50376
1995
13

NUMERO D'ORDRE : 1471

USTL

ANNEE : 1995

LIFL 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

Christophe Gransart

BOX : Un Modèle et un Langage à Objets pour la Programmation Parallèle et Distribuée

Thèse soutenue le 5 janvier 1995, devant la commission d'examen :

Président :	J.L. DEKEYSER	Université de Lille 1
Directeur de Thèse :	J.M. GEIB	Université de Lille 1
Rapporteurs :	J.P. BRIOT	University of Tokyo & LITP
	M. RIVEILL	Université de Savoie
Examinateurs :	B. CARRE	Université de Lille 1
	J.F. MEHAUT	Université de Lille 1
	M. SHAPIRO	INRIA

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 20.43.47.24 Fax. 20.43.65.66

Remerciements

Je remercie les membres du jury :

- Monsieur Jean-Luc Dekeyser, professeur à l'Université de Lille I pour m'avoir fait l'honneur de présider ce jury.
- Monsieur Jean-Pierre Briot, professeur à l'Université Pierre et Marie Curie, d'avoir bien voulu être rapporteur de ce travail.
- Monsieur Michel Riveill, professeur à l'Université de Savoie, d'avoir accepté (depuis si longtemps) de rapporter cette thèse.
- Monsieur Bernard Carré, maître de conférences à l'Université de Lille I, d'être examinateur de cette thèse et d'apporter son *point de vue* sur ce travail.
- Monsieur Jean-Marc Geib, professeur à l'Université de Lille I, d'avoir accepté de diriger mes recherches, de m'avoir suivi et conseillé durant toutes ces années.
- Monsieur Jean-François Méhaut, maître de conférences à l'Université de Lille I, pour sa disponibilité et ses conseils éclairés tout au long de cette thèse.
- Monsieur Marc Shapiro, professeur à l'INRIA, d'avoir accepté d'examiner cette thèse.

Je tiens également à remercier Chrystel Grenot pour les quatre années de collaboration au projet ainsi que pour toutes nos discussions qui ont permis de travailler dans un climat d'amitié.

Je remercie les membres de l'équipe, Yves Denneulin, Cédric Dumoulin, Fred Hemery, Raymond Namyst, Jean-François Roos pour toutes nos discussions et leur aide à la réalisation de cette thèse et plus particulièrement Philippe Merle pour sa collaboration à la réalisation de la première version du compilateur.

Merci à mes relectrices, Chrystel Grenot & Isabelle Ryl pour leurs nombreuses remarques constructives et encouragements durant la rédaction de cette thèse. Merci également à Henri Glanc pour le support logistique.

Enfin, je remercie mes parents pour leur aide continue au fil du temps ; qu'ils en soient remerciés au travers de ce document.

Table des matières

Liste des figures

Introduction

Les architectures décentralisées sont issues de deux révolutions technologiques. Il s'agit d'une part de l'accroissement vertigineux des possibilités d'intégration dans la fabrication des composants et d'autre part de l'émergence d'une technologie de réseau fiable et utilisable à grande échelle. La première révolution, celle des microprocesseurs, a permis, à faible coût, la multiplicité d'unités élémentaires mais puissantes. La deuxième révolution, celle des réseaux, a permis, toujours à faible coût, de les relier. Les architectures décentralisées sont alors nées sous des formes aussi différentes que les réseaux mondiaux d'ordinateurs, les réseaux locaux ou les machines massivement parallèles.

Une architecture décentralisée peut être définie comme étant un ensemble de noeuds (processeur(s) + mémoire) reliés par un réseau de communication. Si aujourd'hui nous arrivons à assembler les composants nécessaires, l'exploitation de ces architectures reste difficile. Nous appelons Application Répartie une application qui exploite ce type d'architecture, c'est-à-dire qui utilise au mieux le réseau et les différents noeuds pour s'exécuter. Il semble évident que la construction d'applications réparties reste sans doute un défi plus important encore que la construction des machines elles-mêmes.

L'objectif de cette thèse est de faciliter la conception des applications réparties en proposant un modèle de programmation d'applications réparties à l'aide d'objets. Un langage nouveau appelé BOX implante ce modèle. L'exécution d'une application BOX utilise un support d'exécution spécifique qui peut être facilement porté sur les systèmes d'exploitation existants.

Notion d'application répartie

Une application répartie exploite un ensemble de sites. Un site est défini comme un ensemble de ressources matérielles composé d'un banc mémoire, d'au moins un processeur et d'une connection avec le réseau. Sur chaque site se trouvent un ou plusieurs processus munis de leurs données locales et ayant connaissance d'un sous-ensemble des autres processus. Les processus se connaissant peuvent communiquer et se synchroniser. Certains processus ont des fonctions de serveurs vis-à-vis d'autres qui sont leurs clients ; tous participent à l'élaboration de l'application.

Des processus peuvent être créés et détruits dynamiquement au cours de l'exécution. Le réseau de liens entre processus peut évoluer au cours du temps.

Pour écrire une application répartie, il faut au minimum disposer

- des primitives permettant de créer et détruire des processus à distance.
- des primitives de communication.
- d'un langage permettant de concevoir le programme de chaque processus.
- d'outils pour gérer l'ensemble.

Une application répartie s'appuie donc sur un logiciel de base appelé Système d'Exploitation Réparti (SER). Un SER peut être un système nouveau (c'est-à-dire conçu comme tel dès sa conception), comme par exemple Mach ou Chorus. Un SER peut aussi être une extension d'un système préexistant, l'Unix des stations de travail en est un exemple. On trouve actuellement des environnements au-dessus d'Unix (tel PVM) qui facilitent la conception d'applications réparties.

L'exploitation d'une architecture distribuée et de son SER se fait donc à l'aide d'outils et de langages qui doivent exploiter la répartition des ressources et le parallélisme physique de l'architecture.

Idéalement, la répartition des ressources doit être complètement cachée aux utilisateurs et aux programmeurs. Par exemple une opération sur une ressource distante doit être identique à une opération sur une ressource locale. Le concept de transparence ou encore masquage de la répartition peut être perçu selon plusieurs points de vue :

- l'accès à des fichiers locaux et distants en utilisant les mêmes opérations
- l'accès à des objets sans connaître leurs emplacements
- la duplication des objets pour obtenir de meilleures performances et de la tolérance aux fautes
- la migration des objets pour une plus grande efficacité
- la reconfiguration du système pour améliorer les performances lorsque les charges varient
- l'ajout de nouvelles ressources sans devoir modifier le système ou les applications

Les SER ne résolvent pas tous les problèmes. Si c'était le cas, un programme "classique", c'est-à-dire non prévu à l'origine pour fonctionner sur un système d'exploitation réparti, y fonctionnerait de manière optimale. Dans l'état actuel, les SER fournissent des outils permettant au programmeur de gérer explicitement la décentralisation et le parallélisme de l'architecture. Cet environnement est le plus souvent réservé au développeur expert car les outils fournis sont de bas niveau et difficiles à utiliser. De plus, devant la diversité des outils disponibles, le programmeur est confronté à un certain nombre de choix de réalisation. Il n'a pas à sa dispostion un modèle clair de programmation.

Cela implique la conception de modèles et de langages nouveaux spécialement conçus pour exploiter les SER. Ces langages doivent masquer les fonctionnalités du SER mais également permettre au programmeur d'influer sur le fonctionnement du SER. Notre proposition est d'utiliser un modèle à base d'objets allant dans ce sens.

Les Objets

Ces dernières années, les langages à objets ont connu un très grand essor, principalement dans le domaine de la programmation séquentielle.

Le modèle orienté objet traditionnel introduit un petit nombre de concepts. On peut le présenter simplement de la manière suivante:

- Le Modèle des Objets se fonde sur des objets créés dynamiquement à partir de classes, on dit qu'un objet est instance d'une classe.
- Une classe décrit la structure interne de ses instances ainsi que les méthodes applicables sur ses instances. L'appel d'une méthode d'un objet est un appel procédural exécuté dans un contexte limité à l'objet en question.
- Un objet référence d'autres objets. Il peut appeler les méthodes des objets qu'il référence.
- Une exécution ne comprend qu'un seul flot d'exécution correspondant à l'appel d'une méthode de création d'un objet racine.

Les intérêts de ce modèle sont connus:

- **modularité** : la structure et les méthodes d'accès sont regroupées dans une seule entité logique.
- **encapsulation** : l'utilisateur d'une classe n'a pas à connaître la structure interne ni l'implantation des méthodes.
- **réutilisabilité** : une classe n'est pas créée pour une application particulière mais bien au contraire peut être (ré)utilisée dans différentes applications.
- **typage** : une classe définit une implantation d'un type et des compilateurs peuvent exploiter ce fait pour vérifier ou optimiser le code.
- **sémantique** : l'accès aux objets utilise la sémantique classique de l'appel procédural, c'est une sémantique familière aux programmeurs, ce qui favorise leur productivité.

On s'intéresse à la conception d'applications réparties avec une approche objet. Il est ici naturel d'imaginer une distribution des objets sur les différents sites de l'architecture répartie. Les objets vont communiquer entre eux par le réseau et vont permettre de créer et de synchroniser les activités parallèles de l'application. L'objet est donc perçu comme:

- l'unité de conception des programmes.
- l'unité de répartition.
- l'interlocuteur pour les communications et la synchronisation.

L'extension du modèle objet à la concurrence et à la distribution laisse néanmoins une grande place au choix:

- On trouvera des modèles incluant la notion d'objet et simultanément celle de processus, alors que d'autres modèles unifient ces deux notions dans celle d'objet actif.

- On trouvera des modèles qui utilisent la communication par messages alors que d'autres utilisent l'appel de méthode à distance.

- On trouvera enfin des modèles qui parlent de synchronisation en termes d'objets partagés alors que d'autres utilisent des communications synchronisantes.

Les langage à objets pour applications réparties de la littérature se positionnent par rapport à ces choix. Cela donne différents modèles de programmation que nous classifions dans notre travail selon trois directions orthogonales : Modélisation, Communication, et Synchronisation.

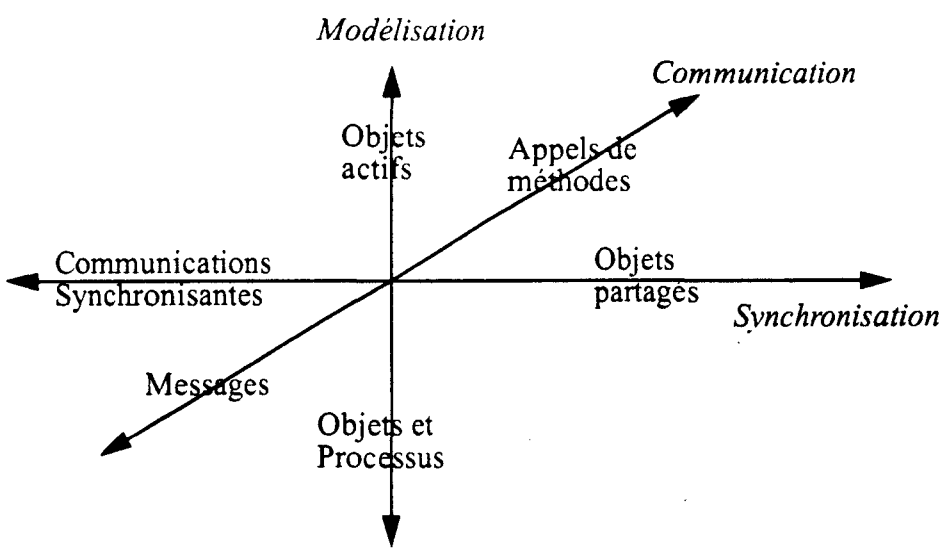


figure 1.1 *Classification*

Il nous paraît indispensable que dans la conception à objets d'une application répartie, on puisse utiliser le modèle le plus approprié. Et donc de ne plus être limité dans le choix du paradigme souhaité. Le modèle BOX que nous introduisons dans cette thèse tente de

rendre cette liberté au programmeur.

Le modèle BOX

Le concepteur d'applications séquentielles se doit de structurer son application en d'une part les données qu'il doit utiliser, et d'autre part les procédures qui manipulent ces données. L'approche objet l'aide dans ce travail en lui fournissant une entité unique de structuration (l'objet) qui regroupe logiquement ces deux aspects de la programmation.

Le concepteur d'applications réparties est devant un double problème de conception. Il doit représenter les données manipulées par son application (ainsi que les procédures qui les manipulent), mais il doit aussi représenter les activités qui vont coopérer à la réalisation de l'application.

Pour répondre à ce besoin, nous introduisons un modèle prenant en charge de manière uniforme ces deux aspects de la conception d'applications réparties. Ce modèle introduit explicitement des entités de représentation des données et des entités de représentation des activités. L'ensemble est vu "sur le mode objet" puisque, on le verra dans le langage, les données et les activités sont créées par une opération d'instanciation à partir de classes.

Le modèle BOX introduit donc deux types distincts d'entités : des entités qualifiées de passives et appelées simplement **Objets**, et des entités qualifiées d'actives et appelées **Fragments**. Les premières correspondent aux objets traditionnels des langages séquentiels à objets, mais elles pourront être ici utilisées en présence de flots d'exécution concurrents. Les secondes correspondent à une approche objet de la représentation des activités d'une application répartie. Le modèle introduit aussi des possibilités multiples de coopération entre Objets et Fragments.

Le modèle ne pose pas de prérequis sur l'architecture, celle-ci est supposée être composée 1) de noeuds (processeur+mémoire) pouvant accueillir des Objets et des Fragments et 2) d'un système de communication permettant de faire coopérer directement des entités situées sur des noeuds différents.

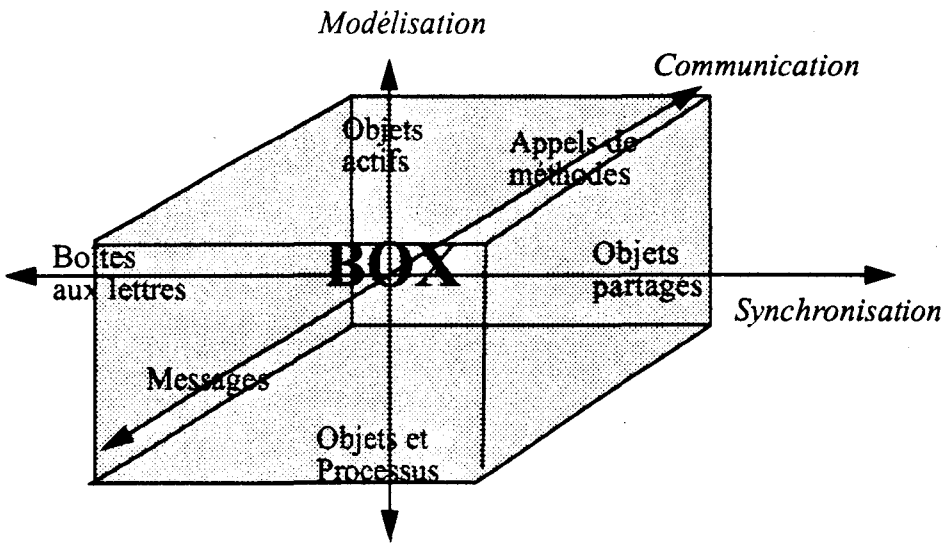


figure 1.2 *Le modèle BOX*

Le langage BOX est un langage à objets pour applications distribuées. Il inclut des caractéristiques nouvelles, spécifiques pour ce type d'applications, mais reste un langage pleinement objet. Le langage implante le modèle précédent. Le langage se focalise sur l'aspect programmation des classes en ne tenant pas compte, conformément au modèle, de la répartition de l'architecture sous-jacente. Une étape importante de la création d'une application répartie est cependant son installation sur la machine cible. Il s'agit de la phase de configuration. L'environnement BOX offre un certain nombre d'outils permettant au concepteur, à partir des classes produites dans la phase de programmation, de produire l'ensemble des exécutables nécessaires à l'exécution répartie, et cela en minimisant les retours sur la phase de programmation. On trouve :

- un compilateur de classes qui permet la compilation individuelle de chaque classe.
- un constructeur d'applications qui se charge de compléter les compilations individuelles par une édition de liens globale pour l'application.
- un langage de description de la répartition des classes sur des sites logiques et permettant donc la fabrication des différents exécutables.
- un lanceur d'applications qui installe les sites logiques sur les sites physiques de l'architecture et qui lance l'exécution répartie.

C'est l'aspect environnement BOX que nous décrivons dans cette thèse.

Un programme BOX s'exécute en utilisant un run-time spécifique appelé PVC. Les principaux objectifs de PVC sont :

- l'indépendance vis-à-vis des systèmes d'exploitation distribués
- la facilité d'adaptation à différentes architectures (multicomputers, réseaux de stations de travail, ...)
- la transparence de l'architecture
- une distribution statique du code et dynamique des entités

Notre système est donc structuré de la manière suivante :

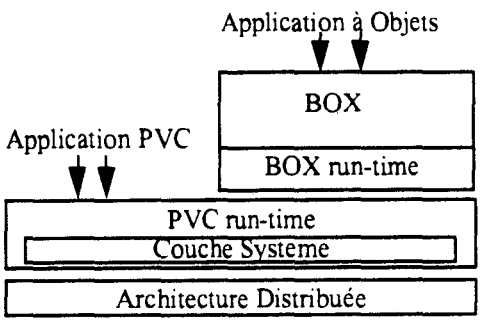


figure 1.3 *Le système BOX en couches*

Face au problème de conception des applications réparties, nous présentons donc dans cette thèse un système complet facilitant cette conception. Ce système BOX se fonde sur

un Modèle de Programmation, un Langage, un Environnement de Développement et un Support d'Exécution permettant l'utilisation des SER existants.

Plan de la thèse

Cette thèse comprend sept chapitres et une annexe :

- Le premier chapitre est consacré à l'étude de la notion d'application répartie. Nous présentons les systèmes d'exploitation répartis et les outils basés sur UNIX pour la réalisation d'applications réparties.
- Le second chapitre traite de l'adéquation du modèle objet avec ces applications. Nous présentons ensuite divers environnements orientés objets pour la réalisation d'applications distribuées.
- Dans le troisième chapitre, nous présentons le modèle BOX pour la programmation distribuée. Nous présentons les différents concepts que le programmeur peut manipuler pour arriver à concevoir son application.
- Le quatrième chapitre est consacré au langage. Nous présentons la syntaxe et la sémantique des différentes instructions.
- Le chapitre cinq présente l'environnement de développement BOX. Nous présentons les différents outils mis à la disposition du programmeur. Ce chapitre présente également une solution pour distribuer les composants logiciels sur une architecture répartie.
- Le sixième chapitre fournit des exemples d'applications réalisées à l'aide du langage. Nous présentons un embryon de méthodologie appelé Objets Actifs Complexes (OAC).
- Le dernier chapitre présente l'implémentation du langage sur le noyau PVC à l'aide des Composants Actifs de Communication (CAC).
- Une annexe présente l'interfaçage du langage BOX avec le langage C, et la bibliothèque d'entrée/sortie construite en utilisant cette interface.

Environnement

Les travaux de cette thèse ont été réalisés au sein de l'équipe PVC-BOX du Laboratoire d'Informatique Fondamentale de Lille. Eric Delattre et Jean-Marc Geib avaient lancé en 1989 PVC-BOX, projet qui avait pour objectif la conception et la réalisation d'un environnement de développement à objets pour architectures fortement distribuées. Dans le cadre de cet environnement, Jean-Marc Geib m'a proposé de définir et de réaliser un langage à objets pour l'exploitation de ces machines.

Le projet PVC-BOX comprend deux parties :

- PVC, un support système pour le développement d'applications distribuées. Luc Courtrai [Courtrai92] a proposé et implanté lors de sa thèse la notion de CAC (Composant Actif de Communication) comme outil de structuration et de programmation. Luc Courtrai a également montré que le modèle CAC est adapté

pour supporter l'exécution de programmes à objets parallèles.

- BOX, un langage de programmation à objets. La principale caractéristique de ce langage est qu'il permet d'aborder naturellement le paradigme objet et celui des acteurs communicants. Les CAC fournissent un support adapté pour la gestion distribuée des objets et des acteurs communicants de BOX.

Chapitre - I -

Outils pour les applications réparties

Dans ce chapitre, nous introduisons les notions générales concernant la conception d'applications réparties. Une application répartie s'exécute sur un ensemble de machines connectées par un réseau local. Une telle application s'appuie sur un logiciel de base appelé Système d'Exploitation Réparti (SER). Un SER peut être un système nouveau (c'est-à-dire conçu comme tel dès sa conception), nous en décrivons quelques-uns. Un SER peut aussi être une extension d'un système préexistant, l'Unix des stations de travail en est un exemple. Dans une deuxième partie nous décrivons donc l'environnement Unix pour la conception d'applications réparties. Nous décrivons ensuite quelques exemples d'environnements construits au-dessus d'un tel système et facilitant la conception de telles applications. Nous terminons ce chapitre sur les choix qui restent posés au concepteur.

I - 1 Définition d'une application répartie

Une application répartie exploite un ensemble de sites. Un site est défini comme un ensemble de ressources matérielles composé d'un banc mémoire, d'au moins un processeur et d'une connection avec le réseau. Sur chaque site se trouvent un ou

plusieurs processus munis de leurs données locales et ayant connaissance d'un sous-ensemble des autres processus. Les processus se connaissant peuvent communiquer et se synchroniser. Certains processus ont des fonctions de serveurs vis-à-vis d'autres qui sont leurs clients ; tous participent à l'élaboration de l'application.

Des processus peuvent être créés et détruits dynamiquement au cours de l'exécution. Le réseau de liens entre processus peut évoluer au cours du temps.

Pour écrire une application répartie, il faut au minimum disposer

- des primitives permettant de créer et détruire des processus à distance.
- des primitives de communication.
- d'un langage permettant de concevoir le programme de chaque processus.
- d'outils pour gérer l'ensemble.

Cela passe donc par un logiciel de base et un environnement prenant en charge ces problèmes. C'est la notion de Système d'Exploitation Réparti (SER)

I - 2 Introduction aux SERs

La notion de système d'exploitation réparti est apparue lorsqu'il s'est avéré possible de construire à coût réduit et avec une fiabilité suffisante des architectures réparties. Celles-ci sont composées d'un ensemble de machines à base de microprocesseurs reliées par un réseau de communication. On s'est rendu compte qu'interconnecter plusieurs machines de faible coût permettait d'obtenir une machine de la puissance d'un gros ordinateur pour un prix plus faible, voire plus puissante puisqu'on peut l'étendre à volonté en ajoutant des machines. Cette interconnection a été rendue possible également par l'amélioration des performances et de la fiabilité des réseaux.

Malheureusement le système d'exploitation d'une machine n'est pas suffisant pour exploiter un ensemble de machines interconnectées. Il fallait repenser les besoins en logiciel de base. Est apparu alors le concept de système d'exploitation réparti (SER).

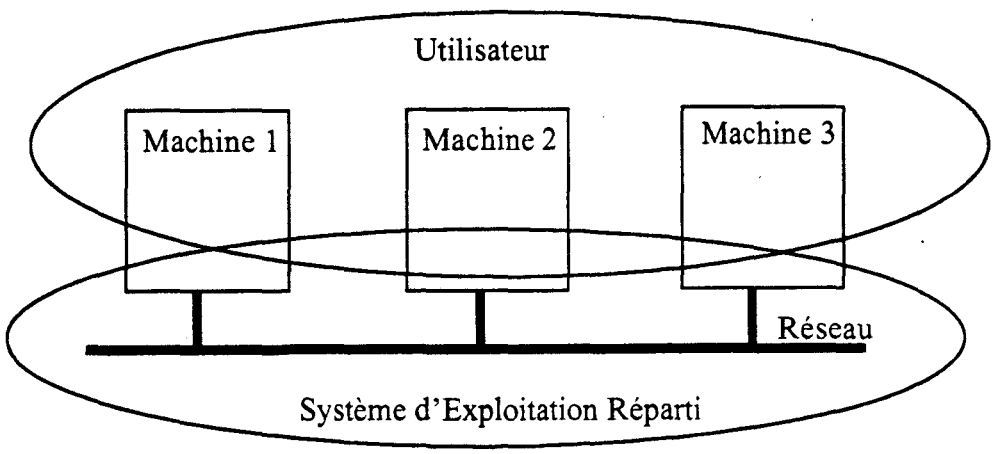


figure 1.1 *Système d'exploitation réparti*

D'un point de vue utilisateur, un système d'exploitation réparti permet d'utiliser un ensemble de machines connectées entre elles comme une seule machine. Lorsque l'utilisateur lance un programme sur sa machine, il ne sait pas - et n'a pas à s'en inquiéter - sur quelle machine physique son programme va se dérouler, ni même s'il va s'exécuter sur un seul processeur physique.

Les avantages des systèmes répartis sont nombreux [Tanenbaum94] :

- Coût : la construction d'une architecture répartie à partir de microprocesseurs est moins coûteuse, à performances théoriques équivalentes, qu'un ordinateur monolithique.
- Performances : un système distribué peut avoir une puissance de calcul aussi importante, sinon plus, que la puissance d'un gros ordinateur.
- Partage de ressources : un système distribué permet de partager des ressources matérielles (ex : imprimante) et logicielles (ex : base de données). Ces ressources sont réparties sur les différents noeuds de l'architecture.
- Fiabilité : le système peut continuer à fonctionner même si une machine tombe en panne (tolérance aux pannes). Si un noeud s'arrête, le système peut continuer à fonctionner en mode dégradé.
- Extensibilité : la puissance de calcul peut être augmentée en fonction des besoins. Il est possible d'ajouter de nouveaux processeurs à l'ensemble déjà installé.
- Souplesse : l'exécution d'un programme se fait en utilisant de manière optimale l'ensemble des ressources disponibles à un instant donné.

Si les SER ont des avantages certains, ils présentent également des inconvénients :

- Exploitation plus difficile: la programmation d'applications réparties reste une tâche difficile malgré les propriétés de transparence qu'offrent les SER. A l'heure actuelle il existe peu de logiciels de haut niveau pour les systèmes distribués.
- Réseau : le réseau peut être rapidement saturé et provoquer des problèmes en cascade. Ce problème peut être résolu en élargissant la bande passante du réseau.
- Sécurité : les SER permettent de partager beaucoup d'informations mais toutes ces données ne doivent pas forcément être accessibles à tous. Il faut que le SER intègre un système de sécurité et de protection ; ce mécanisme est plus difficile à concevoir que pour un système centralisé.

L'exploitation d'une architecture distribuée et de son SER se fait par la conception d'applications réparties, à l'aide d'outils et de langages qui doivent exploiter la répartition des ressources et le parallélisme physique de l'architecture.

Idéalement, la répartition des ressources doit être complètement cachée aux utilisateurs et aux programmeurs. Par exemple une opération sur une ressource distante doit être identique à une opération sur une ressource locale. Le concept de transparence ou encore

masquage de la répartition peut être perçu selon plusieurs points de vue :

- l'accès à des fichiers locaux et distants en utilisant les mêmes opérations
- l'accès à des objets sans connaître leurs emplacements
- la duplication des objets pour obtenir de meilleures performances et de la tolérance aux fautes
- la migration des objets pour une plus grande efficacité
- la reconfiguration du système pour améliorer les performances lorsque les charges varient
- l'ajout de nouvelles ressources sans devoir modifier le système ou les applications

L'exploitation du parallélisme physique de l'architecture doit permettre l'exécution de plusieurs applications en parallèle sans interférences entre elles. De même, une opération complexe doit pouvoir être entreprise en parallèle sur différents noeuds.

Par exemple, lorsque l'utilisateur lance la commande *make* pour compiler une application composée de plusieurs fichiers sources, les différentes compilations doivent pouvoir être faites en parallèle sur les différentes machines. La seule différence perceptible pour l'utilisateur devrait être l'amélioration des performances.

Cette parallélisation peut être transparente à l'utilisateur, mais aussi au programmeur (compilateur parallélisant pour architectures distribuées). Le plus souvent elle sera explicite au programmeur qui pourra utiliser des outils et langages (par exemple à base de processus) pour écrire des applications utilisant l'ensemble des noeuds.

Les deux besoins exprimés ici nécessitent de nouvelles fonctionnalités dans le système d'exploitation de ces machines distribuées. Deux approches sont possibles : concevoir un nouveau système d'exploitation réparti (avec donc des modifications profondes de son architecture logicielle par rapport aux systèmes anciens) ou étendre un système existant en ajoutant les fonctionnalités voulues sans modification profonde de l'existant. Nous présentons d'abord la première approche fondée le plus souvent sur la notion de micro-noyau. La seconde approche sera illustrée par l'utilisation d'UNIX pour les applications réparties.

I - 3 Exemples de Systèmes d'Exploitation Répartis

La caractéristique commune des systèmes d'exploitation répartis récents est qu'ils sont construits sur la notion de micro-noyau. L'élément central du système est un noyau de petite taille ne fournissant qu'un minimum de services, d'où son nom de micro-noyau. Ce noyau est présent sur chaque site du système réparti. Les services offerts sont fournis par des serveurs spécialisés (gestion de la mémoire, gestion des processus, ...). Ces serveurs coopèrent dans le micro-noyau. L'avantage de cette approche est que plusieurs sous-systèmes peuvent fonctionner simultanément (exemple : plusieurs systèmes de fichiers) et ainsi offrir à l'utilisateur plusieurs environnements.

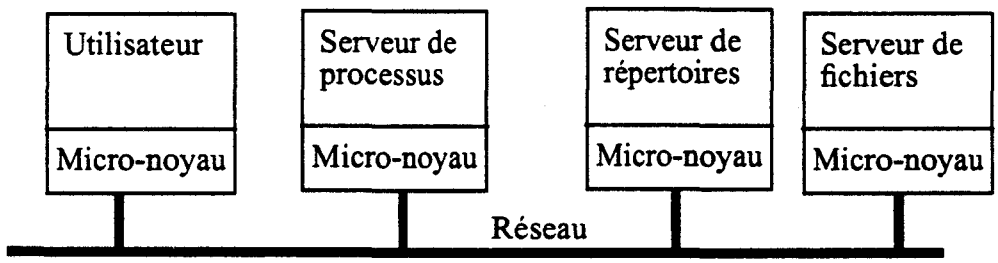


figure 1.2 *Architecture micro-noyau*

I - 3•1 Amoeba

Amoeba [Mullender90][Tanenbaum93] a été développé par l'équipe de A.S. Tanenbaum à la Vrije Univesiteit, Amsterdam. Il a été écrit sans s'appuyer sur des systèmes existants comme Unix, ceci afin de ne pas en subir les limitations. Deux postulats ont été faits à propos du matériel : le système possède un très grand nombre de CPU et chaque CPU a à sa disposition plusieurs méga-octets de mémoire. Même si ces postulats ne sont que peu vérifiés actuellement, ils le seront dans l'avenir. Ils ont des conséquences importantes sur le système : il n'existe pas de pagination mémoire et un processus doit résider entièrement en mémoire pour pouvoir être exécuté.

Amoeba est un nouveau système d'exploitation développé pour exploiter un ensemble d'ordinateurs indépendants en donnant l'illusion à l'utilisateur qu'il travaille sur un simple système à temps partagé. En général, l'utilisateur ne sait pas où ses processus sont en train de s'exécuter, ni même sur quel type de CPU. Il ne sait pas non plus où ses fichiers sont stockés, ni combien de copies sont maintenues pour des raisons de fiabilité ou de performances. Néanmoins, les utilisateurs qui le désirent peuvent exploiter les différents processeurs pour faire de la programmation parallèle ou pour répartir leur travail sur différentes machines.

Amoeba est basé sur un micro-noyau qui s'occupe de la gestion de bas niveau des processus, de la gestion mémoire, des communications et des entrées-sorties.

Amoeba dispose d'un unique mécanisme permettant le nommage et la protection de tous les objets : les capacités (*capabilities*). Chaque capacité contient un droit permettant de déduire les opérations autorisées sur l'objet. Les capacités sont protégées cryptographiquement par l'utilisation d'une fonction non-inversible. Chacune contient une somme de contrôle (cheksum) qui permet d'assurer sa sécurité.

Deux mécanismes de communication sont disponibles : les RPC pour les communications point à point et une communication de groupe fiabilisée. Ces deux types de communication s'appuient sur la couche FLIP (Fast Local Internet Protocol) qui se situe au même niveau que la couche IP (networks layers protocol). Ce nouveau protocole a été développé afin de pouvoir nommer les ports de communication de façon logique au niveau de la couche réseau. Ce nommage logique permet de déplacer un port sur le réseau, sans avoir à modifier son nom, ce qui facilite la création d'un mécanisme de

migration d'objets.

Le système de fichiers d'Amoeba est constitué de trois serveurs : un serveur s'occupant du stockage des fichiers (*bullet server*), un serveur de partitions (*directories server*) pour le nommage des fichiers, et un serveur de copies (*replication server*). Les fichiers sont désignés par leurs capacités, ils ne sont pas modifiables, ils ne peuvent qu'être créés ou détruits. Le *bullet server* les maintient et les stocke de façon continue sur le disque ou dans les caches. Le *directories server* fait la liaison entre les noms ASCII des fichiers et leurs capacités, il permet aussi une organisation hiérarchique des fichiers. Le *replication server* s'occupe de la copie "paresseuse" (*lazy copy*) des fichiers : il tourne en tâche de fond et s'occupe de maintenir un nombre constant de copies d'un objet. Il s'occupe aussi de la récupération des objets en se basant sur leur âge.

Amoeba possède d'autres serveurs : un serveur de lancement (*run server)* permettant de lancer des applications, un serveur d'initialisation (*boot server*) appelé à chaque redémarrage du noyau, un serveur de protocole TCP/IP et d'autres encore (entrées-sorties, nombres aléatoires, ...).

I - 3•2 Mach

Mach [Acceta86] est un système d'exploitation s'appuyant sur un micro-noyau. Il a été développé pour construire de nouveaux systèmes et pour émuler ceux existants. Il fournit aussi un moyen souple d'étendre Unix aux multi-processeurs et aux systèmes distribués. L'idée de départ était de montrer la faisabilité de structurer les systèmes d'exploitation en un ensemble de processus communiquant par messages.

Mach est basé sur les concepts de processus, threads, ports et messages. Un processus Mach est composé d'un espace d'adressage et d'un ensemble de threads s'exécutant à l'intérieur de cet espace. Un processus n'est qu'un simple contenant pour les threads qui sont les vraies entités actives. Chaque processus et thread possèdent un port dans lequel ils peuvent écrire afin d'effectuer un appel système, éliminant ainsi l'accès direct aux appels système.

Mach a une mémoire virtuelle très élaborée, incluant des objets *mémoire* qui peuvent être projetés dans l'espace d'adressage ou en être sortis et qui peuvent aussi être sauvés par des gestionnaires mémoire s'exécutant au niveau de l'utilisateur. Par exemple, les fichiers peuvent être lus ou écrits directement dans l'espace d'adressage en utilisant cette méthode. Le fichier est vu par le programme comme faisant partie de l'espace d'adressage et l'accès aux données du fichier se fait directement en lisant ou en écrivant dans cet espace. Les objets *mémoire* peuvent être partagés de différentes façons, incluant le *copy-on-write*, c'est à dire qu'un objet n'est effectivement copié que lors de la première modification (écriture) de la copie. Des attributs d'héritage déterminent les zones mémoires devant être passées aux enfants du processus.

Les communications dans Mach sont construites à l'aide de "ports" qui stockent les messages et font partie du noyau, ce sont des sortes de boîtes aux lettres. Les ports sont unidirectionnels : un client peut envoyer un message à un serveur par l'intermédiaire d'un port, ce message sera lu par le serveur qui enverra sa réponse par un second port. Un

port est accessible grâce à une capacité (*capabilities*) qui est élaborée par le noyau. L'utilisateur ne peut pas lire ou écrire directement une capacité, ceci afin de préserver la sécurité du système. Il ne peut y accéder que par l'intermédiaire d'un entier de 32 bits, qui est en réalité l'indice de la capacité dans un tableau interne au noyau. Les ports peuvent être passés d'un processus à un autre à l'aide de messages typés complexes. Les ports ne permettent que la communication locale à un noeud. La communication à travers le réseau est assurée par des serveurs (Networks Message Server), mais reste transparente pour l'utilisateur qui continue à utiliser un entier pour référencer un port. Le message est envoyé à un port qui est lu par le serveur. Celui-ci communique grâce à la couche réseau IP avec un autre serveur situé sur le noeud distant. Ce serveur dépose le message dans un port local connu du processus destinataire.

Le principal serveur fonctionnant au-dessus de Mach est un émulateur BSD-Unix. Celui-ci est composé de deux parties : le serveur Unix et une librairie d'émulation des appels système qui est incluse dans toutes les applications Unix. Lors du lancement, le serveur demande au noyau de capturer tous les "traps" système et de les transmettre à la librairie du programme (mécanisme de "trampoline"). La librairie recherche l'appel système et le transforme en message adressé au serveur Unix.

I - 3•3 Chorus

L'architecture Chorus [Rozier88][Chorus91] est structurée en trois couches : la première, qui en constitue les fondations, est un noyau générique s'exécutant sur chaque machine, la seconde regroupe les fonctions des systèmes d'exploitation classiques, ceux-ci sont construits comme des sous-systèmes utilisant les services de base fournis par le noyau. La troisième est constituée des programmes d'application qui s'exécutent dans le contexte d'un sous-système.

A la base du système, le noyau et un ensemble de serveurs implantant les services de bas niveau sont utilisés pour la réalisation de systèmes complets. Un système Chorus est constitué du noyau de petite taille et d'un ensemble de serveurs. Ceux-ci coopèrent dans le contexte de sous-systèmes pour fournir à leurs clients un ensemble cohérent de services et d'interfaces.

Un noyau Chorus offre les bases fondamentales pour construire un système d'exploitation supportant des applications distribuées et temps réel. Ces services sont simples mais puissants. Leur généricité permet d'implémenter différents systèmes. Les services de base du noyau sont :

- Gestion de l'exécution des threads : une interface simple permet de contrôler et de synchroniser l'exécution des threads.
- Communication inter-processus (IPC) : l'interface de communication est la même quelle que soit la localisation du destinataire. Le système optimise les communications locales.
- Gestion de l'espace d'adressage : une interface générique est fournie, permettant de développer différentes stratégies, du schéma simple à la mémoire distribuée.
- Facilités et temps de réponse suffisant pour programmer des applications temps

réel. Un système peut utiliser le mécanisme d'exceptions, les interruptions, les entrées-sorties de bas niveau, et les modifier dynamiquement sans arrêter l'application.

Un sous-système est un système d'exploitation construit au-dessus du noyau Chorus. Il est composé d'un ensemble de serveurs réalisant les services de haut niveau et coopérant ensemble pour offrir une interface cohérente. Une application Chorus dispose de plusieurs niveaux d'interface : l'interface noyau, qui traite directement les "traps" correspondant au service demandé et l'interface fournie par le sous-système qui traite des requêtes de haut niveau. Plusieurs sous-systèmes peuvent être utilisés simultanément : ainsi un système Unix peut fonctionner simultanément avec un système MS-DOS. Le noyau et les interfaces des sous-systèmes sont enrichis par des bibliothèques. Celles-ci permettent d'avoir accès aux différentes fonctionnalités du système depuis un langage donné en fournissant les fonctions de la bibliothèque qui seront associées lors de l'édition de liens et qui seront exécutées dans le contexte de l'utilisateur. En général, ces fonctions copient les paramètres de l'espace utilisateur vers l'espace du système, effectuent un "trap" vers le système, puis une fois revenues, copient les résultats dans le sens inverse.

Le noyau Chorus fournit un certain nombre d'abstractions de base : l'acteur qui est l'unité d'allocation de ressources, l'activité qui est l'unité d'exécution séquentielle, les messages, les ports et les groupes de ports permettant les communications et la région qui est l'unité de structuration mémoire. Tous ces objets sont désignés par un identificateur unique (UI) et sont gérés par le noyau. D'autres objets existent, gérés à la fois par le noyau et par des serveurs externes : les segments qui permettent l'encapsulation des données, et les capacités qui désignent ces données.

I - 3•4 Utilisation des systèmes d'exploitation répartis

Les SER ne résolvent pas tous les problèmes. Si c'était le cas, un programme "classique", c'est-à-dire non prévu à l'origine pour fonctionner sur un système d'exploitation réparti, y fonctionnerait de manière optimale. Par exemple la tolérance aux fautes ne peut pas être gérée uniquement dans le SER. Cela implique l'écriture de langages spécialement conçus pour exploiter les SER. Le SER fournit des mécanismes de distribution automatique des applications. Dans certains cas, le programmeur peut vouloir distribuer lui-même son application sur la machine virtuelle ; ceci pour obtenir un gain de performances.

Les langages doivent masquer les fonctionnalités du SER mais également permettre au programmeur d'influer sur le fonctionnement du SER. Dans l'état actuel, les SER fournissent des outils permettant au programmeur de gérer explicitement la décentralisation et le parallélisme de l'architecture.

I - 4 UNIX réparti

Nous allons nous intéresser aux systèmes formés de noeuds Unix reliés par un réseau local. Le système UNIX augmenté d'un ensemble d'outils aujourd'hui bien connus, fournit des fonctionnalités permettant la programmation d'applications réparties. Un tel système est donc un système d'exploitation réparti et c'est sûrement le système d'exploitation réparti le plus utilisé actuellement.

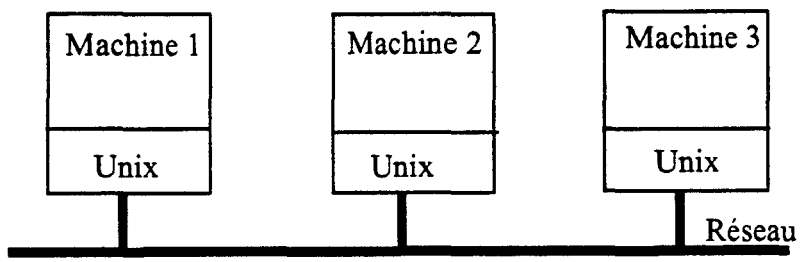


figure 1.3 *Réseau de stations de travail*

I - 4•1 Les outils UNIX de base

Les outils de base d'UNIX pour la programmation répartie sont ceux qui permettent la création dynamique d'activités concurrentes (processus et threads) et la coopération de ces activités (synchronisation) et leurs communications. Nous allons les présenter.

I - 4•1•1 Les processus

Un processus est une entité qui permet d'exécuter un programme sur une machine. Un processus est composé :

- d'un espace d'adressage
- de variables globales
- de descripteurs des fichiers utilisés
- d'une gestion de signaux
- d'un compteur ordinal
- d'une pile

Un processus s'exécute sur une machine. Le processus communique avec le système d'exploitation par appels système. Ces appels permettent par exemple d'allouer de la mémoire pour stocker des données, de lire un fichier ou encore de créer un nouveau processus. Plusieurs processus peuvent s'exécuter simultanément sur une même machine. Chaque processus possède un espace d'adressage dans lequel il stocke ses données. Il n'y a pas de partage de mémoire entre processus. Nous verrons plus loin les

mécanismes disponibles à ce niveau.

UNIX fournit des appels systèmes pour créer de nouveaux processus à partir de processus déjà lancés. Cela se passe sur une seule machine - c'est le mécanisme du système Unix initial. Nous détaillons les principaux.

- L'appel système fork() crée une copie du processus appelant cette fonction. L'espace d'adressage est dupliqué, le code exécutable est partagé. Après l'exécution du fork(), les deux processus s'exécutent indépendamment l'un de l'autre. Il y a une notion de hiérarchie entre les processus : l'initiateur de l'appel à fork() est appelé processus père et le nouveau processus créé est appelé fils.

- L'appel système wait() permet à un processus père d'attendre la terminaison d'un processus fils qu'il a créé. La fonction wait() retourne le numéro du processus (PID) fils terminé et le code de retour de l'exécution.

- L'appel à exec() permet de changer le code du processus qui s'exécute. Cet appel libère la mémoire utilisée par le précédent programme, charge le code du nouveau processus et lance l'exécution. Plusieurs versions de la primitive exec() sont disponibles dans le système UNIX.

En composant les trois primitives présentées ci-dessus, le programmeur peut aborder la programmation concurrente en pseudo-parallélisme. L'extension de ces mécanismes à un ensemble de machines permettrait d'aborder la programmation concurrente en réel parallélisme. Donnons un exemple de lancement d'un traitement asynchrone.

```
if ((pid_fils=fork()) == 0)                    (1)
{
 /* dans le fils */
 execve (programme2, args) ;                   (2a)
}else{
 /* dans le père */                            (2b)
 ...
 /* récupération terminaison du fils */
 retour=wait (&pid) ;                          (3)
}
```

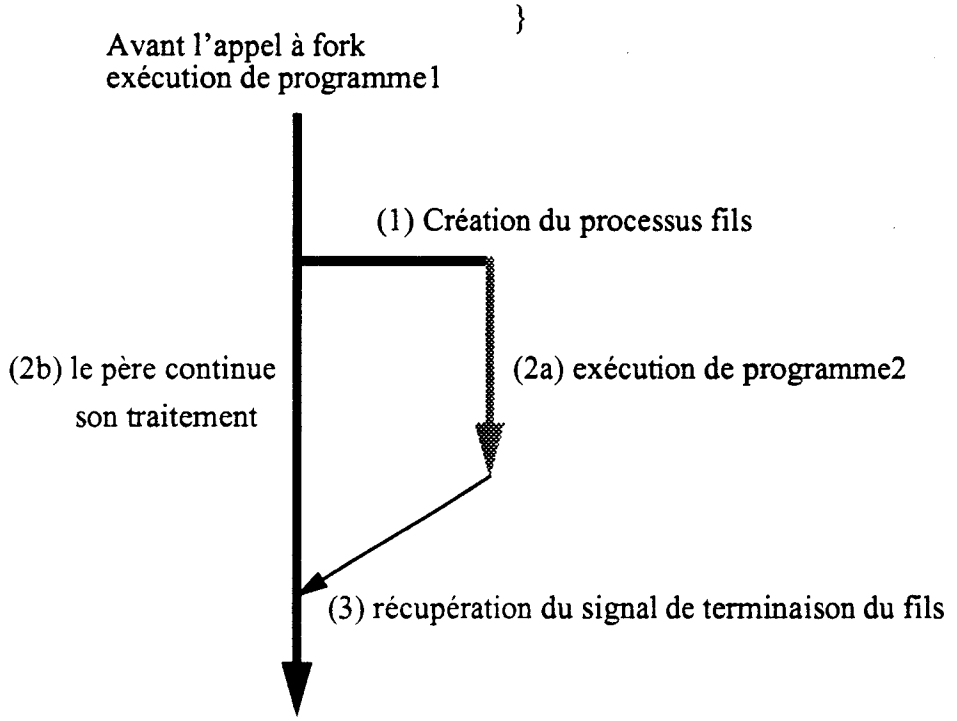


figure 1.4 *Utilisation de fork(), exec() et wait()*

I - 4•1•2 Les processus légers

Les processus légers (ou threads) permettent d'avoir plusieurs activités au sein d'un même processus UNIX. Les processus légers ressemblent beaucoup aux processus UNIX. Ils sont qualifiés de légers parce qu'ils sont composés uniquement :

- d'un compteur ordinal
- d'une pile

Les processus légers permettent la programmation concurrente en pseudo-parallélisme à l'intérieur d'un seul processus. L'extension de ce mécanisme à un ensemble de processus distribués sur l'architecture permettrait d'aborder la programmation concurrente par processus légers en réel parallélisme (voir Chant et PVC).

Les processus légers d'un processus UNIX se partagent les variables globales, les signaux, les descripteurs de fichiers et le code à exécuter.

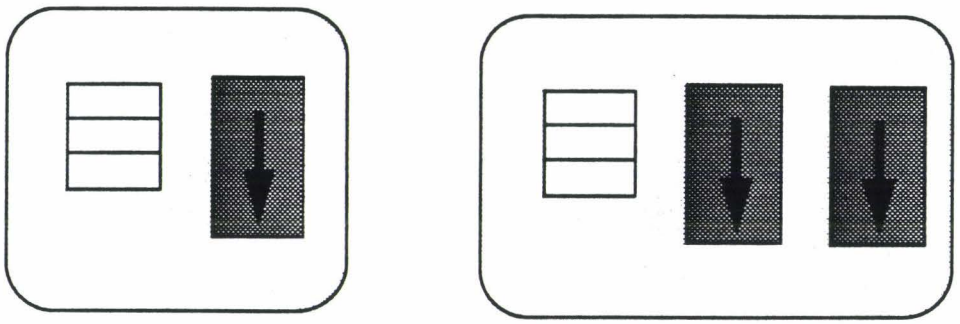


figure 1.5 *Processus simple et processus avec deux processus légers*

Les processus légers peuvent être statiques ou dynamiques. Dans le cas statique, il faut spécifier lors de l'écriture du programme le nombre de processus légers à créer lors de l'exécution du programme. Cette solution est restrictive dans le sens où il faut savoir avant l'exécution le nombre de processus dont le programme va avoir besoin. La solution dynamique permet de créer des processus légers en fonction des besoins du programme.

A la création du thread, on spécifie la procédure qui sera exécutée. Il est possible de transmettre des paramètres et de spécifier une priorité si le système le permet.

La bibliothèque de processus légers doit offrir des primitives pour créer un thread, le suspendre, le détruire.

Nous présentons un exemple de création de processus léger avec attente de la terminaison du nouveau processus. Nous utilisons l'interface pthreads :

```
void fonction2 () { ... }

void fonction1 ()
{
     pthread pid ;
     any_t status ;

     pthread_create (&pid, NULL, fonction2, NULL) ;
     /* Creation d'un processus léger et lancement de
        fonction2 */

     ... suite de la fonction

     pthread_join (pid, &status) ;
     /* attente de la fin de fonction2 */
}
```

figure 1.6 *Exemple de création de processus léger et d'attente de terminaison*

Deux processus légers d'un même processus peuvent communiquer entre-eux par variable partagée. Cette communication par mémoire commune implique une protection de la zone partagée pour garantir la cohérence des données. Plusieurs outils permettent d'implémenter cette protection : moniteur, verrou ou encore sémaphore. Ces mécanismes sont de très bas niveau. Ils impliquent une utilisation dans un contexte de mémoire partagée. Ils sont utilisables pour de petits problèmes de synchronisation. Ils sont difficiles à programmer et à maintenir lorsque la compléxité du problème augmente.

L'utilisation de processus légers avec des bibliothèques pose des problèmes si le code des fonctions des bibliothèques n'est pas réentrant ou si la bibliothèque utilise des variables globales. Par exemple si un mécanisme de communication se sert d'un tampon pour préparer le message à envoyer, ce tampon ne doit pas être global. La solution à ce problème est de protéger à l'aide d'un sémaphore ou d'un verrou les accès à la bibliothèque. Le mécanisme d'allocation mémoire (malloc) peut également poser des problèmes : deux activités ne peuvent pas simultanément demander de la mémoire ; il y a un risque d'incohérence dans la gestion des blocs mémoire.

Les processus légers peuvent être implémentés de deux manières : en tant que bibliothèque (par exemple pthreads [Mueller93]) ou dans le noyau (Solaris 2.3).

Par souci de portabilité, une interface de manipulation de processus légers a été normalisée : c'est la norme POSIX 1003.4. Cette norme définit la liste des fonctionnalités qui permettent de manipuler les processus légers. Ceux-ci peuvent être implémentés par une bibliothèque ou dans le noyau du système d'exploitation.

I - 4•1•3 Les tubes

Les tubes (pipes) sont un moyen de faire communiquer deux processus UNIX situés sur la même machine. Ce mécanisme crée un espace mémoire partagé. Le programme se sert des appels systèmes `read` et `write` pour lire et écrire dans le tube. Les tubes fonctionnent de manière unidirectionnelle ; c'est-à-dire que si deux processus veulent communiquer dans un mode question/réponse, il faut utiliser deux tubes.

Le mécanisme de tube peut être également utilisé avec les processus légers. Il faut protéger l'accès au tube en lecture comme en écriture si plusieurs processus légers d'un même processus UNIX utilisent le tube.

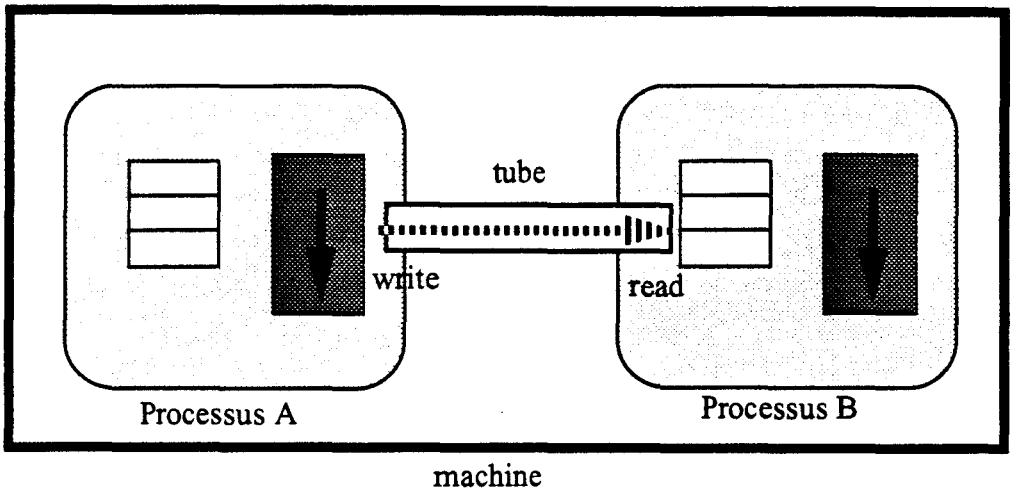


figure 1.7 *Communication entre deux processus à l'aide d'un tube*

Si le tube est vide lorsque le processus B exécute le `read`, le processus reste bloqué jusqu'à ce que des données soient disponibles dans le tube.

De même, si le processus A écrit dans le tube et que le tube est plein (le processus A génère des informations plus vite que le processus B ne les consomme), il reste bloqué jusqu'à ce que le tube ait été vidé par le processus B.

Les pipes permettent à deux processus situés sur la même machine de communiquer. Dans un contexte réparti où deux processus sont sur deux machines différentes, les tubes ne peuvent pas être utilisés (il n'y a pas de mémoire commune). Nous allons voir que les sockets résolvent ce problème.

I - 4•1•4 Inter-Processes Communication (IPC)

La couche IPC offre plusieurs mécanismes de communication entre processus UNIX. Ils permettent à deux processus d'une même machine de communiquer. Ces processus n'ont pas besoin d'avoir un lien de parenté (père - fils). L'accès à un dispositif IPC se fait toujours de la même manière : il faut utiliser la primitive *X*get() où *X* représente le type de mécanisme utilisé. Les primitives d'obtention de ressources IPC utilisent une clé d'accès pour pouvoir retrouver une ressource déjà créée ; si la ressource n'existait pas, elle est créée au premier appel.

Les IPC sont de trois types : sémaphores, messages et mémoire partagée.

I - 4•1•4•1 Les sémaphores

Le sémaphore IPC permet de synchroniser des processus qui utilisent des ressources partageables ou critiques. Il faut noter qu'un sémaphore IPC n'est pas lié à l'existence d'un processus qui l'utilise.

I - 4•1•4•2 Les messages

Une file d'attente de messages permet à un processus de recevoir des données en provenance d'autres processus et également de leur envoyer des données. Les messages sont des chaînes de caractères.

La primitive `msgsnd()` permet à un processus d'envoyer un message dans une file d'attente qu'il aura préalablement obtenue par `msgget()`. La taille de la file d'attente est finie. Selon le choix du programmeur, l'envoi d'un message dans une file pleine peut soit suspendre le processus jusqu'à ce qu'il y ait de la place dans la file, soit retourner immédiatement une erreur. Les messages sont typés, c'est au programmeur de gérer les types des messages.

La primitive `msgrcv()` permet à un processus de réceptionner les messages envoyés par d'autres processus. Grâce au typage des messages, il est possible de filtrer les messages dans la file. Si la file est vide, le programmeur peut choisir de suspendre son processus jusqu'à l'arrivée d'un message correspondant au type désiré ou alors de terminer immédiatement l'opération. Il faut souligner que la lecture d'informations de la file d'attente est destructive. Le système protège les files d'attente contre les accès simultanés.

I - 4•1•4•3 La mémoire partagée

La mémoire partagée permet de mettre en commun un segment mémoire entre plusieurs processus. Ces processus peuvent s'échanger des informations par lecture et écriture dans ce segment. Il n'y a pas de mécanisme de synchronisation de la mémoire partagée contre les accès concurrents. La synchronisation est à réaliser par le programmeur en se servant de sémaphores UNIX.

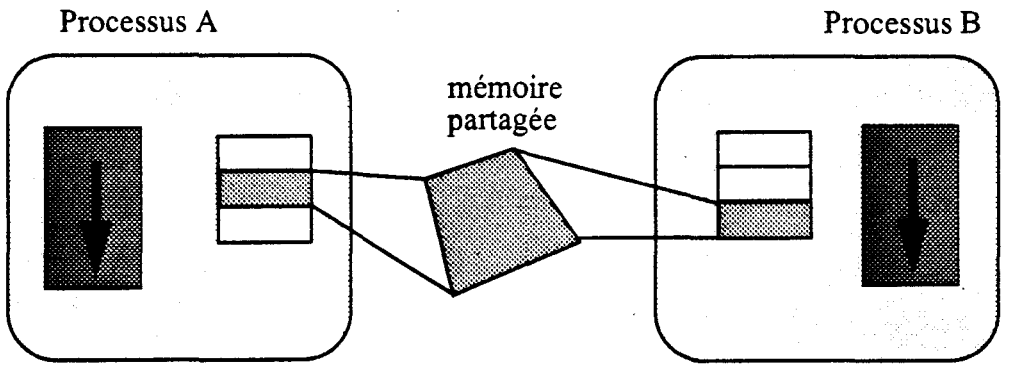


figure 1.8 *Mémoire partagée entre deux processus UNIX*

I - 4•1•5 Les sockets

Les sockets permettent de faire communiquer deux processus qui s'exécutent concurremment sur deux machines différentes. Quand le processus A situé sur la machine 1 veut communiquer avec le processus B situé sur la machine 2, il prépare un message dans son espace d'adressage ; il exécute ensuite un appel système qui va transmettre son message au processus B via le réseau. Le processus B utilisera le message dans son espace d'adressage sur la machine 2.

Le protocole de communication entre deux processus est standardisé : c'est le modèle en couches OSI [Day83]. Le programmeur d'applications distribuées se sert du protocole TCP/IP pour écrire les éléments communicants de son application répartie.

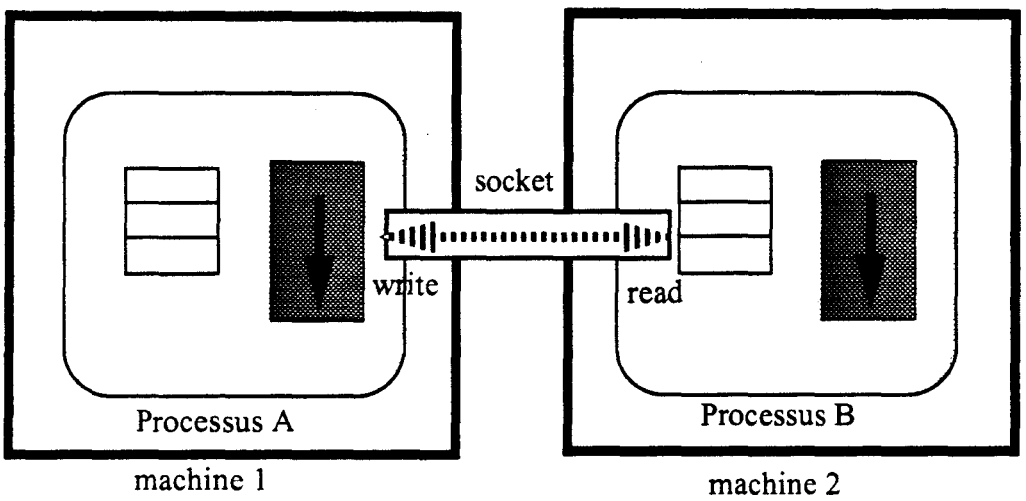


figure 1.9 *Communication entre deux processus à l'aide de socket*

Les sockets permettent à deux processus de communiquer de manière bi-directionelle.

Il existe deux modes d'utilisation des sockets : le mode non connecté (UDP) et le mode connecté (TCP). Dans le premier mode, l'émetteur doit toujours spécifier l'adresse du destinataire. La connection n'est pas continue. Dans le second mode, la connection est tout le temps établie.

Avec les sockets, le programmeur d'applications réparties dispose d'un mécanisme pour réaliser ses applications mais n'a pas de guide méthodologique pour les concevoir. Le modèle client-serveur est fréquemment utilisé pour faciliter le développement d'applications réparties. L'idée est de structurer le système en processus coopérants appelés serveurs. Les serveurs rendent des services à des clients.

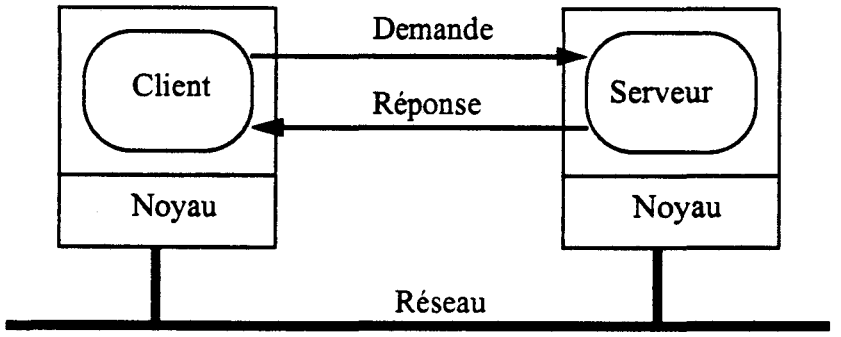


figure 1.10 *Le modèle client-serveur*

C'est au programmeur de définir le protocole de communication entre le client et le serveur. Si le client envoie un message composé d'un entier, le serveur doit s'attendre à recevoir un entier sinon la communication ne fonctionnera pas ! Le serveur restera en attente de réception ou s'arrêtera.

I - 4•1•6 eXternal Data Representation (XDR)

Nous avons vu que les sockets permettent de faire communiquer deux processus sur deux machines différentes. Cette communication est correcte si les deux machines utilisent la même représentation des données en mémoire. Dans un réseau de stations hétérogènes, la représentation peut être différente. Par exemple la communication entre une station Sun (processeur Sparc) et une station Alpha (processeur Alpha) ne peut pas être immédiate. En effet, sur Sun les entiers sont codés en format Big-Endian alors que sur Alpha les entiers sont en Little-Endian. Un entier passé directement entre ces deux machines n'aurait pas la même valeur à la réception qu'à l'émission.

Pour remédier à ce problème, il faut utiliser un codage indépendant des architectures des machines. C'est le format XDR [Corbin90].

La bibliothèque XDR est un ensemble de fonctions C qui convertissent des données entre la représentation de la machine locale et leur représentation en XDR. XDR fournit des primitives pour encoder des données et pour les décoder.

I - 4•1•7 Les Remote Procedure Calls (RPC)

I - 4•1•7•1 Présentation

Les RPC [Birrell83], appels de procédures à distance, permettent aux programmes d'appeler des procédures situées sur d'autres machines. Le but des RPC est de cacher au client le fait que l'appel va s'exécuter à distance. Pour le client, il n'y a pas de différence entre un RPC et un appel de procédure locale.

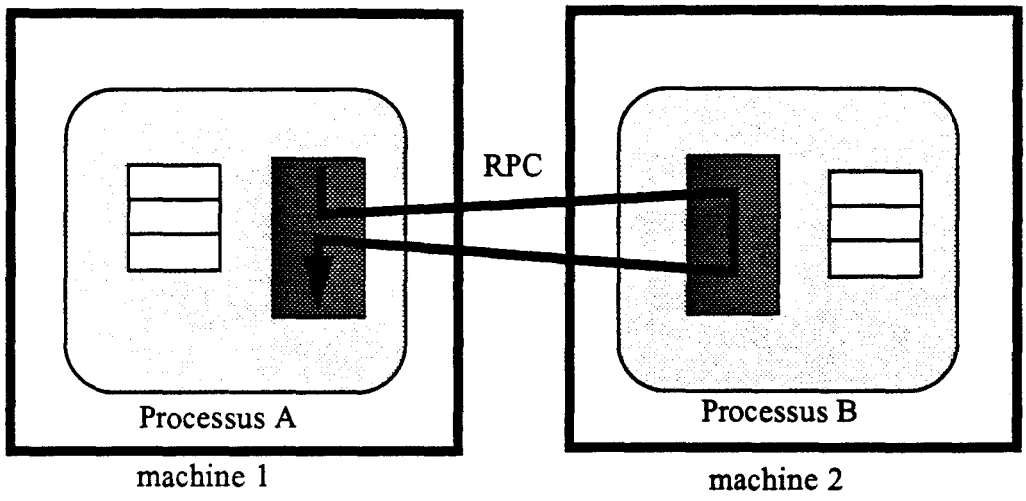


figure 1.11 *Remote Procedure Call*

Lorsque le processus A demande l'exécution d'une procédure sur la machine B, l'appelant est bloqué pendant l'exécution de la procédure sur la machine 2. Des paramètres peuvent être transmis de l'appelant à l'appelé. Les paramètres sont de type de base : entier, chaîne, ... ; un pointeur sur la machine client n'a aucun sens sur la machine serveur. Un résultat peut être retourné de l'appelé à l'appelant. La procédure appelante n'a pas à savoir où se trouve la procédure appelée et inversement.

Pour pouvoir communiquer, le client et le serveur utilisent des stubs. Un stub permet de transmettre de manière transparente des informations entre processus de deux machines différentes en utilisant le réseau

I - 4•1•7•2 Découpe de l'appel à distance

1) la procédure du client appelle le stub par un appel de procédure classique.

2) le stub client construit un message et fait un appel système vers le noyau.

3) le noyau envoie le message vers le noyau distant.

4) le noyau distant donne le message au stub serveur.

5) le stub serveur récupère les paramètres et appelle le serveur.

6) le serveur effectue le travail demandé et retourne le résultat au stub serveur.

7) le stub serveur construit un message contenant le résultat et fait un appel système vers le noyau.

8) le noyau distant envoie le message au noyau du client.

9) le noyau du client donne le message au stub client.

10) le stub client extrait le résultat du message et le donne au client.

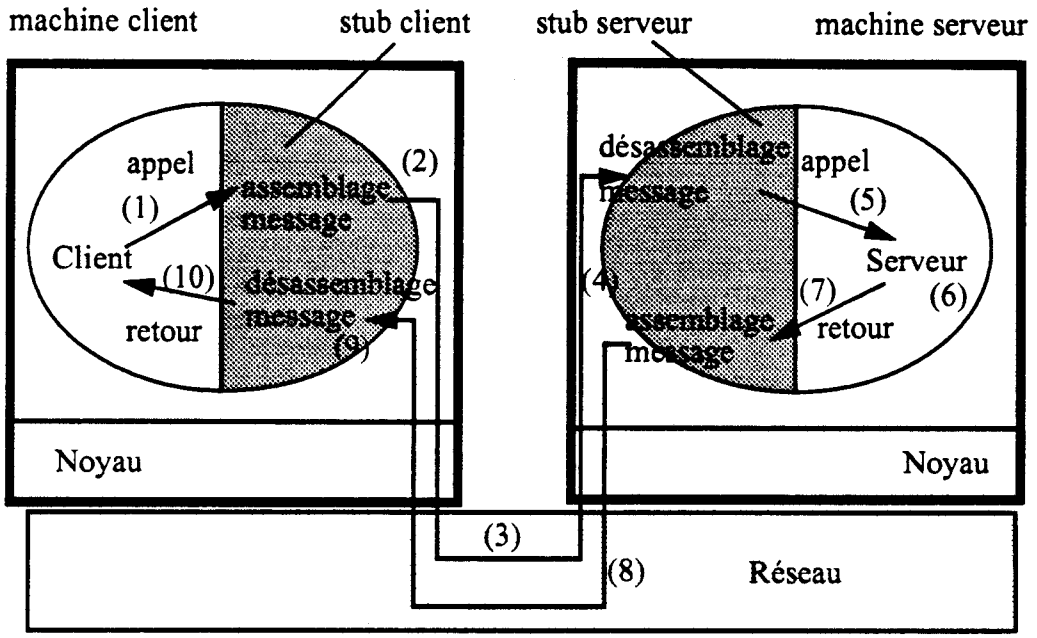


figure 1.12 *Scénario d'un RPC*

I - 4•1•7•3 Nommage des processus client et serveur

Nous avons vu que les RPC étaient transparents pour le client et le serveur. Toutefois, il faut que les stubs client et serveur puissent communiquer et pour cela ils doivent se connaître.

Au lancement du serveur, celui-ci s'enregistre auprès d'un serveur de noms pour être connu des futurs clients. Le serveur exporte l'interface qui est accessible à partir des clients.

Lorsqu'un client veut lancer un RPC, le stub client fait appel au serveur de noms pour importer l'interface du serveur qui pourra répondre à sa demande. Si un serveur est trouvé, le gestionnaire de noms envoie au stub du client l'identificateur et l'adresse du serveur. Si aucun serveur n'est trouvé, la demande du client échoue.

Une fois que le client a l'adresse du serveur, il déroule le scénario présenté ci-dessus. L'adresse du stub client est passée implicitement pour permettre au serveur de retourner le résultat du traitement.

I - 4•1•7•4 Génération des stubs

Le programmeur d'applications distribuées dispose d'outils pour générer les souches (stubs) client et serveur ; c'est l'utilitaire Rpcgen. Rpcgen est un compilateur qui traduit une définition d'interface RPC écrite en RPC Langage (RPCL) en code C. Le code généré est composé des fonctions pour le client, du squelette du serveur, des filtres XDR pour la conversion des données entre machines hétérogènes.

Pour écrire une application cliente, il faut faire appel aux fonctions générées pour le stub client et ensuite faire l'édition de liens avec le code généré par Rpcgen pour le client. Pour concevoir le serveur, le programmeur d'applications doit écrire le code spécifique de son serveur dans le squelette qui a été créé par Rpcgen.

Un serveur peut offrir plusieurs services, c'est-à-dire plusieurs interfaces. Actuellement les serveurs sont mono-programmés : ils peuvent servir un client à la fois. Si un autre client exécute un RPC, celui-ci sera retardé jusqu'à la fin de l'exécution du RPC en cours. La solution pour éviter ce goulot d'étranglement est que le serveur crée un nouveau processus qui va réellement traiter la requête du client. Comme cela, le serveur est de nouveau prêt à accepter de nouvelles requêtes de la part des autres clients.

Depuis peu, Sun offre un mécanisme de RPC multi-programmé avec une gestion de processus légers.

I - 4•2 Résumé

Unix fournit un ensemble d'outils pour l'écriture d'applications réparties. Ces outils permettent de décrire des activités en termes de processus UNIX et légers. UNIX fournit des primitives de communication par messages ; l'utilisation de machines hétérogènes est assurée par la bibliothèque XDR. L'appel de procédure à distance permet de lancer des traitements sur d'autres machines.

L'ensemble de ces outils donne un environnement d'applications réparties sous UNIX. Cet environnement est réservé au développeur expert car les outils fournis sont de bas niveau et difficiles à utiliser. Il est de plus difficile de tester les applications créées. Peu d'outils de déverminage existent à ce niveau.

I - 5 Environnements pour applications réparties

Au-dessus d'UNIX, on trouve un ensemble de produits logiciels (bibliothèques, langages) qui facilitent l'écriture d'applications réparties en offrant un cadre conceptuel, ou au moins des appels de plus haut niveau. Nous en décrivons ici certains.

I - 5•1 Parallel Virtual Machine (PVM)

PVM (Parallel Virtual Machine) est un exemple d'un tel système [Geist93]. PVM permet de voir un ensemble de machines hétérogènes comme une seule machine parallèle. Les noeuds de la machine virtuelle peuvent être des multiprocesseurs, des stations de travail, des machines spécialisées. Ces machines doivent être connectées entre elles (par exemple par ethernet ou FDDI).

Le programmeur écrit ses programmes en C, C++ ou fortran. L'accès à PVM est réalisé par des fonctions de bibliothèque.

PVM est composé de deux parties : le démon PVM et la bibliothèque PVM.

I - 5•1•1 Le démon PVM

Les fonctionnalités du démon sont :

- la création et la terminaison de tâches.
- la conservation de la base de données des tâches qui tournent.
- la gestion de la synchronisation.
- le routage des messages entre tâches.

Chaque utilisateur possède sa machine virtuelle. Elle est composée d'un ensemble de machines (hosts) sur lesquelles tournent un démon PVM. Il y a un démon par machine.

I - 5•1•2 La bibliothèque PVM

La bibliothèque de communication PVM gère les processus, l'échange de données et la synchronisation.

I - 5•1•2•1 Gestion des processus

Une tâche PVM est implémentée par un processus UNIX. Une fonction de gestion de processus enregistre les processus UNIX comme des composants PVM : les processus UNIX se connectent aux démons PVM. A la terminaison d'un processus, celui-ci doit signaler au démon PVM que son exécution s'achève. Le lancement d'un processus se fait par la fonction pvm_spawn () ; les paramètres de cette fonction sont le nom de l'exécutable à lancer, des paramètres pour le programme et des informations de répartition. Le programmeur peut laisser PVM choisir le site où va s'exécuter son programme ; il peut aussi spécifier sur quel type de machine son programme doit tourner ou encore spécifier une machine particulière.

I - 5•1•2•2 Echange de données

Ces fonctions permettent l'échange de données entre tâches PVM.

Un message est préparé dans un buffer PVM. PVM fournit des procédures de construction de message pour les types de base - int : procédure pvm_pkint - float : procédure pvm_pkfloat, ... Les structures complexes peuvent être transférées. Dans ce cas, c'est au programmeur de placer dans le message chacun des éléments contenus dans la structure. Une dernière primitive (pvm_pkbytes) permet de mettre dans un message une séquence d'octets. Cette dernière primitive ne fonctionne pas entre des machines qui ont des représentations de données différentes.

Lorsque le message est créé, il peut être envoyé à une tâche PVM particulière (coomunication point à point) ou à un groupe de tâches PVM (communication collective).

La réception fonctionne de manière similaire au dépôt : PVM offre un jeu de procédures pour extraire les données d'un message.

La communication peut se faire de deux manières : en se servant du démon ou en envoyant directement un message à une tâche. Dans le premier cas, la tâche envoie son message au démon local qui va le diriger vers le démon qui tourne sur la même machine où tourne la tâche destinatrice. Le protocole utilisé est UDP (Datagrammes). Dans le second cas, le message est directement transmis entre les tâches ; on parle alors de circuit virtuel de communication. La seconde possibilité est plus performante (en temps) que la première.

Il faut noter que des problèmes se posent si on tente d'interfacer PVM avec une bibliothèque de processus légers. En effet plusieurs threads peuvent essayer de lire un message en même temps et PVM ne le permet pas.

I - 5•1•2•3 Synchronisation

Dans PVM, les tâches peuvent être synchronisées à l'aide de rendez-vous.

Les rendez-vous (ou *barriers*) permettent à un groupe de processus PVM de coopérer. Les rendez-vous sont définis par un nom symbolique et un nombre de processus. Les processus qui veulent se synchroniser font appel à cette barrière, ils se bloquent alors. Lorsque le nombre de processus bloqués atteint la valeur de la barrière, les processus bloqués sont libérés et peuvent continuer leur traitement respectif. Pendant l'exécution d'une application PVM, chaque barrière ne peut être utilisée qu'une seule fois. Une barrière ne peut pas être utilisée dans le corps d'une boucle. Pour contourner ce problème, il est possible de générer des noms dynamiques de barrière.

I - 5•1•3 Avantages et inconvénients de PVM

Avantages

- Le modèle de programmation est simple : une application est modélisée en termes de processus communicants.
- PVM est portable : une version existe sur beaucoup de systèmes UNIX.
- PVM est fourni avec un environnement graphique : HeNCE, Heterogeneous Network Computing Environment qui permet au programmeur de concevoir son application parallèle et distribuée. Le programmeur décrit le parallélisme de son application à l'aide d'un graphe acyclique qui représente la structure logique de son programme. HeNCE génére automatiquement le code PVM.
- XPVM est une interface Xwindow pour le shell PVM. Des fonctionnalités de trace des programmes sont disponibles.
- et enfin PVM est gratuit !! Il est disponible en version source.

Inconvénients

- PVM ne fournit pas de mécanisme de tolérance aux pannes, pas de diffusion

atomique, pas de migration de processus.

- La sécurité dans PVM est basée sur la sécurité du système UNIX.
- PVM ne permet de recevoir qu'un seul message à la fois.
- Il n'y a pas de migration de tâche.
- Il n'y a pas de fonctionnalités d'équilibrage de charge dynamique.
- PVM permet d'écrire des applications réparties mais le grain des éléments de l'application peut être qualifié de gros : les éléments sont des processus UNIX.

ISIS [Birman93] est un autre outil qui possède des fonctionnalités similaires à PVM. Dans ISIS, l'accent est mis sur la notion de groupes de processus et sur des mécanismes de diffusion atomique, ceci afin de permettre la conception d'applications tolérantes aux pannes. ISIS intègre les processus légers [Birman93b]. Plusieurs activités peuvent se dérouler simultanément dans un processus ISIS. Il faut remarquer que les threads de ISIS ne sont pas ordonnancés : un processus léger s'exécute tant qu'il ne demande pas explicitement à être interrompu pour permettre l'exécution d'un autre processus léger.

I - 5•2 Synchronizing Ressources (SR)

SR [Olsson92] est un langage pour écrire des programmes parallèles. Un programme peut s'exécuter dans plusieurs espaces d'adressage qui peuvent être situés sur plusieurs machines physiques différentes.

Un programme est découpé en un ou plusieurs espaces d'adressage appelés machines virtuelles. Chaque machine virtuelle définit un espace d'adressage sur une machine physique. Les machines virtuelles sont créées dynamiquement. La forme la plus simple d'un programme SR est une machine virtuelle qui s'exécute sur une machine réelle. Généralement un programme est composé de plusieurs machines virtuelles s'exécutant sur plusieurs machines réelles.

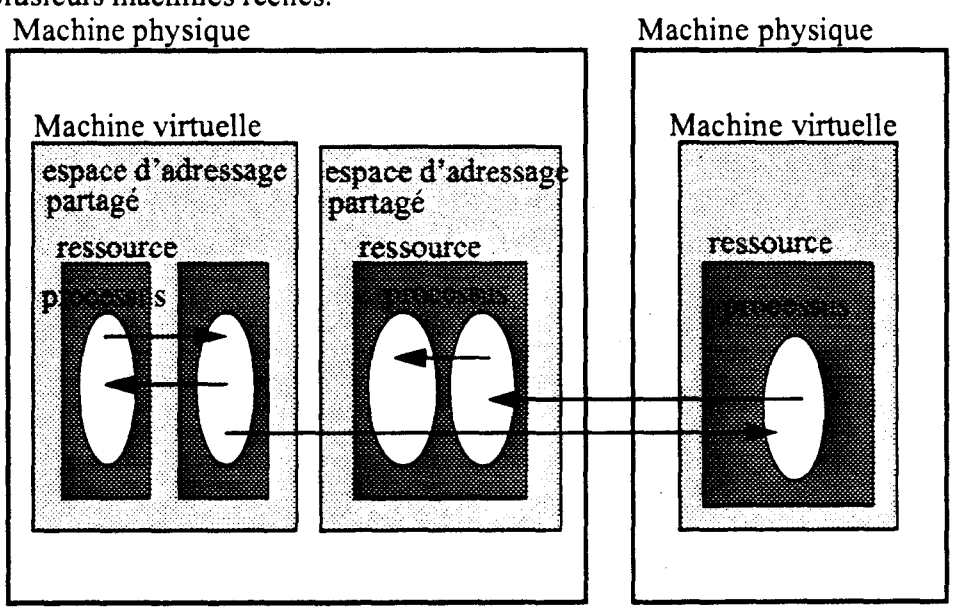


figure 1.13 *Le modèle de SR*

Les entités principales du langage sont les ressources et les opérations. Les ressources encapsulent les processus et les données qu'ils partagent. Les opérations fournissent un mécanisme pour faire intéragir des processus. Les opérations sont l'appel de procédure, les rendez-vous, l'envoi de message, la création dynamique de processus, le multicast et les sémaphores. SR autorise aussi l'utilisation de variables globales.

Les processus d'une même machine virtuelle ou de différentes machines virtuelles peuvent communiquer à l'aide d'opérations. La communication entre deux processus est indépendante de leur localisation.

Les opérations peuvent être appelées de manière synchrone (`call`) ou asynchrone (`send`) et peuvent être servies par procédures (`procs`) ou par instructions d'entrées (`in`). En combinant l'appel et le service, nous obtenons quatre possibilités :

Appel	Service	Résultat
`call`	`proc`	appel de procédure (distant ou local)
`call`	`in`	rendez-vous
`send`	`proc`	création dynamique de processus
`send`	`in`	envoi de message asynchrone

SR sépare la déclaration des opérations de la manière dont elles seront utilisées.

Partage de ressources

Les processus d'une ressource peuvent partager des variables. Ils se synchronisent pour accéder aux variables à l'aide de sémaphores ou à l'aide d'autres opérations déclarées dans la ressource. Des processus dans différentes ressources mais situés dans la même machine virtuelle peuvent partager des variables ou des opérations définies de manière globale. Ce mécanisme est utile, par exemple, sur une machine multiprocesseurs à mémoire partagée.

Distribution

Dans SR, les machines virtuelles servent d'unité de répartition de programme. Elles peuvent être créées et détruites dynamiquement en fonction des besoins du programme. Des ressources et des variables globales peuvent être créées sur les machines virtuelles. Des processus de différentes machines virtuelles communiquent par le biais d'opérations. Les opérations peuvent être appelées indépendamment de l'endroit où elles sont définies. La demande d'une opération de la part d'un client à un serveur se fait de la même manière, indépendamment de l'endroit où se trouvent le client et le serveur (même machine virtuelle, même machine physique ou différente).

SR a été étendu pour introduire des processus légers (appelés Filaments) [Engler93]. Cette extension fonctionne actuellement avec des machines multiprocesseurs à mémoire partagée. Les processus légers de SR ne possèdent pas de pile privée. L'argument avancé en faveur de ce choix est que les threads avec pile consomment beaucoup de mémoire. SR offre trois types de processus légers :

- *run to completion* : le thread exécute un code et se termine.
- *barrier* : le thread s'exécute de manière répétitive avec une synchronisation basée sur une barrière. Il y a un test de détection de terminaison à la fin de chaque exécution.
- *fork/join* : ces processus légers créent de nouveaux threads et attendent un retour de résultat.

Les threads sont gérés dans les processus SR. Au lancement d'un processus, un serveur de processus légers est lancé. Ensuite, les processus légers sont lancés. Un processus léger est constitué d'un pointeur sur une fonction à exécuter et d'un ensemble d'arguments. Lorsque les processus légers initiaux sont créés, le serveur s'active. Le serveur s'arrête lorsque tous les threads ont terminé leur exécution. Les processus légers de SR ne sont pas préemptifs. Dans un processus, il y a au plus un processus léger actif à la fois. Si une activité veut accéder à une structure partagée qui est verrouillée, elle restera bloquée jusqu'à ce qu'elle arrive à verrouiller elle-même cette structure.

Le travail du serveur de threads consiste à parcourir continuellement les 3 queues de threads et à exécuter les activités de la queue la plus grande. Le placement des activités est spécifié par un paramètre de la fonction de création. Ce paramètre consiste en un numéro de serveur sur lequel sera créée l'activité.

I - 5•3 ADA

Ada [Ada83] est un langage qui permet de concevoir des applications concurrentes grâce aux tâches. Dans la version actuelle (Ada83) la programmation distribuée n'est pas abordée. Nous rappelons la notion de tâche du langage, ensuite nous présenterons une extension du langage pour l'écriture d'applications distribuées (STRAda).

I - 5•3•1 Les tâches ADA

Le langage ADA permet l'écriture de programmes parallèles à l'aide de tâches. Le mécanisme utilisé pour faire communiquer deux tâches est le rendez-vous ou les variables partagées.

Rendez-vous

Ada permet de décrire des activités parallèles appelées tâches, chacune étant exécutée

sur un processeur logique. Les tâches se déroulent indépendamment les unes des autres sauf en des points où elles se synchronisent.

Une tâche peut communiquer avec une autre en appelant un point d'entrée de la tâche destinatrice. Une tâche accepte un appel à une de ses entrées en exécutant l'instruction `accept`. La synchronisation est réalisée par rendez-vous entre une tâche faisant un appel d'entrée et une tâche acceptant l'appel. Les entrées peuvent avoir des paramètres ; ils servent de moyen de transmission de données entre tâches. Une tâche qui accepte un rendez-vous ne pourra se synchroniser avec une autre tâche qu'après avoir terminé le rendez-vous en cours.

```
TASK TYPE variable IS
     ENTRY ecrire(e : IN integer) ;
     ENTRY lire (e: OUT integer) ;
END ;

TASK BODY variable IS
BEGIN
     LOOP
          SELECT
               ACCEPT ecrire (e : IN integer) ;
               -- mise à jour de la variable
               OR ACCEPT lire (e : OUT integer) ;
               -- lecture de la variable
          END SELECT ;
     END LOOP ;
END variable;
```

figure 1.14 *Exemple ADA*

Variables partagées

Ada permet également à deux tâches de communiquer par variables partagées. Dans le manuel de référence, il est spécifié que les tâches utilisatrices d'une variable partagée doivent, entre deux points de synchronisation, utiliser la variable en écriture exclusive ou en lectures parallèles. Il faut en conclure que Ada ne spécifie pas de mécanisme de protection des données partagées entre tâches.

Ada introduit quand même une notion d'atomicité pour l'accès aux variables partagées. Le programmeur doit spécifier dans son programme quelles sont les variables qui sont partagées entre tâches (`pragma SHARED (nom_de_variable)`). Dans ce cas, toutes les tâches qui connaissent la variable partagée peuvent la lire ou l'écrire à tout moment. Cette opération n'est implémentée que pour des variables scalaires.

Ada permet l'écriture de programmes concurrents. Toutefois le problème de l'écriture de programmes distribués n'est pas abordé dans Ada. On suppose que le programme va

s'exécuter dans un seul processus. Le programme peut tourner sur une machine à multiprocesseurs, dans ce cas, les tâches pourront tourner en réel parallélisme sur les différents processeurs mais l'ensemble des données sera toujours dans un seul banc mémoire.

Nous présentons maintenant une extension de Ada83 qui permet l'écriture de programmes distribués.

I - 5•3•2 STRAda

Le projet STRAda [Bekele94] (Système de Transformation et de Répartition Ada) a été développé à l'IRIT (Toulouse). Le but de ce projet est la construction d'outils de répartition de programmes Ada. Les programmes répartis tournent sur un réseau de stations de travail homogènes.

Dans ce projet, l'accent est mis sur la réutilisation des programmes Ada existants. Le modèle de répartition est basé sur la notion de tâche Ada. L'écriture d'un programme réparti se fait en deux phases : dans la première phase, le programmeur se concentre sur les fonctionnalités et le parallélisme logique de son application , il obtient un programme distribuable ; dans la seconde phase, le programmeur s'occupe du placement des différentes tâches de son application ; le programme obtenu est appelé programme de configuration. Un changement de configuration ne nécessite que la modification du programme de configuration et laisse inchangé le programme distribuable.

La sémantique de Ada est quasiment conservée ; les différences sont dans la gestion des exceptions et dans l'instruction de sélection d'entrée avec délai. Les tâches qui s'exécutent sur différentes machines peuvent se référencer : le nom d'une tâche est global à l'application. Par contre le passage de pointeurs (type access) n'est réalisé que pour les types limités privés (autres que les tâches).

Le placement des tâches peut se faire de trois manières : interactivement, par programmation de la configuration ou réalisé automatiquement par le système.

Dans STRAda, la création d'une application répartie se fait à l'aide d'un transformateur de programme. Le programmeur conçoit son application en Ada de manière classique avec des tâches. Le transformateur prend le code source et génère un source ADA qui fait des appels à un noyau de répartition plutôt que d'utiliser les tâches du langage. Le noyau de répartition fournit les primitives concernant le parallélisme, la synchronisation, la communication, les exceptions, la terminaison des tâches et les variables partagées. Une tâche Ada est gérée par un processus lourd UNIX.[1] Les variables partagées sont implantées à l'aide d'une mémoire virtuelle partagée distribuée. Un algorithme de détection de terminaison a été écrit pour fonctionner dans un environnement distribué.

STRAda est un environnement intéressant car le modèle de distribution choisi (les

1. Dans un développement futur, il est envisagé d'utiliser des processus légers pour la gestion des tâches.

tâches) est le même que celui permettant de décrire la concurrence dans le langage. Cela facilite fortement la conception d'applications réparties sans utiliser des notions différentes pour exploiter le parallélisme et la distribution. Le problème de STRAda est qu'il n'a pas été possible de garder toute la sémantique du langage (qui n'a pas été conçu pour la distribution) et que cela handicape sa pleine utilisation.

I - 5•4 Linda

Linda [Carriero89][Carriero90] est un système pour construire des applications parallèles. La notion centrale dans Linda est le mécanisme de communication appelé Espace de Tuples (ET). L'espace de tuples peut être défini comme un ensemble non ordonné de données auxquelles on accède par association. Un élément dans cette mémoire est appelé un tuple.

Les processus sont créés dans un contexte d'ET. Ils utilisent cet espace pour se synchroniser et pour communiquer. Un espace de tuples peut être comparé à une mémoire virtuelle partagée.

Un tuple est composé d'un nom logique et de zéro ou plusieurs valeurs. Par exemple ("N", 10, vrai) est un tuple dont le nom est "N" et qui contient deux valeurs : 10 et un booléen à vrai. Quand un tuple est placé dans l'ET, les valeurs contenues ne peuvent plus changer.

Les opérations de base sont le dépôt et le retrait de tuples de l'espace de tuples.

I - 5•4•1 Dépôt

L'opération `out` dépose un tuple dans l'ET. Exemple : `out ("N", 10, vrai)`. Le programmeur peut spécifier des variables dans l'instruction de dépôt, dans ce cas, ce sont les valeurs des variables qui seront mises dans l'espace de tuples. Cette instruction est non bloquante.

I - 5•4•2 Retrait

L'opération `in` retire de l'espace un tuple qui correspond aux types des paramètres spécifiés dans l'instruction de retrait (`in`). Pour correspondre à un tuple, l'instruction `in` doit avoir le même nombre de paramètres et chaque paramètre doit avoir la même valeur qu'un tuple de l'espace. Les paramètres peuvent être réels ou formels. Les paramètres réels sont des valeurs littérales ou des variables.

Les paramètres formels sont représentés par un point d'interrogation suivi d'un nom de variable ou de type. Les paramètres formels correspondent automatiquement aux valeurs correspondantes dans le tuple. Si le paramètre formel est une variable, celle-ci est affectée avec la valeur correspondante du tuple. Par exemple si i est un entier et b un booléen, l'instruction `in ("N", ?i, ?b)` retirera le tuple appelé N qui a un entier comme premier argument suivi d'un booléen et affectera i et b avec les valeurs respectives du tuple. Si aucun tuple de l'ET ne correspond à l'instruction, le processus qui exécute cette opération sera bloqué jusqu'à l'arrivée d'un tuple correspondant.

L'exemple précédent permet de transmettre des données entre processus. Si les paramètres sont réels, l'opération `in` permet de synchroniser deux processus ; par exemple : `in ("N", 10, vrai)`. Le processus qui exécute cette instruction ne va pas affecter de variables dans son espace d'adressage mais va simplement attendre qu'un tel tuple soit placé dans l'ET avant de continuer son traitement.

I - 5•4•3 Consultation

L'instruction `in` retire le tuple de l'espace. Linda offre une opération qui permet simplement de consulter l'espace sans retirer le tuple qui correspond au format. C'est l'instruction `rd`. Cette opération fonctionne comme `in`.

Les opérations `rd` et `in` sont bloquantes si aucun tuple ne correspond au format de l'instruction. Linda offre deux primitives non bloquantes qui retournent un booléen pour signaler au processus si un tuple correspond au format, ce sont respectivement `rdp` et `inp`.

I - 5•4•4 Création de processus

La primitive Linda `eval` est utilisée pour créer de nouveaux processus. Par exemple `eval (func(args))` crée un nouveau processus pour exécuter `func` avec les arguments `args`. Lorsque cette fonction aura terminé son traitement, un tuple sera ajouté automatiquement avec comme valeur le résultat de la fonction.

Un espace de tuples n'est pas lié, en durée de vie, à un processus. L'espace peut continuer à vivre après la terminaison du processus qui l'a engendré. De même deux processus qui communiquent entre eux ne sont pas obligés de tourner simultanément. Le producteur de tuples peut être terminé au moment du lancement du processus qui va consommer les tuples créés (découplage temporel). Linda permet aussi un découplage spatial. Un processus n'a pas besoin de savoir où se trouve le processus avec lequel il communique.

SCA's (Scientific Computing Associates) Network Linda est une implémentation de Linda qui fonctionne sur un réseau de stations de travail hétérogènes. Les langages de programmation disponibles sont : C-Linda et C-fortran ; ce sont des langages compilés. La conversion de données entre différentes machines est assurée par XDR. L'espace de tuples est localisé sur un site. Une version pour machines à mémoire distribuée est en cours de réalisation (nom probable : Paradise).

Piranha Linda est le successeur de Linda. Cet environnement doit fonctionner sur un réseau de stations de travail et permettre la migration automatique des activités vers le site le moins chargé[1].

1. A l'instar des poissons du même nom, Piranha Linda veut dévorer les cycles CPU libres sur les noeuds de l'architecture parallèle.

I - 5•5 CSP et OCCAM

Communicating Sequential Processes (CSP) de Hoare [Hoare78] est un modèle de programmation parallèle basé sur la communication entre processus. La communication entre processus est accomplie par entrées-sorties synchrones à travers des canaux. OCCAM [INMOS88] est un langage basé sur le modèle CSP.

OCCAM permet au programmeur d'exprimer simplement la structure hiérarchique et modulaire de son application par encapsulation d'un ensemble de processus communicants vu comme un seul processus. Dans CSP et OCCAM, on trouve des entités de base qui sont le processus et le canal de communication.

Le processus

Le processus permet d'exprimer des actions. Ces actions peuvent effectuer du calcul ou communiquer avec d'autres processus à l'aide de canaux de communication.

Ces processus peuvent s'exécuter sur un même processeur, dans ce cas il y a partage des ressources CPU du processeur, ou peuvent être sur deux processeurs différents. Cette répartition sur les processeurs n'influe pas sur la phase de conception de l'application parallèle.

Dans OCCAM, il n'y a pas de notion de variable globale. Toutes les variables sont définies dans les processus.

Le canal de communication

Le canal de communication permet à deux processus de converser. Il sert également comme moyen de synchronisation. Un canal permet à deux processus de communiquer de manière unidirectionnelle. Pour que la communication soit réalisée, il faut qu'un des processus écrive dans le canal et que l'autre lise le canal. Cette communication est synchrone. Si l'un des deux processus n'est pas dans une phase de communication, l'autre processus qui a commencé la communication reste en attente.

OCCAM est un langage de processus communiquant de manière synchrone. C'est un langage de bas niveau qui ne manipule ni la généricité, ni l'abstraction de données. Il a été initialement conçu pour des machines à base de Transputers (possédant des liens physiques de communication vers les processeurs voisins).

Une version sur stations UNIX existe ; elle a été développée à l'université de Southampton : Southampton's Portable Occam Compiler (SPOC) [Debbage94]. SPOC est un compilateur pour le langage OCCAM2. Le système génère du code C. Ce code C tournera dans un seul processus UNIX ; c'est donc un système à pseudo-parallélisme. Une version distribuée pour un réseau de stations est en cours d'étude.

I - 5•6 Chant

Chant [Haines94] est une boîte à outils pour la conception d'applications distribuées à base de processus légers. Chant offre des primitives de communications point à point entre threads de processus différents et des RPC entre threads. L'objectif de Chant est la réutilisation des bibliothèques de communication et de processus légers existants aujourd'hui.

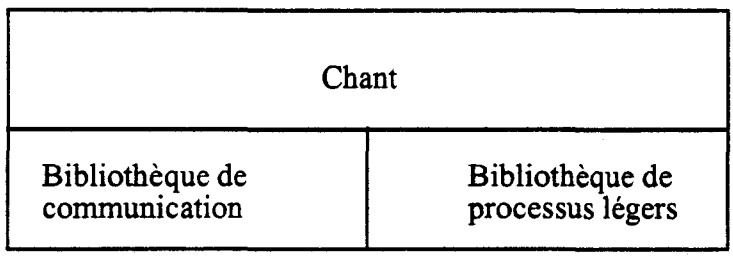


figure 1.15 *Architecture logicielle de Chant*

I - 5•6•1 Communication point à point entre threads

Chant fournit des primitives d'envoi et de réception de messages pour les processus légers. L'opération `send` crée un message et l'envoie sur le réseau à un destinataire. L'opération `receive` prend un message en provenance d'une source déterminée et le retire du réseau.

Pour permettre cette communication, Chant fournit un mécanisme de nommage global des processus légers.

I - 5•6•2 Appel de service à distance

Au-dessus de la communication point à point, nous trouvons un système d'appel de service à distance entre processus légers. Ce mécanisme est réalisé à l'aide d'un processus léger appelé serveur de processus (*server thread*) qui gère les demandes de service à distance. Quand une demande de service arrive au serveur, celui-ci la place dans la file d'attente du processus léger qui doit exécuter le service.

La création d'activité à distance est basée sur ce mécanisme. Un message est envoyé au serveur de processus de la machine distante et celui-ci crée la nouvelle activité demandée.

I - 5•6•3 Implémentation

Chant veut réutiliser les outils de communication et de processus légers existants. Actuellement, Chant fonctionne avec pthreads et Quickthreads comme bibliothèques de processus légers et NX et MPI comme bibliothèques de communication.

Pour être le plus ouvert possible, l'interface de programmation est une extension de l'interface POSIX de pthread.

L'intérêt de Chant est de permettre la programmation concurrente à base de processus légers en réel parallélisme. La notion de processus légers distribués est intéressante car elle permet une répartition de nombreuses activités concurrentes (grain fin), alors que cela n'est pas possible avec les processus (lourds) d'UNIX.

I - 5•7 Concurrent C et C++

Concurrent C [Gehani86] permet de créer des applications réparties. Une application est composée d'un ensemble de processus UNIX. Parmi les fonctionnalités de Concurrent C, nous trouvons : la déclaration et la création de processus, la synchronisation de processus, la terminaison et la possibilité de gérer des événements multiples.

Les interactions entre processus se font à l'aide de transactions[1]. Les transactions sont associées aux processus et peuvent être vues comme des services que les processus clients peuvent demander. Il y a deux types de transaction : synchrone et asynchrone. Dans le premier cas, l'appelant est bloqué jusqu'à la fin du service. Un résultat peut être fourni en retour. Dans le second cas, l'appelant est libéré immédiatement et peut continuer son exécution ; il n'y a pas de retour de résultat.

La déclaration des processus est séparée des aspects fonctionnels de l'application. Le traitement sur le code source est effectué à l'aide d'un préprocesseur.

Concurrent C++ [Gehani88] est une extension de Concurrent C au langage C++. L'implémentation est également réalisée à l'aide d'un préprocesseur. La manipulation de références n'est pas possible dans un contexte de machines à mémoire distribuée. Cela implique que le programmeur ne peut transférer que des éléments de type simple entre processus.

I - 6 Choix de conception

Des outils existent aujourd'hui pour faciliter la tâche du développeur d'applications réparties. Devant la diversité des outils disponibles, le programmeur est confronté à un certain nombre de choix :

I - 6•1 Processus vs Processus légers

Une application répartie va être composée de plusieurs processus UNIX qui vont communiquer. Si le système sous-jacent fournit une notion de processus légers (threads), le programmeur va devoir choisir s'il modélise son application en terme de processus

1. Dans le cadre de Concurrent C, le terme transaction n'a pas le sens des transactions dans un contexte de base de données. Ici, le terme transaction signifie interaction entre processus.

(gros grain) ou en terme de processus légers (grain fin). Ce choix va influer sur les performances de l'application et surtout sur la facilité de conception de l'application.

Une modélisation en terme de processus UNIX restreint le nombre d'activités parallèles qui tourneront dans l'application. En effet, il est difficilement imaginable de créer un très grand nombre de processus UNIX pour une application. Le programmeur est donc limité dans sa conception par les limites du système.

Une modélisation en terme de processus légers repousse la limite précédente. Malheureusement, peu d'environnements permettent complètement ce type de modélisation en offrant une communication point à point entre processus légers (Voir Chant plus haut, et PVC au chapitre VII).

Des environnements comme SR, STRADA, OCCAM introduisent une autre unité de répartition plus conceptuelle. Néanmoins la prise en charge de ces unités est faite le plus souvent par des processus lourds (avec de faibles performances) et très rarement par des processus légers.

I - 6•2 Placement explicite vs Placement implicite

Le placement des activités concurrentes peut être statique ou dynamique. Dans le premier cas, le programmeur spécifie dans son application l'endroit où vont s'exécuter ses activités. Avec la seconde solution, un mécanisme d'équilibrage de charge dynamique fait le choix automatiquement.

Une solution intermédiaire est envisageable : le programmeur spécifie un ou plusieurs sites dans son programme et, à l'exécution, l'équilibrage dynamique choisit le meilleur site pour l'exécution.

L'idéal est qu'une modification du placement des activités n'oblige pas le programmeur à modifier son programme.

Peu de systèmes permettent la migration dynamique de processus parce que c'est actuellement trop coûteux pour en tirer de réels bénéfices.

I - 6•3 RPC vs Communication par messages

Le programmeur dispose de deux principaux modèles outils pour faire communiquer les activités concurrentes.

Le RPC permet de concevoir une application répartie en cachant au programmeur l'aspect communication à distance. D'un point de vue programmation, le concepteur fait un appel d'une procédure, à la fin de l'exécution de la procédure, son programme continue. Tous les aspects communication et lancement d'exécutions distantes sont cachés. Pour aider le programmeur, il existe des outils de génération automatique de RPC.

La communication par messages permet de concevoir une application sur le modèle des processus communicants. Le programmeur manipule des messages qui vont passer

d'un processus à l'autre en utilisant éventuellement le réseau.

I - 6•4 Mémoire partagée vs Mémoire distribuée

Dans un système réparti, il n'y a pas de mémoire partagée. Cela peut influer sur la conception d'une application : un programmeur qui veut partager des données entre plusieurs activités va devoir programmer ce partage. Il peut utiliser la notion de serveur qui sert d'encapsulateur pour ses données. Cela implique un surcoût lors de la conception.

La notion de mémoire virtuelle partagée et distribuée est fortement étudiée. L'idée est de pouvoir accéder à des données partagées à partir de processus tournant sur des machines différentes qui n'ont pas de mémoire commune et ceci sans obliger le programmeur à utiliser des serveurs de données partagées. Ces travaux, bien qu'intéressants, sortent du cadre de cette thèse et ne seront pas abordés.

I - 6•5 Désignation globale ou non

Dans un modèle centralisé, où toutes les entités se trouvent dans le même espace d'adressage, les références sont uniformes : c'est une adresse mémoire. Dans un environnement distribué, une référence peut correspondre à une entité dans la mémoire du site local ou dans la mémoire d'un site distant. De ce fait, une référence composée simplement d'une adresse mémoire ne convient plus. En effet, une adresse correcte sur un site sera sans signification sur un autre site. Nous arrivons donc à deux types d'adresses : des adresses locales et des adresses distantes.

L'uniformisation des références peut se limiter aux processus. Dans ce cas, le passage par référence de données est interdit. Une uniformisation de toutes les références permet une programmation plus aisée pour le concepteur mais elle est difficile à réaliser dans les environnements actuels.

I - 6•6 Type de Synchronisation

La synchronisation d'activités concurrentes peut être réalisée à l'aide de plusieurs outils : par données partagées, par communication synchrone/asynchrone ou encore à l'aide de barrières.

I - 6•6•1 Synchronisation par données partagées

Les activités peuvent se synchroniser sur des données qu'elles ont en commun. Pour garantir la cohérence des données, il faut qu'elles soient protégées de manière logicielle, par exemple à l'aide de sémaphores.

La synchronisation par données partagées est simple à mettre en oeuvre mais oblige les activités concurrentes à travailler dans un espace mémoire commun.

I - 6•6•2 Synchronisation par communication synchrone

La communication synchrone permet à deux processus de se synchroniser. Les deux processus communiquent à l'aide d'un canal commun. Un des processus écrit dans le canal et l'autre processus lit le canal. Les deux processus reprennent leur traitement respectif à la fin de la phase de synchronisation. Si un des deux processus n'est pas prêt à se synchroniser alors que l'autre l'est, ce dernier reste bloqué.

I - 6•6•3 Synchronisation par communication asynchrone

Dans la synchronisation par communication asynchrone, le processus se bloque sur l'attente de l'arrivée d'un message particulier. Lorsque le message est arrivé, le processus continue son traitement. Ce type de synchronisation implique une gestion de files de messages, par exemple à l'aide de boîtes aux lettres.

I - 6•6•4 Synchronisation par barrière

La synchronisation par barrière permet à plusieurs processus (plus de deux) de se synchroniser. Ce mécanisme est intéressant pour des programmes utilisant le modèle SPMD.

I - 7 Conclusion

UNIX peut être vu comme un système d'exploitation réparti. Le programmeur peut concevoir des applications distribuées. Toutefois, ce travail reste difficile parce que les outils sont de bas niveau et donc difficiles à manipuler. L'utilisation du paradigme objet pour la conception d'applications réparties permet de simplifier ce travail : l'objet sert de vision structurante, d'unité de concurrence et de répartition de l'application. Nous présentons cette approche dans le chapitre suivant.

Chapitre - II -

Les Objets pour les applications réparties

Ces dernières années, les langages à objets ont connu un très grand essor, principalement dans le domaine de la programmation séquentielle. Les qualités de l'approche objet sont maintenant connues et reconnues.

On s'intéresse ici à la conception d'applications réparties avec une approche objet. Il est naturel d'imaginer une distribution des objets sur les différents sites de l'architecture répartie. Les objets vont communiquer entre eux par le réseau et vont permettre de créer et de synchroniser les activités parallèles de l'application.

Dans ce cadre, il est naturel de percevoir l'objet comme :

- l'unité de conception des programmes.
- l'unité de répartition.
- l'interlocuteur pour les communications et la synchronisation.

II - 1 Le Modèle Objet Traditionnel

Le modèle orienté objet traditionnel introduit un petit nombre de concepts. On peut le présenter simplement de la manière suivante:

- Le Modèle des Objets se fonde sur des objets créés dynamiquement à partir de classes, on dit qu'un objet est instance d'une classe.
- Une classe décrit la structure interne de ses instances ainsi que les méthodes applicables sur ses instances. L'appel d'une méthode d'un objet est un appel procédural exécuté dans un contexte limité à l'objet en question.
- Un objet référence d'autres objets. Il peut appeler les méthodes des objets qu'il référence.
- Une exécution ne comprend qu'un seul flot d'exécution correspondant à l'appel d'une méthode de création d'un objet racine.

Les intérêts de ce modèle sont connus:

- **modularité** : la structure et les méthodes d'accès sont regroupées dans une seule entité logique.
- **encapsulation** : l'utilisateur d'une classe n'a pas à connaître la structure interne ni l'implantation des méthodes.
- **réutilisabilité** : une classe n'est pas créée pour une application particulière mais bien au contraire peut être (ré)utilisée dans différentes applications.
- **typage** : une classe définit une implantation d'un type et des compilateurs peuvent exploiter ce fait pour vérifier ou optimiser le code.
- **sémantique** : l'accès aux objets utilise la sémantique classique de l'appel procédural, c'est une sémantique familière aux programmeurs, ce qui favorise leur productivité.

II - 2 Extensions possibles du modèle

II - 2•1 Objets et Processus

Le modèle traditionnel est essentiellement séquentiel puisqu'il n'introduit qu'un seul flot d'exécution. Une extension du modèle doit donc permettre de faire apparaître des objets et des flots d'exécution concurrents. Deux écoles s'affrontent: celle qui juxtapose les notions d'objets et d'activités, et celle qui encapsule les activités dans les objets.

Nous identifions ici les termes activité, flot d'exécution, processus ou processus léger.

II - 2•1•1 Activités hors des objets

Ici, les activités restent extérieures au modèle objet. Une activité n'est pas attachée à un objet particulier. Une activité exécute des méthodes sur des objets. Il y a donc deux unités de conception : l'objet pour les données, l'activité pour la concurrence.

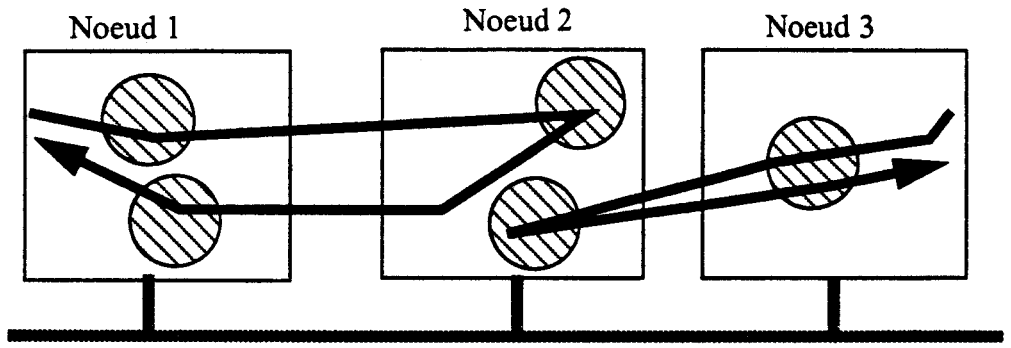


figure 2.1 *Application composée d'objets passifs et d'activités indépendantes*

II - 2•1•2 Activités dans les objets

La seconde possibilité est d'associer une activité à chaque objet. Le processus reste associé à l'objet durant toute la vie de l'objet. Lorsque l'objet est détruit, le processus associé l'est aussi. C'est ce qu'on appelle un objet actif . On peut l'assimiler à un serveur : lorsqu'un client invoque une opération sur un objet actif, celui-ci reçoit la requête et décide, par le comportement qu'il exécute, s'il traite la requête ou non. Tour à tour les objets actifs peuvent être clients et serveurs.

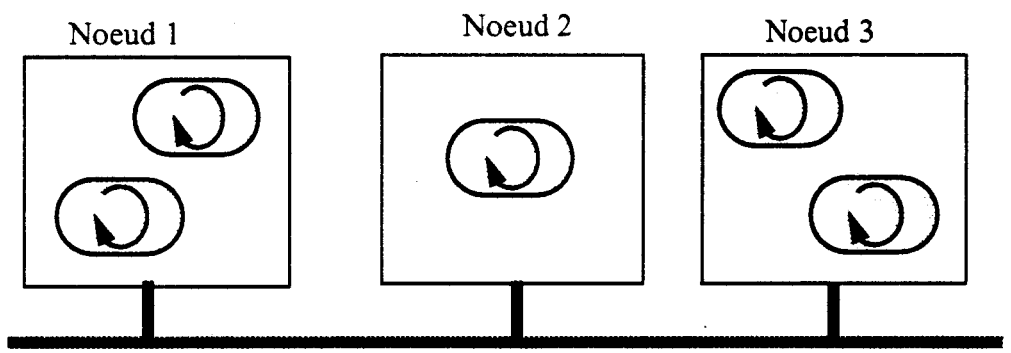


figure 2.2 *Application composée d'objets actifs*

Il est possible d'associer plusieurs activités à un objet, dans ce cas on parle d'objet multi-programmé par opposition à un objet qui ne possède qu'une activité (objet mono-programmé).

II - 2•1•3 Répartition

Un objet est toujours situé sur un site. Dans le modèle séquentiel, toutes les instances se trouvent dans le même processus sur la même machine. En modèle réparti, les objets sont sur des machines différentes. La désignation d'un objet à l'aide d'un pointeur en mémoire n'est généralement plus valable. Dans une application répartie, la désignation d'un objet doit se faire par des références étendues à toute l'application. Cette désignation étendue doit être cachée au programmeur pour simplifier la tâche de conception de son application : d'un point de vue programmation, il manipule des objets et ne sait pas si les objets sont locaux ou distants.

L'objet est donc une unité de répartition naturelle des données. Si les processus existent indépendamment des objets, il existera deux unités de répartition (l'objet et le processus), si les processus sont encapsulés dans des objets, une seule unité de répartition est utilisée.

Notons que les processus utilisent les objets et que donc dans tous les cas la répartition des objets va influencer la répartition des processus.

II - 2•2 Objets et Communication

Dans un modèle objet, les objets appellent des méthodes d'autres objets. En univers réparti ces appels peuvent être pris en charge par des envois de messages ou par des RPC.

II - 2•2•1 Communication par RPC

L'exécution de l'appel d'une méthode sur un objet se fait sur le site où se trouve l'objet. Si l'objet se trouve sur un site différent de l'appelant, la requête transite par le réseau et s'exécute sur le site de l'objet. Le retour du résultat empruntera alors le chemin inverse pour revenir à l'appelant. Pendant l'exécution de la procédure sur l'objet distant, le flot d'exécution dans l'objet appelant est suspendu jusqu'au retour de résultat.

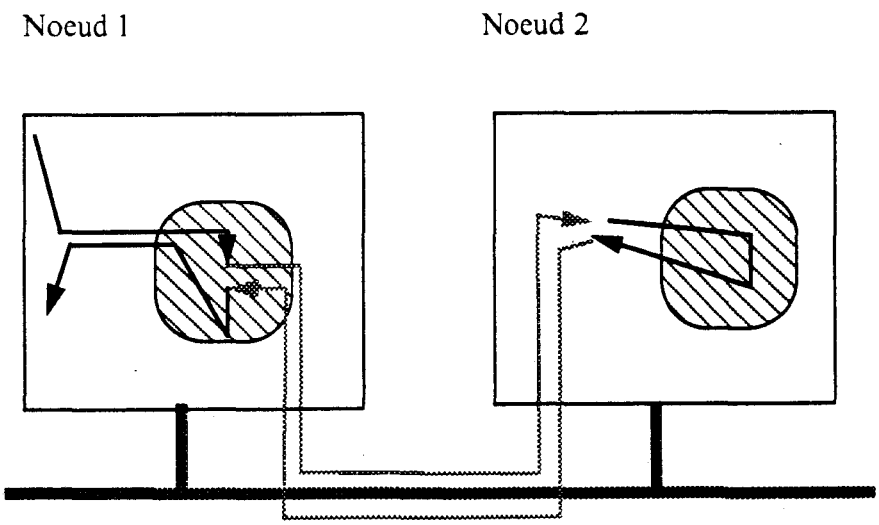


figure 2.3 *Communication entre objets par RPC*

L'exécution sur le site distant implique la création d'une activité temporaire sur ce site pour exécuter le code de la méthode. Du point de vue de l'appelant, c'est son processus qui exécute la procédure. Ce mécanisme est communément appelé *appel de méthode à distance.*

II - 2•2•2 Communication par messages

La communication par messages permet de déclencher des traitements à distance. Le message doit être reçu par une entité active qui effectue le traitement et qui peut retourner un résultat. C'est le cas des objets actifs.

Si des objets actifs communiquent par envois asynchrones de messages, cela induit que le modèle introduise des structures de stockage pour les messages (généralement des files de messages ou boîtes aux lettres). Les messages envoyés à un objet actif sont déposés dans une telle structure qui lui est attachée. L'objet peut explicitement la consulter et en extraire les messages. Cette structure de gestion de messages peut être manipulée explicitement par le programmeur ou peut lui être cachée.

II - 2•3 Objets et Synchronisation

Deux processus peuvent se retrouver simultanément dans un même objet. Une synchronisation est nécessaire pour maintenir l'objet dans un état cohérent. Dans une application répartie, une synchronisation est nécessaire pour contrôler le déroulement de l'application. Les objets peuvent aussi être utilisés pour ce type de synchronisation.

II - 2•3•1 Synchronisation à l'aide d'objets partagés

Le principe est ici que deux processus voulant se synchroniser devront se retrouver dans le même objet. L'utilisation de sémaphore, rendez-vous, ... dans le corps des méthodes permet de synchroniser les processus dans l'objet partagé par les processus, à la fois pour assurer le maintien de sa cohérence et pour ordonnancer les différentes exécutions.

Les sémaphores et autres étant des outils de bas niveau, il est possible d'attacher de la synchronisation aux objets sur la base des activations de méthodes. Un exemple est donné par les conditions d'activation (ou gardes) basées sur des compteurs de synchronisation.

Les compteurs de synchronisation[1] sont des informations associées à chaque méthode qui permettent de savoir le nombre total d'appels depuis l'initialisation de l'objet (`invoked (m)`), le nombre total d'exécutions terminées (`completed (m)`) et le nombre total d'appels commencés depuis l'initialisation (`started (m)`). Ces trois compteurs sont initialisés à zéro à la création de l'objet et ne peuvent qu'augmenter au cours du temps.

1. Nous reprenons ici le formalisme du langage Guide [Nguyen90]

A l'aide de ces 3 compteurs, nous pouvons facilement déduire d'autres informations :

- le nombre de requêtes en cours d'exécution (`current`) :
  `started (m) - completed (m)`
- le nombre de requêtes en attente (`pending`) :
  `invoked (m) - started (m)`

La synchronisation pour une classe d'objets tampon s'écrit donc :

```
METHOD ecrire(e : ELEMENT) IS ... END ;
METHOD lire (e: ELEMENT) IS ... END ;
CONTROL
    ecrire : (completed (ecrire) - completed (lire) <
size) and current (ecrire) = 0 ;
    lire : (completed (ecrire) > completed (lire)) and
current (lire) = 0 ;
```

La méthode *écrire* peut être exécutée s'il y a de la place dans le tampon et s'il n'y a pas d'écriture en cours. La méthode *lire* ne peut être exécutée que s'il y a eu plus de dépôts que de retraits et s'il n'y a pas de lecture en cours.

II - 2•3•2 Synchronisation à l'aide d'objets actifs

Dans le cadre des objets actifs, un objet encapsule une activité qui représente le comportement de l'objet vis-à-vis des sollicitations externes. En fonction de l'état courant de l'objet, le rôle d'un comportement est aussi de décider de traiter un appel de méthode ou de le laisser en attente. Inversement, le comportement peut se mettre en attente d'un message particulier. Un objet actif est donc naturellement un lieu de synchronisation.

Dans un tel schéma, les requêtes en attente sont stockées par l'objet dans une structure particulière souvent appelée boîte aux lettres. La synchronisation est implantée par des opérations qui examinent cette boîte pour en extraire un message particulier. Nous appelons cette synchronisation "synchronisation sur boîte aux lettres".

II - 2•4 Conclusion

Comme on vient de le voir, l'extension du modèle objet à la concurrence et à la distribution laisse une grande place au choix :

- Objets et processus vs objets actifs
- Communication par message ou appel de méthode à distance
- Synchronisation par objets partagés ou sur boîte aux lettres

Chaque langage à objets pour applications réparties doit se positionner par rapport à ces choix. Cela donne différents modèles de programmation que nous classifions dans la section suivante. Nous donnons ensuite quelques exemples de tels langages.

II - 3 Modèles de langages parallèles à objets

A la vue des fonctionnalités présentées ci-dessus, un certain nombre de modèles de langages parallèles à objets peuvent être mis en évidence.

II - 3•1 Classification

Pour les classer, nous pouvons mettre en opposition (figure 2.4) :

- objets + processus vs objets actifs
- communication par messages ou appels de méthodes à distance
- synchronisation par objets partagés ou sur boîte aux lettres.

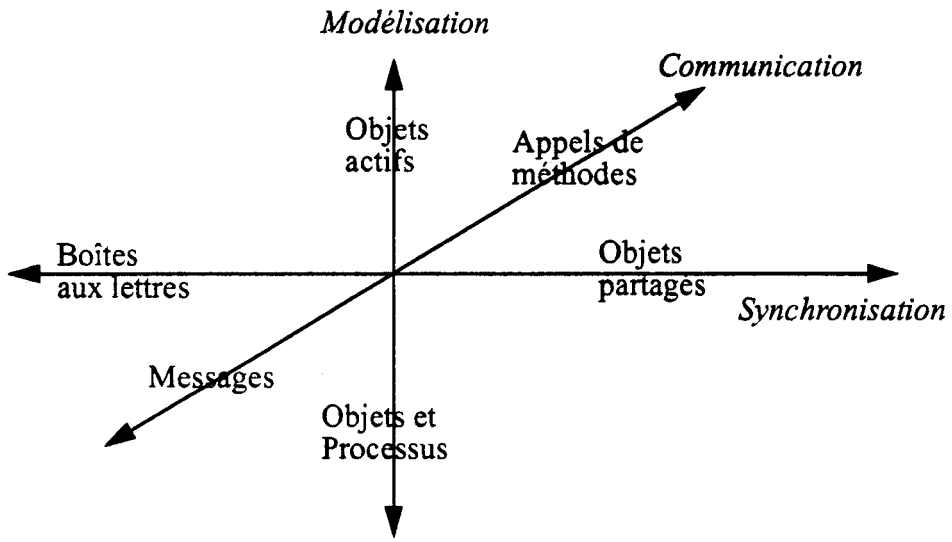


figure 2.4 *Concepts opposés*

II - 3•2 Le modèle Objets et Processus

Dans ce modèle, les objets sont passifs et sont traversés par des activités au gré des appels de méthodes. La synchronisation des activités est assurée par des objets partagés.

Ces objets possèdent une description de leur synchronisation.

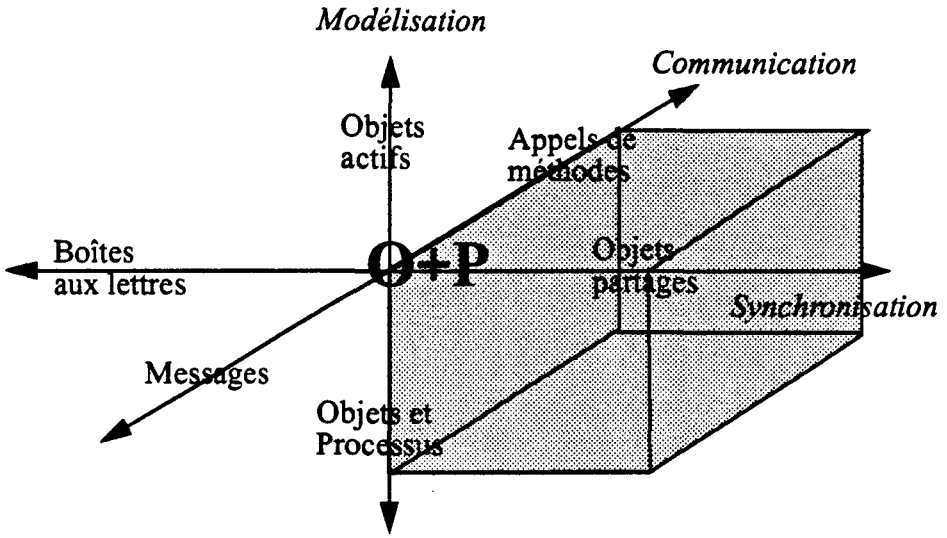


figure 2.5 *Le modèle Objets et Processus*

Dans ce modèle, l'élément principal est l'objet. L'activité est vue comme étant *secondaire*. Si nous utilisons la terminologie de Guide [Krakowiak90], une activité agit sur un ensemble d'objets, appelé domaine. Un objet peut appartenir à plusieurs domaines donc plusieurs activités peuvent le traverser simultanément. Il doit donc être synchronisé.

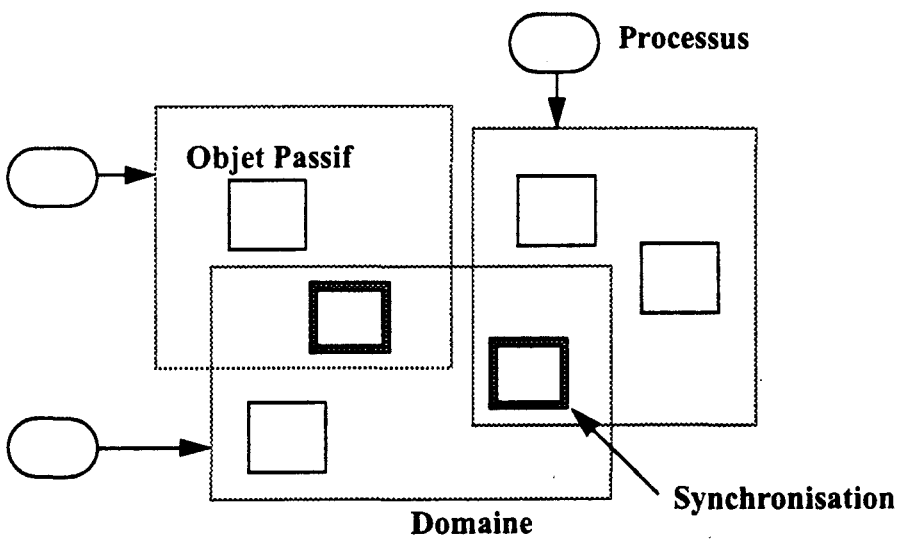


figure 2.6 *Exécution d'une application définie dans le modèle Objets+Processus*

II - 3•3 Le modèle Objets Actifs

Dans ce modèle, les objets sont actifs : ils possèdent tous une activité et sont donc autonomes. Deux grands axes ont émergé de ce modèle : un modèle d'acteurs et un modèle d'objets actifs.

II - 3•3•1 Modèle d'acteurs

Dans ce modèle, tous les objets sont actifs (figure 2.7). Ils communiquent entre-eux par envois de messages asynchrones. La synchronisation s'exprime dans la programmation des comportements en se basant sur l'examen des boîtes aux lettres.

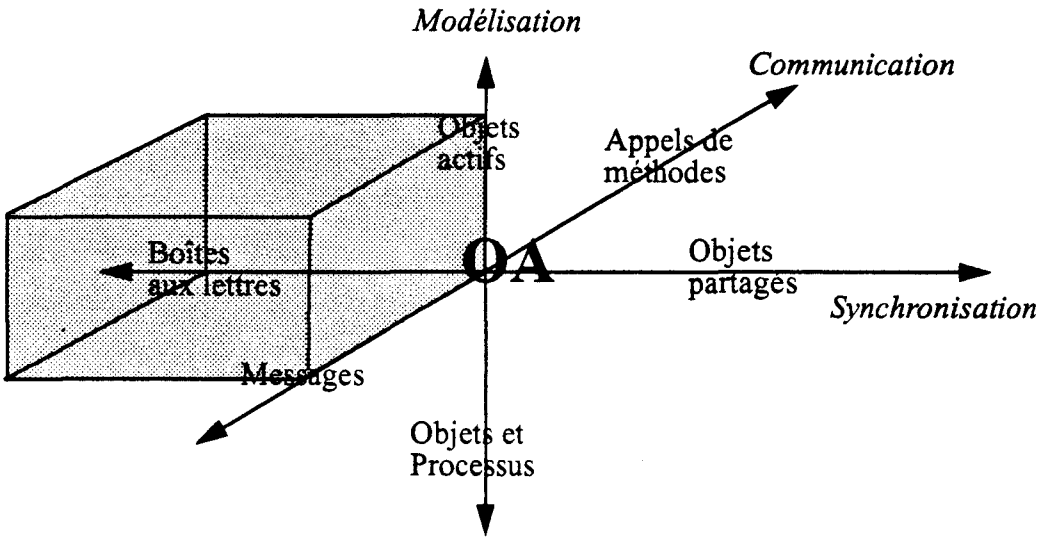


figure 2.7 *Le modèle d'acteurs*

II - 3•3•2 Modèle d'objets actifs

Dans le modèle des objets actifs, les objets communiquent par appels de méthodes. Les appels sont mis en attente à l'entrée de l'objet et les traitements de ces appels sont ordonnés par le processus interne en fonction de l'évolution de l'état de l'objet. De cette manière un objet actif peut être soumis à des sollicitations simultanées - un objet actif est un objet partagé. La programmation de la synchronisation est réalisée par la programmation du comportement de l'objet actif. On peut considérer ici que la synchronisation est à la fois du type boîte aux lettres (stockage des appels) et du type objets partagés.

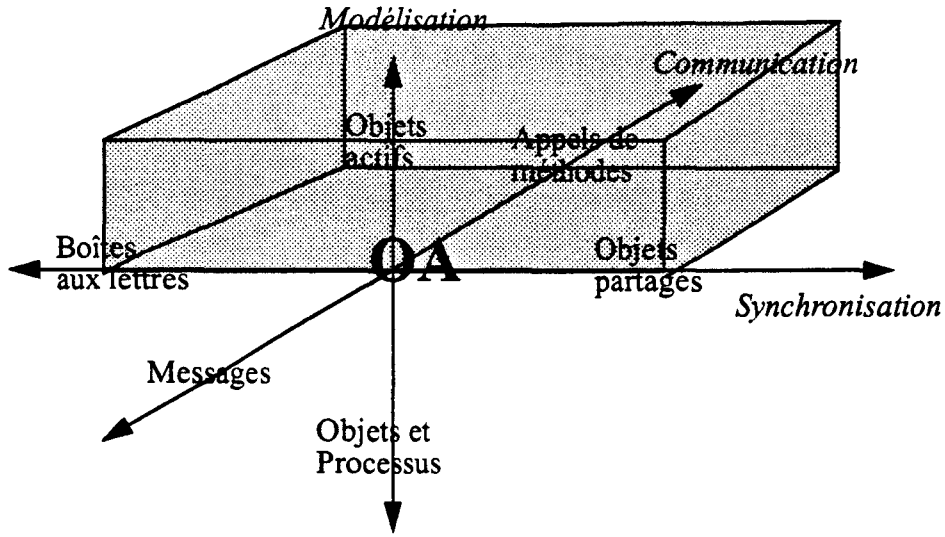


figure 2.8 *Le modèle des Objets Actifs*

Dans une variante de ce modèle, certains objets sont actifs et d'autres restent passifs (figure 2.9). Dans la phase de conception, un objet qui peut être partagé est hissé au rang d'objet actif. Les autres restent passifs. Un objet passif n'est donc plus partagé et ne sera accessible que par un seul objet actif.

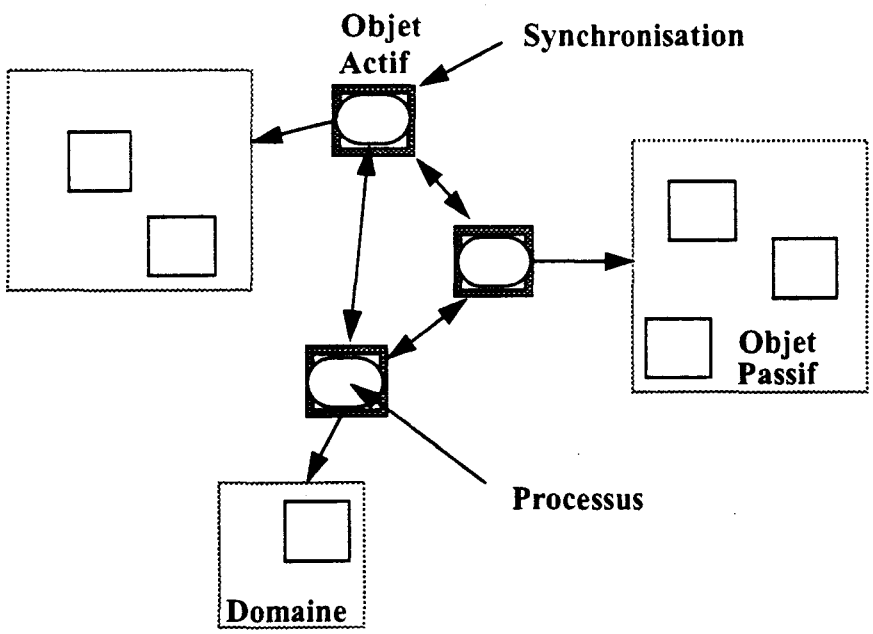


figure 2.9 *Exécution d'une application définie dans le modèle Objets Actifs*

On remarquera que le schéma d'exécution du modèle des objets actifs est dual de celui

du modèle objets et processus. En effet dans le premier cas, les domaines sont disjoints et l'effort de la conception se situe dans les processus des objets actifs qui organisent toute l'exécution, alors que dans le deuxième cas, les domaines ont des intersections non vides et l'effort de la conception doit porter sur les objets passifs aux intersections. Cette remarque montre toute la diversité parfois insoupçonné de la programmation parallèle à objets.

Nous illustrons cette classification par quelques langages parallèles à objets de la littérature.

II - 4 Synthèse des environnements distribués à objets

Dans cette section, nous allons survoler un certain nombre de systèmes distribués à objets disponibles aujourd'hui. Nous allons tenter de les rattacher à un modèle présenté ci-dessus, de donner une description de leurs caractéristiques. Nous essayons de donner des informations concernant :

- les architectures visées
  Type de machines sur lesquelles vont tourner les réalisations.
- le modèle utilisé :
  Objets+Processus, Acteurs ou Objets Actifs.
- l'implémentation réalisée :
  Nouveau langage, extension de langage ou bibliothèques spécifiques.

II - 4•1 ABCL/1

ABCL/1 (An object Based Concurrent Language) [Yonezawa86b][Yonezawa87] [Yonezawa90] est un langage d'acteurs développé par l'équipe de Yonezawa.

Architectures visées : stations de travail

Modèle utilisé : Acteurs

Implémentation : nouveau langage

Commentaires

Tous les objets sont actifs ; lors de la création d'une instance, un processus est créé et exécute le code d'un *script*. Les objets ABCL/1 sont mono-programmés. Les requêtes peuvent être synchrones ou asynchrones.

Dans ce langage nous trouvons deux modes pour les messages : le mode *ordinary* et le mode *express*. Il y a donc une notion de priorité sur les messages.

Emission :

Soit O un objet et M un message :
Envoi asynchrone (past) :
L'envoi du message à l'objet se fait respectivement en mode ordinaire et en mode

express : `[O <= M]` et `[O <<= M]`. L'objet qui exécute cette instruction continue son code juste après sans s'inquiéter des traitements déclenchés dans O.

Envoi synchrone (now) :
L'envoi du message à l'objet se fait respectivement en mode ordinaire et en mode express : `[résultat := [O <== M]]` et `[résultat := [O <<== M]]`. Cet appel sert lorsque l'utilisateur veut immédiatement attendre le résultat de sa requête sur O. Dans le cas de l'exemple, le résultat sera affecté à *résultat*. C'est similaire à un appel de procédure.

Envoi asynchrone avec récupération de résultat (*future*) :
L'envoi du message à l'objet se fait respectivement en mode ordinaire et en mode express : `[O <= M $ résultat]` et `[O <<= M $ résultat]`. Dans ce dernier cas, l'objet qui exécute cette instruction continue son code immédiatement après l'appel. Le résultat sera placé dans la variable *résultat*. Plus tard, l'objet pourra consulter le résultat dans la variable. Il est possible de vérifier si le résultat est arrivé ou pas sans pour autant se bloquer.

Dans chacun des cas, l'émetteur est implicitement transmis en plus du contenu du message.

Réception :
La réception se fait dans un *script* qui correspond au comportement de l'objet.

```
[object nom-de-l-objet
(state representation-de-l-objet)
(script
(=> message-patern where contrainte ... action ... )
(=> message-patern where contrainte ... action ... )
)]
```

`Message-patern` va servir à sélectionner l'action à exécuter. Il est possible d'ajouter des contraintes à la réception : par exemple n'accepter un message qu'en provenance d'un objet bien particulier (ex : `where (= &sender Un-objet-particulier)`). Si un message correspond à plusieurs paterns, le système choisit le premier de la liste en partant du début du script.

Les messages prioritaires (express) peuvent être vus comme des interruptions. La programmation d'une classe devient plus difficile si, en plus du problème à modéliser, il faut penser aux problèmes d'interruptions. Et en définitive, on peut se demander pourquoi avoir deux niveaux de messages et non pas trois ou plus.

II - 4•2 ACOOL

ACOOL (Asynchronous Concurrent Object-Oriented Language) [Maruyama94] a été réalisé par Maruyama aux laboratoires NTT au japon.

Architectures visées : réseau de stations de travail

Modèle utilisé : Objets Actifs

Implémentation : Nouveau langage

Commentaires

ACOOL inclut des objets passifs et des objets actifs. Les appels sur les objets passifs se font de manière synchrone. Les appels sur les objets actifs se font de manière asynchrone. Toutefois lorsqu'un appelant demande l'exécution d'une méthode qui retourne un résultat à un objet actif, il (l'appelant) reste bloqué jusqu'à la fin de l'exécution de la méthode par l'objet actif. La synchronisation entre activités se fait par envoi de messages. Actuellement, il n'y a pas de synchronisation sur les objets passifs. Il est prévu une implémentation des moniteurs de Hoare comme synchronisation sur les objets passifs.

ACOOL utilise un serveur de noms pour enregistrer les objets actifs. Les informations contenues dans ce serveur sont de la forme <référence globale de l'objet actif, nom symbolique>. Ce serveur de noms permet de connaître des objets actifs dans l'application par leur noms symboliques et de leur envoyer des messages pour qu'ils travaillent en parallèle.

II - 4•3 Act3

Les acteurs ont été inventés par Hewitt [Hewitt73]. Ensuite Agha [Agha86][Agha87] a beaucoup contribué au développement des recherches sur les acteurs.

Architectures visées : réseau de machines lisp

Modèle utilisé : Acteurs

Implémentation : en lisp

Commentaires

Les travaux sur Act1 [Lieberman81], Act2 et Act3 [Agha88] sont à la genèse des langages d'acteurs que nous connaissons aujourd'hui. Dans Act3, nous retrouvons tous les concepts présentés ci-dessus concernant les acteurs, à savoir communication par messages entre acteurs, création de nouveaux acteurs et remplacement du comportement. Rappelons que les acteurs ne contiennent pas de variables locales.

De nombreux travaux théoriques ont été réalisés sur la sémantique des acteurs pour permettre l'écriture de preuves formelles des applications écrites à l'aide d'acteurs.

II - 4•4 Actalk

Actalk [Briot89a][Briot89b] est la contraction de Acteur et de Smalltalk [Goldberg83]. Ce langage a été créé par J.P. Briot au LITP.

Architectures visées : station de travail (simulation), réseau de transputers

Modèle utilisé : Acteurs

Implémentation : extension de Smalltalk-80

Commentaires

Le noyau minimal de Actalk [Briot94] est composé de trois classes : `ActiveObject`, `Activity` et `Address`.

- La classe `ActiveObject` représente le comportement interne d'un objet actif c'est-à-dire celui qui traite les messages et donne ainsi la sémantique à l'objet actif. Le programmeur peut définir de nouvelles classes par sous-classement de la classe `ActiveObject`.
- La classe `Activity` représente l'activité interne d'un objet actif. Elle permet l'allocation d'un processus pour l'objet. Elle décrit également la manière dont les messages sont sélectionnés, séquencés et activés.
- La classe `Address` représente l'adresse de l'objet actif (identificateur et boîte aux lettres pour les messages de l'objet). Cette classe définit la manière dont la réception des messages sera interprétée.

Cette décomposition permet le découplage du programme de l'utilisateur du modèle d'activité ainsi que de la spécification de la synchronisation. Il permet également d'associer plusieurs comportements à une même adresse (et ainsi avoir plusieurs comportements distincts pour traiter simultanément plusieurs messages).

Ces classes définissent la sémantique par défaut des objets actifs :

- activité : l'activité est sérialisée et réactive, l'objet traite un message à la fois et l'activité est activée uniquement par transmission de message.
- communication : la communication est asynchrone et unidirectionnelle, l'émetteur d'un message n'est pas bloqué après l'envoi d'un message même si celui-ci n'est pas traité immédiatement. Comme l'envoi est unidirectionnel, il n'y a pas de valeur retournée à l'émetteur du message.

En Actalk, un objet a deux références sur lui-même : `self` et `aself`. `Self` est la référence traditionnelle de Smalltalk. `Aself` permet d'envoyer un message à la propre adresse de l'objet actif. Lorsqu'un objet transmet une référence vers lui-même à un autre objet, il doit utiliser `aself` au lieu de `self`. S'il ne le fait pas, les messages qui lui arriveront ne passeront plus par la synchronisation et risquent de laisser l'objet dans un état incohérent.

Autres travaux sur Actalk

Actalk permet de modéliser différents langages de programmation parallèle : modèle de calcul des acteurs, ABCL/1 [Yonezawa86], POOL, Concurrent Eiffel ou encore ConcurrentSmalltalk.

Différentes stratégies de synchronisation ont été évaluées avec Actalk : par transitions d'états, gardes, suspension/attente, compteurs de synchronisation.

Une version réflexive de Actalk a été réalisée : ReActalk [Giroux91].

Actalk est une bonne plate-forme d'expérimentation de l'expression du parallélisme dans les langages à objets. Il est facile d'essayer de nouveaux mécanismes de synchronisation et de représenter le comportement d'autres langages parallèles en Actalk. Une version distribuée a été réalisée en utilisant GnuSmalltalk sur une machine à base de transputers.

II - 4•5 Charm ++

Charm++ [Kale93] est un langage parallèle orienté objet basé sur C++. Il a été développé à l'université de l'Illinois, Urbana-Champaign par Kale.

Architectures visées : réseau de stations de travail, CM5, Paragon, SP-1

Modèle utilisé : Objets Actifs

Implémentation : translateur Charm++ vers C++ et bibliothèque

Commentaires

Ce langage est une extension de C++. Le modèle d'exécution est basé sur la communication par messages. Le runtime s'appelle Charm [Kale93b]. Il peut fonctionner sur des machines à mémoire partagée ou à mémoire distribuée. Les classes d'objets actifs sont conçues par sous-classement de la classe `chare`. Charm++ permet également l'utilisation d'objets partagés en plus de la communication par messages. Les concepteurs considèrent que l'utilisation de la communication par messages est trop restrictive. Les objets partagés sont initialisés au lancement de l'application et sont répliqués sur l'ensemble des sites de la machine. Différentes versions existent : objets en lecture seule, objets qui peuvent être initialisés avec le résultat d'un traitement, objets monotones : la valeur peut être mise à jour et sera propagée sur l'ensemble des sites. L'utilisation des objets partagés permet une simplification de l'écriture des composants logiciels mais ne permet pas à deux activités de se synchroniser sur un objet commun.

II - 4•6 Concurrent Smalltalk

Architectures visées : station de travail

Modèle utilisé : Objets+Processus

Implémentation : extension de Smalltalk

Commentaires

Concurrent Smalltalk [Yokote86][Yokote87] étend Smalltalk-80 en introduisant l'appel asynchrone et un mécanisme de boîte aux lettres (`CBox`). Un appel asynchrone crée une activité pour exécuter une méthode sur le receveur. L'appelant n'est pas bloqué. Le résultat peut être récupéré par l'appelant dans une boîte aux lettres (`CBox`). Le résultat peut être retourné à l'appelant de manière asynchrone c'est-à-dire que l'activité

peut continuer l'exécution de la méthode en parallèle (concept de *future* de ABCL/1).

II - 4•7 Distributed Smalltalk

Distributed Smalltalk [Bennet87][Bennet90] est, comme son nom l'indique, une version distribuée de l'environnement Smalltalk.

Architectures visées : réseau de stations de travail

Modèle utilisé : Objets+Processus

Implémentation : extension de Smalltalk

Commentaires

Le but est de faire coopérer plusieurs Smalltalk sur un réseau sans modifier la machine virtuelle. Les différentes versions de Smalltalk restent indépendantes. Une classe et toutes ses instances sont localisées sur le même site.

Un objet peut désigner un autre objet qui est sur un site différent. Pour manipuler un objet distant, l'objet client fait appel à un objet *proxy* [Decouchant86]. Celui-ci représente l'objet distant sur le site local. Lors de la création d'un objet distant, le système client envoie un message de création au site distant. Celui-ci retourne le nom de l'instance créée sur le site distant au système client. Le système local crée alors un objet proxy contenant le nom de l'objet distant et le nom du site propriétaire de l'objet.

L'objet proxy peut être utilisé comme un objet local. Tous les messages envoyés au proxy sont transformés en messages réseau réels et sont redirigés vers le site propriétaire de l'objet. L'objet proxy masque donc le réseau. La synchronisation est assurée par des classes Smalltalk.

II - 4•8 Eiffel //

Eiffel Parallèle est une extension du langage Eiffel [Meyer88] au parallélisme. Cette extension a été réalisée par Caromel.

Architectures visées : réseau de stations de travail

Modèle utilisé : Objets Actifs

Implémentation : extension du langage Eiffel (version 2)

Commentaires

Pour définir une classe d'objets actifs, la nouvelle classe doit hériter de la classe PROCESS ou d'une de ses sous-classes [Caromel90][Caromel90b]. Une routine Live est invoquée sur l'objet et un nouveau processus l'exécute.

Les objets passifs sont gérés par un seul objet actif et ne sont pas partagés.

Bien que cette extension n'utilise pas explicitement de communication par message, elle introduit une notion intéressante : l'attente par nécessité [Caromel89]. Le principe est simple : l'attente du résultat d'un appel de routine peut être retardée jusqu'à la

première utilisation du résultat en lecture.

Par exemple :

```
r := o.r (...) ; -- appel asynchrone de la routine r sur
l'objet o
...
<utilisation de r>
```

Au moment de l'appel de la routine r, la variable r est considérée comme une variable attendue c.-à-d. non présente. Le processus n'est bloqué que lorsque la variable r est utilisée et qu'elle est attendue. Lorsque le résultat est retourné, la variable devient présente et le processus est débloqué. Dans cet environnement, il est également possible de tester si le résultat est arrivé sans pour autant être bloqué.

Dans Eiffel //, lorsqu'un objet actif est créé, une routine `Live` est exécutée. Selon l'état de l'objet, le code de cette routine autorise ou non l'exécution de requêtes en provenance des clients de l'objet [Caromel91][Caromel94]. La version la plus simple du comportement est l'exécution des requêtes dans leur ordre d'arrivée (`serve_oldest`). Il est également possible de traiter les requêtes dans un ordre différent de celui d'arrivée. Pour cela, on utilise des primitives comme `serve_old (r)` qui va traiter la plus ancienne requête pour la routine r.

II - 4•9 GARF

GARF [Guerraoui94][Guerraoui94b] (Génération Automatique d'Applications Résistantes aux Fautes) est également une extension de Smalltalk. Elle a été réalisée par Guerraoui à l'EPFL.

Architectures visées : réseau de stations de travail

Modèle utilisé : Objets+Processus

Implémentation : nouvelles classes Smalltalk

Commentaires

Dans GARF, on part d'objets classiques séquentiels (appelés objets *données*) auxquels on associe des objets comportementaux. Les objets comportementaux servent à transmettre les messages d'un site à l'autre de manière transparente pour le client. Ils sont aussi chargés de la gestion du parallélisme. Un objet comportemental est constitué d'une partie se trouvant sur le site du client (le messager) et d'une partie se trouvant sur le site du serveur (l'encapsulateur). L'encapsulateur a une désignation inter-sites que le messager correspondant est seul à connaître. Tous les messages reçus ou émis par un objet *données* sont interceptés, réifiés et envoyés à l'une ou l'autre des parties de l'objet comportemental.

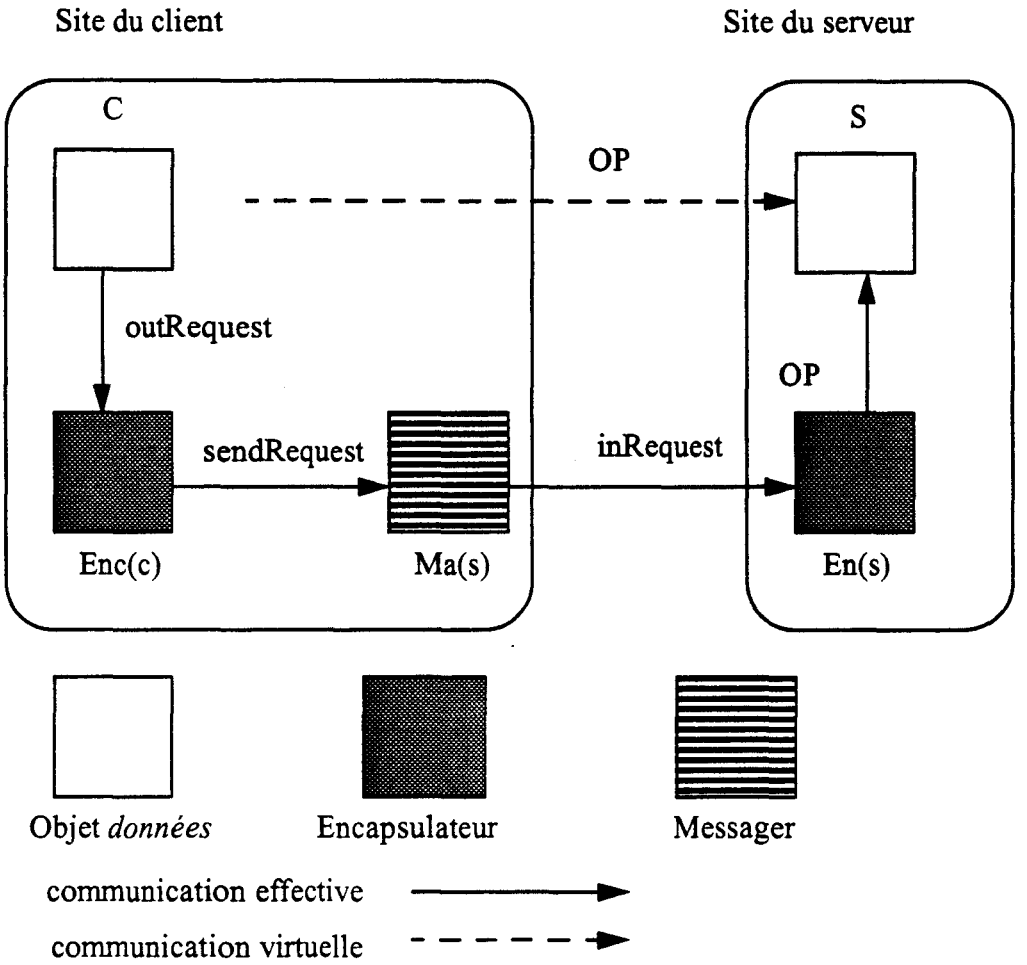


figure 2.10 *Mécanisme d'invocation dans GARF*

L'invocation d'une opération OP sur un objet serveur S par un client C est automatiquement interceptée par le run-time GARF. L'invocation se déroule en plusieurs points :

- invocation de outRequest sur l'encapsulateur Enc(c) du client c en passant le messager Ma(s)
- transmission de l'invocation OP au messager Ma(s) par sendRequest
- transmission de la requête à En(s) par le messager Ma(s)
- invocation de l'opération OP sur S par l'encapsulateur En(s)

Une fois l'exécution de la méthode OP terminée, son résultat revient à l'objet client par le chemin inverse (réponse transmise en cascade, par retour de méthode). Le comportement de l'objet est déterminé par les méthodes outRequest, sendRequest et inRequest.

Dans GARF, les activités sont cachées dans les objets comportementaux. Le

comportement par défaut d'un objet *données* est le même que dans le modèle séquentiel : l'invocation est synchrone. Des classes d'objets comportementaux permettent de lever cette restriction : les classes AsyncEncaps et AsyncMailer.

- la classe AsyncEncap permet au client de lancer ses requêtes de manière asynchrone. A chaque invocation de méthode, l'encapsulateur lance un processus léger Smalltalk qui va faire l'invocation au messager du serveur et se bloque en attente de la réponse. En cas de retour de résultat, celui-ci sera placé dans un objet *futur* (cf section sur la synchronisation). Ce comportement est réalisé par redéfinition de la méthode outRequest.
- la classe AsyncMailer permet au serveur d'être non bloquant pour les clients qui l'invoquent. Ce comportement est réalisé par redéfinition de la méthode sendRequest. S'il y a retour de résultat, celui-ci est manipulé à l'aide d'objets *futur*.

La synchronisation et la protection de l'intégrité des objets sont réalisées grâce à l'encapsulateur. Deux classes sont fournies pour assurer le contrôle de la concurrence intra-objet :

- la classe MonitorEncaps permet d'avoir un accès exclusif sur l'objet.
- la classe ReaderWriterEncaps permet d'avoir un comportement lecteurs/rédacteurs sur un objet. Pour utiliser cette classe, il faut fournir une liste de sélecteurs de messages qui représentent les opérations qui doivent se faire en exclusion mutuelle (les rédacteurs). Cette liste permet à l'encapsulateur de distinguer l'arrivée d'un lecteur ou d'un rédacteur et de choisir le traitement à effectuer.

Dans le cas où une opération implique plusieurs objets et où l'on exige que cette opération soit atomique, on peut utiliser le concept de transaction. Ce mécanisme de synchronisation est réalisé par affinement de la classe MonitorEncaps en ExtendedMonitorEncaps. Pour chaque invocation, l'exclusion mutuelle est garantie sur l'objet mais également sur tous les objets accédés pendant l'exécution de la méthode invoquée.

GARF introduit un mécanisme similaire. Au niveau des objets données, les futurs sont implicites tandis qu'ils sont manipulés au niveau des objets comportementaux.

II - 4•10 Gothic

Gothic est un système développé à l'IRISA à Rennes par l'équipe de J.P. et M. Banâtre.

Architectures visées : machines Bull SPS7 multiprocesseurs

Modèle utilisé : Objets Fragmentés

Implémentation : nouveau langage

Commentaires

La construction de programmes parallèles en Gothic est réalisée à l'aide de classes. La représentation d'un objet peut être centralisée (située sur un seul noeud) ou fragmentée (répartie sur plusieurs noeuds). La programmation en Gothic est essentiellement fondée sur l'utilisation du concept de multiprocédures [Banâtre91]. Cette notion est liée à la fragmentation des objets. Des multiprocédures peuvent être lancées sur des objets fragmentés, ceci afin d'appliquer des traitements en parallèle sur les différents fragments. Les traitements lancés en parallèle peuvent être différents en fonction des fragments.

Les multiprocédures sont le seul moyen de générer du parallélisme. Les appels sont synchrones : l'appelant d'une multiprocédure est bloqué jusqu'à la fin de l'exécution de celle-ci. La règle de synchronisation est simple : deux méthodes s'exécutant sur des fragments différents peuvent être exécutées en parallèle ; par contre deux méthodes se partageant un même objet seront exécutées en exclusion mutuelle.

Arche [Benveniste93] est le langage orienté objet qui permet d'exploiter le système Gothic. Arche est un langage compilé fortement typé. Le langage permet de décrire des objets fragmentés et de leur appliquer des multiprocédures. La synchronisation est décrite à l'aide d'états de synchronisation (*synchronization-state*) qui spécifient un ensemble de méthodes qui peuvent être appliquées sur l'objet lorsqu'il est dans un état particulier. Ce mécanisme n'interfère pas avec le mécanisme d'héritage (voir POOL pour la présentation du problème).

II - 4•11 Guide

Architectures visées : réseau de stations de travail

Modèle utilisé : Objets+Processus

Implémentation : nouveau langage

Commentaires

Dans Guide [Krakowiak90], les objets sont traversés par les processus. Dans ce système, les objets appartiennent à des domaines. Un processus a en charge un domaine. Si un objet appartient à plusieurs domaines, plusieurs exécutions de méthodes peuvent avoir lieu simultanément. Dans ce cas, l'objet doit être synchronisé pour éviter les problèmes d'incohérence de son état. La synchronisation est à base de conditions d'activation [Riveill94][Balter90]. Les conditions d'activation peuvent être étendues de manière incrémentale en utilisant l'héritage.

Le langage Guide sépare la notion de type de la notion de classe. Une classe implémente un type. Cette séparation permet d'avoir des implémentations de classes différentes pour un même type. Ce mécanisme est utilisé pour gérer l'hétérogénéité [Balter94].

Guide introduit également des fonctionnalités de persistance sur les objets.

II - 4•12 Plasma II

Plasma II est un langage d'acteurs développé à l'IRIT par l'équipe de Arcangeli.

Architectures visées : réseau de stations de travail, transputers, Butterfly TC2000

Modèle utilisé : Acteurs

Implémentation : nouveau langage interprété

Commentaires

Plasma II [Arcangeli94] est un langage d'acteurs non typé. La transmission de messages entre acteurs peut être bloquante ou non bloquante. Plasma II introduit une notion de groupe d'acteurs. Un message peut être envoyé à un groupe, l'ensemble des acteurs appartenant au groupe recevra le message émis (notion de diffusion).

II - 4•13 POOL-T

Pool (Parallel Object-Oriented Language) [America87][America87b][America90] a été développé par l'équipe de Pierre America aux laboratoires Philips.

Architectures visées : réseau de stations de travail

Modèle utilisé : Objets Actifs

Implémentation : nouveau langage

Commentaires

POOL est un langage avec un mécanisme de communication synchrone. Tous les objets sont actifs ; lors de la création d'une instance, un processus est créé et exécute le code d'un *body* qui représente le comportement de l'objet. Les objets POOL sont mono-programmés. POOL permet de retourner le résultat d'une fonction à l'appelant et de continuer le déroulement de l'appelé (section `POST`).

America a été un des premiers à se rendre compte qu'il y avait un problème de conflit entre la spécification de comportement par *body* et l'héritage (*inheritance anomaly*). Lorsque le programmeur crée une nouvelle classe d'objets actifs par héritage, très souvent, il y ajoute de nouveaux services que le *body* doit accepter. Cela implique une réécriture du *body*. Cette réécriture entre en conflit avec le principe de réutilisabilité, cher au génie logiciel. La solution choisie dans POOL est de supprimer l'héritage du langage.

II - 4•14 pSather

pSather [Murer93][Lim93] a été développé à l'université de Berkeley par l'équipe de Omohundro. C'est une extension au parallélisme de Sather [Omohundro91], langage compilé pleinement objet qui ressemble à Eiffel.

Architectures visées : réseau de stations de travail, CM5

Modèle utilisé : Objets+Processus

Implémentation : extension du langage Sather

Commentaires

pSather permet la création de nouveaux flots d'activité par l'instruction :

```
:- appel_de_procédure_sur_un_objet
```

Un nouveau thread est créé et exécute l'expression qui suit. Dans pSather, la notion de processus n'apparaît pas comme une notion objet : il n'y a pas de classe THREAD (bien que ce soit prévu dans une version ultérieure).

pSather introduit comme mécanisme de synchronisation des Gates. Gate est en fait un nouveau nom pour moniteur [Feldman91].

Une gate contient :

- une liste de valeurs dont les types doivent être conformes à T (dans le cas où l'on se sert d'une gate générique `GATE{T}`)
- un ensemble de threads associés à la gate
- un status de verrouillage (verrouillé ou déverrouillé)
- un status de liaison (lié, non lié ou cassé)

Une structure de verrou est introduite dans le langage :

```
LOCK expressions THEN actions END
```

La liste d'expressions est constituée d'expressions de gate. Si toutes les gates de la liste sont dans un état non verrouillé ou verrouillé par le thread qui exécute l'instruction `lock` alors elles sont verrouillées de manière atomique et les actions sont exécutées. Après l'exécution des actions, l'état des gates est restauré. C'est-à-dire que si une gate était déjà verrouillée par le thread, elle restera verrouillée après l'exécution des actions, sinon elle est déverrouillée.

Si une des gates n'est pas disponible pour être verrouillée, le thread qui exécute l'instruction lock est bloqué ; il sera réveillé dès que la gate sera disponible pour être verrouillée. Une version non bloquante de cette instruction est disponible :

```
TRY expressions THEN actions1 ELSE actions2 END
```

Une gate peut être associée à un thread :

```
g:GATE {INT} := GATE{INT}::new ; -- Création d'une gate
avec une liste d'entiers
g :- compute ; -- lancement du calcul dans un nouveau
thread
...
result := g.read ; -- récupération du résultat du calcul
; attente si non disponible
```

Dans l'exemple, le thread qui exécute ce code attend un résultat du calcul. Ce type de construction ressemble à l'attente par nécessité de Eiffel //.

D'autres primitives permettent de resynchroniser des threads.

II - 4•15 SOS/FOG

SOS/FOG sont des éléments du projet SOR (Systèmes à Objets Répartis) développé à l'INRIA par l'équipe de Shapiro.

Architectures visées : Réseau de stations de travail

Modèle utilisé : Objets Fragmentés

Implémentation : noyau SOS + nouveau langage (FOG)

Commentaires

SOS [Shapiro89] (SOMIW[1] Operating System) est un système d'exploitation orienté objet. Il fournit les mécanismes pour la manipulation des objets : création, destruction, migration, stockage, localisation, nommage et support pour les objets fragmentés (voir FOG ci-dessous).

Dans SOS, le programmeur programme à l'aide d'objets. Ses objets sont distribués sur plusieurs sites. Pour pouvoir les faire coopérer, il se sert de la notion d'objet fragmenté [Makpangou93]. Un objet fragmenté permet de placer des mandataires de l'objet sur des sites distants, permettant ainsi 1) de réaliser facilement les traitements qui peuvent être locaux aux clients et 2) de cacher les détails des communications inter-sites à l'intérieur des objets fragmentés. SOS offre également des fonctionnalités de migration des objets.

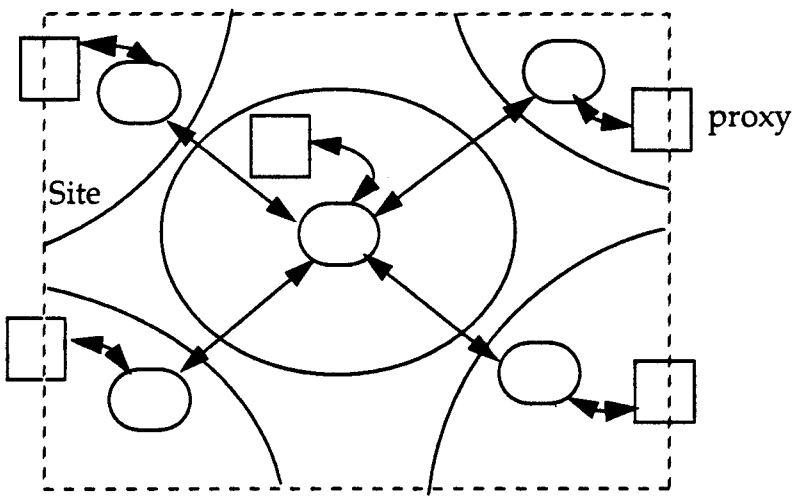


figure 2.11 *Un objet de SOS fragmenté sur plusieurs sites*

La programmation des objets fragmentés (FO : Fragmented Objects) est réalisées à l'aide du langage FOG (Fragmented Object Generator) [Shapiro90]. Le compilateur FOG produit du code C++. Le programmeur décrit ses FO à l'aide de classes.

1. SOMIW (Secure Open Multimedia Integrated Workstation) est un projet Esprit.

Une classe est composée de :

- Une partie locale dans laquelle il peut déclarer des attributs et des fonctions membres. Ce code est identique au code C++.
- Une interface procédurale du fragment pour les autres fragments du même objet fragmenté. Les déclarations de ces méthodes sont traduites dans un stub qui se charge de la réception du message, du décodage, de l'appel proprement dit de la fonction, du codage du résultat et de l'envoi de la réponse.
- La déclaration des méthodes que le client peut appeler sur d'autres fragment du FO (communication avec l'interface procédurale d'un fragment).
- D'une méthode pour la création de proxy au fragment.

Un certain nombre de FO ont été réalisés pour gérer la synchronisation, la réplication, la migration des objets ainsi que le contrôle du parallélisme. Le programmeur d'application réutilise ces différentes abstractions pour réaliser son application [Makpangou91].

II - 4•16 Résumé

Tableau récapitulatif des environnements comparés ; à titre d'information, nous comparons également l'environnement BOX.

Environnement	Entités		Synchronisation			Communication	
	Pass	Act		O.P.	Mess	Sync	Async
ABCL/1		♦	filtrage		♦	♦	♦
ACOOL	♦	♦	filtrage	NYI[a]	♦	♦	♦
Act3		♦	filtrage		♦		♦
Actalk	♦	♦	filtrage, garde		♦		♦
Charm++	♦	♦	filtrage		♦		♦
Concurrent Stk	♦		CBox		♦		♦
Distributed Stk	♦		classes Stk	♦		♦	
Eiffel //	♦	♦	body serve	♦			♦
GARF	♦		biblio. comport.	♦		♦	♦
Gothic	♦		états de synchro.	♦		♦	
Guide	♦		cond. activation	♦		♦	
Plasma II		♦	filtrage		♦	♦	♦
POOL		♦	body	♦		♦	
pSather	♦		gate	♦			♦
SOS/FOG	♦		objets fragmenté	♦		♦	
BOX	♦	♦	filtrage (1) SPROC (2)	♦ (2)	♦(1)	♦	♦

a.NYI : Not Yet Implemented

Entités : (Pass) passives ; (Act) actives
Synchronisation : (O.P.) objet partagé ; (Mess) message
Communication : (Sync) synchrone ; (Async) asynchrone

L'introduction du parallélisme dans les langages à objets n'est pas une chose facile, plusieurs modèles ont émergé, ils ont tous des avantages et des inconvénients. L'approche Objets+Processus permet de partir d'une application séquentielle et d'y ajouter simplement des déclenchements d'activités en faisant toutefois attention à la

synchronisation. L'idée du modèle des Objets Actifs est également alléchante : concevoir une application en considérant les objets comme étant des serveurs.

Toutefois des problèmes subsistent : les aspects génie logiciel et réutilisation de composants logiciels parallèles distribués sont peu abordés. L'expression de la synchronisation en conjonction avec l'héritage est une chose difficile.

Si nous comparons les deux types de modèles (figure 2.12), nous voyons qu'ils sont opposés dans leurs choix de conception.

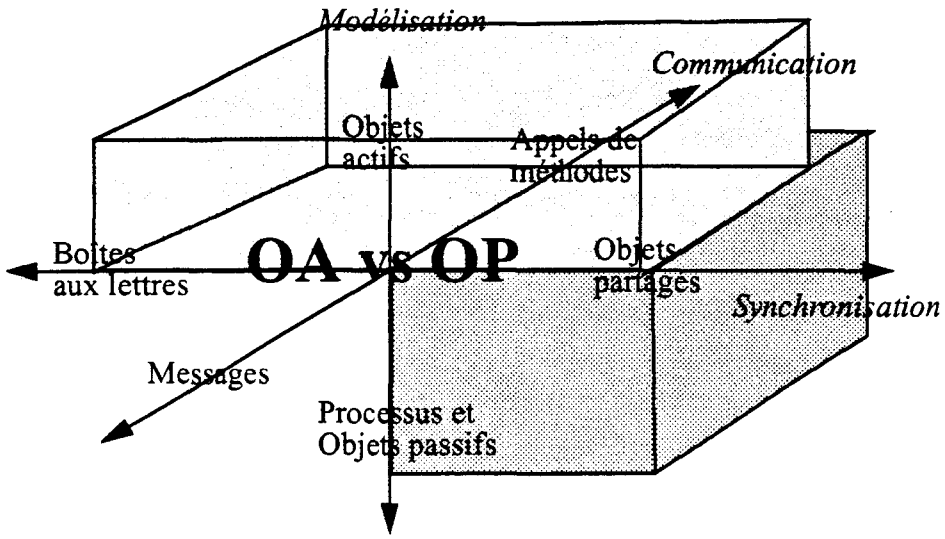


figure 2.12 *Comparaison du modèle Objets Actifs et du modèle Objets+Processus*

La différence principale provient de la manière d'associer une activité à un objet. Dans le cas des objets actifs, l'activité est associée à un objet. Dans le second cas, l'activité passe d'un objet à l'autre en suivant l'ordre des appels de méthodes.

Dans la conception de composants logiciels, on aimerait pouvoir utiliser le paradigme le plus adapté mais le modèle avec lequel il faut réaliser les composants ne le permet pas toujours. Nous verrons que le modèle BOX tente de rendre cette liberté au programmeur.

Chapitre - III -

BOX: Un modèle pour la programmation distribuée

Le concepteur d'applications séquentielles se doit de structurer son application en d'une part les données qu'il doit utiliser, et d'autre part les procédures qui manipulent ces données. L'approche objet, maintenant bien connue l'aide dans ce travail en lui fournissant une entité unique de structuration (l'objet) qui regroupe logiquement ces deux aspects de la programmation.

Le concepteur d'applications réparties est devant un double problème de conception. Il doit représenter les données manipulées par son application (ainsi que les procédures qui les manipulent), mais il doit aussi représenter les activités qui vont coopérer à la réalisation de l'application.

Pour répondre à ce besoin nous introduisons dans ce chapitre un modèle prenant en charge de manière uniforme ces deux aspects de la conception d'applications réparties. Ce modèle, qui est le fondement du langage BOX introduit plus loin, est ici appelé simplement le modèle BOX. Ce modèle introduit explicitement des entités de représentation des données et des entités de représentation des activités. L'ensemble est vu "sur le mode objet" puisque, on le verra dans le langage, les données et les activités sont créées par une opération d'instanciation à partir de classes.

Le modèle BOX introduit donc deux types distincts d'entités : des entités qualifiées de passives et appelées simplement **Objets**, et des entités qualifiées d'actives et appelées **Fragments**. Les premières correspondent aux objets traditionnels des langages séquentiels à objets, mais elles pourront être ici utilisées en présence de flots d'exécution concurrents. Les deuxièmes correspondent à une approche objet de la représentation des activités d'une application répartie. Le modèle introduit aussi des possibilités multiples de coopération entre Objets et Fragments.

Le modèle ne pose pas de prérequis sur l'architecture, celle-ci est supposée être composées 1) de noeuds (processeur+mémoire) pouvant accueillir des Objets et des Fragments et 2) d'un système de communication permettant de faire coopérer directement des entités situées sur des noeuds différents.

III-1 Les Objets

III-1-1 Définitions

Les objets dans le modèle BOX correspondent aux objets des langages à objets séquentiels. Un objet est composé d'un ensemble d'attributs et de procédures qui permettent de le manipuler (figure 3.1).

Un objet est situé complètement sur un noeud de l'architecture et il ne migre pas. Un objet a un nom, il peut donc être référencé. Une référence sur un objet est valable dans toute l'application répartie quelle que soit la localisation de l'objet et des entités qui le référencent.

Les procédures et la lecture des attributs forment l'interface à l'objet. Un objet est donc un domaine d'encapsulation. Lorsqu'un objet A possède une référence sur un objet B, il peut accéder à son interface, et cela quelle que soit la localisation de A et de B.

Les attributs contiennent soit des valeurs de types primitifs soit des références à des entités.

Les procédures et la lecture des attributs sont utilisées par appel procédural de l'interface. Le passage des paramètres et le retour du résultat (s'il s'agit d'une fonction ou d'une lecture d'attribut) se font par valeur pour les types primitifs et par référence pour les entités. Il s'agit ici d'appel procédural distant (RPC) car l'appel procédural est transparent à la localisation. Le flot d'exécution qui effectue l'appel diffuse donc dans l'objet appelé pour exécuter la procédure dans un contexte limité à cet objet.

La création d'un objet se fait par une opération explicite d'instanciation à partir d'une classe d'objet. Cette opération est synchrone et retourne une référence sur l'objet créé. La création s'effectue sur l'un des noeuds de l'architecture. Il est possible de fixer le noeud explicitement au moment de la création. Si cela n'est pas fait, le système se charge de déterminer un noeud.

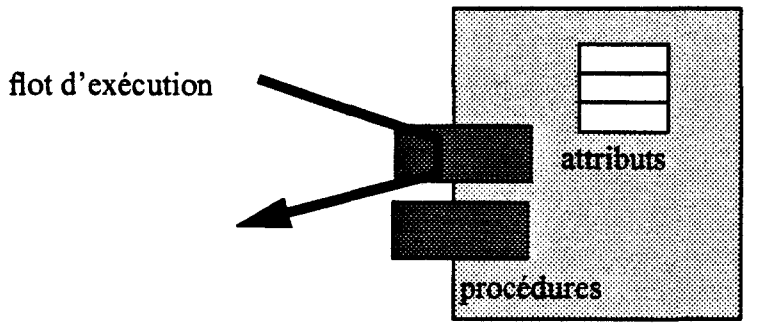


figure 3.1 *Un objet*

Cette définition des objets passifs correspond donc à celle des objets traditionnels des langages séquentiels. La seule différence est que les objets peuvent être distribués de manière quelconque sur les différents noeuds d'une architecture distribuée.

Les objets du modèle BOX peuvent servir à une modélisation traditionnelle des parties séquentielles d'une application répartie (ceux qui sont pris en charge par un seul processus sur une machine par exemple, ou encore un programme principal séquentiel utilisant des composants parallèles). A la limite il est possible ici d'exécuter de manière répartie une application séquentielle conçue à base d'objets.

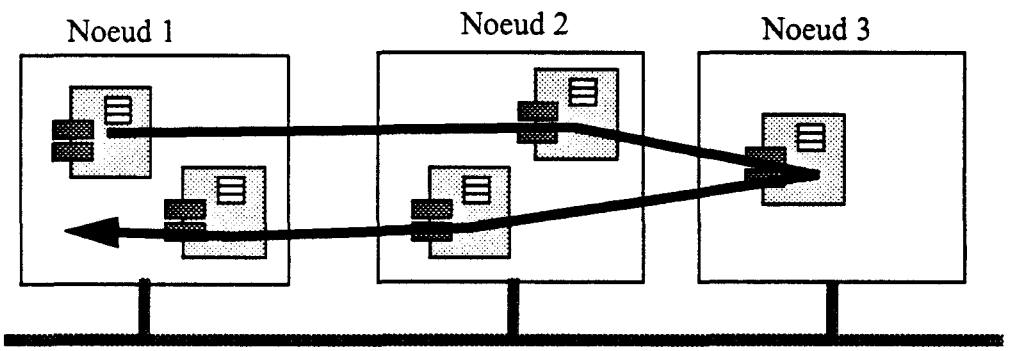


figure 3.2 *Exécution répartie à base d'objets*

Bien évidemment, ils peuvent aussi participer à des parties parallèles d'une application répartie lorsqu'ils sont partagés par plusieurs flots d'exécution. Une synchronisation est alors nécessaire. Cela est introduit dans le prochain paragraphe.

III-1-2 Synchronisation

Comme on le verra plus loin (voir la section fragments), une application répartie comprend généralement plusieurs flots d'exécution concurrents. Ceux-ci peuvent se retrouver simultanément dans un même objet et mettre en cause la cohérence interne de l'objet. Il est donc nécessaire de synchroniser les flots d'exécution à l'intérieur d'un objet. Notre modèle introduit pour cela un minimum de synchronisation dans les objets.

Par défaut, les objets ne sont pas synchronisés ; tous les appels concurrents sur un même objet peuvent s'y dérouler simultanément. Par contre certains objets peuvent être explicitement munis d'une synchronisation de type lecteurs/rédacteurs. Pour ces objets l'interface est partitionnée en deux sous-ensembles. Un premier sous-ensemble comprend les procédures dites lectrices de l'objet (l'accès en lecture aux attributs fait partie de ce sous-ensemble), ce sont les procédures qui ne modifient pas l'état de l'objet et qui donc ne sont pas synchronisées entre elles. Un deuxième sous-ensemble comprend les procédures dites rédactrices de l'objet, ce sont celles qui modifient l'état de l'objet et qui sont assurées d'avoir l'accès exclusif à l'objet.

La synchronisation en lecteurs/rédacteurs s'applique aux appels externes à l'objet, les appels de l'objet à ses propres procédures ne relèvent pas de cette synchronisation.

Cette synchronisation est minimale et n'assure que la cohérence interne des objets partagés. Un modèle plus puissant de synchronisation permettant d'ordonnancer de manière complexe les flots d'exécution (par exemple des conditions d'activation attachées aux procédures) n'est pas introduit dans BOX au niveau des objets. Cette possibilité sera présente par la synchronisation au niveau des fragments.

Les objets BOX sont donc des objets traditionnels protégés des accès concurrents en écriture. Le mécanisme de synchronisation qui protège des accès concurrents en écriture est peu coûteux et ne nécessite pas d'associer un flot d'exécution supplémentaire aux objets partagés.

III-2 Les Fragments

III-2-1 Définitions

Un fragment correspond à la notion d'activité concurrente. La seule manière de créer un nouveau flot d'exécution est de créer un nouveau fragment. Un fragment est composé d'un ensemble d'attributs et d'une procédure en cours d'exécution par un nouveau flot d'exécution. Cette procédure est lancée dès la création du fragment.

Un fragment est situé complètement sur un noeud de l'architecture et il ne migre pas (attention, le flot d'exécution associé au fragment peut, lui, diffuser sur d'autres noeuds s'il appelle des procédures d'objets distants). Un fragment a un nom, il peut donc être référencé. Une référence sur un fragment est valable dans toute l'application répartie quelle que soit la localisation du fragment et des entités qui le référencent.

Un fragment n'a pas d'interface, c'est un domaine d'encapsulation clos (on verra une exception plus loin en termes de boîtes aux lettres).

Les attributs contiennent soit des valeurs de types primitifs soit des références à des entités.

La création d'un fragment se fait par une opération explicite d'instanciation à partir d'une classe de fragments, en précisant obligatoirement la procédure qui doit être prise en charge par le nouveau flot d'exécution. Cette opération est synchrone et retourne une

référence sur le fragment créé. La création s'effectue sur l'un des noeuds de l'architecture. Il est possible de fixer le noeud explicitement au moment de la création. Si cela n'est pas fait, le système se charge de déterminer un noeud.

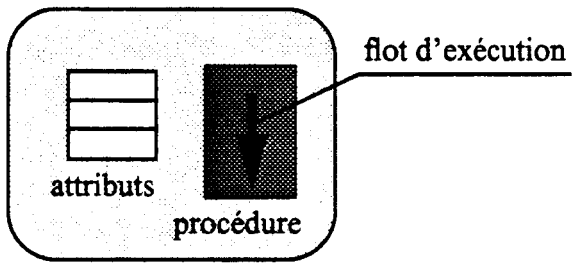


figure 3.3 *Un fragment*

Cette définition des fragments permet une modélisation en termes de processus concurrents et indépendants. Les différents processus sont décrits comme des procédures de fragments et les environnements d'exécution de ces processus comme les attributs de fragments. Les processus peuvent être lancés dynamiquement et sont répartis sur les différents noeuds de l'architecture.

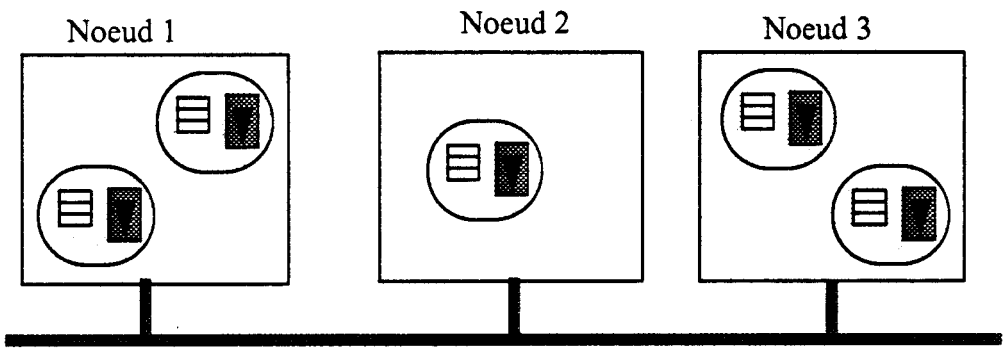


figure 3.4 *Exécution répartie à base de fragments*

III-2-2 Objets et Fragments

Dans le modèle BOX, la notion d'entité dénote les objets et les fragments. Via les attributs d'objets et de fragments, ces entités peuvent se référencer mutuellement : des objets peuvent créer des fragments et les référencer ; des fragments peuvent créer des objets et les référencer. Ces liaisons peuvent aussi être établies par le biais des passages de référence en paramètres. Cette possibilité permet de mettre en oeuvre différents schémas de conception. Les principaux sont décrits ici.

III-2-2-1 Un programme séquentiel lançant des processus parallèles

La modélisation BOX est ici de concevoir le programme séquentiel à l'aide d'objets, les processus comme des fragments, et d'organiser le lancement des processus par la création des fragments par les objets.

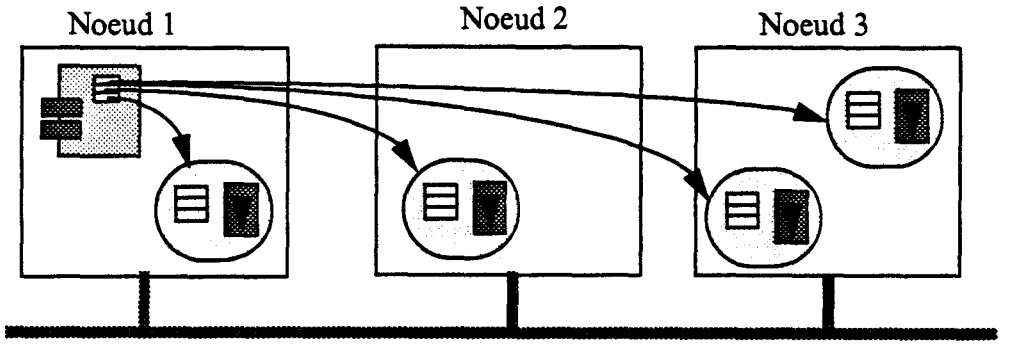


figure 3.5 *Objet et Fragments concurrents*

III-2-2-2 Un processus utilisant des données complexes

La modélisation BOX consiste ici en un fragment qui crée, référence et utilise des objets. Les objets n'ont pas besoin d'être synchronisés.

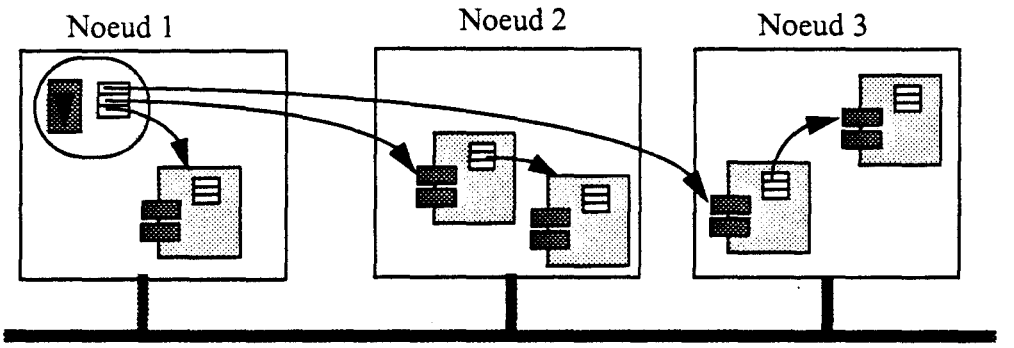


figure 3.6 *Fragment et Objets répartis*

III-2-2-3 Un programme parallèle formé de différents processus qui se partagent des données communes

La modélisation BOX est ici de décrire les processus comme des fragments, les données communes comme des objets, et de fournir des références à ces objets lors de la création des fragments. Les objets doivent être ici synchronisés.

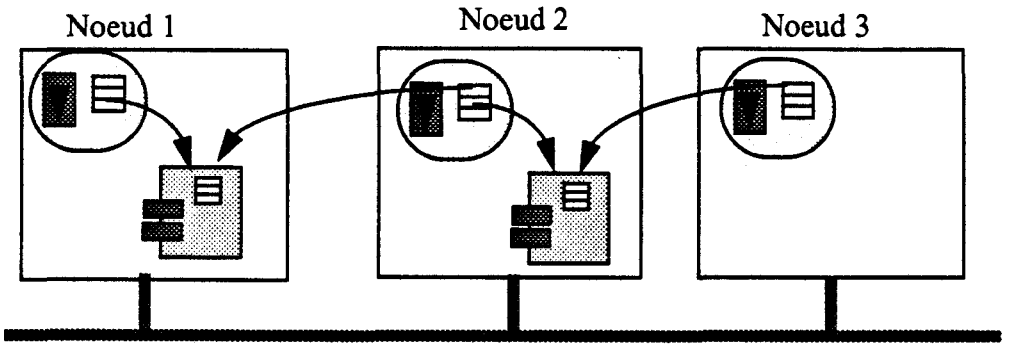


figure 3.7 *Fragments concurrents et Objets partagés*

Ces différentes possibilités montrent l'intérêt d'avoir les deux types d'entités et donc de s'adapter soit à une modélisation en termes d'objets, soit à une modélisation en termes de processus parallèles, soit à une modélisation mixant les deux.

On remarquera que la notion de fragment permet de construire la notion d'appel asynchrone de procédure d'un objet. En effet, imaginons un fragment dont le corps de la procédure (soit PF) ne fait qu'appeler une procédure d'un objet (soit PO), la création de ce fragment (donc le lancement en parallèle de PF) réalise en parallèle l'appel de PO. Pour cette raison BOX n'introduit pas l'appel asynchrone de procédure sur les objets (ce qui aurait modifié le modèle traditionnel des objets).

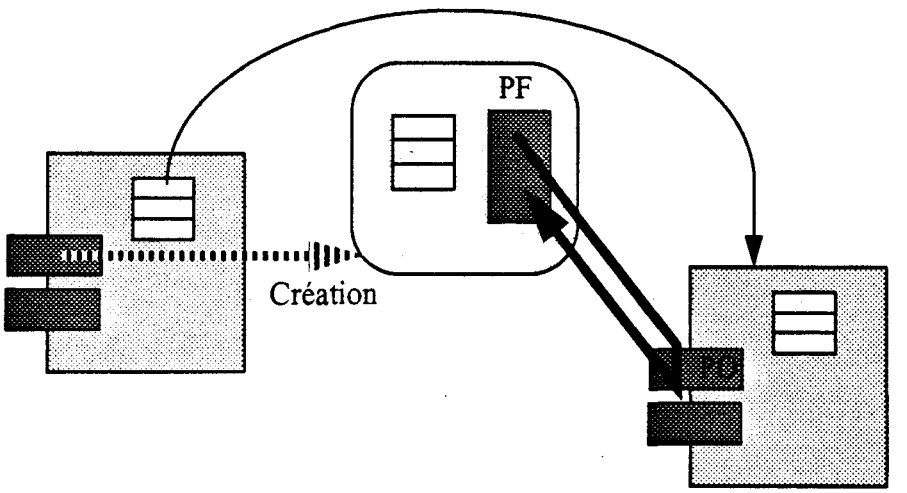


figure 3.8 *Appel asynchrone d'une procédure sur un objet*

Tous les schémas précédents illustrent les possibilités de coopération entre objets et fragments. Tout cela ne saurait être complet s'il n'existait pas une possibilité de coopération directe entre fragments, et ceci pour permettre une modélisation en termes

de processus coopérants (rappelons que les objets partagés, qui peuvent être utilisés à ce propos, n'ont pas dans BOX un mécanisme de synchronisation suffisant pour répondre complètement à ce besoin). Le modèle BOX introduit donc une possibilité de communication explicite par messages via des boîtes aux lettres et associe implicitement des boîtes aux lettres aux fragments. Cela est présenté dans les prochaines sections.

III-2-3 Communication par messages

Dans le modèle BOX, la coopération peut aussi s'exprimer en termes de communication par messages. Les caractéristiques de cette communication sont les suivantes :

- Les entités peuvent créer dynamiquement des boîtes aux lettres. Une boîte aux lettres est créée localement à l'entité qui demande la création. La création retourne une référence à la boîte aux lettres.

- Une entité, quelle que soit sa localisation, qui possède une référence à une boîte aux lettres peut y envoyer des messages de manière asynchrone (c'est-à-dire sans se bloquer). Le contenu d'un message est passé par valeur pour les types primitifs et par référence pour les entités et boîtes aux lettres. Les messages sont placés dans la boîte aux lettres selon l'ordre de leur arrivée.

- Une entité qui a créé une boîte aux lettres peut en examiner le contenu, y retirer des messages ou y attendre des messages. Le retrait de messages ne suit pas obligatoirement l'ordre FIFO, les messages peuvent être attendus, retirés et traités par l'entité dans l'ordre qui lui convient.

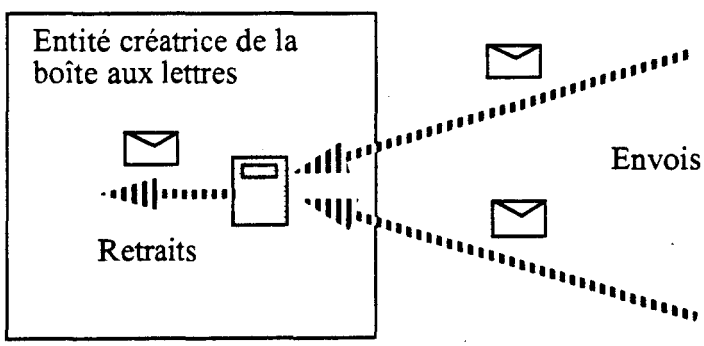


figure 3.9 *Communication par messages*

Ces caractéristiques définissent un mécanisme traditionnel de communication asynchrone par message. Les références aux boîtes aux lettres peuvent être passées dans des messages (et dans les appels procéduraux) pour établir dynamiquement des canaux de communication. De même qu'une entité doit d'abord acquérir une référence à un objet pour pouvoir utiliser son interface, une entité doit d'abord acquérir une référence à une

boîte aux lettres avant de pouvoir y envoyer des messages. Les schémas de coopération doivent donc dans BOX s'établir dynamiquement.

III-2-4 Fragments et Boîtes aux lettres

Par définition chaque fragment est créé avec une boîte aux lettres implicite.

Cette boîte aux lettres implicite a le même nom que le fragment, c'est-à-dire que la référence sur un fragment est aussi une référence sur sa boîte aux lettres implicite. Ainsi lorsqu'une entité crée un fragment, elle peut utiliser la référence résultat de la création pour envoyer des messages au fragment. De façon générale, dès qu'une entité possède une référence sur un fragment, elle peut lui envoyer un message. Par analogie avec les objets, l'interface d'un fragment est ici réduite à sa boîte aux lettres implicite.

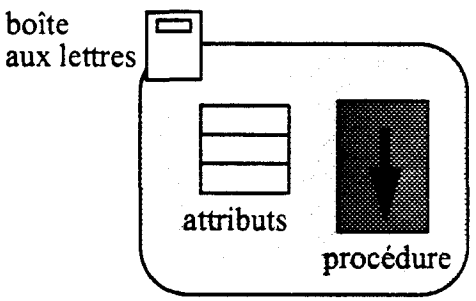


figure 3.10 *Un fragment*

Cette propriété nous donne donc un dernier schéma possible de conception, celui par processus communiquant par message. Les différents processus sont représentés par des fragments répartis sur les différents noeuds, ils communiquent par messages entre leurs boîtes aux lettres. Ce schéma s'apparente à celui des acteurs. Il s'apparente de même à celui des objets actifs, car un fragment peut être vu ici comme un serveur de requêtes arrivant dans sa boîte aux lettres.

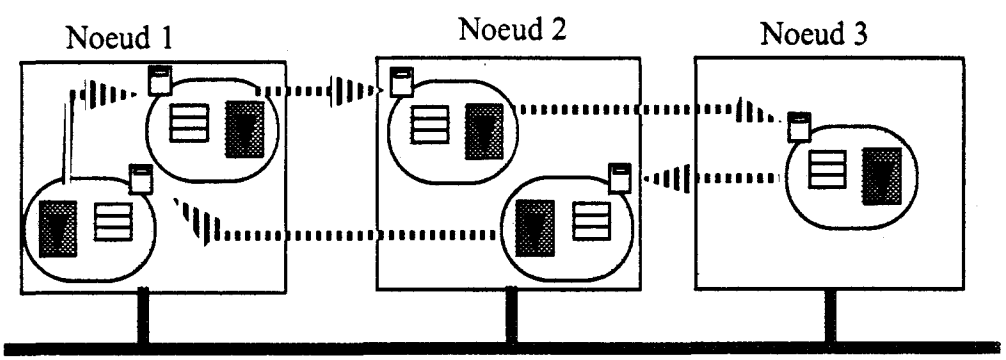


figure 3.11 *Fragments communiquant par messages*

III-2-5 Boîtes aux lettres et appel de service asynchrone

Le modèle BOX, tel qu'il a été présenté ici permet à une entité de lancer une activité en créant un fragment et permet à une entité d'envoyer des messages à un fragment. Ces possibilités peuvent être utilisées pour requérir un service asynchrone (je demande sans me bloquer un service à une entité concurrente). Le message contient alors la requête. Un tel service entraîne, lorsqu'il s'apparente à une fonction, un retour de résultat et donc une nécessaire resynchronisation de l'appelant avec le retour du résultat. Les mécanismes utilisés dans d'autres modèles pour cette resynchronisation (comme les variables futures et l'attente par nécessité par exemple), sont ici remplacés par l'utilisation d'une boîte aux lettres créée dynamiquement pour cela. Le scénario est le suivant:

1- création d'une boîte aux lettres locale par l'appelant.

2- envoi du message de requête contenant aussi une référence à cette boîte.

3- lorsque le résultat est nécessaire à l'appelant, attente de celui-ci sur sa boîte aux lettres.

De son coté, l'appelé envoie le résultat de manière asynchrone sur la boîte aux lettres dont il a obtenu dynamiquement la référence.

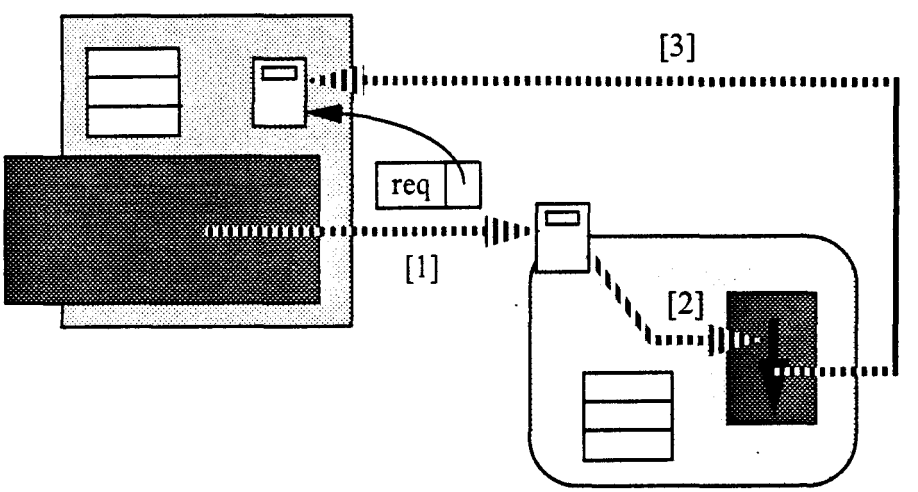


figure 3.12 *Appel de service asynchrone*

De manière générale, la création dynamique de boîtes aux lettres permet de créer des canaux de communication privés entre entités, et la synchronisation de ces entités par l'attente d'un message sur ces boîtes.

III-2-6 Synchronisation

Comme on vient déjà de le voir, les boîtes aux lettres sont des lieux de synchronisation. Le mécanisme de base est l'attente d'un message sur une boîte aux lettres. Comme il a été dit plus haut, le créateur d'une boîte aux lettres a la possibilité d'en examiner le contenu et d'extraire les messages dans l'ordre qui correspond à ses besoins. Il a aussi la possibilité de se mettre en attente d'un message non présent dans la boîte aux lettres au moment de l'examen de celle-ci. Il s'agit donc plus ici d'un mécanisme d'attente d'un message particulier que de l'attente d'un message quelconque. Ces deux possibilités, retrait ou attente d'un message particulier, sont appelées, dans le modèle BOX, réception avec filtrage et sont introduites dans le langage BOX par les constructions syntaxiques spécifiques qui se basent sur le type des informations contenues dans les messages.

Au niveau du modèle cela induit que l'entité créatrice d'une boîte aux lettres peut exécuter un comportement complexe dont les différentes étapes seront synchronisées sur les arrivées des messages les déclenchant. Par exemple une entité qui exécuterait séquentiellement les opérations attendre_message1, traitement1, attendre_message2, traitement2, exécutera effectivement traitement1 puis traitement2 quel que soit l'ordre d'arrivée des messages. Il s'agit donc d'un moyen puissant de synchronisation servant à l'ordonnancement des traitements d'une application répartie. Cette possibilité explique l'absence d'un schéma complexe de synchronisation au niveau des objets, la synchronisation dans le modèle BOX étant plus fondée sur des communications synchronisantes.

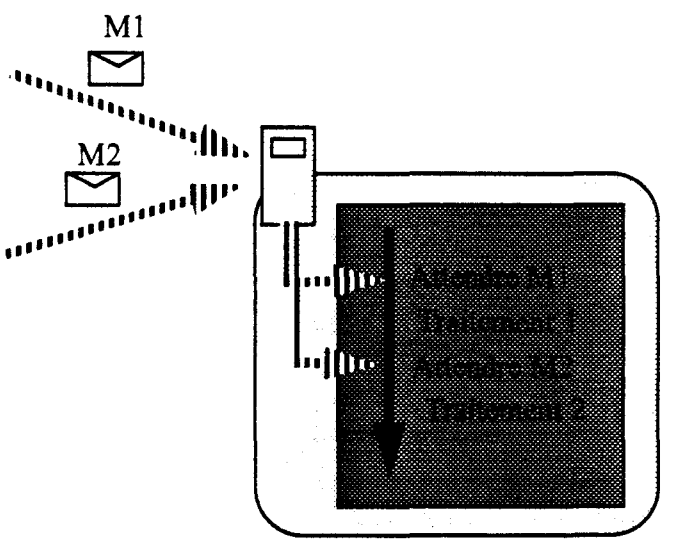


figure 3.13 *Synchronisation par les communications*

III-2-7 Localisation des entités

Une référence peut servir à désigner une entité ou une boîte aux lettres quelle que soit sa localisation. Une création d'une nouvelle entité est effectuée sur un noeud quelle que soit la localisation du créateur, un appel procédural ou un envoi de message est réalisé quelle que soit la localisation des coopérants. Tout cela montre que le modèle BOX est transparent à la distribution de l'architecture sous-jacente et suppose qu'il existe le support d'exécution adéquat rendant possibles des références étendues, des RPC et de la communication point à point, ainsi qu'un minimum d'équilibrage dynamique de la charge pour répartir équitablement les créations.

Il est pourtant souvent utile de bien choisir le noeud de création d'une entité pour limiter par exemple l'éloignement d'entités communiquant fortement, ou pour accroître le parallélisme réel.

Le modèle BOX précise donc simplement que les entités peuvent avoir accès à la configuration utilisée par l'application. Et cela pour pouvoir préciser, au moment d'une demande de création, un ou plusieurs sites de cette configuration. Les éléments mis en oeuvre pour cela dans le langage seront précisés au chapitre V.

III-3 Conclusion

Notre modèle permet de décrire des entités actives (appelées fragments) et des entités passives (appelées objets). Les objets structurent l'espace des données, les fragments l'espace des activités. D'un point de vue conceptuel, ces deux types d'entités sont à un même niveau et toutes les associations objet/fragment sont possibles.

Les différentes entités (objets et fragments) d'une application parallèle doivent coopérer, c'est-à-dire communiquer et se synchroniser. Pour cela nous mettons à disposition deux modes de coopération : l'appel procédural et la communication par messages. Les objets partagés peuvent être protégés des accès concurrents. L'attente de messages sur boîtes aux lettres sert à l'ordonnancement des tâches.

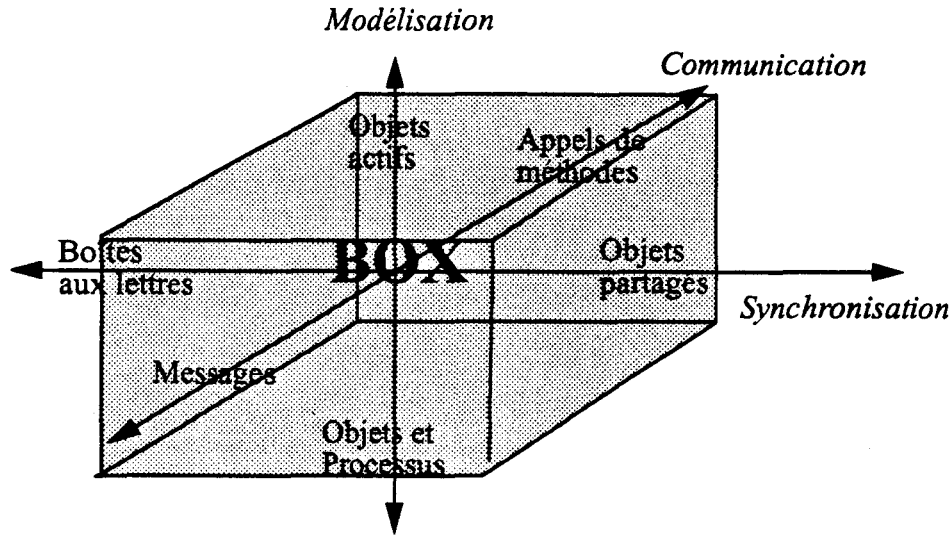


figure 3.14 *Le modèle BOX*

Les entités BOX sont mutuellement référençables. Cela donne la possibilité de concevoir des agrégats complexes et répartis d'objets et d'activités, qui forment la base méthodologique à la conception de composants logiciels réutilisables pour les applications réparties. Par exemple, dans la figure 3.15, Nous avons un composant logiciel ayant une interface procédurale qui englobe des activités concurrentes internes au composant. Ces activités peuvent référencer d'autres activités ou objets. Le client utilise cet objet par appel procédural sans avoir besoin de savoir que des traitements parallèles sont déclenchés. C'est la notion d'Objet Actif Complexe (OAC).

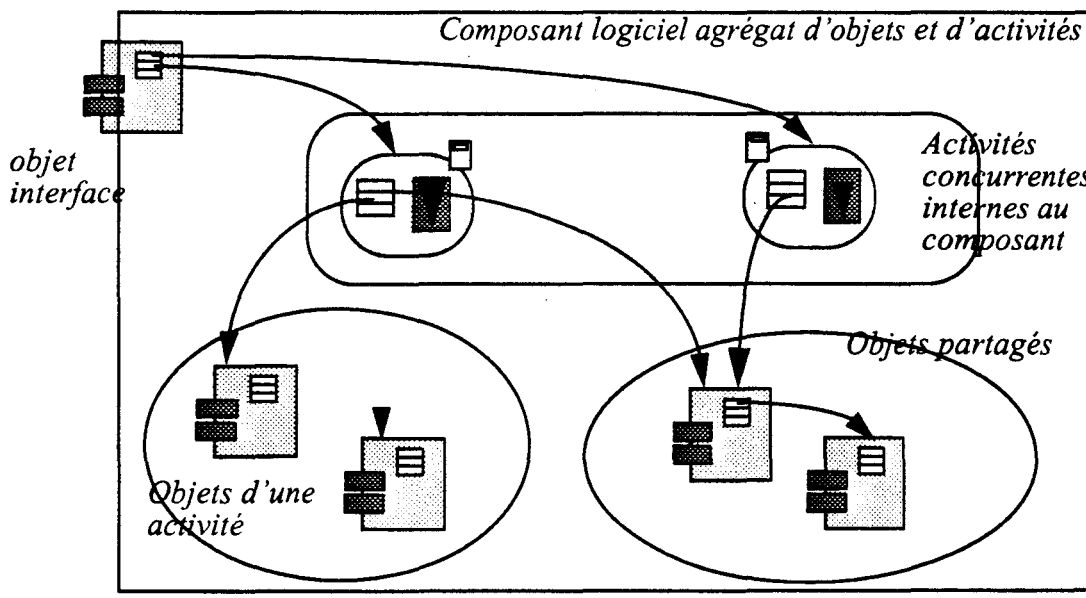


figure 3.15 *Composition d'objets et de fragments*

Chapitre - IV -

BOX: Un langage pour la programmation distribuée

Le langage BOX est un langage à objets pour applications distribuées. Il inclut des caractéristiques nouvelles, spécifiques pour ce type d'applications, mais reste un langage pleinement objet. Le langage implante le modèle du chapitre précédent et ce chapitre a pour vocation d'introduire le langage de programmation. Comme on l'a dit au chapitre précédent, le modèle est indépendant de l'architecture sous-jacente. Le langage BOX va dans le même sens et dissocie l'aspect programmation des classes de l'aspect installation du logiciel produit sur une architecture particulière. Ce chapitre est donc consacré à l'aspect programmation des classes, l'autre aspect est examiné dans le chapitre suivant.

IV-1 Introduction

Le langage BOX juxtapose dans un seul langage différents paradigmes de la programmation par objets et de la programmation distribuée.

1) Sur un axe 'composant', le langage BOX permet à la fois la définition :

1-1) d'entités passives. Elles correspondent aux objets classiques des langages à objets. Une telle entité, appelée simplement **objet**, est instance d'une classe d'objets

qui précise des attributs et des procédures.

1-2) d'entités actives. Ce sont des entités munies d'une activité propre. Une telle entité, appelée par la suite **fragment**, est instance d'une classe de fragments qui précise des attributs et des comportements.

2) Sur un axe 'communication', le langage BOX permet à la fois l'utilisation de :

2-1) l'appel de procédure. Il s'agit d'un appel synchrone d'une procédure d'un objet.

2-2) la communication explicite par messages. Il s'agit d'un envoi asynchrone de messages à destination d'une boîte aux lettres.

3) Sur un axe 'synchronisation', le langage BOX offre à la fois :

3-1) une politique implicite de synchronisation dans chaque objet. Il s'agit d'une synchronisation de type lecteurs/rédacteurs pour l'accès aux procédures de l'objet.

3-2) une opération puissante d'attente sélective de message sur une boîte aux lettres. Cette opération permet une synchronisation basée sur l'envoi de messages.

Le chapitre examine différents points du langage. BOX est un langage compilé fortement typé. Nous introduisons d'abord le système de typage, puis les opérations de créations, les instructions dont les aspects communication, et enfin les caractéristiques des objets et des fragments. Les aspects héritage et généricité sont introduits en fin de chapitre.

IV-2 Types

Le langage BOX utilise les notions de classe et de type. Ces deux notions sont néanmoins fortement liées : chaque classe met en oeuvre un type, et chaque type est réalisé par une classe.

La différence est d'ordre syntaxique : un type est dénoté par le nom de sa classe entre crochets.

Syntaxe :

```
<TYPE>::= [<CLASSE>]
```

Un programme BOX est un ensemble de déclarations. Une déclaration utilise un type, soit un type de base, soit un type défini par une classe.

Les types de base sont soit le type minimal : [ANY], soit les types prédéfinis suivants:

- les types simples :
    - type entier : [INT]
    - type réel : [REAL]
    - type caractère : [CHAR]
    - type booléen : [BOOL]
- les types procédures :

- type procédure simple : [PROC]
- type procédure synchronisée : [SPROC]

- les types complexes :
  - type chaîne : [STR]
  - type tableau : [ARR]
  - type boîte aux lettres : [BOX]
  - type message : [MESS]

-les types héritables de base :
  - type fragment : [FRAG]
  - type objet : [OBJ]

Une relation d'héritage (simple) entre classes définit une relation de sous-typage entre types. Cette relation de sous-typage induit celle de conformité : un sous-type est conforme à tous ses sur-types.

Le graphe d'héritage initial est le suivant:

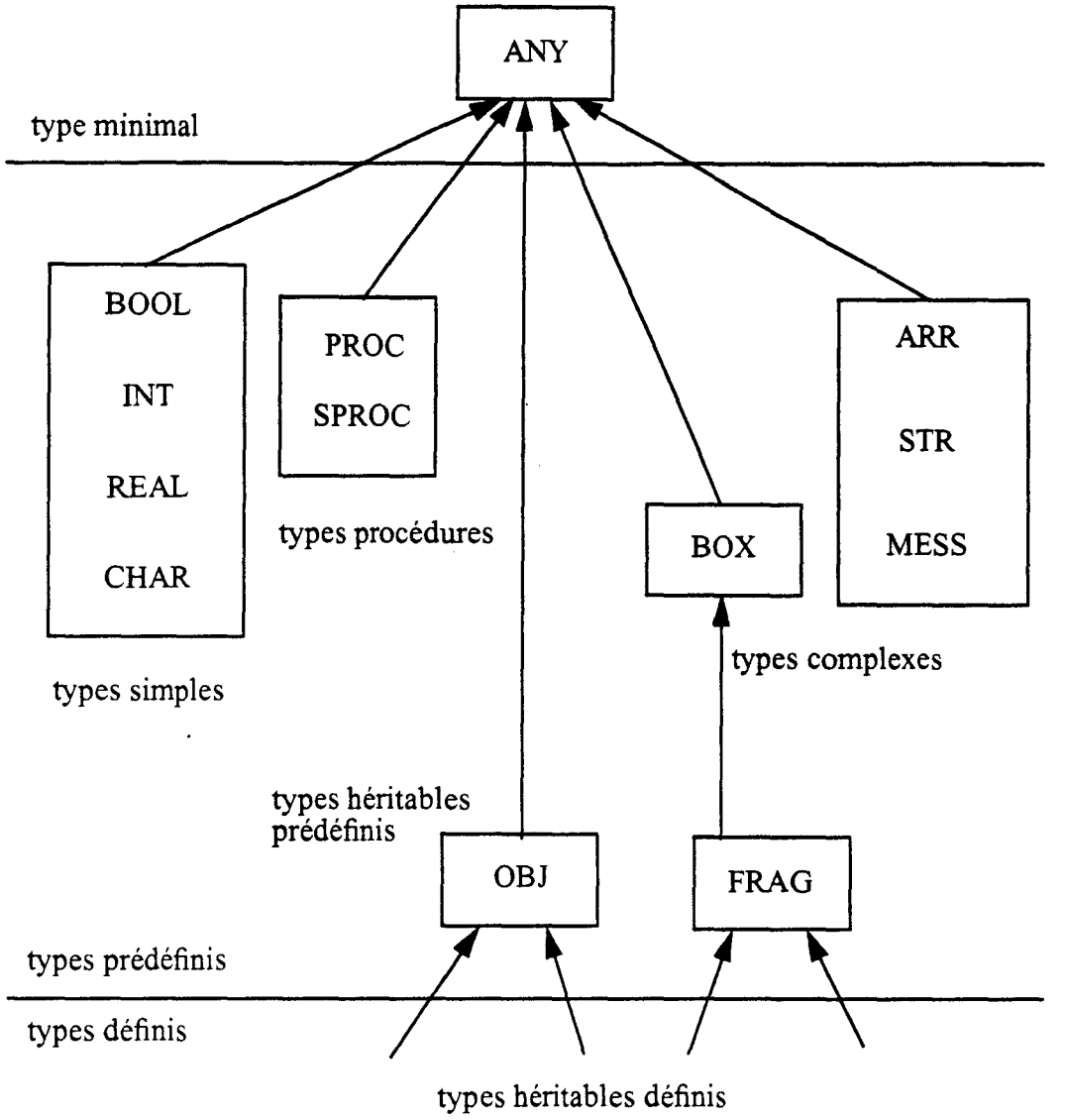


figure 3.16 *Graphe d'héritage en BOX*

Le programmeur a la possibilité de créer des sous-classes UNIQUEMENT de OBJ et de FRAG. Les autres classes sont primitives et ne peuvent être utilisées par héritage. Autrement dit les types correspondants sont primitifs. Tandis que les autres (obtenus directement ou indirectement à partir de OBJ et FRAG) sont dits héritables.

On remarquera que FRAG est sous-classe de BOX. Ce qui constitue donc une exception... permettant de rendre [FRAG] conforme à [BOX], comme le précise le modèle.

De même, toutes les classes héritent de ANY. Tout type est donc conforme au type minimal [ANY].

IV-3 Déclarations

Une déclaration donne un nom à une nouvelle entité du langage. En BOX, on peut trouver des déclarations d'attributs, de constantes, de procédures et de classes. Une déclaration demande un type, un symbole et une description.

Syntaxe :

```
<DECLARATION_ATTRIB>  ::= <TYPE> <SYMBOLE> <DESCRIPTION_ATTRIB>
<DECLARATION_CONST>   ::= <TYPE> <SYMBOLE> <DESCRIPTION_CONST>
<DECLARATION_PROC>    ::= <TYPE> <SYMBOLE> <DESCRIPTION_PROC>
<DECLARATION_CLASSE>  ::= <TYPE> <SYMBOLE> <DESCRIPTION_CLASSE>
```

IV-3-1 Description des attributs

Un attribut est déclaré avec une description vide.

Syntaxe :

```
<DESCRIPTION_ATTRIB> ::=
```

Règle 1 : Le type utilisé dans la déclaration d'un attribut peut être un type simple, un type complexe, un type héritable prédéfini ou défini par le programmeur.

Règle 2 : Un attribut ne peut être déclaré que dans la description d'une classe ou d'une procédure.

Règle 3 : La déclaration d'un attribut entraîne l'allocation de l'espace mémoire nécessaire s'il s'agit d'un type simple, l'allocation d'une référence dans les autres cas.

Règle 4 : Les valeurs par défaut des attributs sont indiquées dans le tableau suivant.

[INT]	0
[CHAR]	\0
[BOOL]	false
[REAL]	0.0
[STR]	référence nulle
[ARR]	référence nulle
[MESS]	référence nulle
[BOX]	référence nulle
[OBJ]	référence nulle
[FRAG]	référence nulle

Exemples :

```
[INT]i --attribut entier
[BOX]b1, b2 -- deux attributs de type boîte aux lettres
```

IV-3-2 Description des constantes

La description d'une constante précise une valeur littérale.

Syntaxe :

```
<DESCRIPTION_CONST> ::= = <VALEUR>
```

Règle 1 : Le type utilisé dans la déclaration d'une constante doit être un type possédant une forme littérale, c'est-à-dire un type simple ou le type chaîne.

Règle 2 : Une constante ne peut être déclarée que dans la description d'une classe.

Exemples :

```
[INT] n = 0
[STR] prompt = "bonjour"
```

IV-3-3 Description des procédures

La description d'une procédure précise des paramètres d'appels (entre ":") et un corps (entre accolades).

Syntaxe :

```
<DESCRIPTION_PROC> ::= : <PARAMETRES> : { <CORPS> }
```

Règle 1 : Le type utilisé pour la déclaration d'une procédure ne peut être qu'un des types procédures prédéfinis (voir le chapitre sur les objets pour l'utilisation des différents types procédures).

Règle 2 : Une procédure ne peut être déclarée que dans la description d'une classe.

Règle 3 : Les paramètres sont indiqués avec la même syntaxe que les attributs, c'est-à-dire un type suivi d'un symbole. Un paramètre peut être déclaré du type minimal [ANY]. Les paramètres d'une procédure doivent être considérés comme des attributs locaux accessibles en lecture uniquement.

Règle 4 : Le corps d'une procédure est une suite d'instructions (voir la section IV-5 sur les instructions pour la liste des différentes instructions BOX) et de déclarations d'attributs dits locaux. Une déclaration d'un attribut local peut apparaître comme une déclaration d'attribut, ou apparaître dans une instruction à la première apparition du symbole en le faisant précéder de son type. Un attribut local peut être déclaré du type [ANY].

Règle 5 : Les types procédures sont implicitement génériques, le paramètre de généricité étant le type du résultat de la procédure considérée comme une fonction (voir la section IV-8-2 sur la généricité). Lorsque la procédure est une fonction, le pseudo attribut local nommé `Result` sert à contenir le résultat retourné par la fonction. Il est implicitement déclaré du type générique utilisé.

Exemple :

```
[PROC[INT]] ppcm: [INT]x, y: -- fonction entière à deux paramètres
   {
     [INT]a := x; [INT]b := y;
     loop
       if a = b
         then result := a; exit
         else
           if a < b then a := a+x else b := b+y end_if
       end_if
     end_loop
   }
```

IV-3-4 Description des classes

La description d'une classe contient des déclarations d'attributs, de constantes et de procédures.

Syntaxe :

```
<DESCRIPTION_CLASSE> ::= : : <ATTRIBUTS, CONSTANTES ET PROCEDURES>
```

Règle 1 : Le type utilisé pour une déclaration de classe doit être un type héritable, soit prédéfini ([OBJ] ou [FRAG]), soit un type défini par l'utilisateur. La nouvelle classe implante un sous-type de ce type.

Règle 2 : Une déclaration de classe ne peut apparaître dans aucune autre déclaration. Les déclarations de classes ne peuvent donc pas être imbriquées : on ne peut déclarer de classes dans les classes - par contre les classes elles-mêmes sont hiérarchisées par le lien d'héritage.

Règle 3 : Une classe peut être générique (voir la section sur la généricité).

Exemple :

```
[OBJ]point:: -- classe d'objets points
  [INT]x,y;
  [PROC]position:[INT] nx,ny:
    { x := nx; y := ny }
  [PROC]deplacer:[INT]dx,dy:
    { x := x+dx; y := y+dy }
  [PROC[BOOL]]egal:[POINT]autre:
    { result := (x = autre.x) and (y = autre.y) }
```

IV-4 La création des entités BOX

La création d'une entité BOX se fait par une requête de création. Il y a globalement deux manières de faire cette requête : soit explicitement, la requête est alors une expression qui retourne une référence à l'entité créée, soit en demandant la création dès la déclaration de l'attribut qui référencera l'entité créée.

Remarque : Cette deuxième manière n'est autorisée que dans le corps des procédures et non dans la description des classes.

Une requête de création est une forme particulière d'indication de type, le type étant celui de l'entité qui sera créée. Une partie initialisation permet de particulariser la création.

Syntaxe :

```
<TYPE> ::= [ <CLASSE>]
<REQUETE_CREATION> ::= [ <CLASSE> <- <INITIALISATION> ]
```

Dans le cas d'une création à la déclaration, la requête de création remplace simplement le type dans la déclaration.

Exemples :

```
[BOX] b1; -- déclaration d'une boîte aux lettres
b1 := [BOX <-] ; -- création explicite d'une boîte aux lettres
[BOX <-] b2; -- déclaration et création d'une boîte aux lettres
```

Nous passons en revue l'application de cette règle pour les différentes catégories de types.

IV-4-1 Types simples

Pour les types simples, il n'y a pas de création explicite puisqu'une déclaration d'un type simple entraîne l'allocation automatique d'une entité du type dans l'entité où est élaborée la création.

L'entité est initialisée avec une valeur par défaut (voir le tableau section IV-3-1). Les entités de type simple peuvent cependant recevoir une valeur initiale au moment de leur déclaration. La partie initialisation de la requête de création est alors réduite à la valeur initiale voulue.

Exemple :

```
[INT <- 10] i ; -- déclaration (et création) d'un entier initialisé à
10
```

IV-4-2 Type tableau

Le type tableau étant un type générique (voir la section généricité), la requête de

création doit préciser le paramètre générique. La partie initialisation doit préciser le nombre d'éléments.

Exemples :

```
mon_tableau := [ARR[INT] <- 10] -- création d'un tableau de 10 entiers
[ARR[INT]<- 10]tab ; -- déclaration et création d'un tableau de 10
entiers
```

L'accès à un élément du tableau utilise la notation 'indice entre crochets'. L'indice doit être compris entre 0 et `NB_ELEMENT-1`.

IV-4-3 Type chaîne

La partie initialisation pour la création d'une chaîne peut contenir une valeur littérale de chaîne.

Exemples :

```
ma_chaine := [STR <- "essai"]; -- création explicite d'une chaîne
[STR <- "bonjour"]prompt ;
    -- déclaration et création d'une chaîne initialisée.
```

Syntaxiquement le premier exemple aurait pu être simplifié en:

```
ma_chaine := "essai"; -- création explicite d'une chaîne
```

IV-4-4 Types BOX et MESS

Voir la partie IV-5-3 sur la communication.

IV-4-5 Types procédures

Il n'y a pas d'opération de création pour les types procédures.

IV-4-6 Types héritables

Une requête de création d'une entité d'un type héritable, c'est-à-dire un objet ou un fragment, suit les mêmes règles. Le résultat est une référence à l'entité créée. La nouvelle entité est créée sur un site de l'architecture distribuée.

Exemples :

```
r := [CARRE <-]; -- création explicite d'un carré
[POINT <- ] p; -- déclaration et création d'un point
```

La partie initialisation contient, s'il le faut, un message (voir la partie communication pour la syntaxe) qui est envoyé au site choisi pour la création. Ce message indique une

procédure de la classe concernée ainsi que des paramètres d'appel. Pour un objet cette procédure est exécutée dans une phase d'initialisation de l'objet. Pour un fragment cette procédure est prise en charge par un nouveau flot d'exécution et correspond donc au comportement du fragment créé. Pour un fragment le message d'initialisation est donc obligatoire.

Exemples :

```
[RECTANGLE] r := [CARRE <- position !3, 4!]
[POINT <- position !2, 3! ] p
```

Le premier exemple montre l'affectation d'un carré à un attribut déclaré de type rectangle. Le type carré est ici supposé conforme au type rectangle, la classe carré sous-classe de rectangle.

IV-5 Les instructions BOX

Le corps d'une procédure est décrit par un programme séquentiel formé d'instructions séparées par des ';' (point virgule).

Il existe 3 sortes d'instructions : les instructions de contrôle, les instructions d'appels de procédure et les instructions pour la communication par messages.

```
<INSTRUCTION> ::= <CONTROLE>
<INSTRUCTION> ::= <APPEL_PROCEDURE>
<INSTRUCTION> ::= <COMMUNICATION>
```

IV-5-1 Les instructions de contrôle

Les instructions de contrôle sont l'alternative, la répétitive et la sortie de boucle. Ce sont les instructions traditionnelles de la programmation impérative.

Syntaxe :

```
<CONTROLE> ::= <ALTERNATIVE>
               <REPETITIVE>
               <SORTIE_BOUCLE>

<ALTERNATIVE> :: if <EXPRESSION_BOOLEENNE>
                            then <INSTRUCTIONS>
                            else_if <INSTRUCTIONS>
                            ....
                            else <INSTRUCTIONS>
                 end_if
<REPETITIVE> ::= loop <INSTRUCTIONS> end_loop
<SORTIE_BOUCLE> ::= exit
```

IV-5-2 Les instructions d'appels de procédure

L'appel de procédure se fait par la notation pointée qui désigne l'objet et la procédure concernée.

Syntaxe :

```
APPEL_PROCEDURE> ::= <SYMBOLE>.<SYMBOLE> ( <EXPRESSIONS> )
```

Remarque : l'appel d'une procédure sans paramètre ne nécessite pas les parenthèses.

Règle 1 : Le premier symbole doit avoir été déclaré d'un type héritable objet, c'est-à-dire [OBJ] ou l'un de ses descendants, et l'objet doit avoir été créé. Le deuxième symbole doit avoir été déclaré d'un type procédure dans la classe correspondante.

Règle 2 : Les résultats des expressions (valeurs ou références) forment les paramètres d'appel de la procédure. Ils doivent être de types conformes à la déclaration de la procédure.

Exemple :

```
test_origine := un_point.egal([POINT <- position !0,0!])
```

IV-5-3 Les instructions de communication

La communication par messages est un aspect novateur important de BOX. Nous y consacrons la section suivante. Les instructions correspondantes y seront décrites.

IV-6 La communication par messages

IV-6-1 Introduction

Le langage BOX permet la communication asynchrone par messages à destination de boîtes aux lettres.

- La communication est point à point. Il n'y a pas d'opération de diffusion globale de messages. Une entité ne peut pas envoyer un message vers une boîte aux lettres qu'elle ne connaît pas.

- Le système de communication sous-jacent est supposé fiable, c'est-à-dire qu'il n'y a ni altération, ni duplication, ni perte de messages. Un message arrivera à son destinataire au bout d'un temps fini (si le destinataire existe). Deux messages issus d'une même source et à destination d'une même boîte arriveront dans le même ordre que l'ordre d'émission.

IV-6-2 Les boîtes aux lettres

Une boîte aux lettres est une instance de la classe primitive BOX. Une boîte aux lettres est manipulée par l'intermédiaire d'une référence.

Les boîtes aux lettres doivent être créées dynamiquement. Une boîte aux lettres est créée dans l'espace de l'entité qui réclame la création (objet ou fragment). Cette entité en devient l'unique propriétaire.

Exemple :

```
une_boite := [BOX <-];
```

Une boîte aux lettres sert à contenir des messages. Les opérations de base pour accéder à cette boîte sont 1) le dépôt de message (opération d'envoi) et 2) l'extraction de message (opération de réception). Ces opérations sont décrites ci-dessous. Ensuite sont décrites des extensions de ces opérations à la réception conditionnelle et aux contraintes.

IV-6-3 L'envoi de message

Une boîte aux lettres est une file de messages (de taille non bornée). L'opération d'envoi permet de passer un message au système de communication, qui le déposera en queue de la file des messages de la boîte désignée. Cette opération peut être entreprise par toute entité possédant une référence pour la boîte réceptrice. Un message est décrit en BOX entre "!" (point d'exclamation). L'opérateur d'envoi est "<-".

Syntaxe :

```
<ENVOI> ::= <boîte> <- ! <EXPRESSIONS> !
```

Les valeurs des expressions (en nombre quelconque) formeront les données du message expédié vers la boîte désignée, quelle que soit sa localisation. L'envoi est immédiat et asynchrone. L'activité exécutant cette instruction n'est suspendue que pendant le temps de prise en charge du message par le système de communication.

Selon le type des résultats des expressions, les données envoyées seront soit des valeurs des types de base, soit des références. Les types des données transmises sont aussi inclus dans le message de manière transparente à l'utilisateur. Les types de base pour lesquels le passage se fait par valeur sont: [INT], [REAL], [BOOL], [CHAR], [STR], [MESS] et [ARR].

Exemple :

```
b <- ! 5, "bonjour" ,self !;
```

Le message transmis contient ici un entier, une chaîne (recopiée dans le message) et une référence à l'objet qui exécute l'instruction.

IV-6-4 La réception de message

Un message véhiculé par le système de communication est déposé dans la boîte aux lettres destinataire sans perturbation de l'entité qui possède cette boîte. Cette entité doit utiliser explicitement l'opération de réception pour obtenir les messages déposés dans sa boîte. Seul le propriétaire d'une boîte peut utiliser l'opération de réception sur cette boîte.

Il s'agit en fait d'une opération d'extraction qui retourne un message dans l'espace de l'entité. L'ordre des autres messages dans la boîte n'est pas modifié. L'opérateur de réception est "->".

Le message extrait n'est pas obligatoirement celui en tête de la file de messages (gestion non FIFO) comme on va le voir.

Syntaxe :

```
<RECEPTION> ::= <boîte> -> ! <ATTRIBUTS> !
```

Les attributs sont des attributs ou des variables locales de l'entité qui reçoit le message. Ces attributs vont recevoir les données du message reçu. Ils sont donc écrits (modifiés) par l'opération de réception.

IV-6-5 Règles de réception non FIFO

Les attributs indiqués dans une opération de réception sont typés. Il doit donc y avoir conformité (au sens du système de typage) entre la liste d'attributs et le message qui sera extrait. Notre stratégie se fonde sur le fait que la liste des types des attributs utilisés va servir de filtre pour l'opération de réception.

L'opération de réception utilise donc les règles suivantes :

- Un filtre est une suite de types.
- Le message reçu lors d'une opération de réception à partir d'une boîte aux lettres est le message le plus ancien dans la boîte qui vérifie le filtre utilisé (ce n'est donc pas obligatoirement le plus ancien de tous les messages).
- Un message vérifie un filtre si, donnée après donnée, les types des données qu'il contient sont conformes aux types du filtre utilisé. Il peut y avoir plus de données que de types dans le filtre ; dans ce cas les données supplémentaires ne sont pas filtrées (ou plus précisément sont filtrées avec [ANY]).
- Si aucun message ne vérifie le filtre, l'opération est suspendue dans l'attente d'un message le vérifiant.

On remarquera:

1) que l'opération de réception est une opération bloquante si aucun message ne vérifie le filtre. C'est donc fondamentalement une opération d'attente de message. Une opération non bloquante de réception conditionnelle sera introduite plus loin.

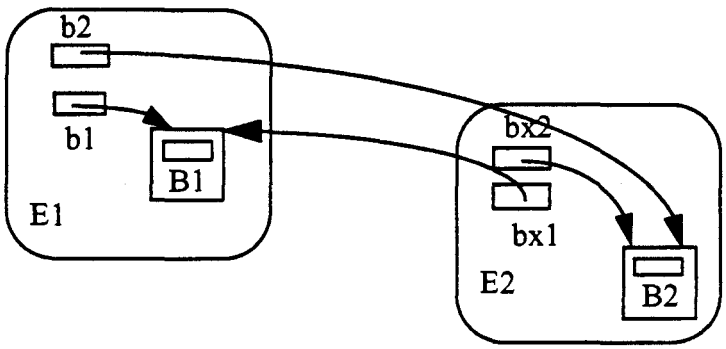
2) que les données supplémentaires (non filtrées) d'un message sont perdues car il n'y a pas d'indication de contenants pour elles. Les formes étendues décrites plus loin vont permettre de lever ce problème.

3) que la discrimination entre messages ne peut se faire que sur la forme des messages

(au sens suite de types) et non sur le contenu. Cette deuxième possibilité sera aussi introduite par les formes étendues, en particulier aux contraintes.

Exemple1 :

Prenons l'exemple d'un protocole simple de question/réponse entre deux entités E1 et E2 possédant respectivement les boîtes B1 et B2. Les boîtes étant référencées par les deux entités comme l'indique le schéma ci-dessous.



Dans cet état, un protocole question/réponse (ici une demande de la multiplication de deux nombres) doit s'établir de la manière suivante :

Du côté E1 (le questionneur) :

```
b2 <- ! n1, n2 !; --envoi du message contenant deux entiers
b1 -> ! result !; --attente du résultat
```

Du côté E2 (le répondeur) :

```
bx2 -> ! a, b !;    --attente de deux entiers
bx1 <- ! a * b !;   -- envoi du résultat
```

Exemple 2 :

Prenons maintenant l'exemple d'un protocole client/serveur. Une entité S est capable de traiter des multiplications d'entiers comme ci-dessus pour le compte d'entités clientes Ci. Dans ce cas les Ci qui connaissent la boîte b pour le service de S vont passer dans le message une de leurs propres boîtes pour la réponse :

Du côté des clients Ci :

```
b <- ! n1, n2, rep !; -- envoi
rep -> ! result !      -- attente résultat
```

Du côté du serveur S :

```
loop -- fonctionnement serveur
  my_box -> ! a, b, a_box !; --attente requête
  a_box <- ! a * b !;        --envoi résultat
end_loop
```

IV-6-6 La réception avec contraintes

Des contraintes, ajoutées à l'opération de réception, permettent une discrimination des messages (vérifiant un même filtre) à partir de leurs contenus.

Une contrainte est une expression booléenne sans effet de bord, exprimée à partir des données du message examiné.

Une opération de réception avec contraintes permet de recevoir le plus ancien message de la boîte utilisée qui 1) vérifie le filtre, 2) vérifie les contraintes.

Une contrainte est dite vérifiée si son résultat donne vrai. Un ensemble de contraintes est vérifié si toutes les contraintes donnent vrai.

Syntaxe :

```
<RECEPTION> ::= <boîte> -> ! <ATTRIBUTS> ! { <CONTRAINTES> }
```

Dans l'expression des contraintes, les différentes données du message testé s'écrivent sous la forme de pseudo variables `$1, $2, ..., $n` . Elles sont implicitement déclarées avec les types du filtre.

Exemples :

```
b -> ! n1, n2 ! { $1 > $2}
  -- attente d'un message de deux entiers
  -- dont le premier est supérieur au second
b -> ! op, n1, n2 !
        { $1 = "addition" or $1 = "multiplication" }
  -- attente d'une requête d'addition ou de
  -- multiplication de deux nombres
b -> ! point_recu ! { $1.x < x , $1.y <y }
  -- dans un objet point:
  -- attente d'un point plus proche de l'origine que
  -- le point courant
b -> ! date, value ! { $1 = date + 1 }
  -- attente d'un message à date + 1 contenant une
  -- nouvelle valeur. Les attributs date et value
  -- sont modifiés par la réception.
```

En reprenant l'exemple du serveur de multiplication vu plus haut, et en supposant maintenant que 1) les deux nombres à multiplier et la boîte pour la réponse arrivent dans des messages différents, 2) que ces paires de messages peuvent être entrelacées pour les différents clients, le serveur s'écrira de la manière suivante:

```
loop
  b -> ! client, n1, n2 ! ;
  b -> ! client, a_box ! { $1 = client } ;
  a_box <- ! n1* n2 ! ;
end_loop
```

Dans un cas comme celui-ci, dans la deuxième opération de réception, il est inutile de recevoir le nom du client du message dans l'attribut `client`. En effet seule la comparaison avec l'ancien nom dans `$1 = client` est importante pour identifier la deuxième partie de la requête.

Pour des raisons comme celle-ci, il est possible d'utiliser la pseudo variable $ pour indiquer qu'une donnée d'un message n'a pas à être mémorisée après l'opération de réception. Lorsqu'elle est utilisée, cette pseudo variable doit être explicitement typée.

Pour le serveur précédent, le code peut donc s'écrire :

```
loop
  b -> ! client, n1, n2 ! ;
  b -> ! [OBJ]$, a_box ! {$1 = client} ;
  a_box <- ! n1* n2 ! ;
end_loop
```

IV-6-7 La réception conditionnelle

La réception conditionnelle est une opération non bloquante qui extrait un message qui vérifie un filtre et des contraintes ; s'il en existe un! Dans le cas contraire, l'opération ne fait rien. Ce n'est donc pas une opération d'attente.

La syntaxe est identique à celle de l'opération de réception présentée dans le paragraphe précédent, avec deux différences: 1) l'opérateur utilisé doit être l'opérateur de réception conditionnelle `"-?>"`, et 2) l'opération est ici une expression qui retourne vrai si une réception a eu lieu, sinon retourne faux si une réception n'a pas eu lieu.

Syntaxe :

```
<RECEPTION> ::= <boîte> -?> ! <ATTRIBUTS> ! { <CONTRAINTES> }
```

Exemple :

```
if b -?> !reponse! {$1 = "ok"}
  then -- message ok reçu
  else -- message ok non reçu
end_if
```

IV-6-8 L'attente simple

L'opération d'attente sur boîte est une opération permettant d'attendre l'arrivée d'un nouveau message dans la boîte. C'est une opération d'attente qui ne réalise pas d'extraction de message. L'opérateur est "?".

Syntaxe :

```
<ATTENTE> ::= <boîte> ?
```

L'opération de réception peut être décrite en utilisant la réception conditionnelle et l'attente simple sur boîte de la manière suivante :

```
b -> ! a1, a2, ...an ! { c1, c2, ..., cm}
```

est équivalent à:

```
loop
  if b -?> ! a1, a2, ...an ! { c1, c2, ..., cm}
    then exit
    else b ?
    end_if
end_loop
```

Exemple :

Prenons l'exemple d'un serveur qui reçoit dans des messages différents des nombres à multiplier avec la stratégie suivante : dès que le serveur reçoit un nom de boîte aux lettres, il retourne dans celle-ci le résultat des opérations précédentes.

Le code pourrait être le suivant :

```
multiplication := 1;
loop
  b ?;
  if b -?> ! n ! then
    multiplication := multiplication * n
  else_if b -?> ! a_box ! then
    a_box <- ! multiplication !;
    multiplication := 1
  end_if;
end_loop
```

On remarquera l'opération d'attente sur boîte qui évite ici l'attente active dans la boucle.

IV-6-9 L'utilisation du type message

Les opérations de réception précédentes peuvent utiliser une variable de type message [MESS] pour récupérer le message reçu plutôt que d'indiquer une liste d'attributs.

Le filtre n'ayant plus à indiquer de contenants, celui-ci s'exprime donc ici comme une suite de types.

Syntaxe :

```
<RECEPTION> ::= <boîte> -> <MESSAGE> !<TYPES>! {<CONTRAINTES>}
```

ou

```
<RECEPTION> ::= <boîte> -?> <MESSAGE> !<TYPES>! {<CONTRAINTES>}
```

La sémantique de réception reste la même sauf que les données du message reçu vont être accessibles à partir du message indiqué.

Dans ce contexte, le filtre et les contraintes peuvent être absents. L'opération consistant alors à récupérer, dans la variable de type message, le premier message de la boîte aux lettres.

Syntaxe réduite de réception:

```
<RECEPTION> ::= <boîte> -> <MESSAGE>
```

De la même manière, l'opération d'envoi peut utiliser une variable de type message pour désigner la suite des données à envoyer.

Syntaxe réduite d'envoi :

```
<ENVOI> ::= <boîte> <- <MESSAGE>
```

IV-6-9-1 Les messages

Un message est une instance de la classe primitive MESS. Un message est manipulé par l'intermédiaire d'un attribut dont la valeur est une référence au message.

Une déclaration d'une variable m de type message s'écrit : [MESS] m. Le message n'est pas ici créé. Seul l'espace nécessaire pour contenir une référence de message est alloué.

Les messages doivent être créés dynamiquement. Un message est créé dans l'espace de l'entité (objet ou fragment) qui réclame la création. Un message peut être créé initialisé, c'est-à-dire contenant déjà des valeurs.

Exemples :

```
[MESS <- ] un_message; -- déclaration et création d'un message vide.
m := [MESS <- ! 2, 3 !]; -- création d'un message initialisé
```

Un message sert à contenir des données. Les opérations de base pour accéder à ce message sont 1) le dépôt de données et 2) l'extraction (conditionnelle ou non) de données. Ces opérations sont décrites ci-dessous. Ces opérations sur les données du message sont analogues à celles sur les messages d'une boîte aux lettres.

IV-6-9-2 Le dépôt de données

Cette opération permet d'ajouter un ensemble de données à la fin d'un message.

Syntaxe :

```
<DEPOT> ::= <MESSAGE> <- ! <EXPRESSIONS>!
```

ou

```
<DEPOT> ::= <MESSAGE> <- <MESSAGE>
```

Les valeurs des expressions (ou le contenu du deuxième message) sont ajoutées séquentiellement en queue du message. Après l'ajout, le message contiendra les données préalablement présentes, suivies des données ajoutées.

Si les résultats sont de type primitif (INT, CHAR, REAL, BOOL, STR, ARR, MESS),

ce sont les valeurs qui sont stockées dans le message ; sinon ce sont des références qui sont déposées.

Exemple :

```
m <- ! 2, [POINT <- position !2,3!], [POINT <- position !7,4!] !;
```

IV-6-9-3 L'extraction de données

L'opération d'extraction dans un message permet d'extraire les données en tête du message et de les placer dans des attributs.

Syntaxe :

```
<EXTRACTION> ::= <MESSAGE> -> ! <ATTRIBUTS>! {<CONTRAINTES>}
```

Elle ressemble syntaxiquement et sémantiquement à l'opération d'extraction de message d'une boîte aux lettres (utilisation de filtre et de contraintes). Il y a toutefois une différence majeure : si le message dans lequel on tente d'extraire les données ne vérifie pas les conditions indiquées, une erreur est déclenchée.

Exemple :

```
m -> !nb_point!;
loop
  if nb_point = 0 then exit
    else
      m -> ![POINT]point_courant!;
      point_courant.afficher;
      nb_point := nb_point-1;
  end_if;
end_loop
```

IV-6-9-4 L'extraction conditionnelle de données

L'extraction conditionnelle est une opération qui extrait d'un message des données vérifiant un filtre et des contraintes, si elles existent ! Dans le cas contraire, l'opération ne fait rien.

La syntaxe est identique à l'opération d'extraction présentée dans le paragraphe précédent, avec deux différences : 1) l'opérateur utilisé doit être l'opérateur d'extraction conditionnelle `"-?>"`, et 2) l'opération est ici une expression qui retourne vrai si une extraction a eu lieu, faux dans le cas contraire.

Syntaxe :

```
<EXTRACTION> ::= <MESSAGE> -?> ! <ATTRIBUTS>! {<CONTRAINTES>}
```

IV-7 Les objets et les fragments

IV-7-1 Introduction

BOX permet la déclaration de classes d'objets et de classes de fragments. Une classe d'objets est construite à partir de la classe OBJ. Une classe de fragments est construite à partir de la classe FRAG. Un objet est une instance d'une classe d'objets. Un fragment est une instance d'une classe de fragments.

Bien que les syntaxes des classes d'objet et de fragment soient identiques, les objets et les fragments se distinguent fortement par leur protocole d'utilisation. Les objets sont utilisés par l'appel de procédure, les fragments utilisent la communication par messages.

IV-7-2 Classes d'objets et de fragments

Une classe a la forme syntaxique suivante :

```
[sur_classe] nouvelle_classe : :
-- déclarations d'attributs
-- déclarations de procédures
```

`nouvelle_classe` est le nom de la nouvelle classe définie à partir de la classe de nom `sur_classe`.

`[nouvelle_classe]` est un nouveau type, sous-type de `[sur_classe]`.

IV-7-3 Les attributs

Les déclarations d'attributs déclarent les connaissances locales de chaque instance. Ces attributs ne peuvent être modifiés que par les procédures de la classe. Les déclarations d'attributs ne peuvent pas contenir de requêtes de création.

IV-7-4 Les procédures

Une procédure est déclarée à partir du type [PROC] (ou [SPROC] pour les objets synchronisés). Le corps d'une procédure est un programme séquentiel qui peut :

- déclarer et initialiser des attributs locaux.
- manipuler en lecture et écriture les attributs de la classe (en particulier les initialiser).
- appeler les procédures des objets référencés par les attributs.
- envoyer des messages aux boîtes aux lettres ou fragments référencés par les attributs.
- recevoir des messages des boîtes aux lettres créées par l'instance.

Une procédure déclare des paramètres formels qui sont manipulables dans le corps de la procédure comme des attributs locaux accessibles en lecture seulement.

Une procédure peut être une fonction. L'attribut local de nom `Result` est alors automatiquement déclaré du type du résultat. La valeur de `Result` est fournie à

l'appelant au retour de la fonction.

IV-7-5 Politique d'utilisation des objets

Un objet est une entité passive soumise aux requêtes des activités concurrentes d'un programme. Un objet fournit une interface constituée des attributs (en lecture) et des procédures déclarés dans sa classe et une politique implicite de synchronisation des accès concurrents.

IV-7-5-1 Création d'objets

Un objet est créé par l'envoi d'un message à la classe concernée. La requête de création peut désigner une procédure de la classe comme procédure d'initialisation de l'objet créé.

Une création est une expression qui retourne une référence à l'objet créé.

A la suite de la création, l'objet existe et est accessible à partir de tout attribut qui le référence. L'objet est automatiquement détruit par un garbage collector, lorsqu'il n'est plus référençable.

IV-7-5-2 Accès à l'interface

L'interface d'un objet est constituée des attributs déclarés dans sa classe - ils ne sont accessibles qu'en lecture - et des procédures déclarées dans sa classe.

L'accès à un élément de l'interface se fait à partir d'une référence à l'objet, et en utilisant la notation pointée.

Syntaxe pour les procédures :

```
<ACCES_PROCEDURE> ::= <REF_INSTANCE> . <PROCEDURE> ( <PARAMETRES> )
```

La partie `(paramètres)` doit être absente si la procédure ne déclare pas de paramètres.

S'il s'agit d'une fonction, l'accès est une expression, sinon c'est une instruction.

Syntaxe pour les attributs :

```
<ACCES_ATTRIBUT> ::= <REF_INSTANCE> . <ATTRIBUT> [ <VALEUR> ]
```

La partie `[valeur]` ne doit être utilisée que pour les attributs de type `[ARR]` pour fournir la valeur de l'index.

L'accès à un attribut est une expression.

L'accès d'un objet à sa propre interface ne nécessite pas la désignation d'un objet. Néanmoins le pseudo attribut nommé `Self` peut être utilisé. Cet attribut, implicitement déclaré, contient toujours une référence à l'objet courant.

IV-7-5-3 La synchronisation

Un objet peut être soumis à des accès concurrents provenant d'activités parallèles. Pour maintenir la cohérence interne de l'objet, il est nécessaire d'offrir une politique de

synchronisation qui va contrôler ces accès.

Pour cela BOX introduit des procédures SPROC (Synchronized Procedure) qui permettent aux objets d'être munis d'une politique implicite de synchronisation de type lecteurs/rédacteurs. Lorsqu'une SPROC est présente dans une classe, cela signifie que les accès aux instances de cette classe seront synchronisés. Les procédures SPROC sont alors considérées comme des procédures rédactrices, les procédures PROC (et la lecture des attributs) comme des procédures lectrices.

La règle de synchronisation est qu'une procédure rédactrice a l'accès exclusif à l'objet.

Les self-invocations ne sont pas concernées par la synchronisation.

Exemple :

```
[OBJ]data [T] :: -- classe générique pour une donnée protégée
  [T]d;
  [SPROC] set_d: [T]new_d:
    { d := new_d }
```

IV-7-6 Politique d'utilisation des fragments

Un fragment est une entité active. Un fragment exécute du code décrit dans sa classe. Il ne peut être sollicité par appel de procédure, mais uniquement par la communication par messages. Tout fragment a une boîte aux lettres implicite qui l'identifie pour le système de communication.

IV-7-6-1 Création de fragments

Les procédures d'une classe de fragments doivent être vues comme des comportements possibles des instances de cette classe. Une de ces procédures doit être choisie à la création du fragment, pour fixer le comportement du fragment créé.

Syntaxe :

```
[<CLASSE_FRAG>  <- <PROCEDURE> ! <PARAMETRES> ! ]
```

La création d'un fragment entraîne la création d'un nouveau processus prenant en charge le comportement indiqué. Après création de ce processus, une référence au fragment est retournée au demandeur de la création. Les références à un fragment sont utilisées pour lui envoyer des messages.

Le fragment devient inactif lorsque son comportement se termine. Il sera ramassé par le garbage collector. C'est aussi le cas lorsque le fragment est bloqué et plus référencé.

IV-7-6-2 Communication avec les fragments

Tout fragment est créé avec une boîte aux lettres implicite. Il est possible d'envoyer un message vers cette boîte dès que l'on possède une référence au fragment, en utilisant cette référence comme une référence à sa boîte aux lettres. Plus généralement, le type [FRAG] est conforme au type [BOX].

Le fragment lui-même connaît sa boîte sous le nom prédéfini `BOX`. `BOX` référence

toujours la boîte aux lettres implicite du fragment courant. BOX joue un rôle équivalent à `Self`, mais pour les fragments.

Exemple : la classe simple pour un fragment prenant en charge la multiplication de deux nombres s'écrit :

```
[FRAG]multiplieur::
  [PROC]action::
    {
      BOX -> ! [INT]n1, [INT]n2, [BOX]une_boite !;
      une_boite <- ! n1 * n2 !
    }
```

L'utilisation de ce type de fragment prendra la forme:

```
-- création et envoi de la requête
[multiplieur <- action !!] <- ! a, b, ma_boite !;
....
-- attente du résultat
ma_boite -> ! resultat !;
```

IV-7-6-3 Extension du système de communication aux procédures de fragments

Ce qui est proposé ici, c'est de rendre visible les procédures de fragments pour qu'elles puissent être appelées grâce à l'envoi de messages. Un tel appel se fera par un message à destination du fragment et ne sera traité qu'après une réception explicite du message par le comportement du fragment destinataire. Il s'agit ici d'une extension du système de communication à des messages explicites de requêtes des procédures des fragments. L'extension proposée permet au compilateur de vérifier la légalité de ces requêtes.

Exemple : pour demander à un fragment de réaliser une opération "op", il faut lui envoyer un message explicitant cette requête. Voici une façon de procéder (mais tout autre codage de la désignation de l'opération est possible) :

```
le_fragment <- ! "op" , param1, param2 !;
```

L'extension proposée est d'écrire cela de la manière suivante :

```
le_fragment <- op ! param1, param 2!
```

en imposant que op soit une procédure du fragment. Contrairement à la première manière, le compilateur peut ici vérifier la légalité de la requête.

Du côté du fragment, la réception explicite de la demande se fera par une réception de message en filtrant le type [PROC] de la manière suivante :

```
BOX -> ! [PROC] !;
```

Le lancement de la procédure est fait automatiquement dès que l'extraction du message de requête est terminée.

Exemple :

```
[FRAG] util::
  [PROC]multiplication:[INT]n1,n2; [BOX]a_box:
     { a_box <- ! n1*n2 ! }
```

```
[PROC]addition:[INT]n1,n2; [BOX]a_box:
   { a_box <- ! n1+n2 ! }
[PROC]comportement::
   { loop
        BOX -> ![PROC]!
              {$1 = multiplication or $1 = addition};
     end_loop
   }
```

Cette extension permet de concevoir les fragments comme des objets actifs au sens de Caromel [Caromel90b] par exemple, et donc de pouvoir implémenter facilement et de façon sûre les modélisations faites sur cette base.

IV-8 L'héritage et la généricité

IV-8-1 L'héritage

L'héritage simple est utilisé pour concevoir de nouvelles classes. Les classes prédéfinies OBJ et FRAG sont les classes initiales de toutes les classes héritables.

Syntaxe :

```
[sur_classe] nouvelle_classe : :
-- autres attributs
-- autres procédures
```

Une nouvelle classe hérite de tous les attributs et de toutes les procédures de sa sur-classe. Elle peut définir de nouveaux attributs et de nouvelles procédures. La redéfinition d'un élément hérité est autorisée, à condition que la nouvelle signature soit conforme à l'ancienne en respectant la règle de covariance pour les paramètres en entrée et en sortie.

La nouvelle classe (ici `nouvelle_classe`) définit un nouveau type (ici `[nouvelle_classe]`), sous-type du précédent (ici `[sur_classe]`)

Cette notion de sous-typage induit celle de polymorphisme : les attributs peuvent référencer des entités de types différents lors de l'exécution. Ce polymorphisme est ici contrôlé par l'héritage : le type dynamique d'un attribut (type de l'entité référencée) doit être sous-type du type statique (de déclaration) de l'attribut.

En cas de redéfinition d'une procédure héritée, il est possible d'accéder à la définition de la sur-classe en utilisant la notation pointée avec le mot clé `old` comme receveur. L'utilisation de `old` est valable pour les objets et pour les fragments.

IV-8-2 La généricité

La généricité permet de définir des classes génériques, souvent utilisées pour définir des structures de données indépendantes des données manipulées.

La syntaxe à utiliser est:

```
[<SUR_CLASSE>] <NOUVELLE_CLASSE> <TYPES_GENERIQUES> : :
```

Exemple :

```
[OBJ] pile [T] :: -- etc
```

A partir de cette classe il est possible de définir un attribut:

```
[pile[INT]] pile_entiers;
```

Le nombre de paramètres formels de généricité n'est pas limité.

Comme la généricité est non contrainte, il est impossible d'appliquer des procédures dans la classe générique sur un objet du type générique. Une généricité contrainte, c'est-à-dire précisant un type minimal pour un paramètre générique devrait être introduite dans les évolutions futures du langage.

IV-9 Conclusion

Dans notre modèle, des objets et des processus sont décrits par des classes. L'appel de méthode et l'envoi de messages coexistent et des aggrégats d'objets et de processus peuvent être facilement construits et réutilisés. *Contrairement à d'autres travaux, les problèmes de synchronisation ne sont plus réglés dans des objets partagés mais par des communications synchronisantes.* Ces aspects permettent d'envisager la conception efficace et à grain fin d'applications parallèles ou distribuées avec les apports de l'approche objet. L'apport méthodologique, encore à développer, prend corps sous la forme d'Objets Actifs Complexes, véritables objets parallèles potentiellement fragmentables sur un ensemble de sites.

BOX est un langage qui supporte ce modèle et cette méthodologie. C'est un langage facile à utiliser, hautement modulaire et muni d'un environnement qui favorise la réutilisation aussi bien des développements de classes que d'applications. Il est efficacement supporté par un système de processus légers distribués adaptables à différents types d'architectures.

L'environnement et l'implantation sont décrits dans les chapitres suivants.

Chapitre - V -

BOX: Un environnement pour la programmation distribuée

V-1 Introduction

Le langage introduit dans le chapitre précédent se focalise sur l'aspect programmation des classes en ne tenant pas compte, conformément au modèle, de la répartition de l'architecture sous-jacente. Une étape importante de la création d'une application répartie est son installation sur la machine cible. Il s'agit de la phase de configuration.

L'environnement BOX offre un certain nombre d'outils permettant au concepteur, à partir des classes produites dans la phase de programmation, de produire l'ensemble des exécutables nécessaires à l'exécution répartie, et cela en minimisant les retours sur la phase de programmation. On trouve :

- un compilateur de classes qui permet la compilation individuelle de chaque classe.
- un constructeur d'applications qui se charge de compléter les compilations individuelles par une édition de liens globale pour l'application.
- un langage de description de la répartition des classes sur des sites logiques et

permettant donc la fabrication des différents exécutables.

- un lanceur d'applications qui installe les sites logiques sur les sites physiques de l'architecture et qui lance l'application répartie.

La flexibilité de l'environnement est importante et permet par exemple :

- de construire des applications différentes à partir d'un ensemble de classes précompilées.
- de changer facilement la répartition du code dans les sites logiques sans recompilation des classes.
- de changer la configuration physique (nombre de processeurs) sans changer la répartition logique de l'application.
- et enfin d'avoir une démarche de réutilisabilité par la possibilité de récupérer des descriptions d'applications préexistantes pour les adapter et les étendre. Cette possibilité est due au langage ensembliste utilisé pour décrire les applications.

Ces outils et possibilités sont introduits dans les sections suivantes.

Un autre aspect de la prise en compte de la répartition est traditionnellement la possibilité de fixer un site précis lors de la création d'une instance. Cette technique a été évaluée pour le langage BOX. Il s'agit dans la programmation d'associer aux opérations de création un paramètre de localisation. Cela est décrit dans la section V-6.

V-2 Compilateur de classes

Les différentes classes d'une application peuvent être placées dans un nombre quelconque de fichiers sources BOX, d'extension `.bx`. Il peut donc y avoir plusieurs classes par fichier. Syntaxiquement, dans un fichier, deux classes doivent être séparées par au moins un '|'.

Le compilateur de classes BOX s'appelle bc. Il prend en entrée des fichiers d'extension .bx et produit un certain nombre de fichiers résultats qui seront utilisés par le constructeur d'applications.

Commande :

bc [fichier, ...] [-d destination]

fichier
: fichier source BOX (extension .bx)

-d destination
: spécifie le répertoire dans lequel seront stockés les résultats.
Répertoire par défaut : 'result'.

La compilation d'une classe ne nécessite aucune autre information que la classe elle-même. En particulier la compilation d'une classe ne tient pas compte des dépendances possibles entre classes et donc ne nécessite pas, par exemple, qu'une classe utilisée par une autre classe soit compilée au préalable. L'ordre de compilation des classes peut être

quelconque.

Pour cela, la compilation d'une classe génère un fichier contenant les éléments exportés par cette classe, ainsi qu'un fichier contenant les éléments importés par cette classe. Ce n'est qu'au niveau de la construction d'une application que sera vérifiée la bonne correspondance entre les éléments importés et exportés par chaque classe au vu de la liste de classes impliquées dans l'application.

Une possibilité intéressante, conséquence de ce type de compilation, est que les différentes classes d'une application peuvent être compilées, non seulement dans un ordre quelconque, mais aussi en parallèle sur un environnement de type réseau de stations de travail (voir l'option -c du constructeur d'applications).

V-2-1 Génération du code

Nous allons présenter les résultats de la compilation d'un fichier source BOX.

V-2-1-1 Informations par fichier source BOX

Pour chaque fichier contenant des classes, le compilateur génère un fichier de même nom que le fichier source avec l'extension .BX qui énumère l'ensemble des noms des classes définies dans le fichier.

V-2-1-2 Informations par classe BOX

Pour chaque classe, le compilateur va générer :

- la liste des types utilisés
- les fonctionnalités offertes par la classe
- les dépendances
- le code C

Pour une classe de nom 'X', nous allons obtenir quatre fichiers.

V - 2•1•2•1 La liste des types utilisés

Le fichier 'X.types' contient la liste des types utilisés par une classe. Les types sont des noms de classes d'objets ou de fragments définies par l'utilisateur.

V - 2•1•2•2 Les fonctionnalités offertes par la classe

Le fichier 'X.give' contient la description de l'ensemble des fonctionnalités de la classe. Nous y trouvons la sur-classe utilisée[1], la liste des attributs et des procédures de la

classe.

Pour un attribut ou une constante, nous trouvons son nom symbolique et son type.

Les informations relatives à une procédure ou fonction sont :

- le nom symbolique de la procédure ou fonction.
- le type de retour s'il y a lieu d'en avoir un.
- la liste des paramètres avec leurs noms symboliques et leurs types.

V - 2•1•2•3 Les dépendances

Nous trouvons dans le fichier X.need toutes les expressions qui n'ont pu être vérifiées pendant la compilation de la classe. Par exemple, un appel de routine sur une instance d'une autre classe.

Les dépendances sont :

- La lecture d'un attribut d'un objet référencé.
- L'appel d'une procédure ou d'une fonction sur un objet.
- L'appel d'une procédure de fragment sur un fragment.
- La création d'un objet ou d'un fragment.
- L'accès en lecture ou en écriture à un attribut hérité.
- L'appel d'une procédure ou fonction héritée.

V - 2•1•2•4 Le code C

Le compilateur BOX génère le code C (fichier 'X.c') correspondant à la classe. Les expressions qui n'ont pu être vérifiées sont codées sous forme d'une macro : en effet le code non vérifié est supposé correct à ce niveau. Cette macro sera générée par l'éditeur de liens lorsqu'il vérifiera les dépendances.

Il faut souligner que le code C et les autres fichiers générés par le compilateur sont les mêmes quelle que soit l'architecture sur laquelle va tourner l'application. La création d'une application pour différentes architectures ne nécessite qu'une seule compilation BOX et autant de compilation du code C qu'il y a d'architectures cibles ceci en vue d'obtenir des exécutables spécifiques à chaque type de machine.

---

1. Simplement une référence au type de la sur-classe ; la forme symbolique du type est dans le fichier des types.

V-3 Constructeur d'applications

Une application réunit un ensemble de classes BOX. Une de ces classes (obligatoirement une classe d'objet) est appelée la classe racine de l'application. Une de ses procédures servira au lancement de l'application. Cela signifie que l'exécution commencera par la création d'une instance de la classe racine et par la création d'un premier flot d'exécution exécutant la procédure désignée. Les autres objets et flots d'exécution seront ensuite dynamiquement créés en fonction de l'évolution de l'application.

Une application répartie est typiquement formée d'un ensemble d'exécutables installés sur les différents noeuds de l'architecture. Nous avons choisi de ne pas installer l'ensemble du code sur tous les noeuds, mais de laisser la possibilité au programmeur de décrire explicitement la répartition du code sur les différents noeuds. Nous introduisons pour cela la notion de module dans la section suivante.

Une application doit être décrite par un fichier de description d'application d'extension `.cluc`. Ce fichier peut comporter:

- la liste des fichiers `.bx` contenant les classes de l'application.
- une liste d'autres fichiers `.cluc` à inclure permettant ainsi de réutiliser des fichiers de description.
- la classe d'objet racine de l'application
- la procédure de cette classe qui sera lancée au démarrage de l'application
- la répartition en modules des classes de l'application

La syntaxe d'un fichier `.cluc` est la suivante:

```
(INCLUDE)
      -- liste fichiers .cluc à inclure
      "nom_fichier.cluc"
(BOXFILES)
      -- liste fichiers .bx à utiliser
      "nom_fichier.bx"
(APPLICATION)
      -- classe d'objet racine
      nom_classe
(START)
      -- procédure de lancement de la classe racine
      nom_procedure
(MODULES)
      -- description de la répartition en modules
```

Commande:

cluc fichier [-c number]

fichier
fichier de description d'application (extension .cluc)

-c number

Spécifie le nombre maximum de compilations en parallèle possibles.
La valeur par défaut est 5.

Le CLUC analyse la description d'application, lance les compilations BC pour les classes non encore compilées, vérifie la cohérence de l'ensemble des dépendances entre classes de l'application et produit les exécutables.

V-3-1 Arborescence d'une application

Lorsqu'un utilisateur demande la compilation d'une application (par exemple 'appli.cluc'), un répertoire est créé (appli.CLUC). L'ensemble des résultats de compilation vont y être stockés.

appli.cluc
appli.CLUC/

résultats des
compilations bc
(*.c *.types *.give *.need *.BX)

résultats du
constructeur d'application
graphe de conformité
initialisation des modules
macros pour les dépendances
makefile

ALPHA/
fichiers objets
exécutables ALPHA

SUN4/
fichiers objets
exécutables SUN4

SUN4SOL2/
fichiers objets
exécutables SOLARIS

figure 5.1 *Hiérarchie de fichiers pour la compilation d'une application*

A un premier niveau (appli.CLUC), nous trouvons l'ensemble des résultats de compilation de *bc* ainsi que les informations générées par le *CLUC* pour construire l'application. Ces informations sont indépendantes de l'architecture sur laquelle vont

s'effectuer les compilations du code généré. A un niveau inférieur (par exemple appli.CLUC/ALPHA), nous trouvons les résultats des compilations en langage C ; nous trouvons les différentes parties de l'application parallèle.

V-3-2 Informations générées

Les informations générées par le CLUC sont :

- Le graphe de conformité entre classes.
- Les structures de classes pour le run-time.
- Le remplissage des macros.
- L'initialisation des modules.
- Le makefile et le fichier de configuration pour le lanceur d'application.
- Le fichier récapitulatif pour le SHAKER[1].

V-3-2-1 Le graphe de conformité entre classes

Le graphe de conformité permet de vérifier pendant l'exécution si une classe est sous-classe d'une autre. Ce graphe est utilisé dans les opérations de réception de messages pour appliquer les filtres.

V-3-2-2 Les structures de classes pour le run-time

Les structures de classes utiles pendant l'exécution sont le tableau de pointeurs de fonctions, le tableau des noms de routines et le tableau des noms d'attributs. Les deux derniers tableaux permettent de connaître à l'exécution la structure des objets.

Le tableau de pointeurs de fonctions permet de sélectionner la fonction appelée selon le type de l'objet utilisé (polymorphisme). Ce travail est fait en temps constant quelle que soit la profondeur de l'arbre d'héritage.

Le constructeur d'applications génère également une fonction d'initialisation des instances. Cette fonction est appelée à chaque création d'une instance. Elle permet d'initialiser les attributs avec leurs valeurs par défaut.

Ces structures sont générées pour chaque classe de l'application.

V-3-2-3 Le remplissage des macros

Le CLUC génère les macros qui sont appelées dans le code C généré par BC. Les macros générées consistent en :

- l'accès aux attributs des objets référencés
- des appels dans les tableaux des pointeurs de fonctions générés précédemment

1. Le SHAKER est l'outil graphique de répartition des classes en modules (voir la section V-4-4).

pour les appels aux fonctionnalités héritées ou en tant que client.

V-3-2-4 L'initialisation des modules

Ce traitement génère le code de lancement des différents exécutables (voir la section V-4) qui composent l'application. L'initialisation d'un exécutable consiste en l'appel des fonctions d'initialisation des classes qui sont présentes dans cet exécutable.

V-3-2-5 Le makefile et le fichier de configuration pour le lanceur d'application

Le makefile permettant la compilation de l'ensemble du code de l'application est généré automatiquement ainsi que le *cdl_gen* qui permet de générer le fichier '.cdl' correspondant à l'application. Ce fichier servira à l'outil C.D.L. qui permet de lancer une application distribuée sur un réseau de processeurs (voir section V-5).

V-3-2-6 Le fichier récapitulatif pour le SHAKER

Ce fichier contient l'ensemble des répartitions en modules des classes de l'application. Il sert de point de départ pour l'outil de répartition des classes. Le fichier contient la liste des classes utilisées, l'ensemble des opérations ensemblistes décrites dans les différents fichiers .cluc de l'application (création de module, ajout/retrait de classe ...). La répartition est présentée dans la section suivante.

V-4 Répartition en modules

V-4-1 Présentation

La notion de module recoupe ici celle d'exécutable. A chaque module indiqué dans le fichier de description de l'application correspondra, à l'exécution, un processus Unix supportant 1) les fragments créés dans ce module (c'est-à-dire des flots d'exécution pris en charge ici par des processus légers) et 2) les exécutions des procédures sur les objets créés dans ce module (c'est-à-dire des RPC pris en charge ici aussi par des processus légers).

La description d'un module liste les classes qui formeront le code exécutable accessible dans ce module. La description des modules est donc une opération de répartition des classes utilisées par une application sur les sites logiques que sont les modules. Supposons qu'une application utilise cinq classes A, B, C, D, E, une répartition en 5 modules peut être par exemple :

Module 1: classes A, B et C

Module 2: classes B et C

Module 3: classes B et C

Module 4: classes D et E

Module 5: classes D et E

Une telle répartition indique que 5 sites logiques sont souhaités, que les instances de la classe A seront sur le site 1, que les instances des classes B et C seront réparties sur les sites 1, 2 et 3, et que les instances des classes D et E utiliseront les sites 4 et 5.

Pour une exécution, les modules doivent être installés sur les sites physiques de l'architecture (voir la partie Lancement d'une application). En cours d'exécution, un objet ou un fragment ne peut être créé que dans un module qui comprend la classe correspondante. Lorsque la classe est présente dans plusieurs modules, le système choisit un de ces modules pour y réaliser la création. Différentes stratégies peuvent être mises en place : choix aléatoire, cyclique, équilibrage, ... La thèse de Fred Hémery [Hemery94] a fait le point sur les différentes possibilités de régulation dynamique de la charge et leurs implantations techniques. Les différentes solutions retenues peuvent être récupérées ici.

Une répartition en modules doit être conçue sachant que :

- la communication intra-module est beaucoup plus performante que la communication inter-module, puisqu'il s'agit de communication à l'intérieur d'un processus UNIX sans utiliser le réseau de communication. Des instances qui communiquent beaucoup doivent donc au maximum être regroupées à l'intérieur d'un même module.

- la répartition des instances d'une classe dans plusieurs modules permet d'espérer un parallélisme réel entre ces instances (si les modules sont installés sur des processeurs différents), alors que si toutes les instances d'une classe sont dans un seul module, il n'y aura qu'un pseudo-parallélisme à l'intérieur d'un processus[1]. Des instances qui calculent beaucoup doivent donc être au maximum réparties dans plusieurs modules.

- des classes particulières peuvent avoir besoin de ressources matérielles particulières qui imposent que les instances soient toujours présentes sur un site particulier.

Déterminer une bonne répartition des classes n'est pas un travail facile. L'éventail des répartitions possibles est très large. Les extrêmes sont d'une part l'installation de toutes les classes sur tous les modules, ce qui laisse la plus grande marge de manoeuvre à une stratégie dynamique de régulation de la charge (lorsqu'une instance doit être créée, n'importe quel site peut être choisi), et d'autre part une répartition très particularisée résultat d'une analyse très fine de l'application et du comportement de ses entités. Dans ce dernier cas, la démarche peut s'apparenter à la résolution d'un problème de placement statique. Fred Hemery, dans sa thèse, a montré que les algorithmes classiques de placement statique, comme le recuit simulé, peuvent être utilisés ici.

Une remarque importante à faire est que la répartition en modules est un travail qui ne remet pas en cause la programmation des classes. En modifiant la répartition en modules, en essayant différentes répartitions, le programmeur peut facilement tester différentes configurations sans avoir à reprogrammer (ni à recompiler) quoi que ce soit, et donc déterminer de manière cyclique la configuration qui convient le mieux à son application.

1. Sauf si le site est multi-processeurs.

V-4-2 Le langage de description

La description de la répartition des classes en modules utilise un langage ensembliste permettant la réutilisation et l'extension des répartitions trouvées dans les fichiers .cluc inclus.

Les opérations que l'on trouve pour manipuler un ensemble de classes sont divisées en trois catégories :

- La gestion des modules
  - création d'un nouveau module
- La manipulation de classes dans un module
  - ajout d'une classe
  - retrait d'une classe
- La manipulation de modules
  - fusion d'un module dans un autre
  - extraction d'un module d'un autre module
  - duplication

La syntaxe de ce langage de description de la répartition reprend celle du langage BOX lui-même.

- création d'un nouveau module (nom)

```
[MODULE] un_module
```

Un nouveau type de module est créé. Il peut être manipulé à l'aide du nom *un_module*.

- ajout d'une classe

```
un_module <- ! classe_1 !
```

Ajout de la classe *classe_1* au module *un_module*. Plusieurs classes peuvent être ajoutées simultanément. Il faut séparer les noms par des virgules.

- retrait d'une classe

```
un_module -> ! classe_2 !
```

Retrait de la classe *classe_2* du module *un_module*. Plusieurs classes peuvent être retirées simultanément. Il faut séparer les noms par des virgules.

- fusion d'un module dans un autre

```
un_module <- un_autre_module
```

L'ensemble des classes contenues dans *un_autre_module* va être recopié dans le module *un_module*.

- extraction d'un module d'un autre module

```
un_module -> un _autre_module
```

L'ensemble des classes contenues dans *un_autre_module* va être retiré du module *un_module*.

-spécifier un facteur de duplication

```
un_module # 3
```

Le type de module *un_module* existera en trois exemplaires à l'exécution. La valeur par défaut est 1. Cela évite ici de décrire plusieurs fois des modules identiques. Un module dont le facteur de duplication vaut zéro n'existera pas à l'exécution. Toutefois le module peut servir dans d'autres descriptions d'applications.

Lorsqu'une classe est placée dans un module, ses sur-classes le sont aussi ; ce travail est fait de manière automatique. Les classes qui n'ont pas été réparties explicitement dans des modules par le programmeur sont automatiquement placées dans le module de lancement de l'application.

V-4-3 Exemple

Si nous reprenons l'exemple présenté ci-dessus, la description de la répartition peut être faite de la manière suivante :

```
[MODULE] m1, m2, m3 -- Déclaration de trois types de
                    -- modules
m2 <- ! B, C !      -- classes B et C dans m2
m1 <- ! A !         -- classe A dans m1
m1 <- m2            -- recopie du contenu de m2 dans m1
m3 <- ! D, E !      -- classes D et E dans m3
m2 # 2              -- 2 exemplaires de m2 à l'exécution
m3 # 2              -- 2exemplaires de m3 à l'exécution
```

Nous obtenons alors la répartition suivante :

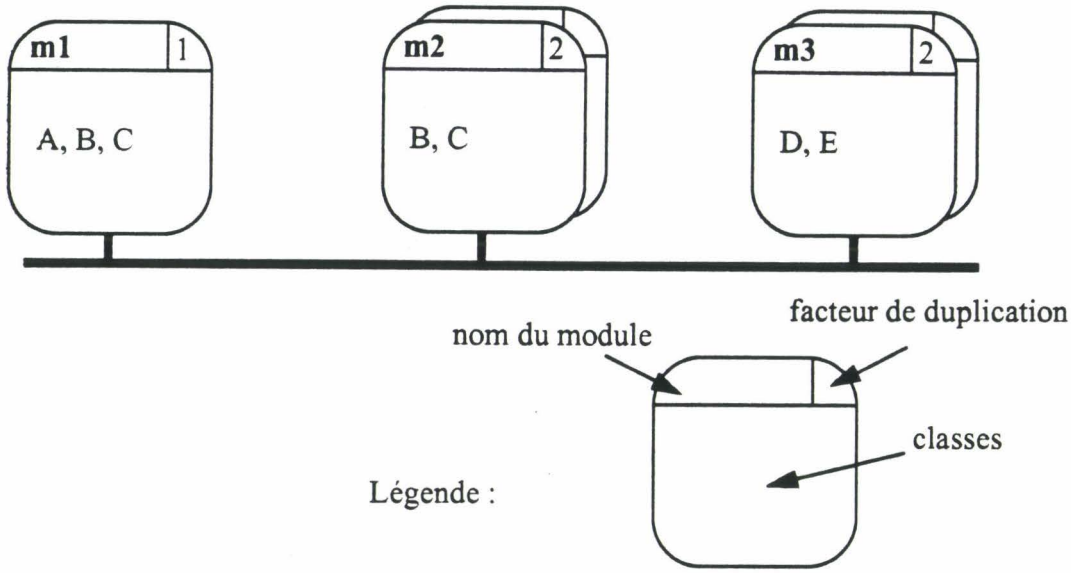


figure 5.2 *Exemple de répartition en modules*

V-4-4 Le SHAKER

Le SHAKER est l'outil graphique qui permet de répartir les classes en modules. Toutes les opérations du langage précédent sont sélectionnables à la souris.

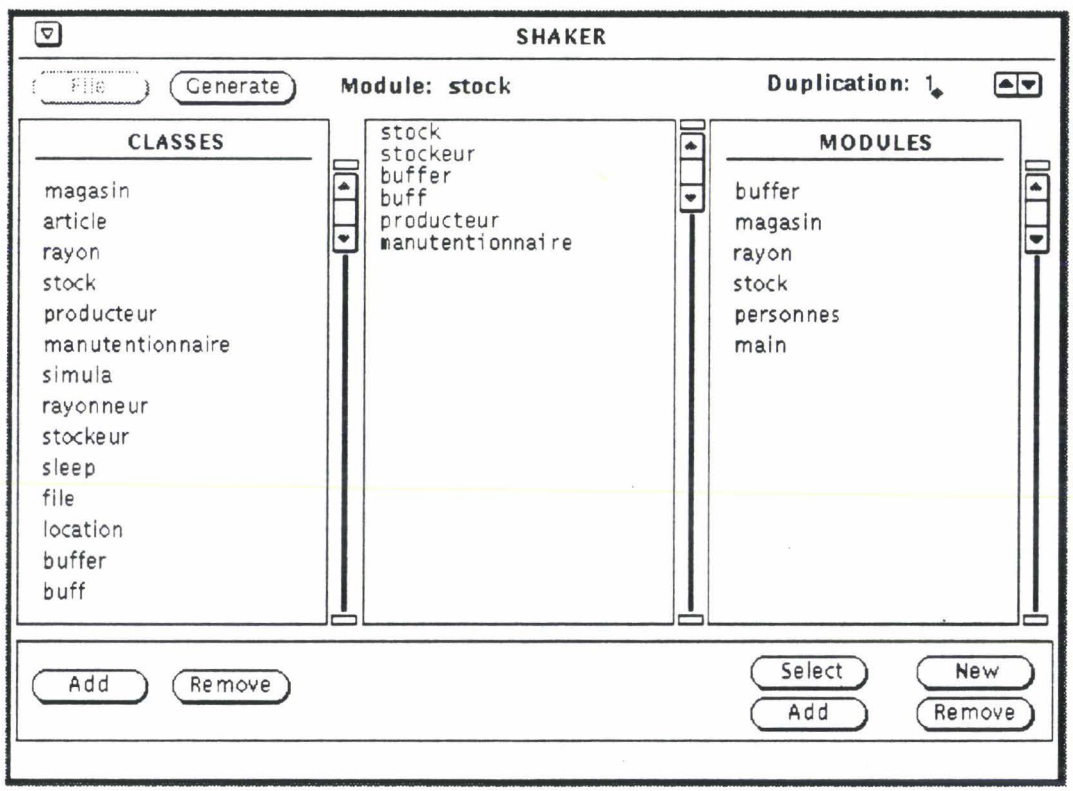


figure 5.3 *L'outil de répartition des classes (SHAKER)*

Le SHAKER génère des instructions ensemblistes qui correspondent aux opérations effectuées par l'utilisateur avec la souris. L'utilisateur peut voir en temps réel le code généré ainsi que la répartition de son application.

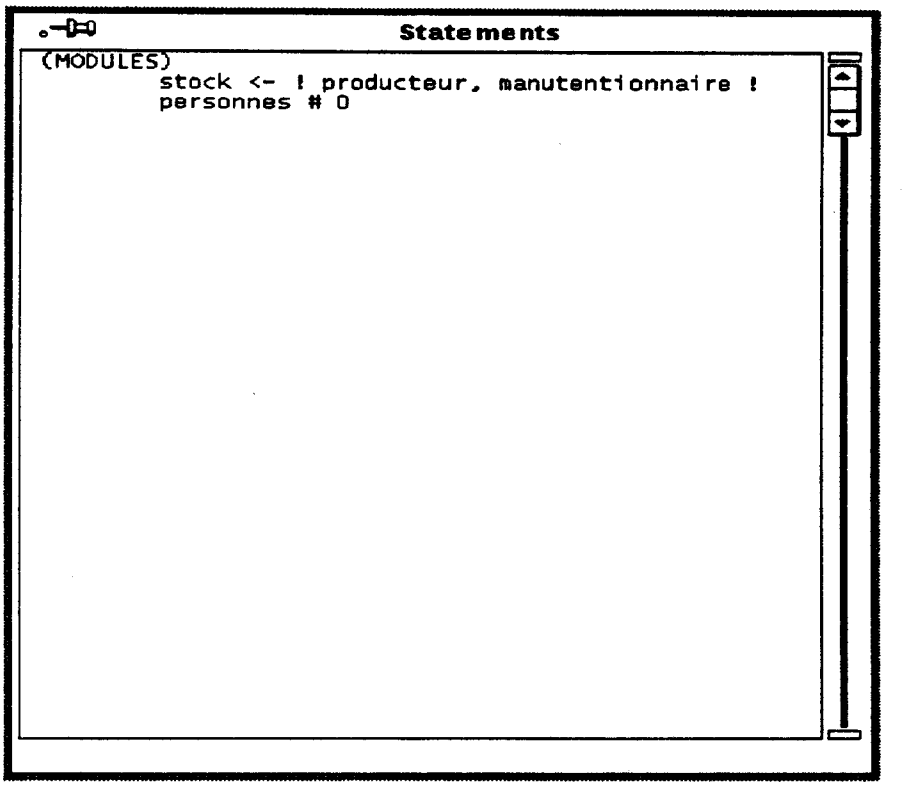


figure 5.4 *Exemple de code généré par le SHAKER*

V-5 Lancement d'une application

Le lancement d'une application nécessite d'abord de donner la liste des machines qui supporteront l'application répartie. L'utilisateur peut spécifier les noms sur la ligne de commande ou dans un fichier. Sur chaque machine, le programme de réseau installe un démon qui se charge de lancer les exécutables des applications réparties.

Un réseau est propre à un utilisateur. Pour une architecture donnée, il n'y a qu'un réseau opérationnel à la fois. Par contre un utilisateur peut avoir plusieurs réseaux simultanément pour des architectures différentes.

C'est la commande *netinst* qui permet de lancer un réseau. Les autres commandes de gestion du réseau sont :

- *netstat* qui donne la liste des modules sur chaque machine.
- *netdel* qui permet de détruire un réseau.
- *netkill* qui permet de tuer un processus d'une application.

Pour rendre la gestion du réseau plus facile, un outil graphique a été développé : Netmaster. Cet outil offre les fonctionnalités présentées ci-dessus.

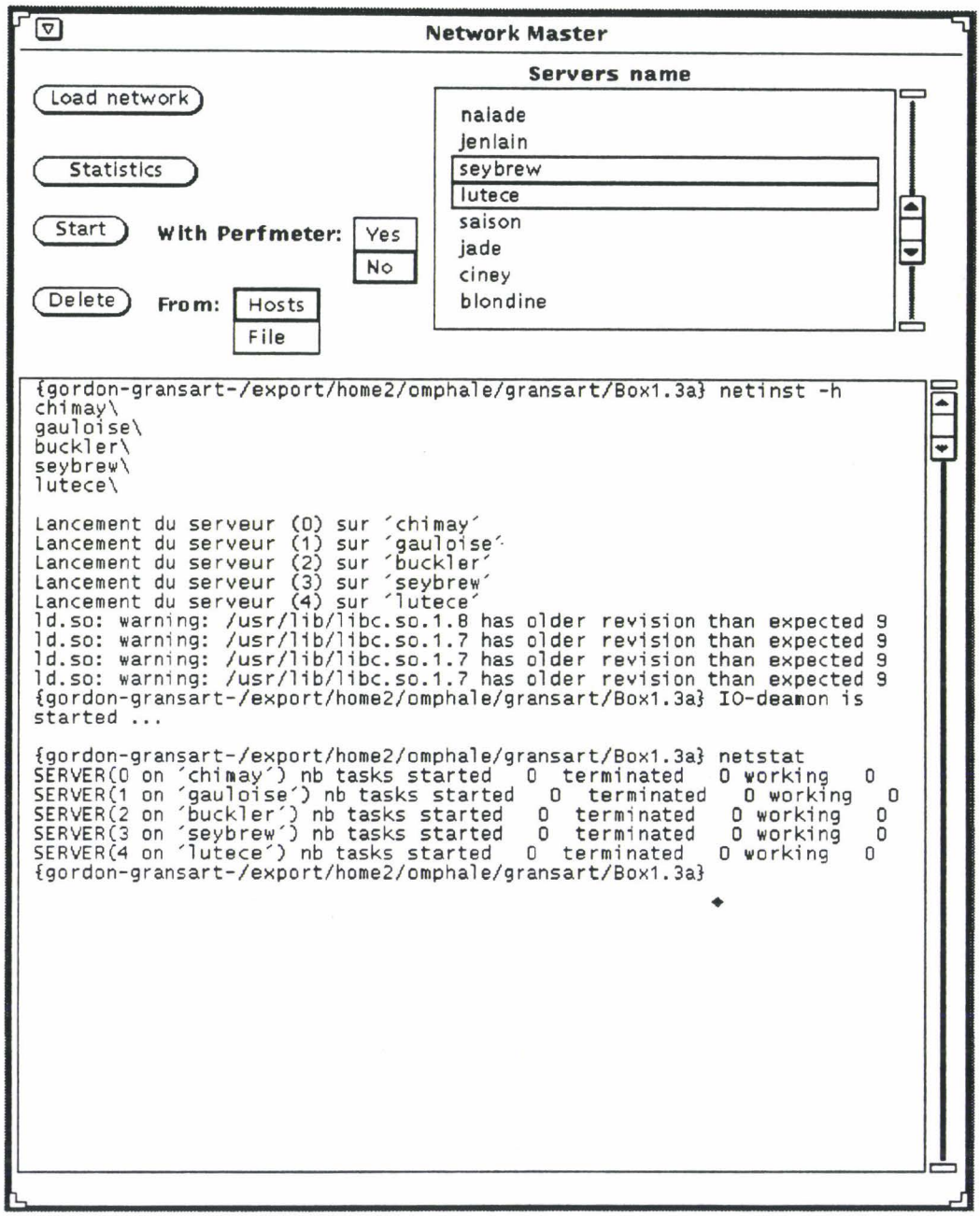


figure 5.5 *Le gestionnaire de réseau*

Lorsqu'une application est réalisée, il faut l'exécuter. La commande *start* se charge du lancement de l'application. Cette commande prend en paramètre le fichier de configuration de l'application (fichier cdl) et va lancer cycliquement un module sur chaque noeud de la machine virtuelle. Si le nombre de noeuds est inférieur au nombre de modules, plusieurs modules se trouveront sur la même machine. Dans le cas contraire, il y aura au plus un module par machine. La commande start établit également les

connections pour les communications entre les différents modules de l'application. Les connections forment un réseau maillé.

Pour améliorer les conditions de travail du développeur, un outil d'automatisation de compilations et de lancement d'applications a été réalisé (*Yawoc : Yet Another Way Of Compiling*). Ce programme permet de lancer des compilations et des exécutions à l'aide de la souris.

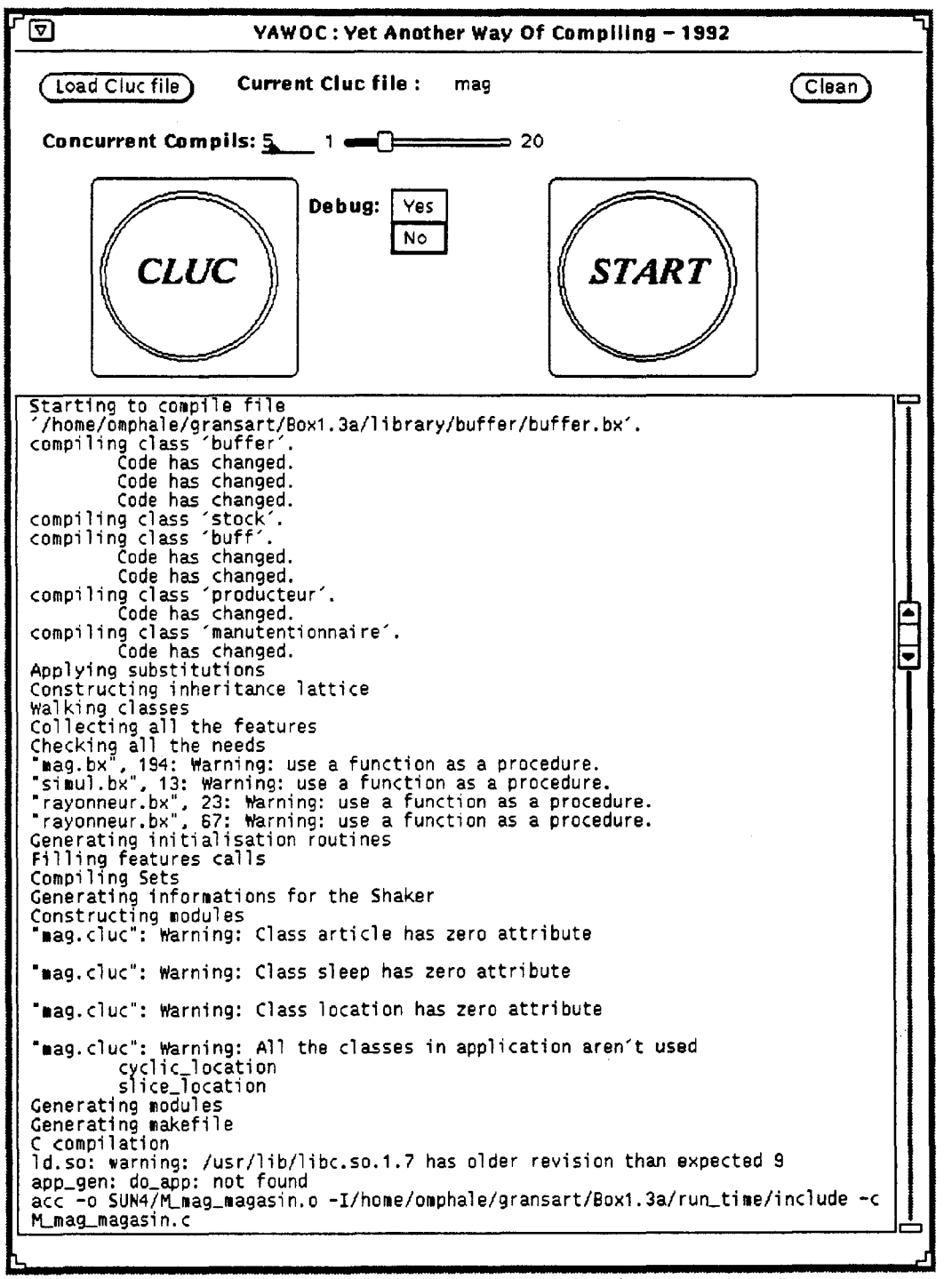


figure 5.6 *Le gestionnaire de compilation*

Résumé

L'environnement de développement BOX permet au concepteur de créer des applications basées sur le langage décrit dans le chapitre précédent. L'environnement offre divers outils pour permettre un développement plus aisé. Pour construire son application répartie, le programmeur dispose d'un langage de description d'application qui lui permet de répartir le code de son programme dans différents exécutables. Un outil graphique lui rend la tâche plus facile. Dans la section suivante, nous présentons le second outil de placement d'une application BOX : le mécanisme de placement à l'exécution au niveau des instances.

V-6 Localisation des instances à la création

V-6-1 Le problème de la localisation des instances

Pendant l'exécution, quand une création d'objet ou de fragment est demandée, un module doit être choisi pour accueillir l'instance. Si aucune autre information n'est transmise au run-time et si plusieurs modules peuvent créer l'instance (parce qu'ils disposent du code de la classe), le run-time choisit un site de manière arbitraire ou selon une stratégie de régulation de charge.

Cette stratégie ne tient pas compte du coût des accès distants à l'objet et ne préserve pas les relations de voisinage entre objets définies pas l'utilisateur. Par exemple, un programmeur d'une structure d'arbre binaire de recherche peut vouloir que les noeuds de son arbre soient distribués par niveaux sur les noeuds de la machine virtuelle[1]. Dans ce cas, des informations pour une distribution particulière doivent être incluses dans le code source et préservées à l'exécution. Nous présentons maintenant les directives de localisation dans le langage BOX.

V-6-2 La localisation de base

Dans BOX, une localisation[2] est un objet qui représente un ensemble de modules. Une localisation peut être transmise à une instruction de création pour restreindre le choix, à l'exécution, de l'ensemble potentiel de modules qui peuvent accueillir la nouvelle instance. Si la localisation ne contient qu'un module, alors la nouvelle instance sera créée sur ce module. Si la localisation contient plusieurs modules, le run-time fera un choix arbitraire parmi les modules possibles.

Au moment de l'écriture de ses composants logiciels, le programmeur ne sait pas quels sont les modules qui seront créés lors de l'exécution de l'application. C'est pourquoi la notion de module n'apparaît pas en tant que telle dans la notion de localisation. Par contre, une localisation peut être spécifiée par rapport à la position d'autres instances

1. Ce n'est pas obligatoirement la meilleure solution !
2. location en anglais.

déjà créées. Par exemple, si une entité E existe sur un module M, le programmeur peut créer un objet de localisation qui permettra la création d'autres objets à la même place que E sans devoir référencer M.

Une localisation est une instance de la classe d'objet LOCATION. A la création d'une instance de localisation, celle-ci est initialisée à zéro module. Après cette initialisation, les fonctionnalités de la classe LOCATION permettent au programmeur d'ajouter des modules dans son objet localisation. L'utilisation la plus courante est de spécifier qu'un objet doit être 'à la même place' qu'un autre objet.

```
[OBJ]LOCATION :: -- set of modules

  [PROC[INT]] nb_modules ::
  -- constant number of modules
  -- in execution

  [PROC [INT]] count ::
  -- number of modules in location

  [PROC] reset ::
  -- reset the set to empty

  [PROC] current ::
  -- set the location
  -- to the current module

  [PROC] as : [ANY] o :
  -- set the location
  -- to the module of o

  [PROC] reverse ::
  -- reverse the set

  [PROC] andi : [ANY] o :
  --current location and module of o

  [PROC] ori : [ANY] o:
  -- current location or module of o

  [PROC] class : [ANY] o :
  -- the modules with the class of o

  [PROC] andl : [LOCATION] l :
  -- current location and location l

  [PROC] orl : [LOCATION] l :
  -- current location or location l

  [PROC [BOOL]] is_present : [INT] i
  -- Is module i present in
  -- the location ?
```

figure 5.7 *Interface de la classe LOCATION*

Dans une implémentation de lecteurs/rédacteurs avec une classe LECTEUR et une classe REDACTEUR installées sur les mêmes modules, le programmeur peut vouloir que, pour chaque couple de lecteur et de rédacteur, ces deux objets ne soient pas installés sur le même noeud. C'est une bonne directive pour obtenir un réel parallélisme entre le lecteur et le rédacteur. Ce choix est implémenté en BOX par :

```
[LOCATION <- ]lo;              -- création d'un objet LOCATION
[LECTEUR <- init ] L;          -- création du lecteur quelque part
lo.as (R) ;                    -- positionnement de la localisation
                               -- avec le module de L
lo.reverse ;                   -- inversion de la localisation
[REDACTEUR <- init at lo] R;-- création du rédacteur quelque part
                               -- mais pas sur le module de L
```

figure 5.8 *Création d'objets avec localisation*

L'information de localisation suit le mot clé `at` dans la demande de création du rédacteur.

V-6-3 Les localisation complexes

Un autre problème classique de distribution est la manière d'exprimer des distributions complexes de certains objets sur des noeuds. BOX permet de décrire facilement des distributions cycliques ou par blocs.

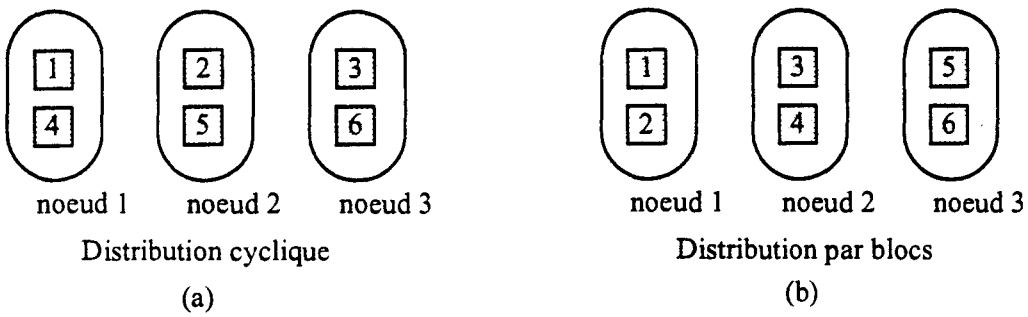


figure 5.9 *Distributions complexes*

La solution que nous avons choisie est de construire des itérateurs de localisation. Deux classes sont fournies dans la bibliothèque BOX : CYCLIC_LOCATION pour une distribution cyclique et SLICE_LOCATION pour une distribution par blocs. Ces deux classes sont construites par sous-classement de la classe LOCATION.

V-6-3-1 La localisation cyclique

La localisation cyclique va fournir un module à chaque appel de la méthode `next`. Le module sur lequel va se faire la création est modifié par l'opération `next` qui sélectionne le prochain module dans la localisation passée en argument.

```
[LOCATION] CYCLIC_LOCATION :: -- one-module location

  [PROC] initialize : [LOCATION] l :
       -- Initialize iterator with the first module of l

  [PROC [BOOL]] end_of_iter ::
       -- End of iteration

  [PROC] next : [LOCATION] l :
       -- Next one-module location from l
```

figure 5.10 *Interface de la classe CYCLIC_LOCATION*

A l'aide de cette classe, nous pouvons distribuer les objets cycliquement. Les objets seront distribués sur les modules qui contiennent le code de la classe concernée.

```
-- On suppose que o est une référence déclarée d'une classe
-- particulière
lo.class (o) ; -- affectation dynamique de la localisation lo à
               -- l'ensemble des modules qui contiennent la classe
               -- de o.

  [CYCLIC_LOCATION] cyc ;
  cyc.initialize (lo) ;
  loop
    if cyc.end_of_iter then exit end_if ;
    o := [UN_OBJET <- init at cyc] ;
    cyc.next (lo) ;
  end_loop ;
```

figure 5.11 *Exemple de distribution cyclique*

V-6-3-2 La localisation par blocs

La localisation par blocs permet de distribuer des instances par groupe. Cette distribution est réalisée à l'aide de la classe SLICE_LOCATION. L'opération next calcule le module à partir des arguments. Il faut fournir les bornes min et max, et le numéro de l'instance que l'on veut créer.

```
[LOCATION] SLICE_LOCATION :: -- one-module location

  [PROC] initialize : [LOCATION] l :
       -- Initialize with the first module of l

  [PROC] next: [LOCATION] l ; [INT] current  ;[INT] bmin, bmax :
       -- Value of the location according to current, bmin, bmax
```

figure 5.12 *Interface de la classe SLICE_LOCATION*

Le code ci-dessous réalise une distribution de six instances sur trois modules. Le nombre de modules est obtenu dynamiquement à l'exécution.

```
-- On suppose que o est une référence déclarée d'une classe
-- particulière
lo.class (o) ; -- affectation dynamique de la localisation lo à
               -- l'ensemble des modules qui contiennent la classe
               -- de o.

  [SLICE_LOCATION] sli ;
  [INT <- 1] i ;
  sli.initialize (lo) ;
  loop
    if i > 6 then exit end_if ;
    o := [AN_OBJECT <- init at sli] ;
    i := i + 1; sli.next(lo,i, 1, 6) ;
  end_loop ;
```

figure 5.13 *Exemple de distribution par blocs*

V-6-3-3 Résumé

Les exemples présentés ci-dessus montrent que notre mécanisme de contrôle de la distribution permet au programmeur de concevoir des distributions complexes de ses objets. La distribution peut être implicite (le run-time choisit seul) ou explicite avec des directives de localisation dans les requêtes de création. Des solutions intermédiaires sont possibles : une localisation contient plusieurs modules et le run-time choisit librement parmi les modules indiqués.

Le mécanisme de localisation fonctionne avec les objets et les fragments de la même manière. Dans le premier cas, les localisations sont utilisées pour distribuer des données ; dans le second cas, les localisations sont utilisées pour distribuer des tâches indépendantes. Ainsi dans BOX, nous trouvons un seul mécanisme pour la distribution de tâches et de données sur une architecture distribuée. De plus, de nouvelles distributions peuvent être créées par le programmeur par sous-classement de la classe LOCATION.

V-6-4 Implémentation

La question que le lecteur peut se poser est de savoir où sont les instances de la classe localisation. La classe LOCATION est une classe "comme" toutes les autres que le programmeur peut écrire. Toutefois cette classe possède une propriété supplémentaire : les instances n'ont pas d'attributs. Dans la terminologie BOX, nous appelons une telle classe "Erwan class"[1]. Nous avons décidé que le code d'une classe avec cette propriété est disponible sur l'ensemble des modules[2] et que les manipulations des instances se font

1. Le nom Erwan est apparu en même temps que le fils de Chrystel Grenot, collaboratrice du projet BOX.
2. Cette répartition du code est réalisée automatiquement par le constructeur d'application.

toujours de manière locale. L'implémentation est détaillée en VII-2-6.

V-7 Conclusion

Le placement des composants d'une application répartie est une étape importante dans sa réalisation : une mauvaise distribution peut donner des résultats médiocres en temps d'exécution. Dans l'environnement BOX, le programmeur dispose de deux outils : le premier permet de répartir les classes dans différents modules qui, à l'exécution, seront représentés par des processus UNIX ; le second permet au programmeur de spécifier un placement fin au niveau des instances.

L'application doit s'exécuter sur un ensemble de noeuds. La liaison entre les noeuds et les exécutables qui forment l'application n'est pas spécifiée dans l'application. Cela permet de changer les machines sur lesquelles vont tourner les exécutables sans devoir modifier l'application.

Chapitre - VI -

BOX: Des exemples

Dans ce chapitre, nous allons présenter quelques exemples réalisés avec le langage BOX. Chacun de ces exemples met l'accent sur un point particulier du modèle et du langage : le calcul de factorielle montre le parallélisme que BOX permet de générer ; la simulation de parking permet une modélisation en terme de processus communicants ; l'exemple du magasin met en valeur le principe de conception à l'aide d'Objets Actifs Complexes ; l'arbre binaire montre qu'il est possible de programmer en BOX avec le principe de délégation.

VI-1 Calcul de factorielle

VI-1-1 Présentation du problème

Une méthode de calcul récursive et parallèle de la fonction factorielle s'effectue par dichotomie. 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 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 deux valeurs est alors retournée.

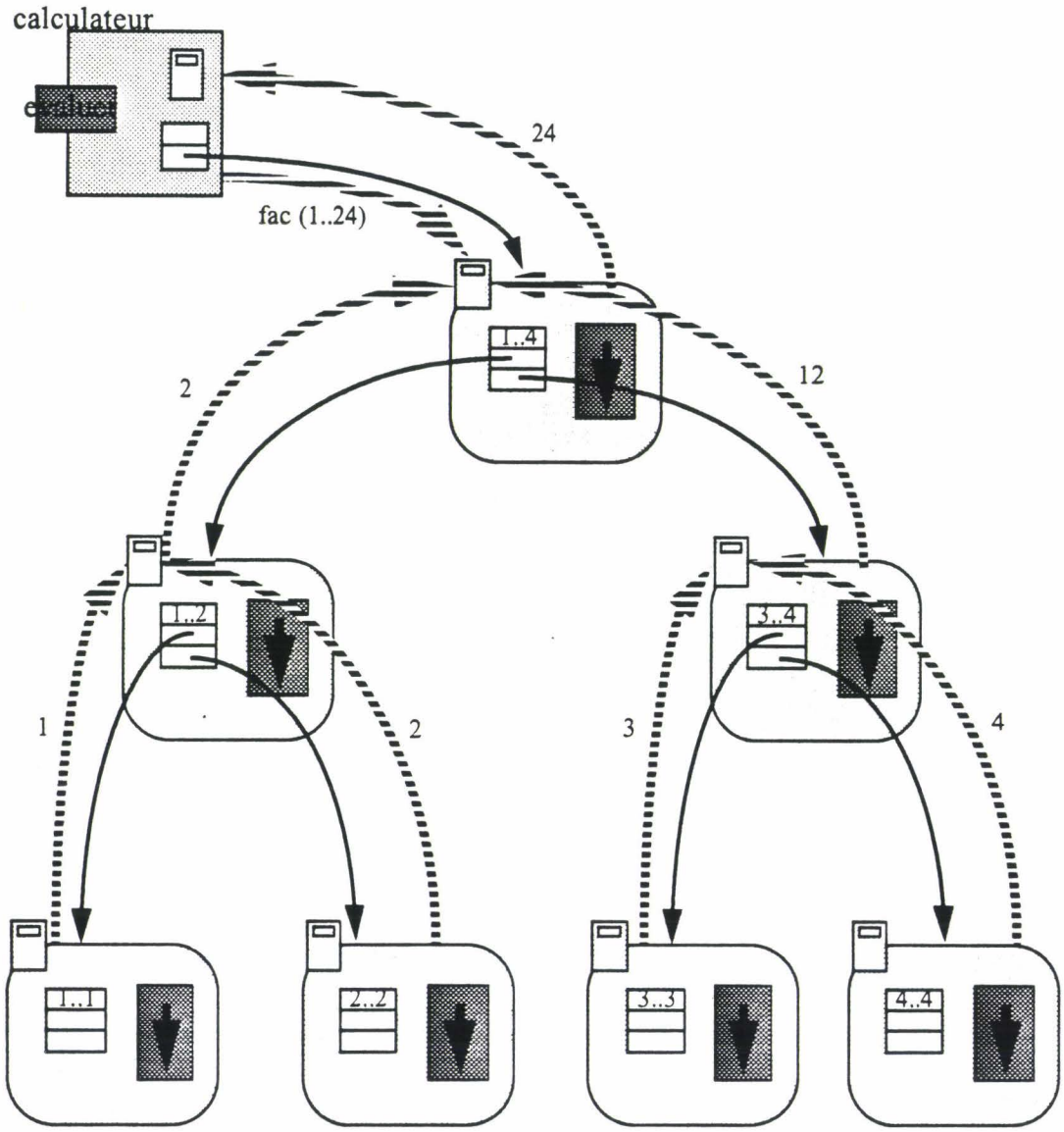


figure 5.14 *Calcul de fac (1..4)*

Cette méthode implique un fort parallélisme du calcul. Le composant actif Fac est réalisé grâce à un fragment.

VI-1-2 Code des classes

VI-1-2-1 La classe de fragment fac

```
[FRAG] fac ::

  [PROC] create :-- comportement du composant FAC.
       [BOX] box_reply ;-- boîte pour retourner le résultat.
       [INT] min, max    -- borne de l'intervalle [min..max].
```

```
    :
  {
    if min = max then          -- terminaison de la récursivité.
      box_reply <- ! min ! ;-- on retourne une des 2 valeurs.
    else
      [FAC <- create ! BOX, min, (min + max) div 2 !] facg ;
      [FAC <- create ! BOX, (min + max) div 2 + 1, max!] facd ;
      BOX -> ! [INT] retour1 ! ;-- 1ière réponse.
      BOX -> ! [INT] retour2 ! ;-- 2ième réponse.
      box_reply <- ! retour1 * retour2 ! ;-- retourne le résultat.
    end_if ;
  }
```

Un exemple d'utilisation de ce composant est :

```
[OBJ] calculateur ::
  [PROC] evaluer ::
  {
    [BOX <-] box_reply ;-- création d'une boîte aux lettres.
    [FAC <- create ! box_reply, 1, 20 !] une_fac ; -- calcul de
fac(20).
    box-reply -> ! [INT] resultat ! ;-- attente du résultat.
    -- afficher n!.
  }
```

VI-1-3 Description de l'application

```
(BOX_FILES)
    "fac.bx"
    "calculateur.bx"

(APPLICATION) calculateur

(START) evaluer

(MODULES)
  [MODULE] calcul, factorielle
  calcul <- ! calculateur !
  factorielle <- ! fac !
  factorielle # 15
```

Cet exemple d'application est composé de 16 modules (1 module calculateur, 15 modules de calcul de factorielle). Ainsi, à un instant donné, 15 composants FAC peuvent fonctionner en réel parallélisme si la machine est munie d'au moins 15 processeurs.

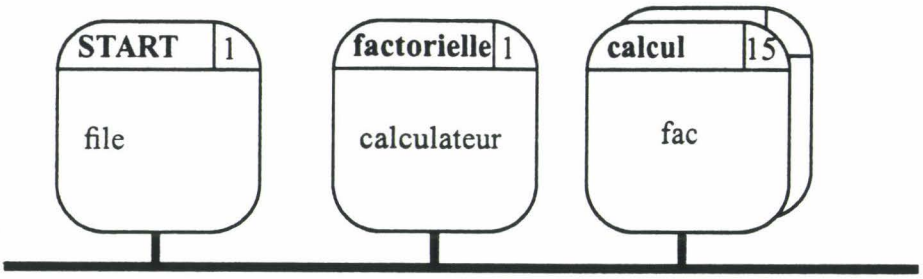


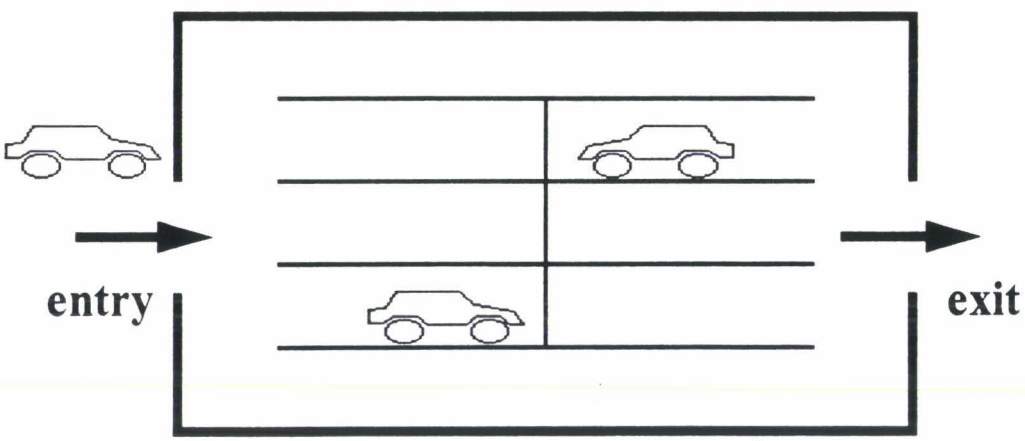
figure 5.15 *Répartition en modules de l'application calcul de factorielle*

VI-2 Simulation d'un parking

Nous allons représenter une simulation de parking à l'aide de processus communicants.

VI-2-1 Présentation du problème

Des voitures peuvent venir se garer dans le parking. Le parking accepte un nombre limité de voitures. Lorsque le parking est plein, une voiture qui désire se garer reste à l'entrée et attend qu'une place se libère. On désire simuler un parking pour lequel le point d'entrée est physiquement distant du point de sortie.



Le parking sera représenté par deux fragments qui contrôlent l'entrée et la sortie (Ctrl_in et Ctrl_out) regroupés dans un objet parking. Les voitures sont des fragments qui tentent de passer par le parking.

Le principe est ici que les deux fragments de contrôle utilisent des tickets pour contrôler le nombre de voitures dans le parking. Chaque voiture qui entre prend un ticket,

chaque voiture qui sort rend un ticket. Au départ il y a autant de tickets que de places, ils sont dans le processus d'entrée. Lorsqu'une voiture sort, le processus de sortie renvoie le ticket au processus d'entrée.

Les tickets sont ici représentés par des messages (vides) qui circulent entre les boîtes des processus voitures, entrée et sortie. La synchronisation est donc ici complètement basée sur la communication.

VI-2-2 Découpe du scénario

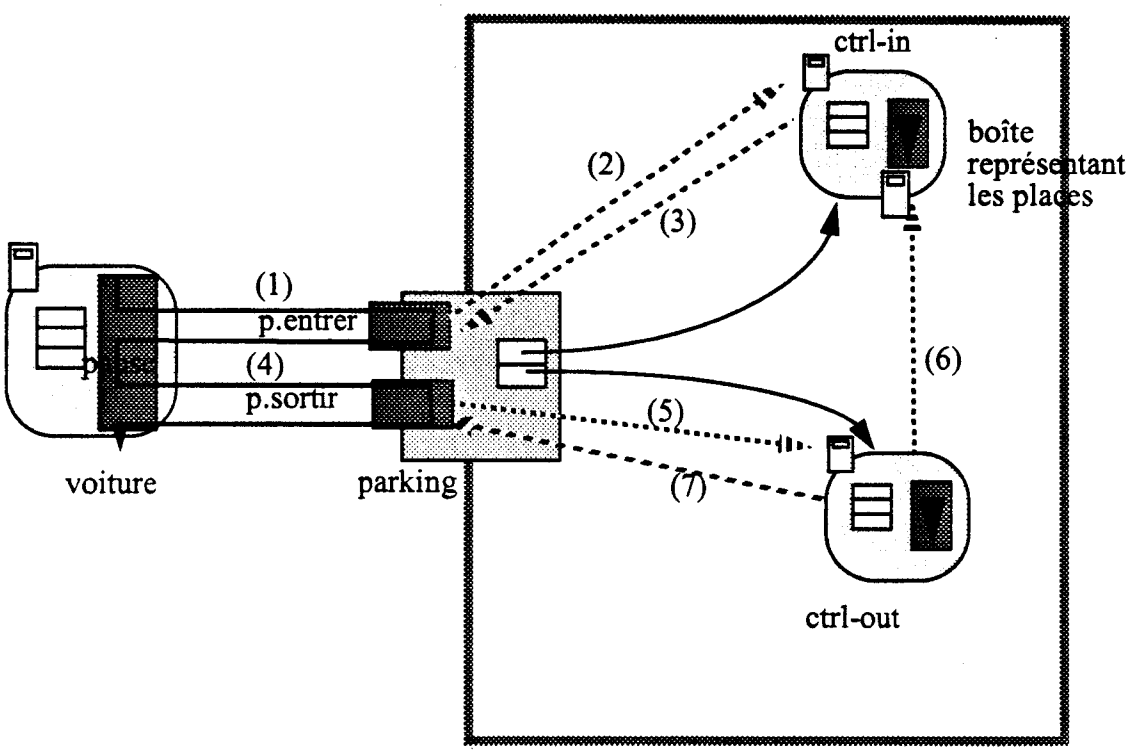


figure 5.16 *Communication entre les entités*

La voiture veut entrer dans le parking (1). Le parking va chercher un ticket à la borne d'entrée (2). S'il reste de la place, la borne d'entrée retourne un ticket au parking (3) et termine l'appel de méthode p.entrer. La voiture est dans le parking. La voiture marque une pause. Elle demande ensuite au parking de sortir (4). Le parking signale cette demande au contrôleur de sortie (5). Celui-ci retourne le ticket de la voiture à la borne d'entrée (6) et signale au parking que la voiture peut sortir (7). La voiture est alors sortie du parking.

Si le parking est plein lors de l'entrée de la voiture, il y a un temps d'attente entre l'opération (2) et (3). L'opération (3) est effectuée lorsqu'une voiture garée cherche à sortir et rend son ticket à la borne d'entrée (6).

Remarque : les différentes voitures peuvent appeler simultanément p.entrer ou p.sortir sans qu'il y ait besoin de synchronisation sur le parking. Toute la synchronisation va se faire au niveau des fragments *dans* l'Objet Actif Complexe parking.

VI-2-3 Code des classes

VI-2-3-1 La classe ctrl_in

```
[FRAG] CTRL_IN:: -- classe pour le fragment d'entrée

     [BOX] transfert; -- boîte pour le transfert de tickets entre
                      -- la sortie et l'entrée

     [PROC] init: [INT] N; [BOX] exit_point : -- N = nombres de places
     {
       [INT <- N]i;
       transfert := [BOX <-]; -- création de transfert
       loop                   -- on met N tickets à l'entrée
         if i = 0 then
           exit
         else
           transfert <- ;
           i := i-1
         end_if
       end_loop;
       exit_point <- !transfert! -- on passe la boîte à la sortie
     }

     [PROC] body: [INT] N; [BOX] exit_point : -- activité de contrôle
     {
       init(N,exit_point);
       loop
         BOX -> ![BOX] voiture!; -- attente d'une entrée
         transfert ->;           -- on prend un ticket
         voiture <-              -- on le donne à la voiture
       end_loop
     }
```

VI-2-3-2 La classe ctrl_out

```
[FRAG] CTRL_OUT::

     [BOX] transfert; -- boîte pour le transfert de tickets entre
                      -- la sortie et l'entrée

     [PROC] init::
     { BOX -> !transfert! }  -- on reçoit la boîte de l'entrée

     [PROC] body::
     {
        init;
```

```
        loop
          BOX -> ![BOX] voiture!;  -- attente d'une sortie
          transfert <-;             -- on envoie le ticket à l'entrée
          voiture <-               -- on libère la voiture
        end_loop
    }
```

VI-2-3-3 La classe parking

L'objet parking renferme les deux processus de contrôle et fournit une interface procédurale pour l'utilisation du parking.

```
[OBJ] parking::

    [CTRL_IN]ctrl_in;   -- processus à l'entrée
    [CTRL_OUT]ctrl_out; -- processus à la sortie
    [SLEEP] clock;  -- horloge pour les attentes

    [PROC]entry:: -- pour entrer
    {
       ctrl_in <- ![BOX <-]voiture!; -- on demande un ticket
       voiture ->                    -- puis on l'attend
    } .

    [PROC]go_out:: -- pour sortir
    {
       ctrl_out <- ![BOX <-]voiture!; -- on rend un ticket
       voiture ->                     -- on attend
    }

    [PROC] park_during: [INT] time :
    {
      entry;
      clock.sleep(time);
      go_out
    }

    [PROC]init: [INT] nb_places :
    {
      ctrl_out := [CTRL_OUT<- body!!];
      ctrl_in  := [CTRL_IN <- body!nb_places,ctrl_out!];
      clock := [SLEEP<-]
    }
```

VI-2-3-4 La classe voiture

Les voitures sont des fragments qui stationnent pendant un certain temps dans le parking.

```
[FRAG]car::
```

```
[PROC]create:[INT]num,wait_out,wait_in;[PARKING]the_parking;[BOX]end :
    {
          [FILE<-out!!]out; -- out est utilisé ici pour imprimer
                            -- les traces
          out.m([MESS<-!"Car ",num," :created.\n"!]);
      sleep(wait_out);
          out.m([MESS<-!"Car ",num," :arrived.\n"!]);
      the_parking.park_during(wait_in);
          out.m([MESS<-!"Car ",num," :exited.\n"!]);
      -- end est une boite pour la terminaison de l'application
      end <-;
    }
```

VI-2-3-5 La classe principale pour lancer la simulation

```
[OBJ]simulation::

    [PROC]go::
    {
      [parking <- init!6!] the_parking; -- parking à 6 places

      -- boucle pour lancer toutes les voitures
      [INT <- 20]i ;
      [BOX <-]conclusion;
      loop
        if i = 0 then exit
        else
          [CAR <- create! i, i mod 3, i mod 7, the_parking
,conclusion!];
          i := i-1
        end_if
      end_loop;

      -- boucle pour attendre la sortie de toutes les voitures
      [INT <- 20] j;
      loop
        if j = 0 then exit
        else
          conclusion ->;
          j := j-1
        end_if
      end_loop
    }
```

VI-2-4 Description de l'application

```
    (INCLUDE)
       "$BOX/library/io.cluc"
       "$BOX/library/sleep.cluc"

    (BOX_FILES)
       "voiture.bx"
```

```
   "simul.bx"
   "parking.bx"

(APPLICATION)
   simulation

(START)
   go

(MODULES)
  [MODULE] simul, le_parking, borne_sortie, les_voitures
  simul <- ! simulation !
  le_parking <- ! parking, ctrl_in !
  borne_sortie <- ! ctrl_out !
  les_voitures <- ! car !
  les_voitures # 2
```

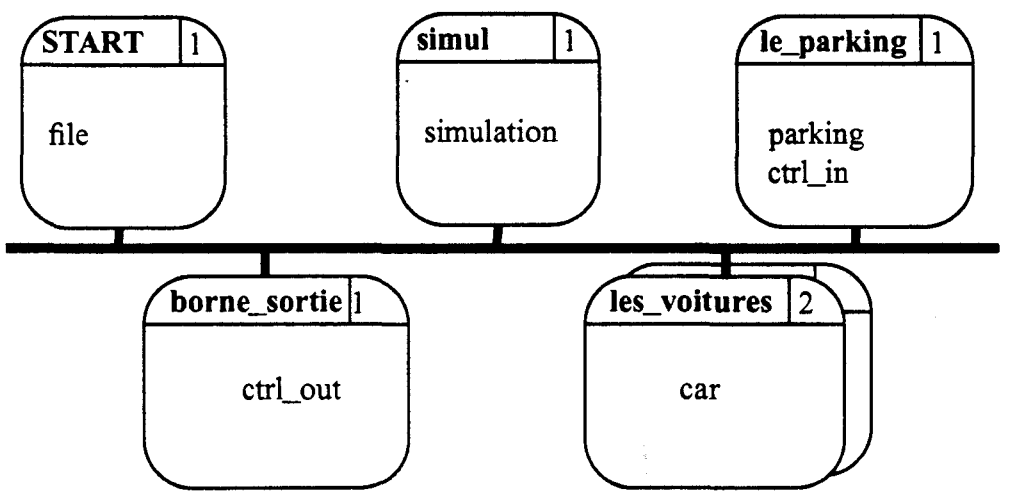


figure 5.17 *Répartition en modules de la simulation de parking*

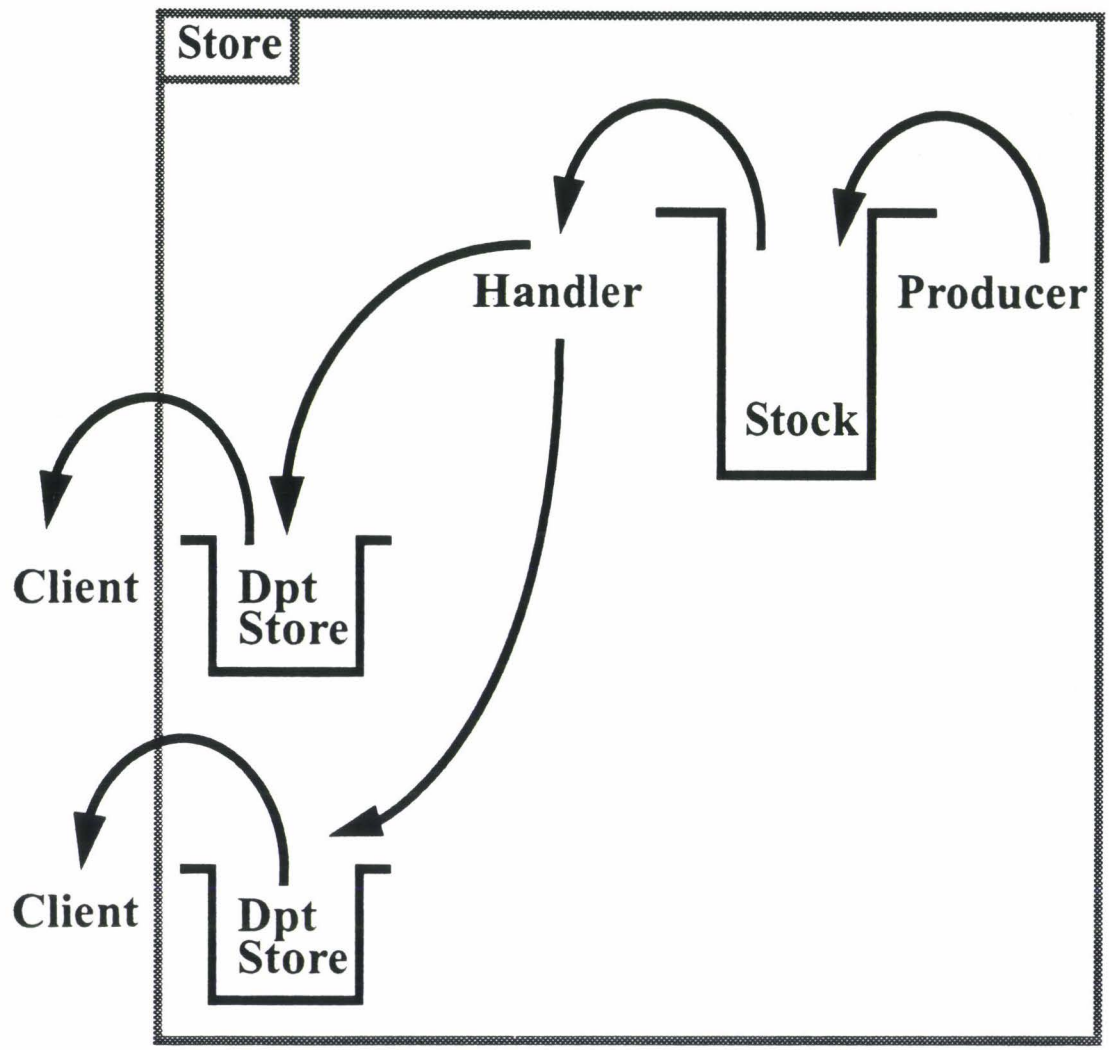
VI-2-5 Conclusion

BOX permet de concevoir des applications en utilisant comme moyen de conception la métaphore des processus communicants. Les fragments vont se synchroniser sur la réception de messages qui leur sont adressés.

VI-3 Simulation d'un magasin

VI-3-1 Présentation du problème

Les clients retirent des articles des rayons. Lorsque le niveau d'un rayon diminue, un manutentionnaire transfère des articles du stock vers le rayon. Lorsque le niveau du stock devient trop bas, le producteur crée de nouveaux articles et les place dans le stock.



Une première analyse permet de déterminer que le client, le manutentionnaire et le producteur sont des entités actives. Le magasin, le rayon et le stock sont des entités passives. Le rayon et le stock ont des structures similaires. Les articles vont être stockés dans le rayon et le stock. Il faut donc une structure qui permette de maintenir un ensemble cohérent d'articles. Un buffer convient très bien ici.

VI-3-1-1 Le buffer

Le buffer peut être vu comme une entité active. En BOX, c'est une classe de fragment. Un buffer répond à deux types de messages : dépôt ou retrait d'information.

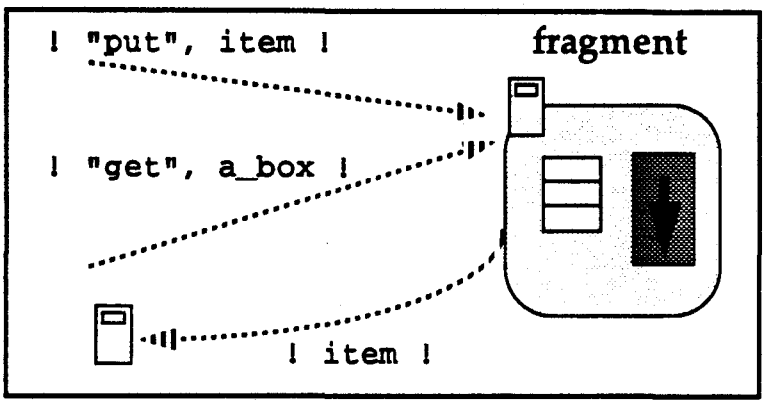


figure 5.18 *Fragment buffer*

VI-3-1-2 La classe de fragment buff

```
[FRAG] BUFF [T] ::

  [PROC] behaviour ::
       {
       loop
         BOX -> ! [STR], [BOX] b ! { $1 = "get" } ;
         BOX -> ! [STR], [T] v ! { $1 = "put" } ;
         b <- ! v ! ;
       end_loop ;
       }
```

Chaque instance de fragment possède une activité qui déroule un code pour le fragment. Cette implémentation du buffer est correcte mais ne répond pas aux attentes du génie logiciel en particulier d'un point de vue lisibilité et réutilisation. En effet, pour pouvoir dialoguer avec le buffer, il faut connaître le format des messages qui sont acceptés par le buffer. Cette connaissance ne peut se faire qu'en examinant attentivement le code de la classe.

Pour supprimer ces déficiences, nous allons encapsuler cette classe de fragment dans une classe d'objet passif et ainsi créer un Objet Actif Complexe (OAC) tampon.

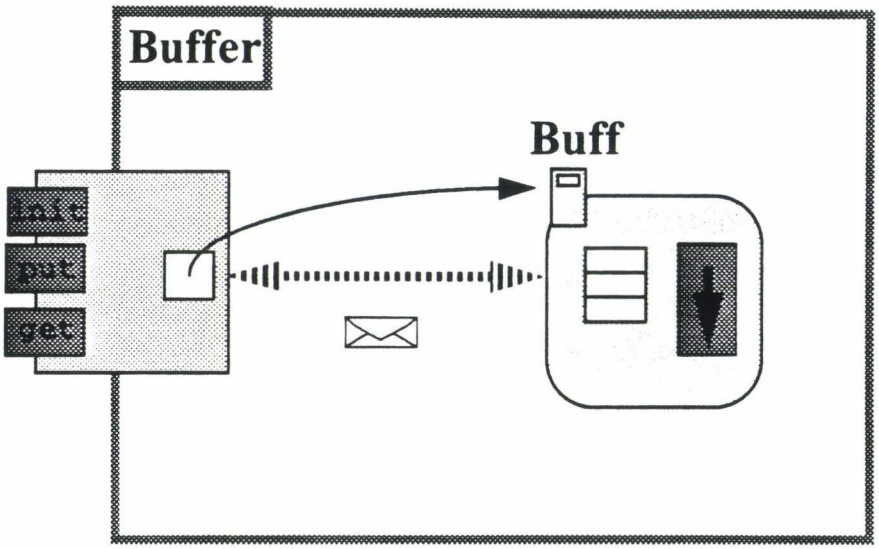


figure 5.19 *OAC buffer*

Nous obtenons une interface procédurale claire. D'un point de vue réutilisation, c'est la seule chose à connaître pour pouvoir se servir de l'OAC buffer. Plusieurs méthodes peuvent se dérouler simultanément sur le buffer, il est à noter que la synchronisation se fait, dans l'OAC, au niveau du fragment et non pas au point d'entrée de l'OAC.

VI-3-1-3 La classe d'objet buffer

```
[OBJ] BUFFER : [T] :
  -- Generic buffer

  [PROC] init ::
       -- Initialization
    { bt := [BUFF [T] <- behaviour !!] ; }

  [PROC] put : [T] item :
       -- Put 'item' in the buffer
    { bt <- ! "put", item ! ; }

  [PROC [T]] get ::
       -- Get item
    { [BOX <-] res ; bt <- ! "get", res ! ;
      res -> ! result ! ; }

  [BUFF [T]] bt ;
```

VI-3-1-4 Le stock

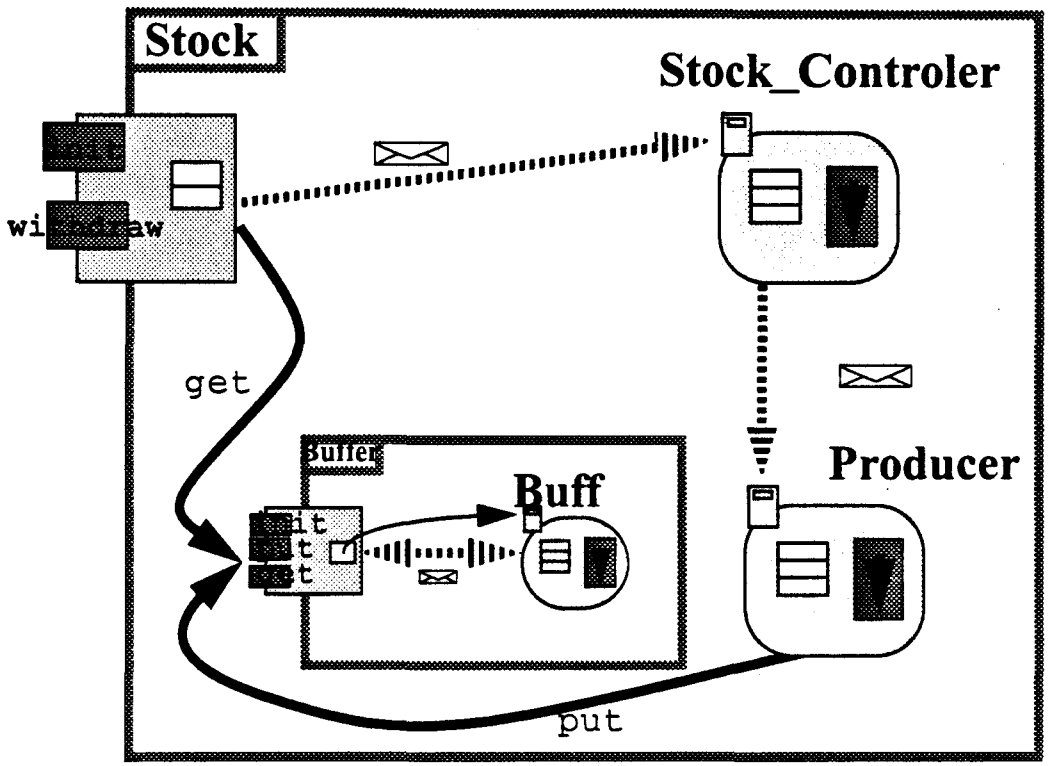


figure 5.20 *OAC stock*

L'OAC stock est composé d'un buffer qui contient les articles. Un contrôleur de stock (Stock_Controler) vérifie en permanence (sans attente active) l'état du stock. Si le niveau du stock descend en-dessous d'un seuil, le contrôleur demande au producteur (Producer) de créer de nouveaux articles et de les insérer dans le stock.

VI-3-1-5 La classe stock

```
[OBJ] STOCK ::

    -- Stock d'articles

    -- Le stock ou il y a des articles dans une grande quantite

    [INT] max = 30 ;-- Nbre max d'articles en stock
    [INT] limite = 10 ; -- Qtte minimale

    [INT] qtte_courante ;-- Quantite courante en stock


    [STOCKEUR] stockeur ; -- Personne responsable de la gestion du stock

    [PROC [ARTICLE]] extraire ::
```

```
        -- Extraire un article
    {
        stockeur <- un_retrait !! ;
        result := buf_a.get ;

    }


[PROC] ajouter : [ARTICLE] art :

        -- Ajouter un article
    {
        buf_a.put (art) ;
    }


[PROC] init ::

        -- Initialisation
    {
        [LOCATION] lo;
        lo.current ;

        buf_a := [BUFFER [ARTICLE] <- init ! ! at lo] ;
        stockeur := [STOCKEUR <- work ! self ! ] ;
    }


-- Implementation

[BUFFER [ARTICLE]] buf_a ;
```

VI-3-1-6 Le rayon

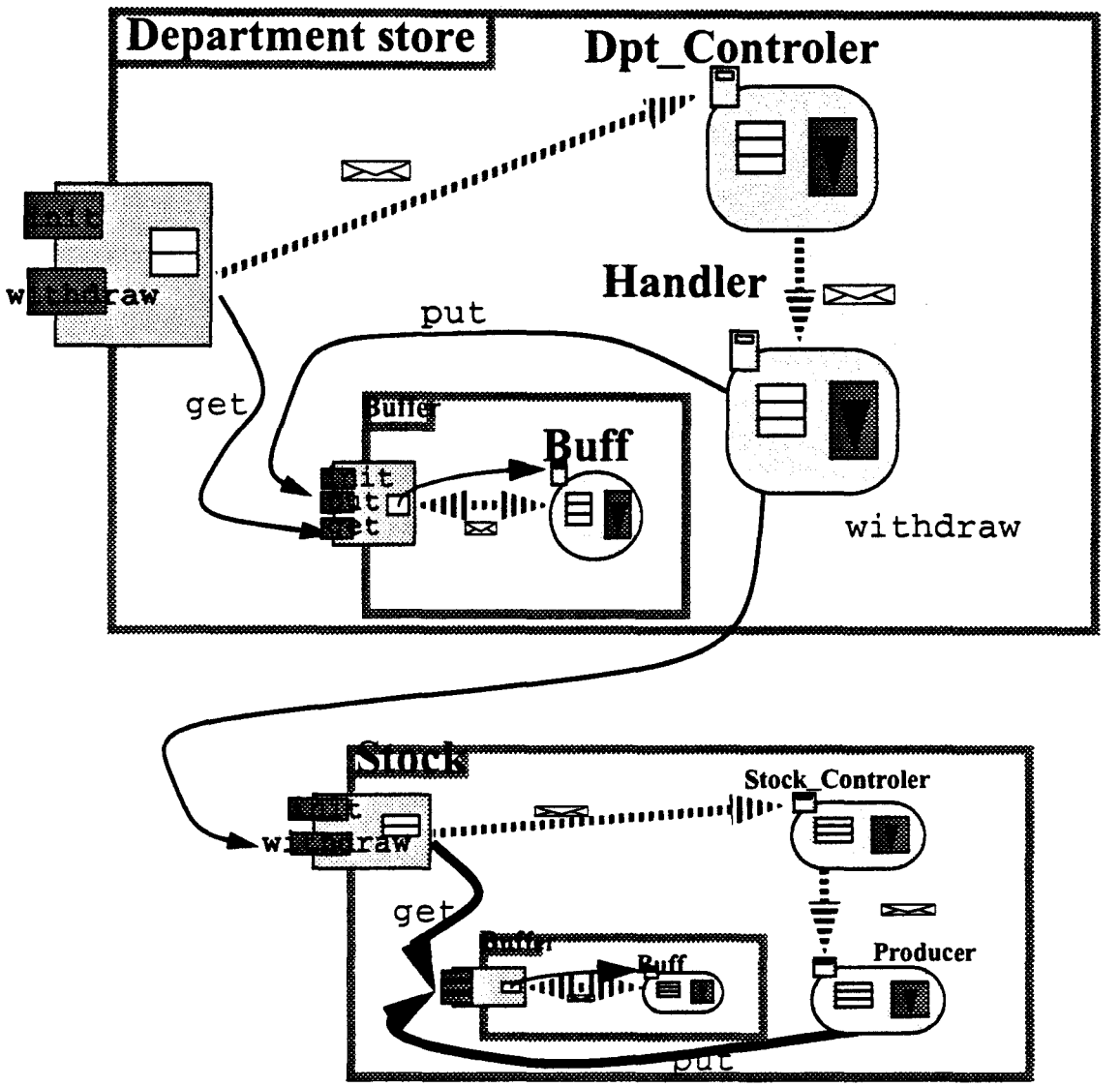


figure 5.21 *OAC rayon*

L'OAC rayon a une structure similaire à celle du stock. Le rayon est composé d'un buffer qui contient les articles du rayon. Une activité (Dpt_controler) vérifie en permanence l'état du rayon. Si celui-ci descend en-dessous d'un certain seuil, le manutentionnaire (handler) déplace des articles du stock vers le rayon.

VI-3-1-7 La classe rayon

```
[OBJ] RAYON ::

     -- Le rayon ou il y a des articles dans une certaine quantite

     [INT] max = 10 ;-- Nbre max d'articles en rayon
     [INT] limite = 2 ; -- Qtte minimale

     [INT] qtte_courante ;-- Quantite courante en rayon

     [STOCK] stock ;-- Adresse du stock d'articles ou il faut recommander

     [RAYONNEUR] rayonneur ; -- Personne responsable de la gestion du
rayon

     [FILE] out ;


     [PROC] retirer : [INT] qtte :

               -- Retirer l'article en quantite 'qtte'
          {
               -- Essayer de retirer
               -- Recommander au STOCK s'il n'y a en pas assez

               out.s ("demande de retrait dans le rayon nb articles : ");
               out.i (qtte) ;
               out.nl ;
               status ;

               [INT <- qtte] i ;
               loop
                    if i = 0 then exit end_if ;

                    rayonneur <- un_retrait !! ;
                    [ARTICLE] aa := buf_a.get ;

                    i := i - 1;
               end_loop ;
               status ;
          }


     [PROC] deposer : [ARTICLE] art :

               -- Ajouter un article dans le rayon
          {
               buf_a.put (art) ;
          }

     [PROC] init : [STOCK] stk :

               -- Initialisation
          {
               [LOCATION] lo;
```

```
                lo.current ;
                out := [FILE <- out !!] ;
                buf_a := [BUFFER [ARTICLE] <- init ! ! at lo] ;
                rayonneur := [RAYONNEUR <- work ! self ! ] ;
                stock := stk ;
            }


        -- Implementation

        [BUFFER [ARTICLE]] buf_a ;

        [PROC] status ::

                -- Status du rayon
            {
                out.m ([MESS <- ! "\t\t\t\t\t Rayon ", buf_a.size, "\n" !
] ) ;
                out.m ([MESS <- ! "\t\t\t\t\t\t\t Stock ", stock.buf_a.si-
ze, "\n" ! ] )
            }
```

VI-3-1-8 Le magasin

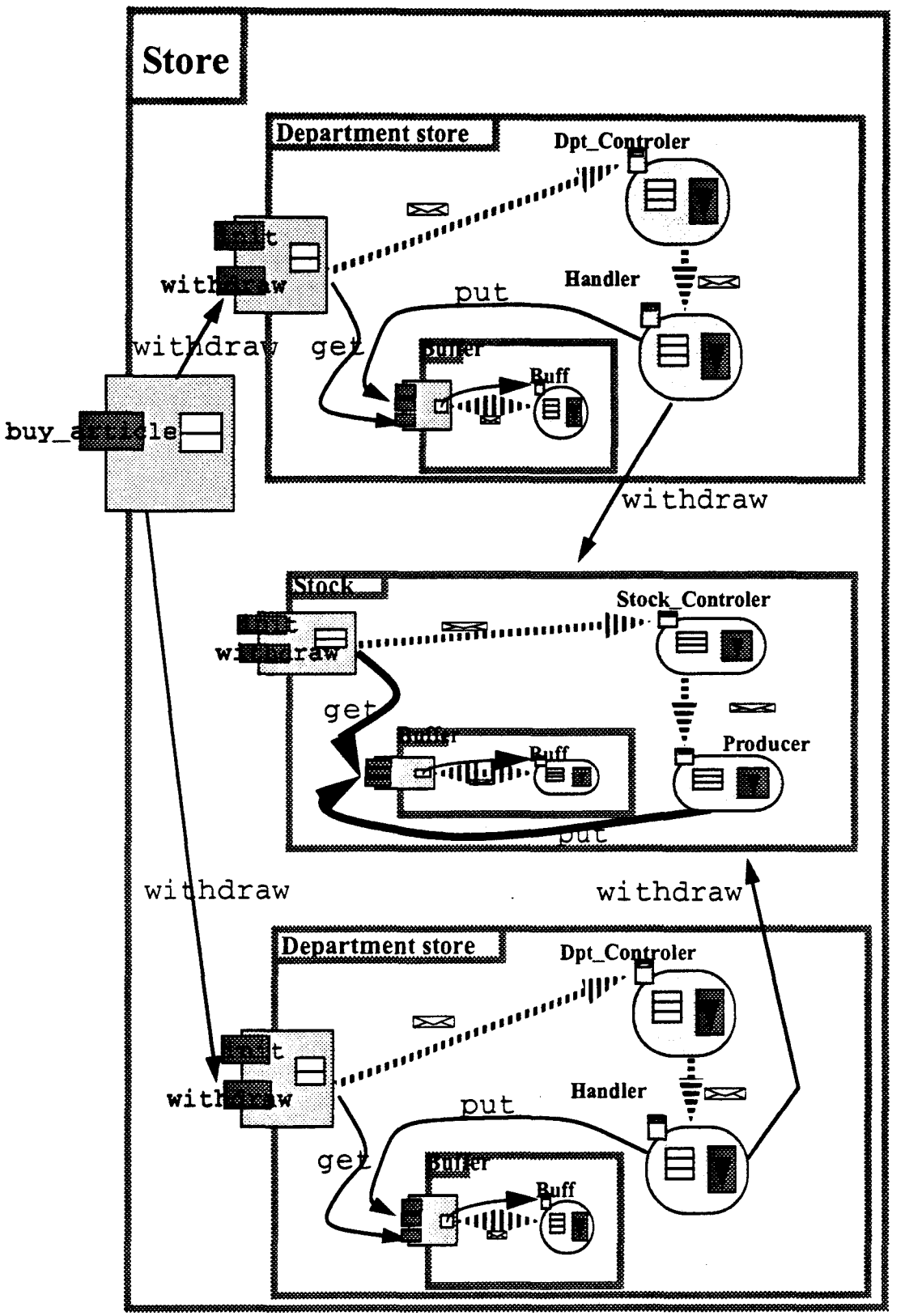


figure 5.22 *OAC magasin*

L'OAC magasin englobe les différentes classes présentées ci-dessus. Un magasin peut contenir plusieurs rayons qui peuvent se réapprovisionner à un même stock. Le magasin offre comme interface une procédure qui permet de retirer des articles d'un rayon. Le client du magasin ne connaît pas les traitements déclenchés suite à un retrait.

VI-3-1-9 La classe magasin

```
[OBJ] Magasin ::

     -- Un magasin ou on peut retirer des articles

     [RAYON] rayon ;
          -- Rayon ou le serveur se sert


     [STOCK] stock ;
          -- Stock ou on peut se reapprovisionner


     [SPROC ] retirer_article : [INT] qtte :

               -- Retirer l'article du type 'type' en quantite 'qtte'
               -- Resultat stocke dans un tableau d'articles
               -- Il n'y a qu'un serveur

          {
               [FILE <- out !!] out.s ("demande de retrait dans le maga-
sin\n");
               rayon.retirer (qtte) ;
               out.s ("Fin de retrait dans le magasin\n");
          }

     [PROC] init ::

               -- Creation de tout
          {
               stock := [STOCK <- init !!] ;
               rayon := [RAYON <- init ! stock !] ;
          }
```

VI-3-2 Code des autres classes

VI-3-2-1 La classe article

```
[OBJ] ARTICLE ::

     -- Un article
```

VI-3-2-2 La classe producteur

```
[FRAG] PRODUCTEUR ::

     -- Creation d'un nouvel article et depot dans le stock

     [PROC] produire : [BUFFER [ARTICLE]] buf_a :

               -- Produire un article et le deposer dans le stock

          {

               -- Le producteur crée un article

               [ARTICLE <-] art ;-- Creation d'un article

               buf_a.put (art) ;
               [FILE <- out !!] out.s("production\n") ;

          }
```

VI-3-2-3 La classe manutentionnaire

```
[FRAG] MANUTENTIONNAIRE ::

     -- Manutentionnaire qui deplace un ARTICLE du STOCK vers le RAYON

     [STOCK] stock ;

     [RAYON] rayon ;

     [PROC] work : [STOCK] stk ; [RAYON] ray :

               -- Comportement d'un travailleur

          {
               -- Initialisation
               stock := stk ;
               rayon := ray ;

               -- Le manutentionnaire déplace un article

               [ARTICLE] art ;

               art := stock.extraire ;

               -- J'ai un article

               rayon.deposer (art) ;
               [FILE <- out !!] out.s("\t\t\tdeplacement\n") ;
          }
```

VI-3-2-4 La classe rayonneur

```
-- Rayonneur ou chef de rayon
-- Responsable du reapprovisionnement des rayons, stocks

[FRAG] RAYONNEUR ::

     -- Chef de rayon

     [INT] etat ;

     [RAYON] rayon ;

     [PROC] un_retrait ::

               -- Un retrait a eu lieu
          {
               etat := etat - 1 ;

               if etat <= rayon.limite then
                    etat := etat + 1 ;
                    loop
                         if etat = rayon.max then exit end_if ;
                         [MANUTENTIONNAIRE <- work ! rayon.stock, rayon !]
man;

                         etat := etat + 1 ;
                    end_loop ;

               end_if ;

          }


     [PROC] work : [RAYON] ray :

               -- Travail
          {
               rayon := ray ;

               loop
                    BOX -> [MESS] m ! [PROC] f ! ;
               end_loop ;

          }
```

VI-3-2-5 La classe stockeur

```
[FRAG] STOCKEUR ::

     -- Chef de stock

     [INT] etat ;
```

```
[STOCK] stock ;

[PROC] un_retrait ::

          -- Un retrait a eu lieu
     {
          etat := etat - 1 ;

          if etat <= stock.limite then
          etat := etat + 1 ;

               loop
                    if etat = stock.max then exit end_if ;
                    [PRODUCTEUR <- produire ! stock.buf_a !] prod;

                    etat := etat + 1 ;
               end_loop ;

          end_if ;

     }


[PROC] work : [STOCK] sto :

          -- Travail
     {
          stock := sto ;

          loop
               BOX -> [MESS] m ! [PROC] f ! ;
          end_loop ;

     }
```

VI-3-2-6 La classe de simulation du magasin

```
[OBJ] SIMULA ::

     [MAGASIN] mag ;


     [PROC] make ::

               --
          {

               [FILE <- out !!] out ;
               [SLEEP] sleep;

               out.s ("Simulation de magasin \n") ;

               mag := [MAGASIN <- init !! ];

               mag.retirer_article (1) ;
```

```
            sleep.sleep (10) ;

            mag.retirer_article (2) ;
            mag.retirer_article (3) ;

            sleep.sleep (3) ;
            mag.retirer_article (10) ;

            sleep.sleep (5) ;

            mag.retirer_article (5) ;

            mag.retirer_article (1) ;

            sleep.sleep (5) ;
            out.s ("On ferme \n") ;

        }
```

VI-3-3 Description de l'application

```
(INCLUDE)
        "$BOX/library/sleep.cluc"
        "$BOX/library/io.cluc"
        "$BOX/library/location.cluc"
        "$BOX/library/buffer.cluc"


(BOX_FILES)
     "mag.bx"
     "simul.bx"
     "rayonneur.bx"

(APPLICATION)
        simula

(START)
        make

(MODULES)

     [MODULE] magasin, rayon, stock, personnes, main

     buffer # 0 -- la description du module buffer est récupérée par la
                -- clause d'inclusion. Le module buffer est intégré dans
                -- d'autres modules et n'existe pas en tant que tel à
                -- l'exécution

     magasin <- ! magasin !

     rayon <- ! rayon, rayonneur !
     rayon <- buffer

     stock <- ! stock, stockeur !
```

```
stock <- buffer

personnes <- ! producteur, manutentionnaire !

main <- ! simula !
```

Dans la description de cette application, nous réutilisons la description du module `buffer`. Celle-ci est incluse dans les modules `rayon` et `stock`. Le module `buffer` en tant qu'exécutable n'existe pas à l'exécution (facteur de duplication mis à zéro).

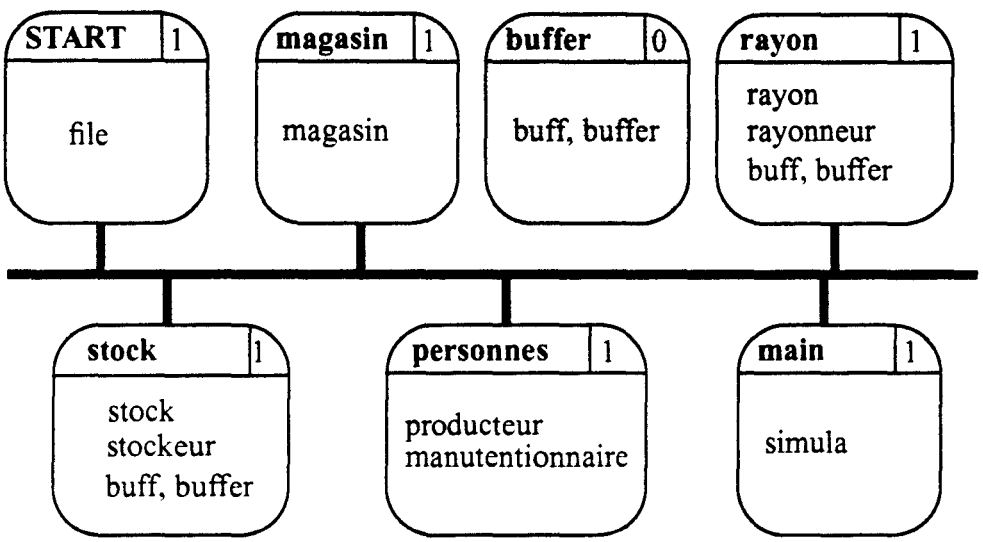


figure 5.23 *Répartition en modules de l'application magasin*

VI-3-4 Conclusion

Cet exemple montre l'utilisation de la méthodologie des Objets Actifs Complexes pour la réalisation d'une application. Le programmeur conçoit chaque composant logiciel en y plaçant les objets et activités nécessaires à sa réalisation et fournit une interface procédurale pour l'utiliser. Les composants clients connaissent uniquement cette interface. C'est le principe d'encapsulation appliqué aux objets et aux activités.

VI-4 L'arbre binaire de recherche

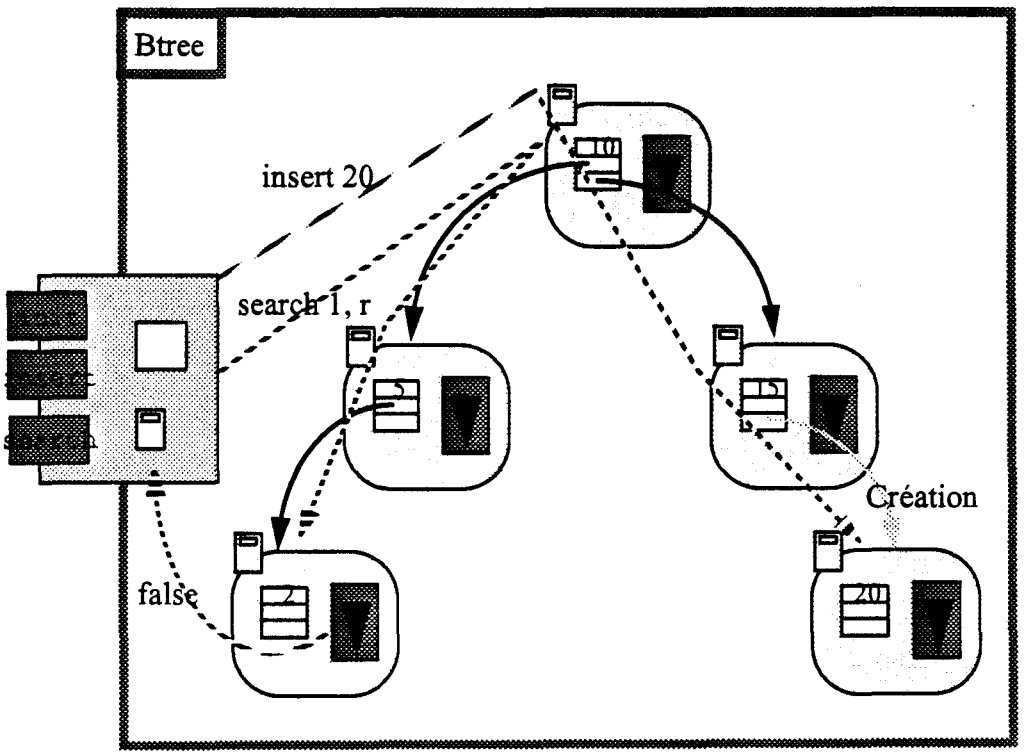
VI-4-1 Présentation

Cette application permet de stocker des informations dans un arbre et de les retrouver (insertion et recherche). Plusieurs implémentations sont possibles [Denneulin94]. Nous présentons la version où les noeuds de l'arbre sont actifs. Dans cette version, chaque noeud est actif ; c'est-à-dire réalisé à l'aide d'un fragment. Plusieurs insertions et recherches peuvent se faire en parallèle. Il n'y a pas de verrouillage de l'arbre.

La synchronisation se fait au niveau de chaque noeud qui traite les demandes de la part

des clients. Si un noeud ne peut pas répondre à la requête qui lui est transmise, il délégue le travail à un de ses fils (pour chacune des opérations, vers le fils gauche si la valeur à insérer ou à chercher est inférieure à la valeur du noeud courant ; sinon à son fils droit).

Cet exemple montre qu'il est possible de concevoir des applications en utilisant le principe de délégation des langages d'acteurs. D'autres versions de l'arbre ont été développées : avec rééquilibrage dynamique ainsi qu'avec localisation des noeuds.



(1) Insertion de la valeur 20
(2) Recherche de la valeur 1

figure 5.24 *Exemple d'arbre binaire*

VI-4-2 Code des classes

L'arbre binaire est composé de deux classes : la classe noeud et la classe arbre.

VI-4-2-1 La classe noeud

La classe noeud est une classe de fragment. Chaque instance va contenir une valeur de l'arbre.

```
[FRAG] NODE ::

    -- Node of a binary tree

    [NODE] left ;
```

```
[NODE] right ;
[INT] val ;

[PROC] init : [INT] v :
     {
          [PROC] f

          val := v ;
          loop
               BOX -> ! f ! {$1 = insert or $1 = search} ;
          end_loop ;
     }


[PROC] insert : [INT] v :
          -- Insert `v'
     {
          if val <> v then
               if v < val then
                    if left = [NODE] null then
                         left := [NODE <- init ! v !] ;
                    else
                         left <- insert ! v ! ;
                    end_if ;
               else
                    if right = [NODE] null then
                         right := [NODE <- init ! v !] ;
                    else
                         right <- insert ! v ! ;
                    end_if ;
               end_if ;
          end_if ;
     }

[PROC] search : [INT] v ; [BOX] r :
          -- Search `v'
     {
          if val <> v then
               if v < val then
                    if left = [NODE] null then
                         r <- ! false ! ;
                    else
                         left <- search ! v, r ! ;
                    end_if ;
               else
                    if right = [NODE] null then
                         r <- ! false ! ;
                    else
                         right <- search ! v, r ! ;
                    end_if ;
               end_if ;
          else
               r <- ! true ! ;
          end_if ;
     }
```

VI-4-2-2 La classe arbre

La classe arbre fournit une interface procédurale pour manipuler les instances d'arbre. C'est donc une classe d'objets. Il n'y a pas de synchronisation dans cette classe, tout va se faire au niveau de chaque noeud de manière indépendante.

```
[OBJ] BTREE ::

     -- Binary tree

     [NODE] root ;

     [PROC] insert : [INT] v :
               -- Insert `v' in BTREE
          {
               root <- insert !  v ! ;
          }

     [PROC [BOOL]] search : [INT] v :
               -- Is `v' exist in BTREE ?
          {
               [BOX <-] r ;
               root <- search ! v, r ! ;
               r -> ! result ! ;
          }

     [PROC] init : [INT] v :
          {
               root := [NODE <- init ! v !] ;
          }
```

VI-4-2-3 Classe principale de l'application

```
[OBJ] SIMULA ::

     [FILE] out ;
     [BTREE] tree ;

     [PROC] make ::

          {
               tree := [BTREE <- init ! 10 !]  ;
               tree.insert (5) ;
               tree.insert (15);
               tree.insert (2) ;

               out := [FILE <- out !!] ;

               tree.insert (20) ;
               chercher (1) ;
          }

     [PROC] chercher : [INT] i :
          {
```

```
        out.s ("recherche : ");
        out.i (i) ;
        out.s (" ") ;
        out.b (tree.search (i)) ;
        out.nl ;
    }
```

VI-4-2-4 Description de l'application

```
(INCLUDE)
     "$BOX/library/io.cluc"

(BOX_FILES)
     "btree.bx"
     "simula.bx"
     "node.bx"

(APPLICATION)
     simula

(START)
     make

(MODULES)
     [MODULE] node, tree, main
     node <- ! node !
     tree <- ! tree !
     main <- ! simula !
     node # 4
```

Dans cette application, les noeuds de l'arbre sont créés sur quatre modules. Le choix du site où créer un nouveau noeud est laissé au système. Nous avons développé d'autres versions où les noeuds sont groupés en sous-arbres ou encore placés par niveaux.

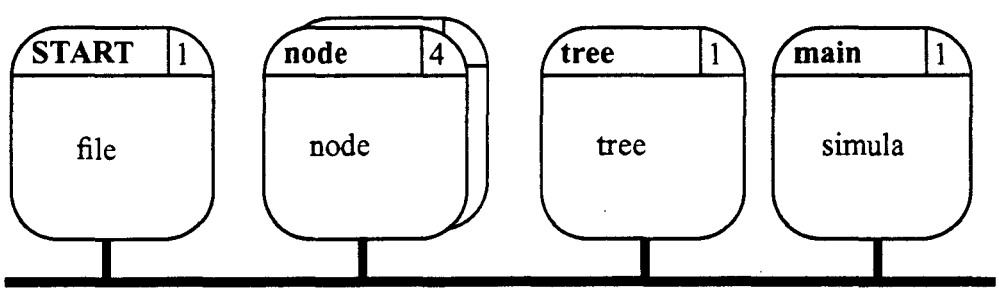


figure 5.25 *Répartition en modules de l'application arbre binaire*

VI-4-3 Conclusion

Cet exemple montre que BOX permet de concevoir des applications à l'aide de la notion d'objet actif. Chaque noeud de l'arbre est un objet actif qui traite des requêtes de

la part de ses clients. Cet exemple montre également que le principe de délégation des langages d'acteurs peut être manipulé en BOX.

VI-5 Conclusion

Ces différents exemples montrent que BOX permet de concevoir des applications réparties en modélisant le problème en terme de classes d'objets et de classes de fragments. Le programmeur peut créer son application en se basant sur le modèle des objets actifs ou des acteurs. Nous avons également présenté une méthodologie pour faciliter la création de composants logiciels parallèles et distribués : les Objets Actifs Complexes (OAC). Le but des OAC est de permettre au programmeur de réutiliser des composants parallèles qui offrent une interface procédurale comme si c'était un composant séquentiel. Les aspects distribution et parallélisme sont cachés dans le composant.

Chapitre - VII -

BOX: Le support d'exécution

Ce chapitre présente l'implantation du langage sur l'environnement PVC à l'aide des CAC/s ainsi que le run-time BOX.

VII-1 Le projet PVC

Le projet a débuté en 1990 par la notion de Processeur Virtuel de Classe (PVC) proposée par Eric Delattre et Jean-Marc Geib. Un Processeur Virtuel de Classe est un serveur encapsulant une classe et ses instances. La proposition était basée sur le principe de localité du code et des données et d'une fragmentation des objets par l'héritage. Cette architecture logicielle devait servir de support pour les langages à objets parallèles.

Le projet a ensuite évolué avec la définition des Composants Actifs de Communication par Luc Courtrai. Le but était de développer un environnement pour faciliter la conception et la construction d'applications parallèles et surtout pour servir de support d'exécution de langages orientés objets parallèles.

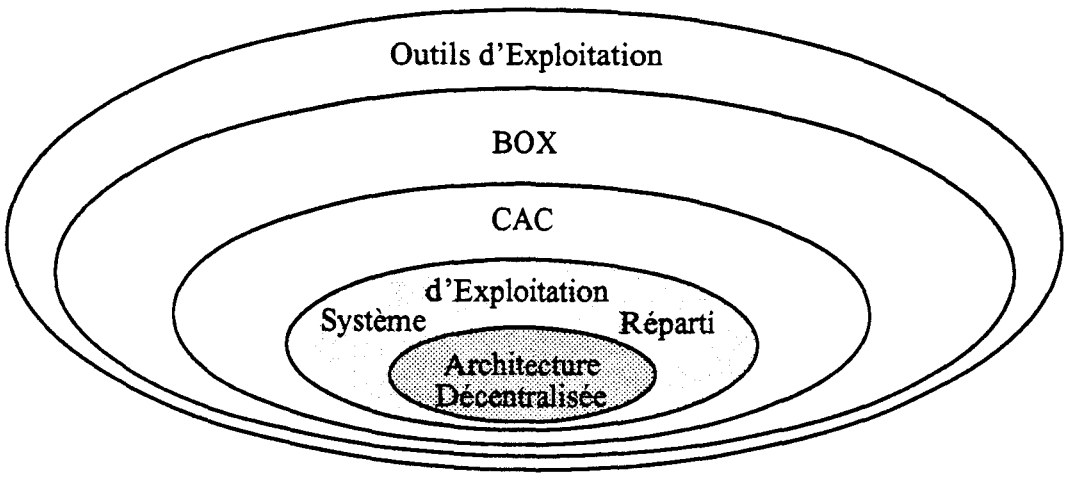


figure 7.1 *Présentation de l'environnement*

Les principaux objectifs des CAC étaient :

- l'indépendance vis-à-vis des systèmes d'exploitation distribués
- la facilité d'adaptation à différentes architectures (multicomputers, réseaux de stations de travail, ...)
- la transparence de l'architecture
- une distribution statique du code et dynamique des activités

Nous présentons tout d'abord les Composants Actifs de Communication (CAC). Ce sont des entités actives et autonomes permettant la construction d'applications parallèles. Elles apportent une vision structurante des données et des activités. C'est le principe objet appliqué aux systèmes. Le CAC est une entité de programmation regroupant une activité (processus léger), un environnement local et une boîte aux lettres. Cette structure est facilement réalisable au-dessus des fonctionnalités traditionnelles des systèmes d'exploitation.

Nous présentons ensuite la notion de module permettant la distribution du code et des activités. Le module est le support pour la répartition des données sur les différents noeuds, mais aussi le support pour la répartition des activités.

Nous décrivons finalement les différentes implémentations du run-time PVC.

VII-1-1 Les Composants Actifs de Communication

Les Composants Actifs de Communication définissent un grain unique de décomposition des applications parallèles en unités autonomes et concurrentes. Cette structure de base est d'un niveau plus évolué que le processus : elle intègre une activité, une boîte aux lettres et un environnement privé. La programmation et la manipulation de ces composants sont réalisées par l'utilisation d'une bibliothèque spécifique. Une application est définie par la spécification du comportement des différents CAC nécessaires et de la coopération des CAC, celle-ci se fondant sur la communication asynchrone par envoi de messages.

L'environnement des Composants Actifs de Communication permet :

- La gestion d'un parallélisme massif : tout CAC possède une activité autonome et toute application est décrite uniquement à l'aide de CAC.
- La structuration des données et des activités : tout CAC possède une activité et un environnement local.
- Une description des applications parallèles indépendamment des spécificités du système d'exploitation et de l'architecture cible.

VII-1-1-1 Structure d'un CAC

La structure d'un Composant Actif de Communication est proche de celle d'un Acteur. 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 7.2).

Le processus : chaque composant possède une activité autonome dont le code représente le comportement du composant. Cette activité est programmée par l'utilisateur en décrivant une fonction comportementale. Elle gère les données locales du composant et les communications avec les autres composants.

La boîte aux lettres : à chaque composant est associée 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 dès la création du composant et son nom identifie le composant dans le système. Les messages contenus dans la boîte d'un composant sont stockés dans l'ordre d'arrivée, mais les primitives de communication permettent au composant de filtrer et d'extraire les messages dans un ordre quelconque.

L'environnement local : l'environnement est créé dynamiquement par le processus du composant. Il regroupe toutes les données locales du composant. L'allocation mémoire s'effectue dans l'espace du processeur où évolue le CAC.

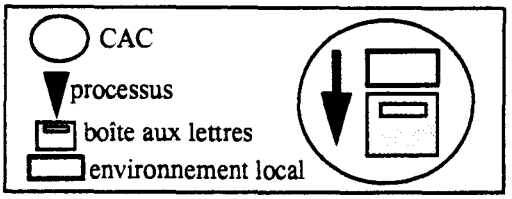


figure 7.2 *Structure d'un CAC*

VII - 1•1•1•1 La désignation

L'environnement CAC requiert la désignation de deux entités distinctes : les comportements pour la création des CAC et les CAC eux-mêmes pour autoriser la communication entre eux.

Les comportements : la création d'un CAC nécessite la désignation d'un comportement. La création sur un noeud quelconque impose une désignation globale des comportements sur l'ensemble du système. La création doit respecter une contrainte, dite de localité : le code du comportement doit résider sur le noeud de création.

Les composants : La primitive de création d'un composant renvoie un nom global au système pour permettre la communication entre CAC. Ce nom est celui de la boîte aux lettres du composant créé. La primitive d'envoi de messages laisse le message au système qui est chargé de le déposer dans la boîte aux lettres du CAC destinataire : la connaissance de la localisation du CAC destinataire n'est pas nécessaire.

VII-1-1-2 Les modules

Nous avons introduit précédemment les CAC comme structure programmable pour la construction d'applications parallèles. Nous présentons maintenant le Module comme le grain logique de répartition du code et des activités. Au départ, un CAC est créé en désignant le code décrivant son comportement. Ce sont les *fonctions comportementales* des CAC utilisés dans l'application. Ces fonctions sont regroupées dans les différents modules utilisés dans l'application. Le code d'une application est ainsi partitionné en modules contenant chacun un sous-ensemble des fonctions comportementales. Cette distribution des comportements en modules peut être retardée jusqu'avant la phase d'exécution de l'application.

Un CAC sera donc créé *dans* un module. Durant la phase d'exécution, un module est instancié sur un des noeuds de la machine cible ; il possède alors le code de certains comportements et un sous-ensemble des CAC exécutant ces comportements.

VII - 1•1•2•1 Structure d'un module

La mémoire d'un module est structurée en deux espaces (figure 7.3) :

- un espace code qui prend la forme d'une table des fonctions comportementales du module : cet espace contient plus exactement le code des comportements du module.

- un espace mémoire de CAC : chaque module définit une zone mémoire réservée aux CAC du module lors de l'exécution de l'application. La structure d'un CAC est entièrement allouée dans cette zone.

De plus, chaque module possède une activité spécifique gérant les créations de composants dans le module. Ce processus, appelé Composant Gestionnaire de Module (CGM), est implémenté par un CAC spécialisé. Il est chargé de traiter les "requêtes au module", en particulier les demandes de création de composant.

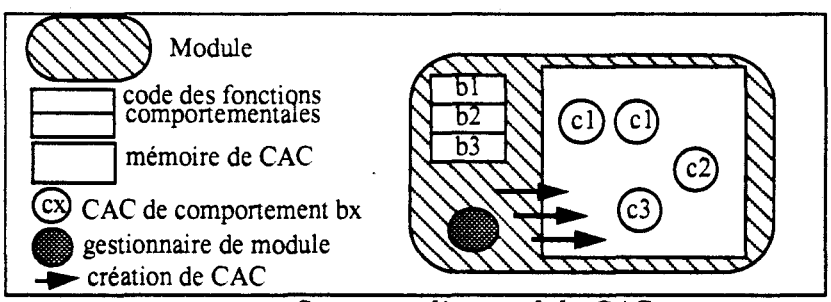


figure 7.3 *Structure d'un module CAC*

VII - 1•1•2•2 Fonctionnalités d'un module

Les modules ont deux fonctionnalités :

- un module est une entité de partage de code : il regroupe un ensemble de fonctions comportementales désignées globalement ; ces noms globaux sont utilisés dans un autre module pour la création à distance de composants ; l'environnement permet difficilement la migration du code à l'exécution et un module réside entièrement sur un noeud de la machine.

- le module est aussi la structure d'accueil des CAC sur les différents noeuds de la machine cible ; chaque module regroupe un ensemble de CAC ayant chacun un comportement parmi ceux contenus dans le module ; un composant vit dans 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'instanciation sur un noeud.

VII - 1•1•2•3 Distribution des fonctions comportementales

Les critères de regroupement des fonctions comportementales sont libres :

- la phase de répartition des fonctions comportementales peut être retardée jusqu'à l'exécution de l'application pour prendre en compte les contraintes de l'architecture cible ; cette distribution peut être le résultat d'un outil de répartition de code et d'équilibrage de charge ayant pour but de répartir au mieux les CAC ; Fred Hemery étudie actuellement les problèmes de répartition de charge au sein de l'environnement CAC ; cette opération peut être cachée au programmeur ; le module est donc un outil de répartition.

- le groupement des fonctions peut être lié à leur programmation ; les modules peuvent explicitement être le résultat d'une programmation modulaire au sens classique, 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 offre de multiples possibilités. Si un module contient un et un seul comportement, il peut être assimilé 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.

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.

L'environnement propose un outil supplémentaire de répartition du code : la duplication de modules. Le programmeur découpe une application en un ensemble de modules puis définit le nombre de duplications de chacun d'eux. Cette phase est indépendante de la conception, c'est un paramètre de l'exécution. Les modules seront ainsi dupliqués sur des noeuds différents de la machine. Ce mécanisme offre plusieurs

avantages :

- la création des composants sera répartie sur les différents noeuds de la machine abritant les modules dupliqués déchargeant ainsi la mémoire de chaque noeud ;
- le parallélisme réel de l'application est augmenté : sans duplication, les CAC de même comportement sont localisés dans le même module, donc sur le même noeud.

VII-1-1-3 Les modules à l'exécution

Après la compilation des modules, le programmeur peut lui-même définir la localisation des modules sur les noeuds de la machine. Il décrit un réseau de modules pour les répartir sur les noeuds de la machine. Cette phase est indépendante de la phase de programmation et permet de prendre en compte les contraintes matérielles de la machine. Il peut aussi laisser cette tâche au système.

VII - 1•1•3•1 Implémentation

Deux implémentations du run-time CAC ont été effectuées. Une application CAC n'a besoin que d'une recompilation pour s'exécuter sur le MultiCluster ou sur le réseau de stations de travail.

Transputer

Une première version du run-time a été implémentée sur une machine parallèle à base de transputers, le MultiCluster II de Parsytec [Parsytec90], équipée de 32 transputers T800 sous le système d'exploitation distribué Helios [Perihelion89]. Sur cette architecture, un module est implémenté par une tâche Helios (processus lourd) et l'activité des CAC par un processus Helios (processus léger). Les processus Helios sont créés dans une tâche et partagent l'espace d'adressage de la tâche. La distribution des modules est réalisée par l'intermédiaire du Component Description Language (CDL), outil du système d'exploitation Helios.

Réseau de stations

Le run-time CAC a aussi été porté sur réseau de stations de travail SUN sous le système d'exploitation SUNOS 4.1, utilisant la bibliothèque pthreads [Mueller93b] et les

communications par sockets. Un module est ici implémenté par un processus Unix et l'activité d'un CAC par un processus léger de la bibliothèque pthreads. La distribution des modules est également réalisée par l'intermédiaire d'un programme cdl qui a été porté sous Unix : un démon sur chaque site est chargé de créer les processus Unix et les liens de communication (socket TCP/IP en mode connecté).

Plus récemment, un portage a été réalisé sur une ferme d'Alpha, toujours en se servant des threads posix. Une version sous Solaris 2 existe aussi, elle se sert des threads du noyau de Solaris et gère les stations multiprocesseurs.

VII-1-2 Outils d'exploitation de PVC

Dans l'environnement PVC, des outils de déverminage d'applications parallèles et de gestion mémoire des CAC ont été développés.

VII-1-2-1 Le déverminage d'applications parallèles

La mise au point d'une application parallèle est une tâche délicate. Pour les programmes séquentiels, des outils utilisant le déverminage cyclique existent, comme dbxtool [Sun86]. Des outils équivalents pour applications distribuées sont difficiles à réaliser. Les difficultés viennent du fait que le programme a un comportement non déterministe et que l'observation d'un système altère son comportement.

VII-1-2-2 Le déverminage dans PVC

Dans PVC, Jean-François Roos a construit un outil de déverminage pour applications écrites à l'aide des CAC [Roos94]. Le mécanisme de réexécution est basé sur une trace d'événements et est de type "dirigé par le contrôle", ceci afin d'éviter la production d'une trace trop importante et donc de perturber l'exécution.

Le déverminage d'une application se fait en deux phases : une phase d'enregistrement et une phase de réexécution. Le mécanisme de réexécution est implanté à l'aide de CAC.

Dans la phase d'enregistrement, le programme s'exécute et nous enregistrons les envois, les réceptions de messages ainsi que les opérations telles que la consultation de boîtes aux lettres ou la création de CAC. Cette trace est sauvegardée sur disque, elle sert de modèle pour la phase de réexécution.

La réexécution consiste en une exécution réelle du programme dirigée par les traces générées lors de l'exécution initiale. Après une première réexécution totale, il est possible d'effectuer des réexécutions partielles de l'application.

VII-1-2-3 Le déverminage dans BOX

Le déverminage des applications BOX se sert du mécanisme de réexécution de PVC.

De nouveaux événements propres au langage doivent être gérés, ceux-ci vont permettre l'écriture de points d'arrêts distribués : la création d'objet, le début et la fin de l'exécution d'une méthode, l'appel d'une méthode du côté du client, la création et la

terminaison d'un fragment, l'envoi, le dépôt, l'extraction et l'attente de message.

Le programmeur peut spécifier des points d'arrêts dans son programme. Lorsque celui-ci sera stoppé, le programmeur pourra consulter, par exemple, la valeur des attributs d'un objet.

Le déverminage sur PVC est opérationnel, l'implémentation sur BOX est en cours de réalisation.

VII-1-2-4 La gestion mémoire des CAC

Dans [Meyer88], Bertrand Meyer dit : "Il serait si agréable d'oublier tout ce qui concerne la mémoire". Cette maxime peut être appliquée à l'environnement PVC. Cédric Dumoulin a réalisé un glaneur de cellules (GC) pour la gestion des CAC dans une application répartie [Dumoulin92]. Le GC se charge de la récupération mémoire des CAC encore présents en mémoire qui ne sont plus référencés et dont l'activité est nulle.

Le GC est divisé en deux parties : sur chaque module tourne un glaneur de cellules local (GCL) qui s'occupe des activités au sein de son module et au niveau de l'application, nous trouvons un glaneur de cellules global (GCG). Les GCL regroupent les informations de leurs CAC pour constituer un sous-graphe. Ensuite les sous-graphes sont réunis par le GCG dans un graphe complet. Le GCG traite ce graphe pour déterminer les activités récupérables en appliquant l'algorithme de [Kafura90][Kafura91], et fait parvenir à chaque GCL une liste de ces activités. Les GCL récupèrent alors les activités récupérables [Dumoulin93].

Le glaneur de cellules doit être implémenté au niveau BOX pour gérer également les objets passifs.

VII-1-3 Conclusion

Nous avons présenté les Composants Actifs de Communication, environnement de programmation simplifiant la modélisation et la conception des applications parallèles. La figure 7.4 présente la structuration des applications dans l'environnement CAC.

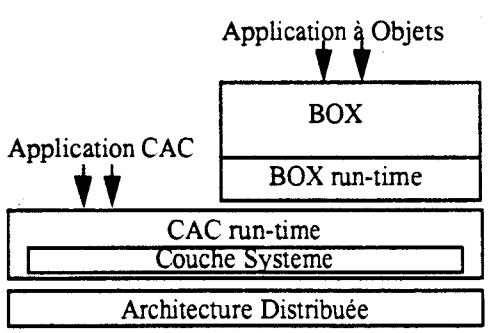


figure 7.4 *L'Environnement Pvc*

Il est bien évidemment possible de programmer directement à l'aide de la bibliothèque CAC mais très vite on se trouve confronté à des difficultés :

- manque de typage des variables et des messages.
- répartition des fonctions comportementales à la main.
- pas de notion d'objet passif.
- pas d'héritage ni de généricité.
- environnement de programmation assez pauvre.

Nous allons voir maintenant les solutions qu'apportent le run-time BOX.

VII-2 Le run-time BOX

Le run-time CAC est un outil intermédiaire. Ce run-time définit un modèle d'exécution exhibant un parallélisme massif d'activités. Mais ce n'est qu'un outil système pour construire des outils de plus haut niveau. Le langage BOX est le premier de ces outils. Le run-time BOX est une extension de celui des CAC/s prenant en charge, en plus, la gestion des objets passifs (objets classiques de l'approche objet), les références entre entités (objets et fragments) et le typage des données contenues dans les messages (pour une extraction plus sélective de ceux-ci).

VII-2-1 Les références entre entités

Dans le langage BOX, les objets sont manipulés par référence. Ces objets sont typés : nous avons des types simples (integer, boolean, real, char), des types complexes (array, string, boîte, message) et enfin des types d'entités définis par l'utilisateur.

La structure DATA englobe toutes les informations pour référencer une entité de type quelconque au sein de notre système. La structure DATA[1] est formée de deux parties : la partie Tag qui indique le type de l'entité et la partie Valeur qui contient la valeur de l'entité pour les types de base, l'adresse étendue pour les autres types.

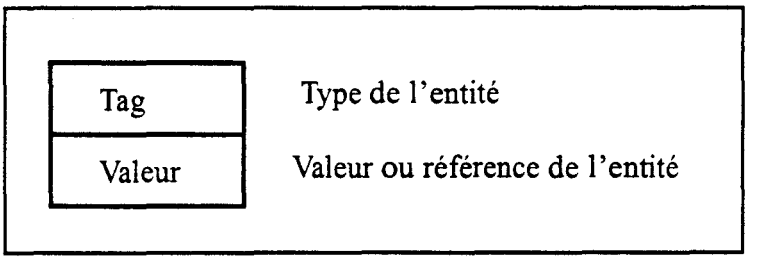


figure 7.5 *Représentation des références: DATA*

Une adresse étendue permet de référencer une entité à partir de n'importe quel point de l'application. Elle est :

- pour un fragment : une adresse CAC de sa boîte aux lettres.
- pour un objet : l'adresse CAC du serveur d'objets sur le site de l'objet suivi de l'adresse mémoire de l'objet dans ce serveur.
- pour un tableau ou un message : la taille et un pointeur mémoire vers le tableau ou le message.
- pour une chaîne : la taille et un pointeur vers la chaîne en langage C.

1. Cette structure fait 16 octets sur SunOs et Solaris ; 24 octets sur Alpha (à cause de l'alignement)

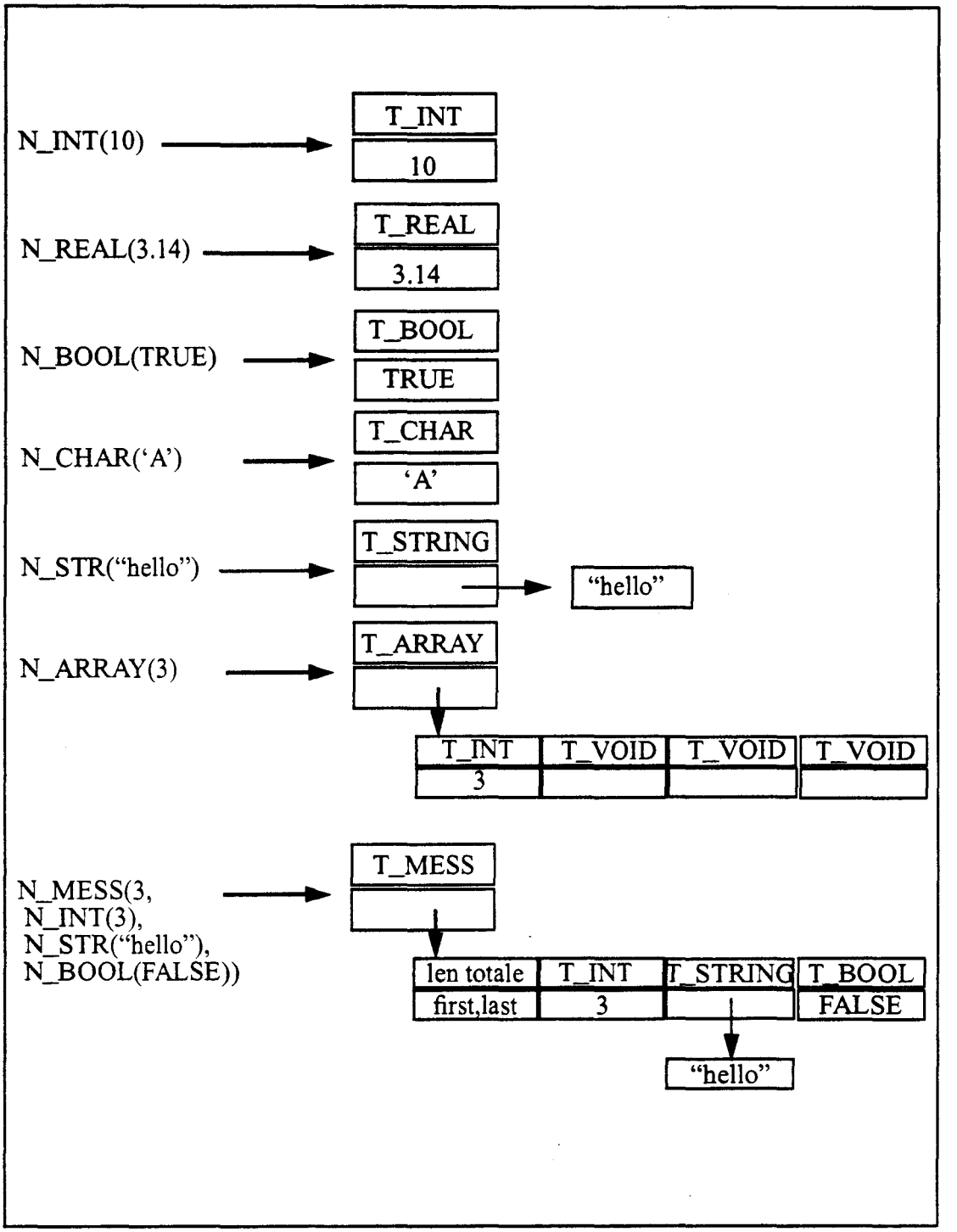


figure 7.6 *Exemples de références à des entités BOX*

Il est nécessaire de donner à chaque type un Tag différent. La numérotation des types est assurée par le compilateur BOX ; cette numérotation est propre à chaque application.

VII-2-2 Structure d'un module BOX

Une application BOX est composée d'un ensemble de modules exécutables. Chaque module contient et gère des classes de fragments et d'objets.

Un module BOX est un module CAC contenant

- un CAC spécialisé appelé serveur d'objets pour gérer tous les objets passifs alloués dans le module.
- les CAC/s supportant l'exécution des fragments actifs alloués dans le module.
- une structure de représentation des classes.

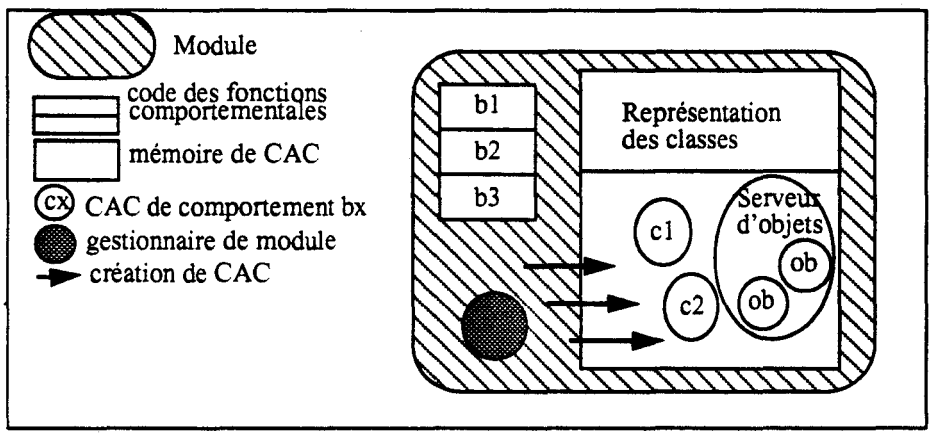


figure 7.7 *Structure d'un module BOX*

VII-2-2-1 Le serveur d'objets

Un CAC spécialisé est appelé "Serveur d'objets". Il fonctionne en tâche de fond dans chaque module et gère les objets localisés sur le module. Ce serveur traite des requêtes à appliquer sur les objets dont il est le gérant. Ces requêtes sont les suivantes :

- ***INIT*** : requête d'initialisation du serveur.
- ***EXECUTE_METHOD (adresse de l'objet, numéro de procédure, arguments)*** : création d'un CAC spécialisé pour exécuter la procédure (le CAC est appelé CEM pour Composant d'Exécution de Méthode). Ce CAC est créé car l'exécution de la méthode peut être bloquante et le serveur doit pouvoir continuer de traiter d'autres requêtes.
  Si l'objet appelé se trouve sur le même module que l'appelant alors la routine est invoquée dans le contexte CAC de l'appelant. Par contre, si l'appelé est distant alors:
  - -l'appelant construit un message d'invocation de routine.
  - -il l'envoie au CAC "Serveur d'objet" du module de l'appelé.
  - -celui-ci crée un CAC CEM (Composant d'Exécution de Méthode) dans lequel la routine sera invoquée.

-à la fin de la routine, le CEM renvoie, au CAC qui a fait l'invocation, un message contenant le résultat renvoyé par la routine.

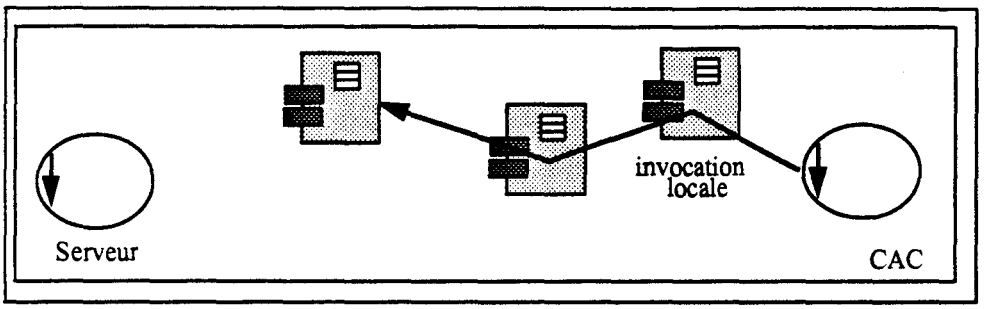


figure 7.8 *Les invocations sur des objets locaux*

Les self invocations (invocation d'une routine locale à l'objet) s'effectuent par l'appel de la fonction représentant la routine invoquée.

- ***GET_ATTRIB (adresse de l'objet, numéro d'attribut)*** : lecture de la valeur d'un attribut d'un objet.
- ***NEW_OBJECT (numéro de classe, numéro de procédure)*** : création d'un objet d'une certaine classe. Si la création est accompagnée d'un appel de procédure d'initialisation, il y a création d'un nouveau CAC au sein duquel la création de l'objet et la procédure d'initialisation seront effectuées.

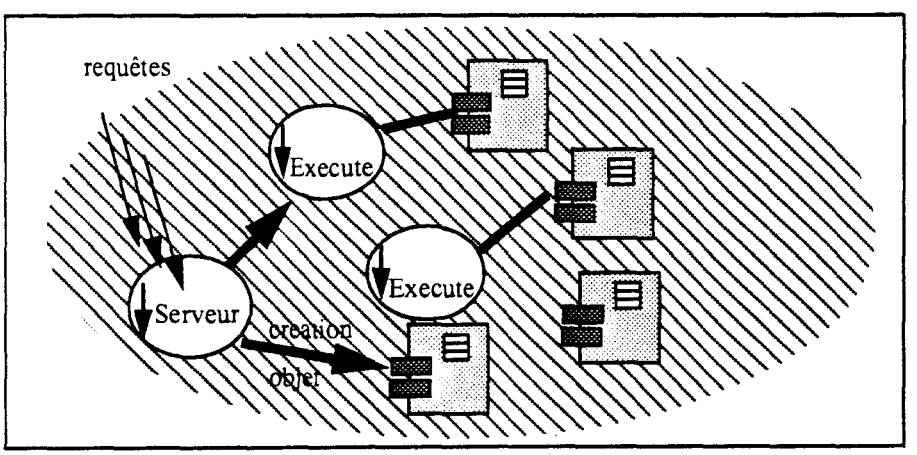


figure 7.9 *Schéma de représentation du "Serveur d'objet"*

VII-2-2-2 Le serveur de fragments

Les fragments sont gérés par le serveur de CAC/s. Les fragments sont créés à l'aide d'un comportement générique. Celui-ci décode les paramètres du message dans lequel il trouve le numéro de la classe de création et le numéro de la procédure à lancer qui est le comportement du fragment. Tous les aspects communications sont gérés par la couche CAC.

VII-2-2-3 Représentation des classes

VII - 2•2•3•1 Table des classes

A l'exécution, les classes sont représentées par une **structure** qui contient les informations suivantes :

- Nature de la classe : une classe peut être une classe d'objets ou de fragments.
- Nombre d'attributs : utilisé lors de la création pour réserver l'espace pour la représentation de l'objet ou du fragment en mémoire.
- Nom de la classe
- Routine d'initialisation : celle-ci permet de fixer le type des attributs et leur valeur par défaut de la nouvelle instance créée.
- La synchronisation : si au moins une des procédures de la classe d'objets est rédactrice, la structure stocke pour chaque procédure de la classe le type de synchronisation (procédure *lectrice* ou *rédactrice*).

Chaque module contient une table des classes définies avec leurs diverses informations pour le système. Les cases du tableau sont remplies si la classe est définie au sein du module et elles sont vides dans le cas contraire.

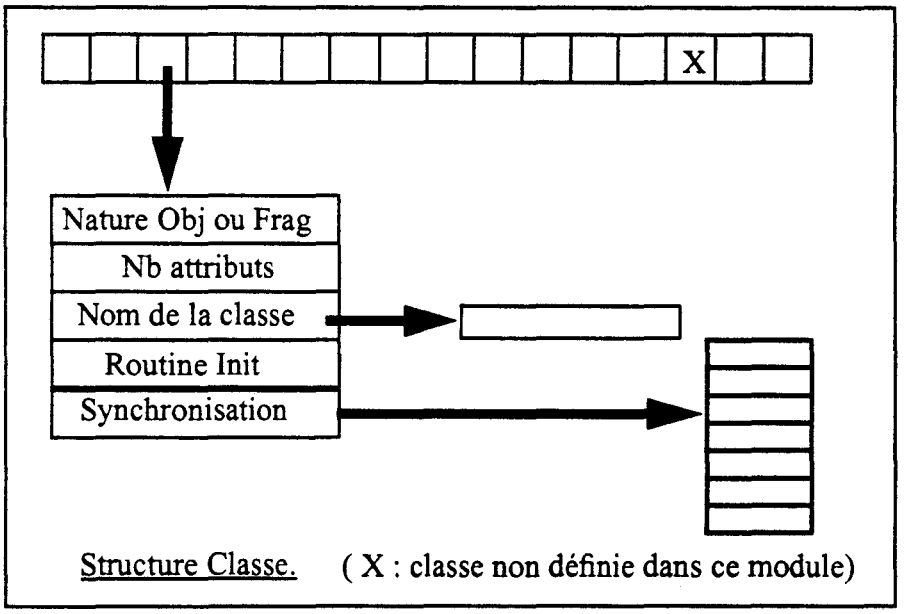


figure 7.10 *Tableau des structures des classes*

VII - 2•2•3•2 Table des méthodes

La table des méthodes d'une classe est une table qui, pour chaque classe présente sur le module, contient une référence à un tableau de pointeurs de fonctions C. Chaque pointeur référence une méthode de la classe. Les méthodes ont le même numéro d'ordre dans une classe B héritant de la classe A que dans A, ceci afin de permettre le

polymorphisme à l'exécution.

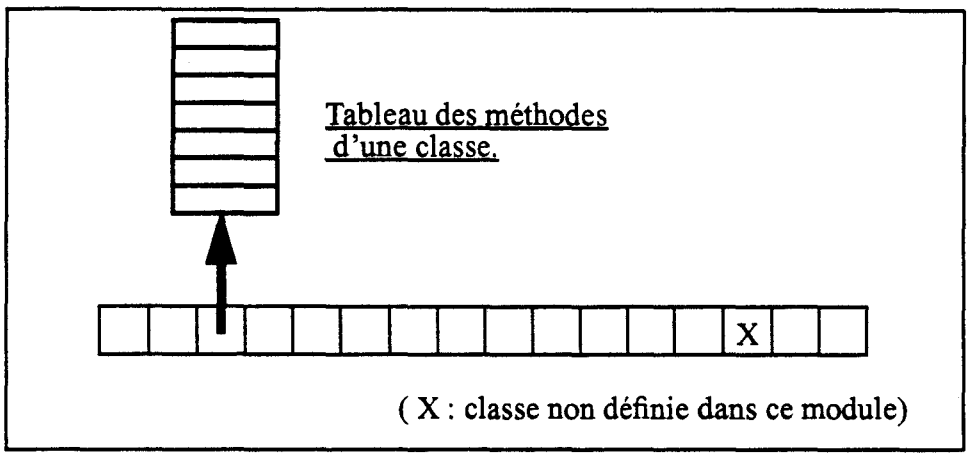


figure 7.11 *Tableau des méthodes des classes d'un module*

Les routines des classes sont compilées en C. Le prototype de chaque fonction C générée contient deux paramètres obligatoires :

- CEM (Composant d'exécution de méthode): indique au sein de quelle activité (CAC) se déroule l'exécution de la routine.
- object : donne une référence sur la représentation mémoire de l'entité sur laquelle porte la procédure.
- les autres paramètres sont les arguments de la procédure ; ils sont optionnels.

```
Data une_routine (BoxCac CEM, T_Object *object, ...)
{
}
```

figure 7.12 *Entête en langage C d'une méthode BOX*

VII-2-2-4 Les instances à l'exécution

VII - 2•2•4•1 Les instances d'une classe d'objets

Une instance est représentée par un tableau de valeurs de ses attributs. Les valeurs contenues par les attributs sont des références vers d'autres instances.

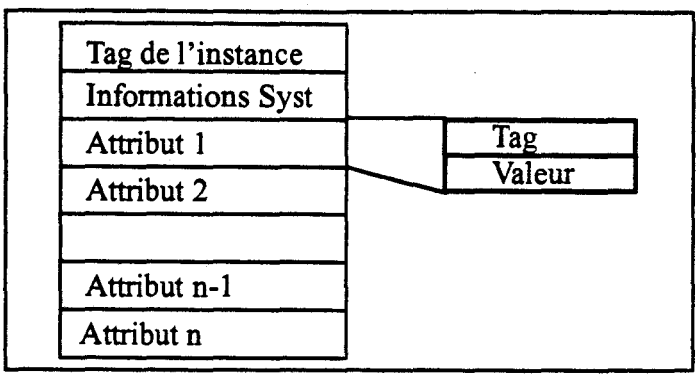


figure 7.13 *Représentation d'une instance d'objet*

Chaque instance connaît le numéro de sa classe.

Les informations système contiennent les données sur l'état de la synchronisation de l'objet.

Les routines d'une classe d'objets peuvent être munies d'une politique de synchronisation. La politique réalisée actuellement est du type lecteurs/rédacteurs. Les routines lectrices peuvent s'exécuter en parallèle. Les routines rédactrices sont exécutées en exclusion mutuelle.

Si une classe ne contient pas de routines rédactrices, alors les invocations sur ses instances ne sont pas synchronisées. Par contre, si l'une de ses routines est rédactrice, alors toutes les invocations sont synchronisées.

Ce mécanisme de synchronisation est basé sur les sémaphores.

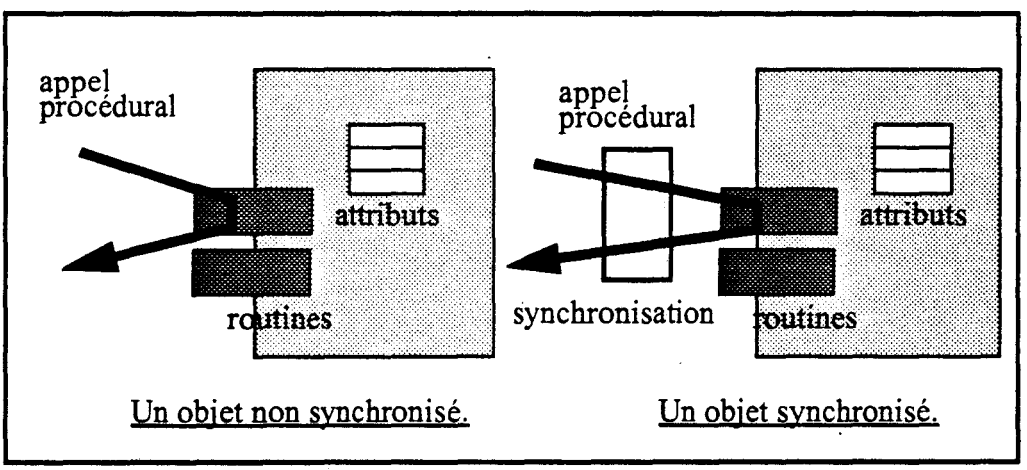


figure 7.14 *Objet synchronisé et non synchronisé*

VII - 2•2•4•2 Les instances d'une classe de fragments

Un fragment a la même représentation mémoire qu'un objet mais il contient une

activité autonome réalisée grâce à un CAC. La partie synchronisation de routines est inutile car la synchronisation du fragment est réalisée par envois de messages asynchrones.

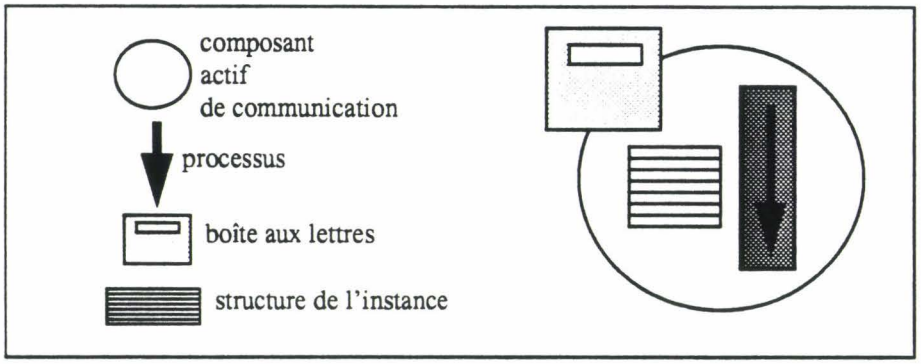


figure 7.15 *Représentation d'un fragment à l'exécution*

VII-2-3 La communication

Dans le langage BOX, pour communiquer avec une entité, on utilise une variable qui contient une DATA. Lorsqu'il s'agit d'un fragment, dans la DATA se trouve l'adresse CAC de la boîte aux lettres. Le run-time CAC véhicule directement le message vers cette boîte aux lettres. Lorsqu'il s'agit d'un objet, dans la DATA se trouve l'adresse CAC de la boîte aux lettres du serveur d'objets du module contenant l'objet ; l'adresse physique de l'objet (qui se trouve aussi dans la DATA) est placée dans le message.

La couche CAC ne permet que le transport de blocs de données. Les messages dans le langage BOX sont plus complexes car un message est composé d'un ensemble de DATA. La couche BOX construit, à partir d'une liste de DATA, un message transportable par la couche CAC. Les types de bases sont placés par valeur, les types construits par le programmeur (classes) sont placés par références et les types complexes sont transmis par valeur également. Pour ces derniers, comme leur valeur a une taille variable, ceux-ci sont placés en fin de message.

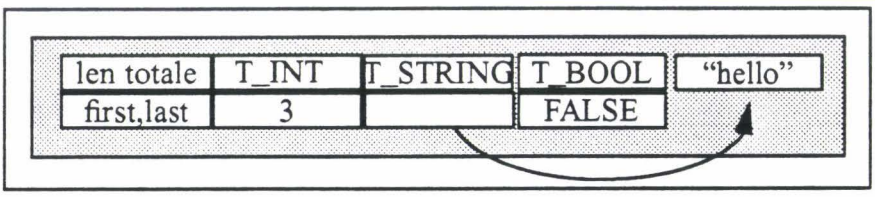


figure 7.16 *Représentation d'un message transporté*

Le déballage de ce message est réalisé à l'arrivée dans la boîte par la couche BOX.

L'opération d'extraction de messages du langage BOX permet d'extraire un message contenant certains types de données (le filtre) et vérifiant une contrainte booléenne.

la table de conformité des types permet, à l'exécution (par exemple lors des extractions de données dans une boîte aux lettres), de vérifier la conformité entre deux types (pour accepter le polymorphisme par héritage). Par exemple, si l'on désire extraire d'une boîte un message contenant une référence sur un "véhicule" alors l'extraction d'une "voiture" sera validée par la table de conformité..

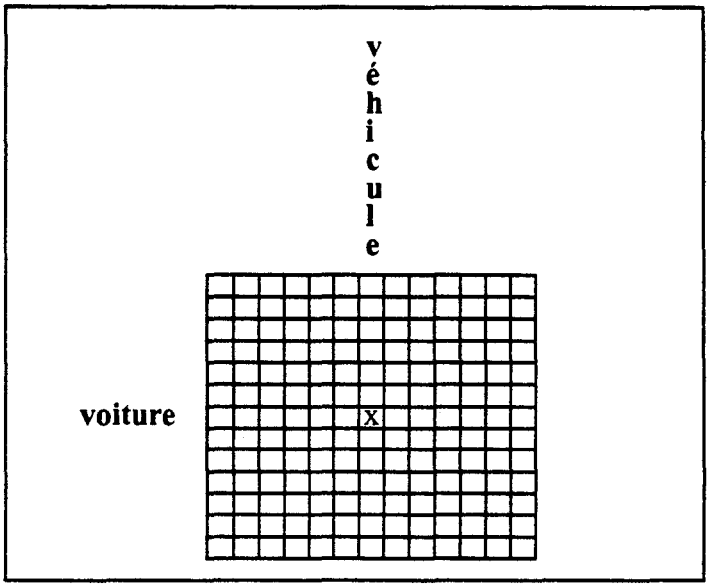


figure 7.17 *Tableau conformité entre les classes*

L'existence de cette table de conformité est due au fait que le langage ne peut faire statiquement le contrôle de type pour la communication par messages (typage dynamique).

VII-2-4 La distribution

A l'exécution, les instances créées sont réparties sur les modules pouvant les accueillir ; c'est-à-dire sur les modules qui contiennent le code relatif à ces instances.

La répartition des fragments et des objets est prise en charge par le run-time BOX. La couche CAC offre des fonctionnalités de répartition pour les fragments mais n'offre pas de fonctionnalités pour les objets. Par souci d'uniformisation, nous avons décidé de gérer la répartition de toutes les entités au niveau BOX.

A l'avenir, nous développerons au sein de la couche CAC une sous-couche spécialisée dans la distribution des entités. Ainsi, les deux politiques seront unifiées en une seule stratégie de répartition d'entités (fragments et objets). Nous comptons étendre les

résultats obtenus par Fred Hemery dans sa thèse [Hemery94] pour gérer la répartition des objets et des fragments.

VII-2-5 Autres CAC systèmes

Un module BOX peut gérer d'autres CAC système. Parmi ceux-ci, nous trouvons le CAC pour la gestion mémoire (glanage de cellules), le CAC chargé de l'enregistrement de la trace d'exécution pour pouvoir reproduire une exécution donnée d'un programme ou encore le CAC de gestion de l'équilibrage de charge.

VII-2-6 Le cas particulier des Erwan classes

VII-2-6-1 Présentation

Une Erwan Class (EC) est une classe d'objets sans attributs. Cette propriété est détectée automatiquement par le compilateur. Cette propriété fait que le code de la classe va être distribué automatiquement dans l'ensemble des modules. L'utilisation classique d'une EC est l'interfaçage avec du code écrit en langage C. Ce type de classe permet de faire des appels au code C de manière locale.

La création d'un objet d'une EC ne va pas consommer de mémoire. Il n'y a en mémoire que la place pour la référence. Une référence est composée du type dynamique de l'objet et d'informations se rapportant à l'entité (cf VII-2-1). Dans le cas d'une EC, il n'y a pas d'informations associées.

L'environnement BOX offre un certain nombre de primitives pour écrire dans la référence et pouvoir ainsi conserver des informations. L'espace disponible pour stocker des informations est très réduit : 96 bits dans l'implémentation actuelle sur Sparc.

Avec les EC, le programmeur peut écrire des classes qui vont implémenter les types de base (entier, caractère, ...). Ce travail a été réalisé mais comme le langage n'offre pas la surcharge des opérateurs, l'utilisation est fastidieuse. Le code généré est actuellement moins efficace que lorsque le programmeur utilise les types prédéfinis.

VII-2-6-2 La localisation des instances

La classe LOCATION est une Erwan Class. Une configuration de modules est définie comme un masque de bits. Si le bit numéro N est à 1 alors le Nième module peut accueillir l'instance ; s'il est à 0, le module ne peut pas accueillir l'instance. Avant de créer une instance, le run-time BOX vérifie que les modules choisis disposent réellement du code de la classe pour laquelle il faut créer une instance. Dans l'implémentation sur architecture Sparc, nous pouvons donc manipuler 96 modules. Sur Alpha, nous pouvons manipuler 160 modules (les références sont plus grandes à cause de l'alignement sur des multiples de 8 octets).

VII-2-7 Conclusion

Nous avons implanté le noyau exécutif BOX sur celui des CAC. Nous l'avons étendu pour permettre la gestion des objets passifs : pour pouvoir les référencer à travers l'application et pour pouvoir y appliquer des procédures. Tous les aspects communications entre entités localisées sur différents modules sont réalisés grâce aux primitives de communication de la couche CAC.

Conclusion

Pour répondre aux problèmes posés par la conception d'applications réparties, notre proposition est dans cette thèse de réutiliser le modèle objet - connu pour ses qualités - pour la conception de ces applications. Le modèle objet traditionnel doit cependant être étendu pour prendre en compte la distribution des architectures visées et pour s'adapter à la réalisation d'applications utilisant plusieurs flots d'exécution dans des espaces mémoires disjoints.

Nous introduisons un modèle objet pour applications réparties qui permet la modélisation en termes d'objets et d'activités en plaçant ces deux notions au même niveau de conception. Ce modèle est à la fois un modèle objet traditionnel et un modèle de processus communicants. Il peut être simplement présenté de la manière suivante :

- Le modèle de BOX se fonde sur des objets et des processus créés dynamiquement à partir de classes d'objets et de classes de processus. Un objet est une instance d'une classe d'objets, un processus est une instance d'une classe de processus.
- Une classe décrit la structure interne de ses instances. De plus une classe d'objets décrit les méthodes applicables sur ses instances et une classe de processus décrit les différents comportements que peuvent prendre ses instances lors de leur création.
- Les objets comme les processus référencent d'autres objets ou processus. Ils communiquent par envoi de messages avec les processus qu'ils référencent, et par appel de méthode avec les objets qu'ils référencent.
- Une exécution comprend autant de flots d'exécution que d'instances de processus. Initialement, un seul flot existe correspondant au lancement d'un objet racine.

Nous avons conçu un langage implantant ce modèle. Le langage BOX est un langage compilé, pleinement objet (à base de classes, avec héritage et généricité...) qui facilite donc le développement d'applications réparties et la conception de composants réutilisables. Le langage s'intéresse à la programmation des classes, indépendamment de la phase de configuration qui installe le logiciel produit sur une architecture particulière. Cette phase de configuration est prise en charge par un ensemble d'outils qui forment l'environnement de développement BOX. L'exécution d'un programme BOX est supportée par un système d'exécution spécifique se basant sur un système de processus légers distribués prenant en charge les nombreux flots d'exécution générés par ces programmes.

L'ensemble formé par ce modèle, ce langage et ces outils forme un cadre conceptuel nouveau et puissant pour concevoir les applications demandées par ces nouvelles architectures. L'ensemble du système BOX est portable et facilement installable sur toute architecture décentralisée (réseau de stations, cluster de processeurs) munie des fonctionnalités traditionnelles des systèmes d'exploitation répartis.

Au travers du modèle et du langage sont revisités certains des problèmes connus des langages parallèles à objets.

- Concernant l'aspect **synchronisation**, le système BOX utilise plutôt une synchronisation basée sur des communications synchronisantes (grâce à une opération puissante de réception avec filtre et contraintes), alors que de nombreux autres langages parallèles à objets se fondent sur des conditions d'activation (toujours difficiles à exprimer) dans des objets partagés. Ce qui modifie profondément le modèle objet initial.

- Concernant l'aspect **distribution**, nous montrons qu'il est possible de dissocier l'aspect programmation des objets et des activités de la phase de configuration de l'application.

- Concernant l'aspect **composant logiciel**, le langage BOX, qui place au même niveau conceptuel les objets et les activités, permet la création d'agrégats actifs (les Objets Actifs Complexes) qui sont de véritables composants logiciels réutilisables pour applications réparties. Il existe à notre connaissance peu de systèmes qui permettent cela. Les Objets Actifs Complexes forment intuitivement un guide méthodologique important qui reste à explorer.

- Concernant enfin la **modélisation**, BOX permet une conception en termes d'acteurs, ou en termes d'objets et de processus, ou encore en termes d'objets actifs. Il permet d'utiliser la communication par messages ou l'appel de méthode à distance. Le concepteur est libre de choisir et de mixer ces modes de conception pour s'adapter à la nature profonde de l'application et de ses composants. Le tout dans un cadre conceptuel clair.

Perspectives

Le système BOX tel qu'il a été décrit dans cette thèse est à ce jour opérationnel. Un certain nombre d'extensions et de réflexions sont ou doivent être envisagées.

Modèle

Nous comptons étudier les impacts méthodologiques des Objets Actifs Complexes dans la phase de conception d'applications réparties. Nous allons évaluer les bénéfices apportés dans le cycle de développement et de réutilisation de composants logiciels.

Langage

Dans BOX, le programmeur doit spécifier à la création de fragment le comportement que celui-ci doit exécuter. Nous avons introduit également un mécanisme qui permet de substituer un nouveau comportement à celui en cours d'exécution (comme le become des acteurs). Nous devons étudier l'intérêt de ce mécanisme dans le but de faciliter la conception des fragments.

De même, ce mécanisme permet de représenter le comportement d'un fragment à l'aide d'un automate. Dans chaque état, le fragment peut effectuer du calcul et accepter certains messages. Le changement de comportement se fait suite à l'arrivée d'un message ou selon l'état courant du fragment. Nous étudierons la relation d'une spécification de comportement sous forme d'automate avec l'héritage ; ceci pour tenter d'apporter une solution au problème d'*inheritance anomaly*.

Nous espérons introduire un dispositif de persistance dans BOX. Les objets et fragments pourraient être utilisés dans plusieurs exécutions d'applications. Des produits pour gérer la persistance des objets existent actuellement sur le marché mais, à notre connaissance, aucun ne gère la persistance d'entités actives.

Nous pensons interfacer BOX avec d'autres langages, ceci afin de pouvoir réutiliser du code écrit dans d'autres langages. Nous pensons également interfacer BOX avec des bases de données. Ces travaux peuvent être rattachés aux perspectives de persistance et de liaison avec CORBA.

Le langage BOX dispose d'un mécanisme d'héritage. Celui-ci est qualifié d'héritage simple. Nous comptons l'étendre pour offrir au programmeur BOX un mécanisme d'héritage multiple.

Environnement

L'environnement BOX doit être complété par un glaneur de cellules pour les objets. Ce GC est actuellement réalisé dans la couche PVC pour récupérer les CAC inutiles. De même, dans la couche PVC existe un CAC spécialisé pour l'équilibrage de charge entre les modules. Cet outil d'équilibrage gère les CAC. Nous devons l'étendre pour qu'il puisse également évaluer la charge des modules en termes d'objets. Ce mécanisme devra être lié avec le mécanisme de localisation des instances de BOX.

Actuellement, une application est figée au moment de sa construction. Nous comptons intégrer un mécanisme qui permettra d'ajouter des modules dynamiquement au cours de l'exécution. Cette amélioration ne modifie en rien le modèle.

La bibliothèque standard doit être étoffée. Actuellement, elle comporte des classes pour la gestion de structures de données à accès parallèles (tampon, arbre binaire, ...). D'autres types de structures doivent être ajoutées. Nous pensons également écrire de nouvelles classes pour la gestion de la localisation des entités.

BOX fonctionne sur un réseau local de stations. Nous comptons étudier la faisabilité d'une version fonctionnant sur un réseau à grande échelle. Nous tentons également d'interfacer BOX avec CORBA. Une thèse sur le développement d'un ORB est en cours [Merle96].

Support d'exécution

Dans sa version actuelle, BOX permet de concevoir des applications sur un ensemble de machines homogènes. Nous comptons permettre l'écriture d'applications pour machines hétérogènes. Cette extension ne modifie en rien le modèle ni le langage. Les modifications portent sur la phase de conception d'applications dans laquelle il faudra (entre autres) pouvoir spécifier sur quel type d'architecture un module devra s'exécuter.

Le concept de mémoire virtuelle distribuée est une alternative à notre mécanisme de nommage des objets. Les références et l'application de méthodes sur les objets seront gérés par ce mécanisme. Des travaux sur ce sujet sont en cours [Dumoulin95].

PVC/BOX n'est pas tolérant aux pannes. Nous comptons introduire dans le run-time des fonctionnalités de tolérances (réplication, notion de groupes d'entités, ...). Au niveau du langage, nous allons introduire un mécanisme d'exceptions et de gestion de groupes.

Utilisation

Nous comptons fournir des abstractions sous forme de classes qui permettront au programmeur de développer des applications à parallélisme de données [Jézéquel93][Lim93] ; par exemple, pouvoir dupliquer des entités sur plusieurs noeuds de manière transparente pour le programmeur et pouvoir, ensuite, accéder à la copie locale. Ces abstractions permettront d'écrire des programmes basés sur le modèle SPMD.

BOX est interfacé avec X-window. Une application peut gérer des fenêtres X. Il est possible dans un même programme de gérer des fenêtres qui vont apparaître sur des écrans différents et donc de faire coopérer des utilisateurs (répartis physiquement !) par l'intermédiaire d'une application BOX. Ce travail est en cours de réalisation [Grenot95]. Nous sommes en train d'étendre la trilogie bien connue MVC de Smalltalk [Krasner88] à BOX pour arriver à une structure $M^nV^oC^p$ permettant la prise en compte du caractère distribué du système. Nous comptons également développer des classes pour gérer les autres médias (le son, ...).

Annexe - A -

Interfaçage avec le langage C

A-1 Introduction

Le langage BOX permet de réutiliser du code écrit en langage C. Ceci implique de pouvoir appeler des fonctions C à partir de BOX.

Il existe deux façons de les spécifier : l'une permet des appels directs aux fonctions C : les paramètres BOX sont automatiquement transformés par le compilateur avant d'être passés aux fonctions C ; de même pour le résultat éventuel d'une fonction. Bien entendu cette méthode nécessite de n'utiliser que des paramètres de types simples (entiers, réels, booléens, caractères). La seconde méthode permet de transmettre des structures BOX qui sont passées en paramètres à des fonctions C (obligatoirement écrites par un programmeur BOX). Ces structures doivent par conséquent être décodées pour pouvoir être utilisées dans les fonctions C. En retour, le résultat doit être codé sous forme d'une structure reconnaissable par le compilateur.

A-2 Syntaxe

La syntaxe de ces appels est la suivante:

```
-- les traitements de transfert sont pris en charge par le compilateur
[PROC[TYPE_RESULTAT]] NOM_FONCTION_BOX : LISTE_PARAMETRES :
        CALL "NOM_FONCTION_C"

-- les traitements de transfert sont à la charge du programmeur
[PROC[TYPE_RESULTAT]] NOM_FONCTION_BOX : LISTE_PARAMETRES :
        EXTERNAL "NOM_FONCTION_C"
```

Dans le cas où le transfert doit être pris en charge par le programmeur, les fonctions C doivent avoir le format suivant :

```
Data NOM_FONCTION_C(BoxCac CEM, T_Object *object, Data par1, Data
par2,..)
{ ...}
/* CEM référence le processus qui exécute la fonction NOM_FONCTION sur
l'objet object ; par1, par2 correspondent aux paramètres passés dans la
fonction BOX */
```

Quelques fonctions du run-time BOX sont accessibles qui permettent de manipuler les structures BOX dans une fonction C :

```
TAG(nom_data) /* renvoie le type de la Data */
T_INT T_BOOL .... T_STR /* types de Data */
V_INT(nom_data)
V_BOOL(nom_data)        /* permettent d'accéder à la valeur d'une data
*/
... V_STR(nom_data)
N_INT(nom_entier)
N_BOOL(nom_entier)      /* permettent de transformer une variable C */
...N_STR(nom_entier)    /* en une Data BOX */
```

A-3 Liaison avec une classe

La liaison entre une classe BOX et un fichier compilé (objet ou biblitohèque), se fait à l'aide du langage de description d'applications dans la clause OBJ_FILES. Le programmeur spécifie les fichiers compilés qui doivent être associés à une classe particulière. Dans la phase de construction des modules, chaque fois que la classe est ajoutée dans un module, l'éditeur de liens fait une édition avec les fichiers spécifiés.

```
(OBJ_FILES)
      -- liste fichiers .o ou .a à utiliser par classe
      nom_classe : "nom_fichier.o"
```

A-4 Elément de la bibliothèque

Nous présentons la classe BOX qui permet de réaliser les entrées/sorties. Cette classe utilise de code écrit en langage C.

Les entrées/sorties sont réalisées à l'aide d'une classe d'objets de nom File.

```
[OBJ]file :      :

  [STR]file_name;    -- le nom du fichier traite
  [ANY]fp;           -- pointeur du fichier

  [INT]mode;         -- I/O mode peut prendre les valeurs suivantes
  [INT]closed;
  [INT]read;
  [INT]write;

  [INT]eof;           -- marqueur de fin de fichier
                      -- valeur : -1
  [INT]error_val;     -- code erreur pour les erreurs de lecture
  [INT]read_error;    -- erreur lors d'une tentative de lecture
                      -- valeur : 1
  [INT]zeof_error;    -- fin de fichier atteinte lors d'une lecture
                      -- valeur : 2
  [INT]open_error;    -- erreur a l'ouverture
                      -- valeur : 3

-------------------------------------------------------------------------
-
---    PROCEDURES INTERFACAGE C
-------------------------------------------------------------------------
-

  [PROC[BOOL]]C_end_of_file :[ANY]fp:
    EXTERNAL "C_end_of_file"

  [PROC[INT]]C_get_ci:[ANY]fp:
    EXTERNAL "C_get_ci"

  [PROC[INT]]C_get_i:[ANY]fp:
    EXTERNAL  "C_get_i"

  [PROC[REAL]]C_get_r:[ANY]fp:
    EXTERNAL  "C_get_r"

  [PROC[STR]]C_get_s:[ANY]fp:
    EXTERNAL  "C_get_s"

  [PROC[STR]]C_get_s_of_len:[ANY]fp;[INT]l:
    EXTERNAL  "C_get_s_of_len"

  [PROC[STR]]C_get_s_with_spaces:[ANY]fp;[INT]taille:
      EXTERNAL "C_get_s_with_spaces"
```

```
[PROC]C_unget_c:[CHAR]c;[ANY]fp:
   EXTERNAL "C_unget_c"

[PROC[ANY]]C_in::-- creation d'un fichier entree standard(stdin)
   EXTERNAL "C_in"

[PROC[ANY]]C_out::
     -- creation d'un fichier sortie standard(stdout)
   EXTERNAL "C_out"

[PROC[ANY]]C_err::
     -- creation d'un fichier erreur standard(stderr)
   EXTERNAL "C_err"

[PROC]C_putc:[CHAR]c;[ANY]fp:
   EXTERNAL "C_putc"

[PROC]C_printi:[ANY]fp;[STR]type;[INT]aff:
   EXTERNAL "C_printi"

[PROC[INT]]C_printr:[ANY]fp;[REAL]aff:
   EXTERNAL "C_printr"

[PROC[INT]]C_prints:[ANY]fp;[STR]type;[STR]aff:
   EXTERNAL "C_prints"

[PROC[ANY]]C_open:[STR]name;[STR]mode:
     -- ouverture du fichier name en lecture
   EXTERNAL "C_open"

[PROC]C_close:[ANY]fp:-- fermer le fichier
   EXTERNAL "C_close"

[PROC[CHAR]]C_i_to_c:[INT]i:
   CALL  "C_i_to_c"

[PROC] C_msg_out : [MESS] msg ; [ANY] fp :
   EXTERNAL " C_msg_out" ;


-------------------------------------------------------------------------
-------  PROCEDURES APPELABLES PAR L'UTILISATEUR
-------------------------------------------------------------------------

[PROC[BOOL]]end_of_file :: -- vrai si une fin de fichier a ete lue
{
  result:=C_end_of_file(fp)
}


[PROC[BOOL]]get_b::-- lecture d'un booleen
{
  [INT]ci:=get_ci;
  if ci = eof then
    error_val := zeof_error;
    result:=[BOOL]null;
  else
    [CHAR]c := C_i_to_c(ci);
```

```
    if c = 'T' or c = 't' then
   result:=true
    else_if c = 'F' or c = 'f' then
   result:=false
    else
   result:=[BOOL]null;
   error_val:=read_error;
    end_if
  end_if
}

[PROC[CHAR]]get_c::          -- lecture d'un caractere
{
  result:=C_i_to_c(C_get_ci(fp))
}

[PROC[INT]]get_ci::-- lecture d'un caractere comme un entier
{
  result:=C_get_ci(fp)
}

[PROC[INT]]get_i::           -- lecture d'un INT
{
  result:=C_get_i(fp)
}

[PROC[REAL]]get_r::          -- lecture d'un reel
{
  result:=C_get_r(fp)
}

[PROC[STR]]get_s::
     -- lecture d'une chaine jusqu'au premier new-line
{
  result:=C_get_s(fp);
}

[PROC[STR]]get_s_of_length:[INT]l:
   -- lecture d'au maximum l caracteres
{
  result:=C_get_s_of_len(fp,l)
}

[PROC[STR]]get_s_with_spaces::
  -- lecture d'une chaine de caracteres pouvant contenir des espaces
  -- au maximum 256 caracteres
{
  result:=C_get_s_with_spaces(fp,256)
}

[PROC[STR]]get_s_with_spaces_of_length:[INT]l:
  -- lecture d'une chaine de caracteres pouvant contenir des espaces
  -- de taille l
{
  result:=C_get_s_with_spaces(fp,l)
}

[PROC]unget_c:[CHAR]c:-- remettre le dernier caractere lu
```

```
{
  C_unget_c(c,fp)
}

[PROC]in::-- creation d'un fichier entree standard(stdin)
          -- in servira a la fois de create et de open
{
  fp:=C_in;
  file_name:="in"
}

[PROC]out::
          -- creation d'un fichier sortie standard(stdout)
          -- out servira a la fois de create et de open
{
  fp:=C_out;
  file_name:="out"
}

[PROC]err::
          -- creation d'un fichier erreur standard(stderr)
          -- err servira a la fois de create et de open
{
  fp:=C_err;
  file_name:="err"
}

[PROC]b:[BOOL]bo:-- sortie d'un booleen
{
  if bo then
    C_putc('T',fp)
  else
    C_putc('F',fp)
  end_if
}

[PROC]c:[CHAR]ch:-- sortie d'un caractere
{
  C_putc(ch,fp)
}

[PROC]i:[INT]in:-- sortie d'un entier
{
  C_printi(fp,"%d",in)
}

[PROC]r : [REAL]re:
{
  [INT]i := C_printr(fp,re);
}

[PROC]s:[STR]st:-- sortie d'une chaine
{
  [INT]i := C_prints(fp,"%s",st)
}

[PROC]nl::-- sortie d'un new_line
{
```

```
    C_putc('\n',fp);
  }

  [PROC] m : [MESS] message :  -- sortie d'un message
  {
    C_msg_out (message, fp);
  }

  [PROC]openr:[STR]name:
       -- ouverture du fichier name en lecture
  {
    fp:=C_open(name,"r");
    file_name:=name;
    if fp = [ANY]null then
      error_val:=open_error
    end_if
  }

  [PROC]openw:[STR]name:-- ouverture du fichier name en ecriture
  {
    fp:=C_open(name,"w");
    file_name:=name;
    if fp = [ANY]null then
      error_val:=open_error
    end_if
  }

  [PROC]opena:[STR]name:-- ouverture du fichier name pour ajouts
  {
    fp:=C_open(name,"a");
    file_name:=name;
    if fp = [ANY]null then
      error_val:=open_error
    end_if

  }

  [PROC] close ::
  {
    C_close (fp) ;
  }
```

L'interface du côté C est la suivante. Cela illustre la manière de faire converser Box et C.

```
#include <Prim_Obj_Box.h>
#include <stdio.h>
#include <floatingpoint.h>

#define ptr_fichier(f) V_OBJ(f).object

Data C_get_i(BoxCac CEM, T_Object *object,Data dfp)
{
  FILE *fp = (FILE *)ptr_fichier(dfp) ;
  int i;

  fscanf(fp,"%d",&i);
  return N_INT(i);
```

```
}

Data C_get_r(BoxCac CEM, T_Object *object,Data dfp)
{
  FILE *fp = (FILE *)ptr_fichier(dfp) ;
  double RrR;
  Data dr;
  char s[20];

  fscanf(fp,"%s",s);
  RrR=(double) atof(s);
  return N_REAL(RrR);
}

Data C_get_s(BoxCac CEM, T_Object *object,Data dfp)
{
  FILE *fp = (FILE *)ptr_fichier(dfp) ;
  char s [256] ;

  fscanf(fp,"%s",s);
  return N_STR(s);
}

Data C_get_s_of_len(BoxCac CEM, T_Object *object,Data dfp,Data in)
{
  FILE *fp = (FILE *)ptr_fichier(dfp) ;
  char s [256] ;
  char m [10] ;

  sprintf (m ,"%%%ds", V_INT(in)) ;
  fscanf(fp,m ,s);
  return N_STR(s);
}

Data C_printr(BoxCac CEM , T_Object *object , Data dfp , Data  dre)
{
  FILE *fp =  (FILE *)ptr_fichier(dfp) ;
  double re = V_REAL(dre);
  char s[20];

  gconvert(re,12,0,s);
  fprintf(fp,"%s",s);
  return N_INT(1);
}



Data C_get_s_with_spaces(BoxCac CEM , T_Object *object , Data dfp , Data
lg)
{
  FILE *fp = (FILE *)ptr_fichier(dfp) ;
  int ln = V_INT(lg);
  char s [256] ;

  fgets(s,ln,fp);
  if (strlen(s) != 0)
  s[strlen(s)-1] = 0;
  return N_STR(s);
```

```
 }

Data C_in(BoxCac CEM , T_Object *object)
{
  Data result ;

  SET_DATA_ADR (result, T_ANY, 0, (long)(stdin)) ;
  return result;
}

Data C_out(BoxCac CEM , T_Object *object)
{
  Data result ;

  SET_DATA_ADR (result, T_ANY, 0, (long)stdout) ;
  return result;
}

Data C_err(BoxCac CEM , T_Object *object)
{
  Data result ;

  SET_DATA_ADR (result, T_ANY, 0, (long)stderr) ;
  return result;
}


char C_i_to_c(int in)
{
  return (char)in;
}

/* Impression d'un message complet */

static void imprime_dans_string (Data d, char *dest)
{
  switch(TAG(d)) {
    case T_VOID:
      break ;
    case T_INT:
      sprintf (dest,"%d",V_INT(d)) ;    break ;
    case T_BOOL:
      sprintf (dest,"%d",V_BOOL(d)) ;   break ;
    case T_CHAR:
      sprintf (dest,"%c",V_CHAR(d)) ;   break ;
    case T_REAL:
     gconvert(V_REAL(d),12,0,dest) ;
      break ;
    case T_STR:
      sprintf (dest,"%s",V_STR(d)) ;
      break ;

    default:
      break;
  }
}

Data C_msg_out (BoxCac CEM, T_Object *object, Data message, Data dfp)
```

```
{
  FILE *fp =  (FILE *)ptr_fichier(dfp) ;

  Data result ;
  int debut = BEGIN_MESS(message) ;
  int fin = END_MESS(message) ;
  char chaine [512] ;
  char *pt = chaine ;
  int i ;

  for(i=debut;i<fin;i++) {
    imprime_dans_string (ARR(message,i),pt) ;
    pt = &chaine[strlen(chaine)] ;
  }
  fprintf (fp, "%s", chaine) ;
  return result ;
}

Data C_end_of_file (BoxCac CEM, T_Object *object, Data dfp)
{
  FILE *fp =  (FILE *)ptr_fichier(dfp) ;

  return N_BOOL(feof (fp)) ;
}

Data C_get_ci (BoxCac CEM, T_Object *object, Data dfp)
{
  FILE *fp =  (FILE *)ptr_fichier(dfp) ;

  return N_INT(getc (fp)) ;
}

Data C_unget_c (BoxCac CEM, T_Object *object, Data car, Data dfp)
{
  FILE *fp =  (FILE *)ptr_fichier(dfp) ;

  Data result ;
  ungetc (V_CHAR(car), fp) ;

  return result ;
}

Data C_putc (BoxCac CEM, T_Object *object, Data car, Data dfp)
{
  Data result ;

  FILE *fp =  (FILE *)ptr_fichier(dfp) ;

  fputc (V_CHAR(car), fp) ;
  return result ;
}

Data C_printi (BoxCac CEM, T_Object *object, Data dfp, Data type, Data
aff)
{
  Data result ;

  FILE *fp =  (FILE *)ptr_fichier(dfp) ;
```

```
  fprintf (fp, V_STR(type), V_INT (aff)) ;
  return result ;
}


Data C_prints (BoxCac CEM, T_Object *object, Data dfp, Data type, Data
aff)
{
  Data result ;

  FILE *fp =  (FILE *)ptr_fichier(dfp) ;

  fprintf (fp, V_STR(type), V_STR (aff)) ;
  return result ;
}

Data C_open (BoxCac CEM, T_Object *object, Data name, Data mode)
{
  Data result ;

  FILE *fp ;

  fp = fopen (V_STR(name), V_STR(mode)) ;
  SET_DATA_ADR (result, T_ANY, 0, (long)fp) ;
  return result ;
}
Data C_close (BoxCac CEM, T_Object *object, Data dfp)
{
  Data result ;

  FILE *fp =  (FILE *)ptr_fichier(dfp) ;

  fclose (fp) ;
  return result ;
}
```

Le code CLUC qui décrit cette classe est :

```
(BOX_FILES)
        "$BOX/library/io/file.bx"

(OBJ_FILES)
        file:"$BOX/library/io/$SYSTEMARCHI/File.o"
```

BOX et SYSTEMARCHI sont des variables d'environnement qui contiennent respectivement le chemin pour atteindre le compilateur BOX et le type d'architecture sur lequel le programmeur travaille.

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

[Agha88] G. Agha, C. Hewitt
Concurrent Programming Using Actors
[Yonezawa87], pp. 37-53.

[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

[Arcangeli94] J.P. Arcangeli, A. Marcoux, C. Maurel, P. Sallé
La programmation concurrente par acteurs : le système PLASMA II
La lettre du Transputer et des calculateurs parallèles, numéro spécial Les Langages à Objets, no 22, juin 1994.

[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

[Balter94] R. Balter, S. Lacourte, M. Riveill
The Guide Language
To appear in The Computer Journal, 1994.

[Banâtre91] J.P. Banâtre, M. Banâtre
Les systèmes distribués - Expérience du Projet GOTHIC
InterEditions 1991

[Bekele94] D. Bekele
Contribution à l'étude de la Répartition d'Applications écrites en langage Ada 83
Thèse de doctorat en informatique, IRIT, Octobre 1994, Toulouse.

[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

[Benveniste93] JM. Benveniste, V. Issarny
Concurrent Programming Notations in the Object-Oriented Language Arche
TR, IRISA, 1993.

[Birman93] K.P. Birman
The process group approach to reliable distributed computing
CACM, 1993

[Birman93b] B.B. Glade, K.P. Birman, R.C.B. Cooper, R. van Renesse
Light-weight process groups in the Isis system
The British Computer Society, The Institution of Electrical Engineers and IOP Publishing Ltd, 1993.

[Birrell83] A.D. Birrell, B.J. Nelson
Implementing Remote Procedure Calls
Xerox report CSL-83-7, October 1983

[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.

[Briot94] J.P. Briot
Modélisation et classification de langages de programmation concurrente à objets : l'expérience Actalk
Langages et Modèles à Objets, 13-14 octobre 1994, Grenoble.

[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

[Caromel94] D. Caromel
Programmation parallèle à objets : Eiffel //
La lettre du Transputer et des calculateurs parallèles, numéro spécial Les Langages à Objets, no 22, juin 1994.

[Carriero89] N. Carriero, D. Gelernter
How to Write Parallel Programs : A Guide to the Perplexed
ACM Computing Surveys, Setpember 1989, pp 323-357.

[Carriero90] N. Carriero, D. Gelernter
How to Write Parallel Programs : A First Course
MIT Press, 1990.

[Chorus91]
Chorus Kernel v3 r4.0 Programmer's Reference Manual
Technical Report CS/TR-91-71, Chrorus systèmes, september 1991.

[Corbin90] J.R. Corbin
The Art of Distributed Applications
Springer-Verlag, 1990.

[Courtrai92] L. Courtrai
Les Composants Actifs de Communication : Outil pour la conception et l'implantation de langages parallèles à objets actifs pour machines MIMD
PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Octobre 1992.

[Day83] J.D. Day, H. Zimmermann
The OSI Reference Model
Proceedings of IEEE, vol 71, December 1983, pp. 1334-1340.

[Debbage94] M. Debbage, M. Hill, S. Wykes, D. Nicole
Southampton's Portable Occam Compiler (SPOC)
Department of Electronics and Computer Science, University of Southampton, Febrary 1994.

[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

[Denneulin94] Y. Denneulin
Introduction des procédures asynchrones dans le langage BOX
Mémoire de DEA, LIFL, Septembre 1994

[Dijkstra65] E. W. Dijkstra
Co-operating sequential Processes
Programming Languages Genuys editor, London 1965

[Dumoulin92] C. Dumoulin
Un ramasse-miettes pour les Composants actifs de communication
Mémoire de DEA , LIFL, Septembre 1992

[Dumoulin93] C. Dumoulin, J.F. Roos, J.F. Méhaut
Le ramasse-miettes du projet PVC/BOX
RENPAR'5, Brest, 1993.

[Dumoulin95] C. Dumoulin
Une mémoire virtuelle distribuée pour le projet PVC/BOX
PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, en préparation.

[Engler93] D.R. Engler, G.R. Andrews, D.K. Lowenthal
Eifficient Support for Fine-Grain Parallelism
University of Arizona, TR 93-13, 1993.

[Feldman91] J.A. Feldman, C.C. Lim, F. Mazzanti
pSather monitors : Design, Tutorial, Rationale and Implementation
TR-91-031, International Computer Science Institute, Berkeley, Ca., 1991.

[Gehani86] N. H. Gehani, W. D. Roome
Concurrent C
Software-Pratice and Experience, Vol 16, 1157-1177, 1986, pp 821-844.

[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

[Geib93a] J.M. Geib, C. Gransart, C. Grenot
Distributed Objects in BOX
TOOLS Europe 93 Workshop on Distributed and Concurrent Objects, Versailles, France, March 8-11, 1993

[Geib93b] J.M. Geib, C. Gransart, C. Grenot
Mixing Objects and Activities in Complex Active Objects
ECOOP 93 Workshop on Object-Based Distributed Programming, Kaiserslautern, Germany,July 26-27 1993

[Geib94] J.M. Geib, C. Gransart, C. Grenot, P. Merle
Une *Introduction au langage BOX*
rapport technique ERA-150, LIFL, Université de Lille, Avril 1994

[Geib94b] J.M. Geib, C. Gransart, C. Grenot
Object-Oriented Distributed Programming with the Language BOX
IASTED 94, Annecy, France, May 18-20, 1994

[Geib94c] J.M. Geib, C. Gransart, C. Grenot
Programmation Parallèle à Objets avec le Langage BOX
La lettre du Transputer et des calculateurs parallèles, numéro spécial Les Langages à Objets, no 22, juin 1994.

[Geib94d] J.M. Geib, C. Gransart, C. Grenot
Survol d'un Langage à Objets pour Applications Parallèles : BOX
poster à RENPAR6, Lyon, France, June 8-10, 1994

[Geist93] A. Geist, J. Dongarra, R. Manchek
PVM 3 User's Guide and Reference Manual
Oak Ridge National Laboratory & University of Tennessee, May 1993.

[Giroux91] S. Giroux, A. Senteni
a Reflexive Version of Actalk
OOPSLA'91, Workshop on Reflection and Metalevel Architectures, Xerox PARC Technical Report, October 1991.

[Goldberg83] A. Goldberg, D. Robson
SMALTALK-80. The language and its implementation
Addison-Wesley, 1983

[Gransart91] C. Gransart
Introduction des objets actifs dans une extension d'Eiffel au parallélisme
Mémoire de DEA (1), LIFL, Septembre 1991

[Gransart91b] C. Gransart
FOCUS, Fragments of Objects in a ConcUrrent System
Mémoire de DEA (2), LIFL, Septembre 1991

[Gransart92] C. Gransart, J.M. Geib
Reusability and Concurrency in Parallel Eiffel
11ème International Eiffel User Conférence, 3 Avril 1992, Dortmund, Germany

[Gransart93] C. Gransart, C. Grenot, L. Courtrai
Le projet PVC/BOX: Environnement pour la Programmation Parallèle
Journées des Jeunes Chercheurs en Systèmes Informatiques Répartis, 14-16 Avril 1993, Grenoble, France, pp 131-136

[Grenot95] C. Grenot
Les interfaces Homme-Machine Distribuées Orientées Objets
PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, en préparation.

[Guerraoui94] B. Garbinato, R. Guerraoui, K.R. Mazouni
Distributed programming in GARF
R. Guerraoui, O. Niestrasz, M. Riveill, editors, Object Based Distributed Programming. Springer Verlag, 1994

[Guerraoui94b] B. Garbinato, X. Défago, R. Guerraoui, K.R. Mazouni
Abstractions pour la Programmation Concurrente dans GARF
La lettre du Transputer et des calculateurs parallèles, numéro spécial Les Langages à Objets, no 22, juin 1994.

[Haines94] M. Haines, D. Cronk, P. Mehrotra
On the Design of Chant : A Talking Threads Package
Institue for Computer Applications in Science and Engineering, NASA Langley Research Center, April 1994.

[Hemery94] F. Hemery
Etude de la répartition dynamique d'activités sur architectures décentralisées.
PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Juin 1994.

[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

[Hoare75] C.A.R. Hoare
Monitors: An operating system structuring concept
Communications ACM vol 18,2, 1975 ,pp 95.

[Hoare78] C. A. R. Hoare.
Communicating Sequential Processes
Communications ACM, vol. 21, 8, pp. 666-677, August 1978.

[HPF93] High Performance Fortran Forum
High Performance Fortran Language Specification, version 1.0
Rice University, Houston, TX, may 1993.

[INMOS88]
Occam2 Reference Manual
Prentice-Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs,1988.

[Jézéquel93] J.M. Jézéquel
Transparent parallelisation through reuse: between a compiler and a library approach
ECOOP 93, Kaiserslautern (D), July 1993

[Kale93] L.V. Kale, S. Krishnan
CHARM++ : A Portable Concurrent Object Oriented System Based On C++
Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications, September 1993.

[Kale93b] L.V. Kale
Parallel Programming with CHARM : An Overview
Department of Computer Science, University of Illinois, Urbana-Champaign, 1993.

[Kafura90] D. Kafura and D. Washabough,
"Incremental Garbage Collection of Concurrent Objects for Real-Time Applications"
11th Real Time System Symposium, pp 21-30, December 1990.

[Kafura91] D. Kafura and D. Washabough
"Progress in the Garbage Collection of Active Objects"
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

[Krasner88] G.E. Krasner, S.T. Pope
A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80
JOOP Journal of Object-Oriented Programming, August/ September 1988, pp. 26-49

[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

[Lim93] C.C. Lim
A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions
PhD, TR-93-063, ICSI, Berkeley, October 1993

[Makpangou91] M. Makpangou, Y. Gourhant, M. Shapiro
BOAR: A Library of Fragmented Object Types for Distributed Abstractions
IWOOOS'91 Proc. of the International Workshop on Object-Orientation in Operating Systems, Paolo Alto, CA (USA), 1991.

[Makpangou93] M. Makpangou, Y. Gourhant, J.P. Le Narzul, M. Shapiro
Fragmented Objects for Distributed Abstarctions
In T.L. Casavant and M. Singhal, editors, Readings in Distributed Computing Systems, IEEE Computer Society Press, July 1993.

[Maruyama94] K. Maruyama, N. Raguideau
Concurrent Object-Oriented Language "COOL"
ACM SIGPLAN Notices, volume 29, No 9, September 1994, pp 105-114.

[Mehrotra94] P. Mehrotra, M. Haines
An Overview of the OPUS Language and Runtime System
Institue for Computer Applications in Science and Engineering, NASA Langley Research Center, 1994.

[Merle96] P. Merle
Système d'objets pour le support d'applications coopératives
PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, en préparation.

[Mueller93] F. Mueller
A library implementation of POSIX threads under UNIX
Proceedings of the USENIX Conference, January 1993, pp. 29-41.

[Mueller93b] F. Mueller
Pthreads Library Interface
TR, Department of Computer Science, B-173 Florida Sate University, July 1993.

[Meyer88] B. Meyer
Object-oriented Software Construction
Prentice-Hall, 1988

[Meyer92] B. Meyer
Eiffel : The Language
Prentice-Hall, 1992

[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

[Murer93] S. Murer, J.A. Feldman, C.C. Lim, M.M. Seidel
pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation
TR-93-028, ICSI, Berkeley, December 1993

[Nguyen90] H. Nguyen Van, M. Riveill, C. Roisin
Manuel du langage Guide (V1.5)
Rapport Technique 3-90, Bull-IMAG, Décembre 1990.

[Olsson92] R.A. Olsson, G.R. Andrews, M.H. Coffin, G.M. Townsend
SR : A Language for Parallel and Distributed Programming
University of Arizona, TR 92-09, 1992.

[Omohundro91] S.M.Omohundro
The Sather Language
International Computer Science Institute, Juin 1991

[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

[Riveill94] M. Riveill
Guide : Un langage à objets pour la programmation concurrente
La lettre du Transputer et des calculateurs parallèles, numéro spécial Les Langages à Objets, no 22, juin 1994.

[Roos94] J.F. Roos
Mise au point d'applications distribuées pour environnement de développement basé sur une technologie objet
PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Février 1994.

[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. 305-37

[Shapiro89] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, C. Valot
SOS : An Object-Oriented Operating System - Assessment and Perspectives
Computing Systems, 2(7), 1989, pp. 287-337

[Shapiro90] M. Shapiro, Y. Gourhant
FOG/C++ : a Fragmented-Object Generator
Usenix C++ Workshop, San Francisco, 1990.

[Sun86] Sun Microsystems
News Preliminary Technical Overview
1986.

[Tanenbaum93] A.S. Tanenbaum
Modern Operating System
Prentice-Hall International Editions, 1993.

[Tanenbaum94] A. Tanenbaum
Systèmes d'exploitation - Systèmes centralisés, Systèmes distribués
InterEditions, 1994.

[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

Bibliographie commentée du projet PVC/BOX

Cette section donne une bibliographie commentée du projet. Nous présentons les résultats les plus significatifs sur PVC et BOX. Dans une première partie, nous présentons des travaux concernant une plate-forme d'évaluation du parallélisme avec le langage Eiffel.

Parallel Eiffel

[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

Cet article présente notre interface sous forme de classes pour introduire du parallélisme dans le langage Eiffel.

[Gransart91] C. Gransart

Introduction des objets actifs dans une extension d'Eiffel au parallélisme

Mémoire de DEA (1), LIFL, Septembre 1991

Dans ce travail, nous avons introduit la notion d'objet actif dans le langage Eiffel. Nous avons créé un mécanisme de conditions d'activation proches de celles définies par Caromel dans Eiffel //.

[Gransart91b] C. Gransart

FOCUS, Fragments of Objects in a ConcUrrent System

Mémoire de DEA (2), LIFL, Septembre 1991

Dans cette seconde partie de mémoire, j'ai évalué un mécanisme de fragmentation d'objet basé sur les liens d'héritage.

[Gransart92] C. Gransart, J.M. Geib

Reusability and Concurrency in Parallel Eiffel

11ème International Eiffel User Conférence, 3 Avril 1992, Dortmund, Germany

Cet article présente les résultats obtenus avec notre plate-forme Parallel Eiffel concernant les objets actifs.

PVC

[Courtrai92] 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

Cet article présente l'environnement PVC et le support d'exécution CAC.

[Courtrai92b] L. Courtrai

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

PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Octobre 1992.

Cette thèse présente les différents systèmes à objets pour l'écriture d'applications distribuées. Elle décrit également notre système PVC et la notion de Composant Actif de Communication.

[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

Cet article présente le mécanisme de réexécution de programmes parallèles CAC.

[Hemery94] F. Hemery

Etude de la répartition dynamique d'activités sur architectures décentralisées.

PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Juin 1994.

Cette thèse présente le mécanisme d'équilibrage de charge implanté dans PVC ainsi qu'un simulateur d'aide au placement statique. Cette thèse montre qu'il est possible d'utiliser des techniques comme le recuit simulé pour aider au placement.

[Hemery94b] F. Hemery

Distribution de code pour architectures décentralisées

RENPAR6, Lyon, France, June 8-10, 1994

Cet article présente une méthode de placement statique de programmes répartis en utilisant un algorithme basé sur le recuit simulé.

[Roos94] J.F. Roos

Mise au point d'applications distribuées pour environnement de développement basé sur une technologie objet

PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Février 1994.

Cette thèse présente le problème de mise au point d'applications distribuées ainsi qu'un mécanisme de réexécution pour des applications CAC et BOX.

BOX

[Geib93a] J.M. Geib, C. Gransart, C. Grenot
Distributed Objects in BOX
TOOLS Europe 93 Workshop on Distributed and Concurrent Objects, Versailles, France, March 8-11, 1993
Cet article fait une comparaison des différents modèles de conception pour applications distribuées à objets. Nous y présentons notre modèle et brièvement le langage BOX.

[Geib93b] J.M. Geib, C. Gransart, C. Grenot
Mixing Objects and Activities in Complex Active Objects
ECOOP 93 Workshop on Object-Based Distributed Programming, Kaiserslautern, Germany,July 26-27 1993
Cet article présente le modèle et le langage BOX ainsi que notre solution pour la répartition de classes en modules.

[Gransart93] C. Gransart, C. Grenot, L. Courtrai
Le projet PVC/BOX: Environnement pour la Programmation Parallèle
Journées des Jeunes Chercheurs en Systèmes Informatiques Répartis, 14-16 Avril 1993, Grenoble, France, pp 131-136
Survol et présentation du projet PVC/BOX.

[Geib94] J.M. Geib, C. Gransart, C. Grenot, P. Merle
Une *Introduction au langage BOX*
rapport technique ERA-150, LIFL, Université de Lille, Avril 1994
Manuel de programmation du langage BOX et présentation des outils de l'environnement de développement.

[Geib94b] J.M. Geib, C. Gransart, C. Grenot
Object-Oriented Distributed Programming with the Language BOX
IASTED 94, Annecy, France, May 18-20, 1994
Dans ce papier, nous présentons le mécanisme de localisation des instances.

[Geib94c] J.M. Geib, C. Gransart, C. Grenot
Programmation Parallèle à Objets avec le Langage BOX
La lettre du Transputer et des calculateurs parallèles, numéro spécial Les Langages à Objets, no 22, juin 1994.
Dans cette publication, nous présentons notre modèle dual de conception d'applications réparties basé sur la fusion du modèle des objets et du modèle des processus communicants.

[Geib94d] J.M. Geib, C. Gransart, C. Grenot
Survol d'un Langage à Objets pour Applications Parallèles : BOX
poster à RENPAR6, Lyon, France, June 8-10, 1994
Présentation du langage BOX.