pp. 157-180

[Bershad88] B.N. Bershad, E.D. Lazowska, H.M. Levy
PRESTO : A system for Object-oriented Parallel Programming
Software - Practice and Experience, Vol. 18, No. 8, August 1988, pp. 713-732

[Birrell83] A.D. Birrell, B.J. Nelson
Implementing Remote Procedure Calls
Xerox report CSL-83-7, October 1983

[Black86] A. Black, N. Hutchinson, E. Jul, H. Levy
Object Structure in the Emerald System
OOPSLA'86 Proc. of the ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon, October 1986, Special Issue of SIGPLAN Notices, Vol. 21, No. 11, 1986, pp. 78-86

[Black87] A. Black, N. Hutchinson, E. Jul, H. Levy, L. Carter
Distribution and Abstract Types in Emerald
IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, January 1987, pp. 65-76

[Briot87] J.P. Briot, A. Yonezawa
Inheritance and Synchronization in Concurrent OOP
ECOOP'87, European Conf. on Object-Oriented Programming, Paris, France, June 1987, Special issue of BIGRE, no. 54, June 1987, pp. 35-43

[Briot89a] J.P. Briot
Actalk: Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment
ECOOP'89, Proc. of the Third European Conf. on Object-Oriented Programming, Nottingham, G.B., July 1989, edited by Stephen Cook, British Computer Society Workshop Series, pp. 109-129.

[Briot89b] J.P. Briot
Des Objets aux Acteurs 1982-1989: 7 ans de reflexion
Habilitation à diriger des recherches mémoire de synthèse. LITP 89.68, September 1989.

[Boyer90] F. Boyer, J. Cayuela, P.Y. Chevalier, A. Freyssinet, D. Hagimont
Supporting an object-oriented distributed system: experience with UNIX, Mach and Chorus
BULL-IMAG Technical Report 7-90, December 1990

[Budd87] T. Budd
A Little Smalltalk
Addison-Wesley Publishing Compagny, 1987

[Burke90] E.J. Burke, T. P. Domane G.F. Johnson
An extensible distributed object management system, Edoms.
TOOLS 2, Technology of Object-Oriented Languages and Systems, Proc., Paris, 1990, pp. 183-197

[Campbell74] R. H. Campbell, A. N. Haberman
The specification of process synchronisation by path expression
Colloque sur les aspects théoriques et pratiques des systèmes d'exploitation, Paris 1974

[Capobianchi92] R. Capobianchi, R. Guerraoui, A. Lannusse, P. Roux
Active Objects on Parallel Machines: a case study
TOOLS Europe'92, Technology of Object-Oriented Languages Dortmund, Germany, March 1992, pp207-216.

[Caromel89] D. Caromel
Service, Asynchrony, and Wait-By-Necessity
JOOP, Journal of Object-Oriented Programming, November/ December 1989, pp. 12-22

[Caromel90] D. Caromel
Concurrency And Reusability: From Sequential To Parallel
JOOP, Journal of Object-Oriented Programming, September/ October 1990, pp. 34-42

[Caromel90b] D. Caromel
Concurrency : An Object-Oriented Approach
TOOLS 2, Technology of Object-Oriented Languages and Systems, Proc., Paris, 1990, pp. 183-197

[Caromel91] D. Caromel
Programmation parallèle asynchrone et impérative; étude et proposition. Une extension parallèle du langage objet Eiffel
Thèse de doctorat de l'université de Nancy I (F), Février 1991

[Carre90] B. Carre
Méthodologie orientée objet pour la représentation des connaissances, concepts de point de vue, de représentation multiple et évolutive d'objet
Thèse de doctorat de l'université de Lille I (F), Janvier 1989

[Chien91] A. A. Chien
Concurrent Aggregates: Using Multiple-Access Data Abstractions to manage complexity in Concurrent Programs
ACM OOPS Messager, 2(2), pp. 31-36, April 1991

[Ciampolini89] A. Ciampolini, A. Corradi, L. Leonardi
Parallel Object System Support on Transputer-Based Architecture
The Euromicro Journal, Microprocessing and Microprogramming, North-Holland, vol 27, no 1-5 , August 1989, pp. 336-346

[Colin90] J.F. Colin
Une interface Eiffel à la bibliothèque des processus légers Unix
Mémoire de DEA, université de Lille I (F), Octobre 1990

[Colin91] J.F. Colin, J.M. Geib
Eiffel Classes for Concurrent Programming
Proc. of TOOLS 4, fourth International Conference on Technology of Object-Oriented Languages and Systems, Paris 1991, Prentice-Hall 1991, pp. 23-35

[Comandos91]
A Guide the Comandos Platform D1-T2.2, Description of Commandos-2 Architecture
Technical report, Comandos Consortium, March 1991

[Cornafion81] F. Andre, J-S Bannio, C. Bétourné, J. Ferrié, D. Herman, C. Kaiser, S. Krakowiak, G. Mazaré, J. Mossière, X. Rousset de Pina, J. Seguin, J-P. Vergus
Systèmes informatiques répartis , concepts et techniques
Dunod informatique 1981

[Corradi87] A. Corradi, L. Leonardi
An Environment on Parallel Objects: PO
IEEE phoenix Conf. on Computers and Communications, Scotlsdale, AZ, Feb 1987, pp. 253 -257

[Corradi89] A. Corradi, L. Leonardi
PO: An Object Model To Express Parallelism
Proc. of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, SIGPLAN Notices, Vol. 24, No. 4, April 1989, pp. 152-154

[Corradi91] A. Corradi, L. Leonardi
PO constraints as tools to synchronize active objects
Journal of Object-Oriented Programming, vol 4 no 6, October 1991, pp. 41-53

[Courtrai91a] L. Courtrai, E. Delattre, J.M. Geib
À Server Based Architecture to Support Object-Oriented Languages on Multicomputers
Internal Report ERA 95, University of Lille 1 (F), April 1991

[Courtrai91b] L. Courtrai, J.M. Geib, J.F. Mehaut
Inheritance of Synchronization Constraints in the VCP system
EastEurOOPe'91 Proceedings, September 1991, pp. 57-60,

[Courtrai92a] L. Courtrai, J.F. Roos, J.M. Geib, J.F. Mehaut
The implementation of Actor Environment on Transputer under Helios: The Communicating Active Components
Poster, in Transputers'92, Besançon (F), May 1992

[Courtrai92b] L. Courtrai, J.F. Roos, J.M. Geib, J.F. Mehaut
Communicating Active Components: An environment for concurrent applications on parallel machines
EUROMICRO'92, Paris (F), September 1992

[Courtrai92c] L. Courtrai, J.F. Roos, J.M. Geib, J.F. Mehaut
An Environment to Support Fragmented Active Objects
I-WOOOS'92, International Workshop on Object-Oriented in Operating Systems, Paris (F), September 1992, published IEEE

[Courtrai92d] L. Courtrai, J.F. Roos, J.M. Geib, J.F. Mehaut
Les Composants Actifs de Communication : Manuel de programmation
Internal Report ERA 115, University of Lille 1 (F), June 1992

[Courtrai92e] L. Courtrai, J.F. Roos, C. Dumoulin, P. Merle, J.M. Geib, J.F. Mehaut
Communicating Active Components: Support for Parallel Object Languages on Distributed Architectures.
EUSUG'92, EUropean Sun User Group Conference 1992, Wiesbaden (D) November 1992

[Courtrai92f] L. Courtrai, J.F. Roos J.M. Geib, J.F. Mehaut
Support of an Actor Environment on Distributed Architectures
Sun User Group, Tenth Annual Conference, San Jose, California (USA) December 1992

[Dally87] W.J. Dally
A VLSI Architecture for Concurrent Data Structures
Kluger Academic Publishers, 1987

[Dally89] W.J. Dally, A.A. Chien
Object-Oriented Concurrent Programming in CST
Proc. of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, SIGPLAN Notices, Vol. 24, No. 4, April 1989, pp. 174-176

[Decouchant86] D. Decouchant
Design of a distributed object manager for the Smalltalk-80 system
OOPSLA'86 Proc. of the ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon, October 1986, Special Issue of SIGPLAN Notices, Vol. 21, No. 11, 1986, pp. 444-452

[Decouchant89] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, X. Rousset de Pina
A synchronisation mechanism for typed objects in a distributed system
Proc. of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, SIGPLAN Notices, Vol. 24, No. 4, April 1989, pp. 105-107

[Delattre89] E. Delattre
Les Langages Orientés Objets Paralléles - Etude bibliographique
Publication Interne N ERA 74, Octobre 89

[Delattre90] E. Delattre
Le Parallélisme dans les Langages à Objets Actifs: un «Survey»,
Publication Interne Lille I (F), December 90

[Dijkstra65] E. W. Dijkstra
Co-operating sequential Processes
Programming Languages Genuys editor, London 1965

[Di Santo91] M. Di Santo, G. Iannello
Implementing actor-based primitives on distributed memory architectures
ACM OOPS Messager, 2(2), pp. 44-50, April 1991

[Faust90] J.E. Faust, H.M. Levy
The Performance of an Object-Oriented Threads Package
OOPSLA/ECOOP'90 Proc., Conf. on Object-Oriented Programming : Systems, Languages and Applications, European Conf. on Object-Oriented Programming, Ottawa, Canada, October 1990, SIGPLAN Notices, Vol. 25, No. 10, October 1990, pp. 278-288

[Gautron92] P. Gautron, J.P. Briot, H. Saleh, S. Lemarié, L. Lescaudron
An Environment for Execution of Active Objects on Parallel Machines
Extented Abstract in EWPC'92 The European Workshops on Parallel Computing "from Theory to Sound Practice", Barcelona (Spain) Mach 1992, pp 376-379

[Gehani88] N. H. Gehani, W. D. Roome
Concurrent C++: Concurrent Programming with Class(es)
Software-Pratice and Experience, Vol 18(120), 1157-1177, December 1988, pp.1157-1177

[Geib92] J.M. Geib, L. Courtrai
Abstractions for Synchronization to Inherit Synchronization Constraints
ECOOP'92 W6, Workshop On Object Based Concurrency and Reuse, Utrecht, Netherlands, June 1992

[Genolini89] S.Genolini, A. Di Maio, C. Cardigno, S. Goldsack, C. Atkinson
Specifying Synchronisation Constraints in a Concurrent Object-Oriented Language
TOOLS'89 Technology of Object-Oriented Language and Systems, Paris, France, pp.371-378

[Glavitto91] J.L. Glavitto, C. Germain, J. Fowier
OAL: an Implementation of an Actor Language on a Massively Parallel Message-Passing Architecture
Distributed Memory Computing, Proc of 2nd European Conf. EDMCC2, Munich, Germany, pp. 347-360

[Goldberg83] A. Goldberg, D. Robson
SMALTALK-80. The language and its implementation
Addison-Wesley, 1983

[Gourhant90a] Y. Gourhant, M. Shapiro
FOG/C++: a Fragmented-Object Generator
Usenix C++ Workshop, San Francisco

[Gourhant90b] Y. Gourhant
Outils pour la programmation d'objets fragmentés
Thèse de doctorat de l'université de Paris-VI, Novembre1990

[Gransart91] C. Gransart
Introduction des objets actifs dans le langage Eiffel
Mémoire de DEA, université de Lille I, Septembre 1991

[Gransart92] C. Gransart, J.M. Geib
Reusability and Concurrency in Parallel Eiffel
X ième international Eiffel User Conference, Dortmund (Germany), April 1992

[Hemery91] F. Hemery, D. Lazure, E. Delattre, J.F. Méhaut
An Analysis of Communication and Multiprogramming in the helios Operationg System
Euromicro, September 1991, pp. 137-147

[Habert90] S. Habert, L. Mosseri, V. Abrossimov
Cool: Kernel Support for Object-Oriented Environments
OOPSLA/ECOOP'90 Conf. on Object-Oriented Programming, European Conf. on Object-Oriented Programming, Ottawa, Canada, October 1990, pp 269-277

[Hewitt73] C. Hewitt, P. Bisshop, R. Steiger
A universal Modular ACTOR Formalism for Artificial Intelligence
in Proc. of the 3rd IJCAI, Stanford California 1973. pp 235-245

[Hewitt77] C. Hewitt
Viewing control structures as patterns of passing messages
Journal of Artificial Intelligence 8-3, June 1977, pp. 323-364

[Hewitt79] C. Hewitt, R. Atkinson
Specification and Proof Techniques for Serializers
IEEE Transactions on Software Engieneering, Vol. SE-5, No 1, January 1979, pp 10-23

[Hoare75] C.A.R. Hoare
Monitors: An operating system structuring concept
Communications ACM vol 18,2, 1975 ,pp 95

[Hur87] J. H. Hur, K. Chon
Overveiw of a Parallel Object Oriented language CLIX
ECOOP'87. European Conf. on Object-Oriented Programming, Proc., Paris, France, Juin 1987, pp 315-323

[INMOS88]
Occam2 Reference Manual
Prentice-Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs,1988

[Jul88] E. Jul, H. Levy, N. Hutchinson, A. Black
Fine-Grained Mobility in the Emerald System
ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, pp. 109-133

[Kafura89] D.G. Kafura, K.H. Lee
Inheritance in Actor Based Concurrent Object-Oriented Languages
The Computer Journal, Vol. 32, No. 4, 1989, pp. 297-304

[Kafura90] D.G. Kafura, K.H.Lee
ACT++: Building a Concurrent C++ with Actors
JOOP Journal of Object-Oriented Programming, May/June 1990 , pp. 25-37

[Kafura91] D. Kafura, G. Lanvender
Recent Progress in Combining Actor-Based Concurrency with Object-Oriented Programming
ACM OOPS Messager, 2(2), pp. 55-58, April 1991.

[Krakowiak90] S. Krakowiak, M. Meysembourg, H. Nguyen Vam, M. Riveill, C. Roisin, X. Rousset de Pina
Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications.
JOOP Journal of Object-Oriented Programming, September/ October 1990 , pp. 11-21

[Lazure90] D. Lazure
Faisabilité de l'implémentation de Pvc sur Transputers et sous Hélios
Mémoire de D.E.A., Université de Lille, Septembre1990

[Lea91] R. Lea, James Weightman
Cool: an object support environment co-existing with Unix
AFUU Convention Unix '91.

[Lieberman81] H. Lieberman
Concurrent Object-Oriented Programming in Act1
Object-Oriented Programming, A. Yonezawa, M. Tokoro, Computer Systems Series, MIT Press, Cambrige MA, pp. 9-36

[Lieberman86] H. Lieberman
Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems
OOPSLA'86 Proc. of the ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon, October 1986, Special Issue of SIGPLAN Notices, Vol. 21, No. 11, 1986, pp. 214-222

[Liskov79] B. Liskov
Primitives for Distributed Computing
7 th A.C.M. Symp. on Operating System Principles, 1979, pp 33-42

[Lucco87] S.E. Lucco
Parallel programming in a Virtual Object Space
OOPSLA'87, Proc. of the second ACM conf. on Object-Oriented Programming Systems , *Languages, and Applications*, Orlando, Florida, October 1987, pp. 26-34

[Makpangou91] M. Makpangou, Y. Gourhant, M. Shapiro
BOAR: A library of fragmented object types for distributed applications
IWOOOS'91 Proc. of the International Workshop on Object-Orientation in Operating Systems, Paolo Alto, CA (USA)

[Markhoff90] B. Markhoff
Implémentation en Occam du paradigme Acteur sur un réseau de transputers,
La lettre du transputer et des calculateurs distribués, pp 17-27 juin1990

[Marques89] J. A. Marques, P. Guedes
Extending the Operating System to Support an Object-Oriented Environment
OOPSLA'89 Conf. Proc., Object-Oriented Programming Systems, Languages, and Applications, New Orleans, October 1989, Special issue of SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 113-122

[Masini90] G.Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre
Les langages à objets, langages de classes, de frames et d'acteurs.
InterEditions Paris 1990

[McAffer90] J. McAffer
Actra - An Industrial Strength Concurrent Object-Oriented Programming System
ACM OOPS Messager, 2(2), pp. 82-85, April 1991

[McCullough87] P.L. McCullough
Transparent forwarding : first steps
OOPSLA'87 Proc. of the Second ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Orlando, Florida, October 1987, Special Issue of SIGPLAN Notices, Vol. 22, No. 12, 1987, pp. 331-341

[Meyer88] B. Meyer
Object-oriented Software Construction
Prentice-Hall, 1988

[Moss87] J. Eliot B.Moss W.H. Kohler
Concurrency Features for Trellis/Owl Language
ECOOP'87. European Conf. on Object-Oriented Programming, Proc., Paris, France, Juin 1987, *pp 223-231*

[Mullender86] S.J. Mullender, A.S. Tanenbaum
The Design of a Capability-Based Operating System
The Computer Journal, Vol. 29, No. 4, 1986, pp. 289-299

[Mullender90] S.J. Mullender, G. Van Rossum, A. S. Tanenbaum, H. Van Staveren
Amoeba: A Distributed Operating System for the 1990s
IEEE Computer, vol 23, no 5, pp 44-53, May 1990

[Namyst92] R. Namyst
Les Abstractions pour la synchronisation dans les Langages à Objects Parallèles
Mémoire de D.E.S.S., Université de Lille, Juin 1992

[Neusius91] C. Neusius
Synchronizing Actions
Proc. of ECOOP'91, European Conference on Object-Oriented Programming, Geneva, Switzerland, July 1991, Pierre America Ed., Lecture Notes in Computer Science, Springer-Verlag 1991, pp. 118-132

[Nelson91] M.L. Nelson
Comcurrency & Object-Oriented programming
ACM SIGPLAN Notices, Vol 26, No 10, October 1991

[Nguyen90] H. Nguyen Van, M. Riveill, C. Roisin
Manuel du langage Guide (V1.5)
Rapport Technique 3-90, Bull-IMAG, Décembre 1990.

[Nierstrasz87] O.M. Nierstrasz
Active Objects in Hybrid
OOPSLA'87 Proc. of the Second ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Orlando, Florida, October 1987, Special Issue of SIGPLAN Notices, Vol. 22, No. 12, 1987, pp. 243-253

[Nierstrasz89] O.M. Nierstrasz
Two Models of Concurrent Objects
Proc. of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, SIGPLAN Notices, Vol. 24, No. 4, April 1989, pp. 174-176

[Parsytec90] Parsytec GmbH
Multicluster-2. Technical Documentation. Installation, expansion and maintenance manual
Rev. 1.1, May 1990, Juelicher straße 338, D-5100 Aachen

[Perihelion89] Perihelion Software
The HELIOS Operating System
1989, Prentice-Hall

[Ritchie74] D.M. Ritchie, K. Thompson
The UNIX Time-Sharing System
1974, Communications of the ACM, Volume 17

[Robert77] P. Robert, J.P. Verjus
Togard autonomous descriptions of synchronization modules
Proc. IFIP. Congress (B. Gilchrist, ed.), North-Holland, 1977, pp. 981-986

[Roos92] J.F. Roos, L. Courtrai, J.F. Mehaut
Réexécution de programmes parallèles
RenPar4. 4èmes Rencontres du Parallélisme, Université des Sciences et Technologies de Lille, Villeneuve d'Ascq, Mars 1992, pp 17-20

[Rozier88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser
The Chorus distributed operating system
Chorus Distributed Operating System, Chorus System, february 1989, pp. 305-370

[Shapiro89] M. Shapiro, Y. Gourhant, S. Habert, L. Misseri, M. Ruffin, C. Valot
SOS: An object-oriented operating system - assessment and perspectives
Computing Systems, 2(7), 1989, pp. 287-337

[Shapiro91] M. Shapiro, Y. Courhant, J.P. Le Marzul, M. Makpangou, M. Ruffin, C. Valot
Un bilan du système réparti à objet SOS
AFCET/ INTERFACE No 103/104 mai/juin 1991 pp. 46-53

[Schelvis88] M. Schelvis, E. Bledoeg
The Implementation of a Distributed Smalltalk
ECOOP'88. European Conf. on Object-Oriented Programming, Proc., Oslo, Norway, August 1988, Lectures Notes in Computer Science 322, Springer-Verlag, 1988, pp. 212-232

[Sun88] Sun Microsystems
System Services Overview. Chapter 6 : Lightweight Processes
1988, Part Number 800-1753-10

[Sun91] Sun Microsystems
C++
1991, Part Number 800-5174-10

[Stroustrup86] B. Stroustrup
The C++ Programming Language
Addison-Wesley Publishing Company 1986

[Takada90] T. Takada, A. Yonezawa
An Implementation of an Object-Oriented Concurrent Programming Language in Distributed Environments
in [Yonezawa90], pp. 133-155

[Tanenbaum85] A.S. Tanenbaum, R. Van Renesse
Distributed Operating Systems
Computing Surveys, Vol. 17, No. 4, December 1985, pp. 419-47

[Thomas89] D.A. Thomas, W.R. Lalonde, J. Duimovich, M. Wilson
Actra - A Multitasking/Multiprocessing Smalltalk
Proc. of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, SIGPLAN Notices, Vol. 24, No. 4, April 1989, pp. 87-90

[Tomlinson89] C. Tomlinson, V. Singh
Inheritance and Synchronisation with Enabled-Sets
OOPSLA'89 Conf. Proc., Object-Oriented Programming Systems, Languages, and Applications, New Orleans, October 1989, Special issue of SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 103-112

[T9000-91]
The T9000 transputer architecture
La Lettre du Transputer et des Calculateurs Distribués, Numéro hors série «Spécial T9000», Laboratoire d'Informatique de Besançon, Avril 91, pp 7-28

[Wegner87] P. Wegner
Dimensions of Object-Based Language Design
OOPSLA'87, Proc. of the second ACM conf. on Object-Oriented Programming Systems , *Languages, and Applications*, Orlando, Florida, October 1987, pp. 168-182

[Wegner90] P. Wegner
Concepts and Paradigms of Object-Oriented Programming
OOPS messsenger, A Quarterly Publication of the Special Interest Group on Programming Languages, ACM press, V 1, No 1 AUGUST 1990 pp8-87

[Yokote86] Y. Yokote, M. Tokoro
The Design and Implementation of ConcurrentSmalltalk
OOPSLA'86 Proc. of the ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon, October 1986, Special Issue of SIGPLAN Notices, Vol. 21, No. 11, 1986, pp. 331-339

[Yokote87] Y. Yokote, M. Tokoro
Experience and Evolution of ConcurrentSmalltalk
OOPSLA'87, Proc. of the second ACM conf. on Object-Oriented Programming Systems , Languages, and Applications, Orlando, Florida, October 1987

[Yonezawa86] A. Yonezawa, J. P. Briot, E. Shibayama
Object-Oriented Concurrent Programming in ABCL/1
OOPSLA'86 Proc. of the ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon, October 1986, Special Issue of SIGPLAN Notices, Vol. 21, No. 11, 1986, pp. 258-268

[Yonezawa86b] A. Yonezawa, H. Matsuda, E. Shibayama
An approach to Object Concurrent Programming. A language ABCL
Bigre + Globule, No 48, Pages 125-134

[Yonezawa87] A. Yonezawa, M. Tokoro
Object-Oriented Concurrent Programming
The MIT Press, 1987

[Yonezawa90] A. Yonezawa
ABCL, An Object-Oriented Concurrent System
The MIT Press, 1990