Partage et migration de l'information dans un système réparti à objets

Dominique Decouchant

▶ To cite this version:

Dominique Decouchant. Partage et migration de l'information dans un système réparti à objets. Modélisation et simulation. Université Joseph-Fourier - Grenoble I, 1987. Français. NNT : . tel-00324431

HAL Id: tel-00324431
https://theses.hal.science/tel-00324431

Submitted on 25 Sep 2008

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

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

THESE

Présentée à

L'UNIVERSITE SCIENTIFIQUE, TECHNOLOGIQUE ET MEDICALE DE GRENOBLE

pour obtenir le grade de
docteur de l'Université Scientifique,
Technologique et Médicale de Grenoble
(Spécialité : Informatique)

par

DECOUCHANT Dominique

OOOOO

PARTAGE ET MIGRATION DE L'INFORMATION DANS UN SYSTEME REPARTI A OBJETS

OOOOO

Thèse soutenue le 25 Juin 1987 devant la commission d'examen.

J. MOSSIERE	**Président**
J.P. BANATRE **R. BALTER**	**Rapporteurs**
S. KRAKOWIAK	**Directeur de thèse**

UNIVERSITE SCIENTIFIQUE TECHNOLOGIQUE ET MEDICALE DE GRENOBLE

Président de l'Université :
M. TANCHE

Année Universitaire 1986 - 1987

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ère Classe

ARNAUD Paul	Chimie Organique
ARVIEU ROBERT	Physique Nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S
AURIAULT Jean-Louis	Mécanique
AYANT Yves	Physique Approfondie
BARBIER Marie-Jeanne	Electrochimie
BARBIER Jean-Claude	Physique Expérimentale CNRS
BARJON Robert	Physique Nucléaire ISN
BARNOUD Fernand	Boichimie Macromoléculaire Végétale
BARRA Jean-René	Statistiques-Mathématiques Appliquées
BELORISKY Elie	Physique C.E.N.G- D.R.F.
BENZAKEN Claude	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean-Paul	Mathématiques Pures
BILLET Jean	Géographie
BOELHER Jean-Paul	Mécanique
BONNIER Jane Marie	Chimie Générale
BOUCHEZ Robert	Physique Nucléaire ISN
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CAUQUIS Georges	Chimie Organique`
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean-Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DELOBEL Claude	Mathématiques Appliquées
DEPORTES Charles	Chimie Minérale
DESRE Pierre	Electrochimie
DOLIQUE Jean-Michel	Physique des Plasmas
DOUCE Rolland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FONTAINE Jean-Marc	Mathématiques Pures
GAGNAIRE Didier	Chimie Physique
GERMAIN Jean-Pierre	Mécanique,
GIRAUD Pierre	Géologie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques Pures
KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées
KUPKA Yvon	Mathématiques Pures
LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre-Jean	Mathématiques Appliquées
DE LEIRIS Joel	Biologie

LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences Nucléaires I.S.N.
MACHE Régis	Physiologie Végétale
MAYNARD Roger	Physique du Solide
MICHEL Robert	Minéralogie et Pétrographie (Géologie)
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIAUX Jacques	Géologie
PERRIER Guy	Géophysique
PIERRARD Jean-Marie	Mécanique
PIERRE Jean-Louis	Chimie Organique
RASSAT André	Chimie Systématique
RENARD Michel	Thermodynamique
RINAUDO Marguerite	Chimie CERMAV
ROSSI André	Biologie
SAKAROVITCH Michel	Mathématiques Appliquées
SAXOD Raimard	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
VALENTIN Jacques	Physique Nucléaire I.S.N.
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ADIBA Michel	Mathématiques Pures
ANTOINE Pierre	Géologie
ARMAND Gilbert	Géographie
BARET Paul	Chimie
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BLANCHI J.Pierre	STAPS
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BRUANDET J.François	Physique
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHARDON Michel	Géographie
CHIARAMELLA Yves	Mathématiques Appliquées
COURT Jean	Chimie
DEMAILLY Jean-Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPASSEL Roger	Mécanique des Fluides
DERRIEN Jacques	Physique
DUFREYNOY Alain	Mathématiques Pures
GASPARD François	Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GUIGO Maryse	Géographie
GUMUCHAIN Hervé	Géographie
GUITTON Jacques	Chimie
HACQUES Gérard	Mathématiques Appliquées

HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
JARDON Pierre	Chimie
JOSELEAU Jean-Paul	Biochimie
KERCKHOVE Claude	Géologie
LEBRETON Alain	Mathématiques Appliquées
LONGEQUEUE Nicole	Sciences Nucléaires I.S.N.
LUCAS Robert	Physique
LUNA Domingo	Mathématiques Pures
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
MASCLE Georges	Géologie
NEMOZ Alain	Thermodynamique CNRS - CRTBT
OUDET Bruno	Mathématiques Appliquées
PELMONT Jean	Biochimie
PERRIN Claude	Sciences Nucléaires I.S.N.
PFISTER Jean-Claude	Physique du Solide
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
RIEDIMANN Christine	Mathématiques Pures
ROBERT Gilles	Mathématiques Pures
ROBERT Jean-Bernard	Chimie Physique
SARROT-REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SOUTIF Jeanne	Physique
SCHOLL Pierre-Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L' IUT 1

PROFESSEURS de 1ere Classe

BUISSON Roger	Physique IUT 1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1

PROFESSEURS de 2ème classe

BOUTHINON Michel	EEA. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHEHIKIAN Alain	EEA. IUT 1
CHENAVAS Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
GOSSE Jean-Pierre	EEA.IUT 1
GROS Yves	Physique IUT 1
KUHN Gérard, (Détaché)	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	EEA.IUT 1
NOUGARET Marcel	Automatique IUT 1
PEFFEN René	Métallurgie IUT 1
PERARD Jacques	EEA. IUT 1
PERRAUD Robert	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
VINCENDON Marc	Chimie IUT 1

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS CLASSE EXEPTIONNELLE ET 1ère CLASSE

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puericulture	C.H.R.G.
BEZEZ Henri	Orthopédie-Traumatologie	Hopital SUD
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté La Merci
	Chirurgie Générale et Digestive	C.H.R.G.
BUTEL Jean	Orthopédie-Traumatologie	C.H.R.G.
CHAMPETIER Jean	Anatomie-Topographique et Appliquée	C.H.R.G.
CHARACHON Robert	O.R.L.	C.H.R.G.
COUDERC Pierre	Anatomie-Pathologique	C.H.R.G.
DELORMAS Pierre	Pneumophtisiologique	C.H.R.G.
DENIS Bernard	Cardiologie	C.H.R.G.
GAVEND Michel	Pharmacologie	Faculté La Merci
HOLLARD Daniel	Hématologie	C.H.R.G.
LATREILLE René	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
LE NOC Pierre	Bactériologie-Virologie	C.H.R.G.
MALINAS Yves	Gynécologie et Qbstétrique	C.H.R.G.
MALLION Jean-Michel	Médecine du Travail	C.H.R.G.
MICOUD Max	Clinique Médicale et Maladies Infectieuses	C.H.R.G.
MOURIQUAND Claude	Histologie	Faculté La Merci
PARAMELLE Bernard	Pneumologie	C.H.R.G.
PERRET Jean	Neurologie	C.H.R.G.
RACHAIL Michel	Hépato-Gastro-Entérologie	C.H.R.G.
DE ROUGEMONT Jacques	Neurochirurgie	C.H.R.G.
SARRAZIN Roger	Clinique Chirurgicale	C.H.R.G.
STIEGLITZ Paul	Anestésiologie	C.H.R.G.
TANCHE Maurice	Physiologie	Faculté La Merci
VERAIN André	Biophysique	Faculté La Merci
VIGNAIS Pierre	Biochimie	Faculté La Merci

PROFESSEURS 2ème CLASSE

BACHELOT Yvan	Endocrinologie	C.H.R.G.
BARGE Michel	Neurochirurgie	C.H.R.G.
BENABID Alim Louis	Biophysique	Faculté La Merci
BENSA Jean-Claude	Immunologie	Hopital Sud
BERNARD Pierre	Gynécologie-Obstétrique	C.H.R.G.
BESSARD Germain	Pharmacologie	ABIDJAN
BOLLA Michel	Radiothérapie	C.H.R.G.
BOST Michel	Pédiatrie	C.H.R.G.
BOUCHARLAT Jacques	Psychiatrie Adultes	Hopital Sud
BRAMBILLA Christian	Pneumologie	C.H.R.G.
CHAMBAZ Edmond	Biochimie	C.H.R.G.
CHIROSSEL Jean-Paul	Anatomie-Neurochirurgie	C.H.R.G.
COLOMB Maurice	Immunologie	Hopital Sud
COMET Michel	Biophysique	Faculté La Merci
CONTAMIN Charles	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Néphrologie	C.H.R.G.
COULOMB Max	Radiologie	C.H.R.G.
CROUZET Guy	Radiologie	C.H.R.G.
DEBRU Jean-Luc	Médecine Interne et Toxicologie	C.H.R.G.
DEMONGEOT Jacques	Biostatistiques et Informatique Médicale	Faculté La Merci

DUPRE Alain	Chirurgie Générale	C.H.R.G.
DYON Jean-François	Chirurgie Infantile	C.H.R.G.
ETERRADOSSI Jacqueline	Physiologie	Faculté La Merci
FAURE Claude	Anatomie et Organogénèse	C.H.R.G.
FAURE Gilbert	Urologie	C.H.R.G.
FOURNET Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
FRANCO Alain	Médecine Interne	C.H.R.G.
GIRARDET Pierre	Anesthésiologie	C.H.R.G.
GUIDICELLI Henri	Chirurgie Générale et Vasculaire	C.H.R.G.
GUIGNIER Michel	Thérapeutique et Réanimation Médicale	C.H.R.G.
HADJIAN Arthur	Biochimie	Faculté La Merci
HALIMI Serge	Endocrinologie et Maladies Métaboliques	C.H.R.G.
HOSTEIN Jean	Hépato-Gastro-Entérologie	C.H.R.G.
HUGONOT Robert	Médecine Interne	C.H.R.G.
JALBERT Pierre	Histologie-Cytogénétique	C.H.R.G.
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirurgie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christian	Médecine Interne	C.H.R.G.
MOUILLON Michel	Ophtalmologie	C.H.R.G.
PELLAT Jacques	Neurologie	C.H.R.G.
PHELIP Xavier	Rhumatologie	C.H.R.G.
RACINET Claude	Gynécologie	C.H.R.G.
RAMBAUD Pierre	Pédiatrie	C.H.R.G.
RAPHAEL Bernard	Stomatologie	C.H.R.G.
SCHAERER René	Cancérologie	C.H.R.G.
SEIGNEURIN Jean-Marie	Bactériologie-Virologie	Faculté La Merci
SELE Bernard	Cytogénétique	Faculté La Merci
SOTTO Jean-Jacques	Hématologie	C.H.R.G.
STOEBNER Pierre	Anatomie Pathologique	C.H.R.G.
VROUSOS Constantin	Radiothérapie	C.H.R.G.

MEMBRES DU CORPS ENSEIGNANT PHARMACIE

AGNIUS-DELORD Claudine	Physique	Faculté La Tronche
ALARY Josette	Chimie Analytique	Faculté La Tronche
BERIEL Hélène	Physiologie et Pharmacologie	Faculté La Tronche
BOUCHERLE André	Chimie et Toxicologie	Faculté Meylan
CUSSAC Max	Chimie Therapeutique	Faculté La Tronche
DEMENGE Pierre	Pharmacodynamie	Faculté La Tronche
JEANNIN Charles	Pharmacie Galénique	Faculté Meylan
LATURAZE Jean	Biochimie	Faculté La Tronche
LUU DUC Cuong	Chimie Générale	Faculté La Tronche
MARIOTTE Anne-Marie	Pharmacognosie	Faculté La Tronche
MARZIN Daniel	Toxicologie	Faculté Meylan
RENAUDET Jacqueline	Bactériologie	Faculté La Tronche
ROCHAT Jacques	Hygiéne et Hydrologie	Faculté La Tronche
SEIGLE-MURANDI Françoise	Botanique et Cryptogamie	Faculté Meylan
VERAIN Alice	Pharmacie Galénique	Faculté Meylan

je voudrais remercier,

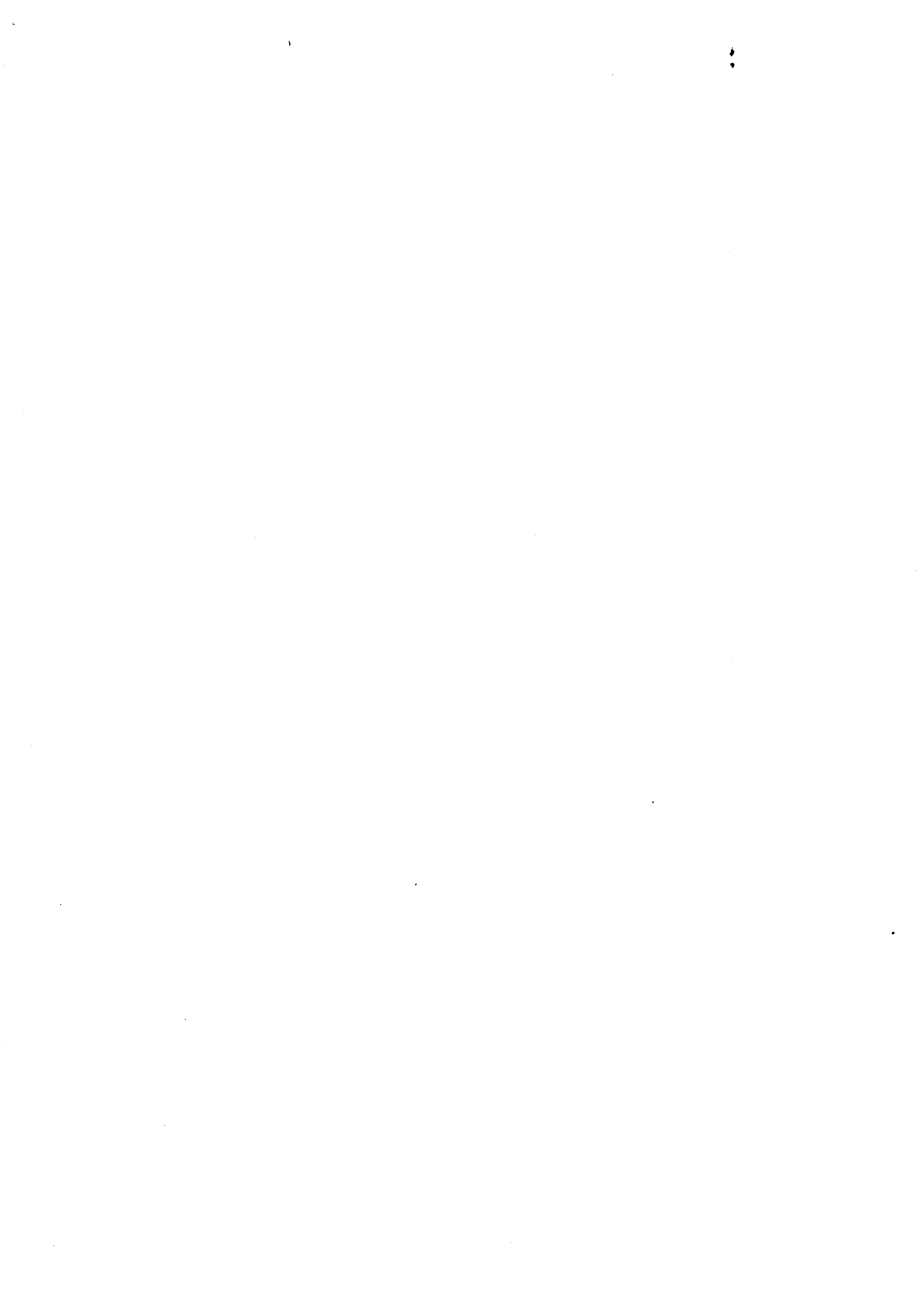
Monsieur Jacques Mossière, directeur du Laboratoire de Génie Informatique, pour m'avoir fait confiance en m'acceuillant au sein du laboratoire qu'il dirige, et pour avoir accepté de présider le jury de cette thèse,

Monsieur Jean-Pierre Banâtre, professeur à l'Université de Rennes, pour les critiques pertinentes qui ont permis d'élaborer cette thèse, et pour avoir accepté de faire partie du jury,

Monsieur Roland Balter, directeur du Centre de Recherches Bull de Grenoble, pour les conseils qu'il a apporté à la rédaction de ce document,

Monsieur Sacha Krakowiak, professeur à l'Université de Grenoble, pour m'avoir accepté dans son équipe et constamment soutenu dans mes recherches.

Je tiens de plus à remercier toutes les personnes du Laboratoire de Génie Informatique et du Centre de Recherche Bull, pour l'aide qu'elles ont pu me fournir et pour leur constante sympathie.



I. INTRODUCTION.

1. Motivations.

La chute du coût des composants électroniques (processeurs, mémoires, etc...) a facilité le développement d'ordinateurs individuels de grande puissance. Ainsi le traditionnel terminal, qui permettait de se connecter à un système en temps partagé, a fait place à un micro-ordinateur. L'avantage acquis réside dans le gain d'autonomie et la garantie de disponibilité des ressources locales telles qu'unité de traitement, mémoire de masse, ou imprimante. Cependant, cette mutation de l'environnement de travail présente certains inconvénients:

- les ressources de ces micro-ordinateurs sont limitées, ce qui interdit de développer de grandes applications qui nécessitent des besoins en taille de mémoire centrale et en temps d'exécution plus importants que ceux disponibles localement,
- la mémoire secondaire d'un micro-ordinateur a une capacité limitée et les risques de dégradation de son contenu ne sont pas négligeables,
- les systèmes en temps partagé offraient la possibilité aux usagers de communiquer et de partager des données. La communication et le partage sont rendus plus difficiles par la décentralisation des environnements de travail.

La recherche de solutions pour combler ces lacunes a débouché sur la définition de systèmes répartis composés de postes de travail individuels et de serveurs connectés à un réseau de communication à grand débit. Chaque usager élabore la majeure partie de son travail sur son poste individuel, et fait ponctuellement appel aux services offerts par les serveurs spécialisés. Ces derniers fournissent par exemple un stockage fiable de l'information (gestion de versions, sauvegarde, ...) et une puissance de calcul élevée. Ils peuvent également posséder des périphériques tels que des disques de grande capacité ou des imprimantes, trop coûteux pour que chaque poste en soit équipé. Le réseau permet également la communication entre usagers.

Cette évolution de l'environnement de travail a été rendue possible par le développement de nouveaux supports de communication rapides (cable coaxial et fibre optique), qui ont permis de construire des systèmes de communication fiables et efficaces.

L'évolution des méthodes de conception de logiciel conduit d'autre part, à une description indépendante des services offerts (connus par leurs interfaces) et de leurs réalisations. C'est par exemple le cas de NFS (Network File System) [Sun84a], développé par Sun qui permet d'accéder sous Unix à un système de fichiers distant comme s'il était local. C'est la même idée qui a conduit la définition du système DOMAIN [Nelson83], par Apollo: des postes de travail indépendants partagent le système de mémoire virtuelle, ce qui permet à des groupes de partager des programmes, des données et des terminaux virtuels. Dans le cas du partage d'une structure de données, le service offert est l'accès à cette structure de données à l'aide d'un nom symbolique, et la réalisation est sa représentation en mémoire physique, dont la localisation peut éventuellement changer au cours du temps. C'est cet aspect d'encapsulation et de masquage des données qui rend intéressant l'usage des méthodes de programmation fondées sur les objets pour la conception des systèmes répartis.

La programmation par objets est un modèle de conception où chaque entité manipulée est autonome ("objet") et utilise des requêtes ("messages") pour interagir avec d'autres objets. Un mécanisme de définition et de création des objets permet de créer des modèles d'objets ("classe") décrivant la structure des objets et leurs comportements lors de l'arrivée de requêtes. La programmation par objets consiste donc à regrouper sous une description structurelle et comportementale commune des entités, actives ou passives, partageant des propriétés communes. La granularité de ce regroupement peut être très variée: elle peut être très fine comme dans Smalltalk où les entités de base telles que les entiers sont des objets, ou être plus grossière, les objets étant alors analogues à des modules. Un mécanisme supplémentaire permet de définir une classe d'objets par la récupération de la définition d'une ou de plusieurs autres classes ("héritage").

Dans un système à objets, une erreur de programmation se traduit toujours par le fait qu'une requête adressée à un objet ne peut pas être prise en compte. On connaît donc immédiatement l'objet source de l'erreur. La modularité assurée dans les langages et systèmes à objets permet donc un prototypage aisé et un développement incrémental des applications.

Il est important de remarquer que le champ d'application de la programmation par les objets s'étend progressivement à des domaines de recherches très variés tels que l'intelligence artificielle [Bobrow83, Fikes85], l'interface homme-machine [Coutaz87, Hullot86, Sibert86], ou la conception de bases de données [CCA83, Albano85, Mylopoulos80, Copeland84].

Le modèle de structuration en objets peut apporter des éléments de solution aux problèmes des systèmes répartis par la modularité qu'il introduit dans la conception d'applications, et par l'homogénéité de la manipulation d'entités fonctionnellement différentes. A l'époque (1984) où ce travail a été entrepris, nous disposions de peu d'expérience sur les systèmes répartis à objets. Pour cette raison, deux choix étaient possibles pour tester et valider les idées developpées dans cette thèse: soit étendre à un environnement réparti un modèle existant de système à objets, soit concevoir

complètement un nouveau système. Compte tenu du temps qui était alloué à ces recherches, il n'était pas raisonnable de vouloir développer un nouveau système. Il convient de noter que ce travail ne s'inscrivait pas dans le cadre d'un projet, mais qu'il s'agissait d'une pré-étude destinée à préparer un projet à plus long terme.

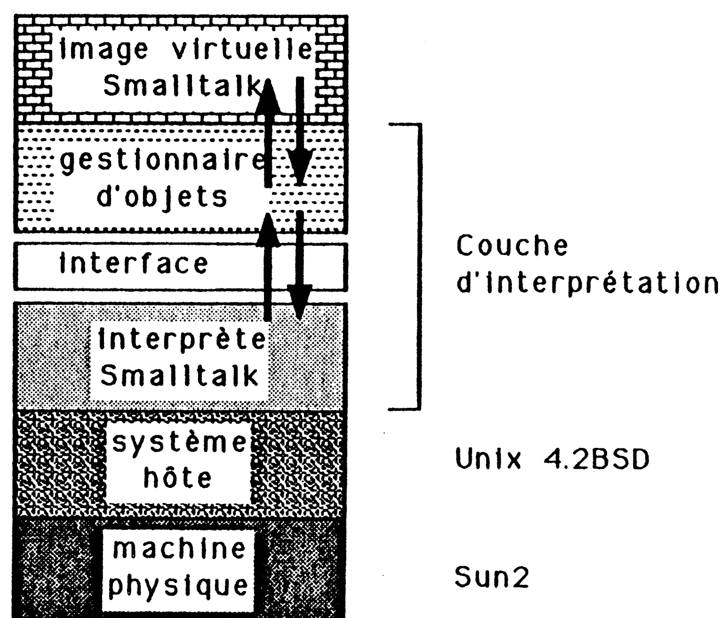
Cette étude est donc partie d'un système existant (Smalltalk-80). Deux éléments ont guidé ce choix: d'une part Smalltalk-80 peut être considéré comme un système "modèle" pour la mise en œuvre de la notion d'objet, et d'autre part nous disposions du texte source d'une réalisation particulière de Smalltalk-80 développée à l'Université de Berkeley (USA) sur une station de travail Sun2. Nous avons étudié sur ce système les divers problèmes que posaient la répartition et le partage des objets.

L'expérience acquise dans ce travail pourra être réinvestie dans le projet Guide qui vient de démarrer au sein du Laboratoire de Génie Informatique en collaboration avec le Centre de Recherches Bull de Grenoble. Ce projet a pour but la définition d'un langage et d'un système pour la conception et le développement d'applications réparties, et doit utiliser un modèle fondé sur les objets.

2. Définition des objectifs du travail.

Nous nous proposons d'étudier l'adaptation du modèle des objets à l'expression de la répartition en prenant comme support un réseau de systèmes Smalltalk répartis. Le système Smalltalk a été conçu comme un système mono-utilisateur, où l'utilisateur peut interagir non seulement avec les objets qu'il a construits dans le système, mais aussi avec le système lui-même, qui est décrit de la même façon en termes d'objets. L'utilisateur peut donc modifier le système, l'enrichir et éventuellement l'adapter à ses besoins.

Smalltalk-80 est un système défini au dessus d'une machine virtuelle, qui a pour fonctions l'interprétation des instructions Smalltalk et la gestion des objets. La couche de gestion des objets offre une interface d'accès aux objets composée d'un ensemble de primitives.



Nous désirons concevoir un système réparti, extension du système centralisé existant, comme le regroupement de plusieurs stations de travail Smalltalk-80 individuelles sur un réseau local. Pour rester compatible avec le système existant, chaque site se présente à l'utilisateur avec la même interface. Cependant, celle-ci est complétée avec des fonctions permettant à l'utilisateur d'un site de partager des ressources avec des utilisateurs d'autres sites.

Toute entité du système Smalltalk étant un objet, ce partage s'exprime tout naturellement en terme d'objets. Le problème ainsi défini peut se résumer à la définition d'un gestionnaire d'objets distribué permettant à plusieurs sites de partager des objets. Ce partage est rendu possible par la définition de nouvelles primitives au niveau de l'interface du gestionnaire d'objets avec la couche d'interprétation. Ce gestionnaire d'objets doit assurer la protection entre les divers sites, effectuer l'accès aux objets distants, permettre la migration d'objets et assurer que les manipulations des objets restent correctes. D'autres tâches moins visibles à l'utilisateur doivent également être assurées telles que la récupération des ressources (mémoire et noms) des objets du système réparti ainsi constitué. Ce dernier point est déjà un problème difficile en milieu centralisé et cette difficulté est augmentée par la répartition.

3. Travail réalisé.

Le travail exposé dans cette thèse est le résultat de deux phases de réflexion successives. La première phase consistait à définir un modèle de mémoire virtuelle à objets adapté à la structure du système Smalltalk-80. Dans un deuxième temps, le modèle ainsi défini a été étendu pour permettre le partage et la migration d'information entre plusieurs systèmes Smalltalk indépendants.

La définition d'une mémoire virtuelle pour le système Smalltalk-80 a été présentée dans [Decouchant84]. L'objet étant l'unité élémentaire du système, cette mémoire virtuelle est analogue à une mémoire segmentée constituée des objets composant le système. Nous essayons d'adapter au mieux la gestion de cette mémoire aux caractéristiques inhérentes aux systèmes à objets. L'unité d'allocation et d'échange entre les niveaux de mémoire a donc été choisie, tout naturellement, comme étant l'objet. La gestion de ceux-ci peut tout à fait s'assimiler à la gestion de segments dans un système traditionnel, car un objet est par définition une entité de taille variable. L'aspect nouveau de cette gestion réside dans ses caractéristiques quantitatives. En effet, dans les systèmes traditionnels à mémoire segmentée, les segments gérés sont en quantité limitée et leurs tailles sont importantes (plusieurs centaines ou milliers d'octets). A l'inverse, dans le système Smalltalk, les entités manipulées sont en nombre très important et leur taille est petite (parfois quelques octets).

Cette multiplicité des entités manipulables par une activité influe directement sur les choix à faire pour réaliser la mémoire virtuelle des objets: le choix de l'algorithme d'allocation par zones et de l'algorithme de recomposition des zones libres est primordial,

car les systèmes à objets émiettent la mémoire gérée en engendrant avec une forte fréquence des demandes d'allocations de zones. De plus, dans ces systèmes, la destruction des objets récupérables est laissée à la charge du système, l'utilisateur n'indiquant pas les objets qu'il n'utilise plus. Le système gère, pour chaque objet, un compteur d'utilisation dont la nullité indique que la place occupée par l'objet est récupérable. Cependant, cela ne permet pas de repérer tous les objets inutilisés (c'est par exemple le cas des paires d'objets qui se référencent mutuellement sans qu'aucun des deux ne soit accessible à partir d'autres objets).

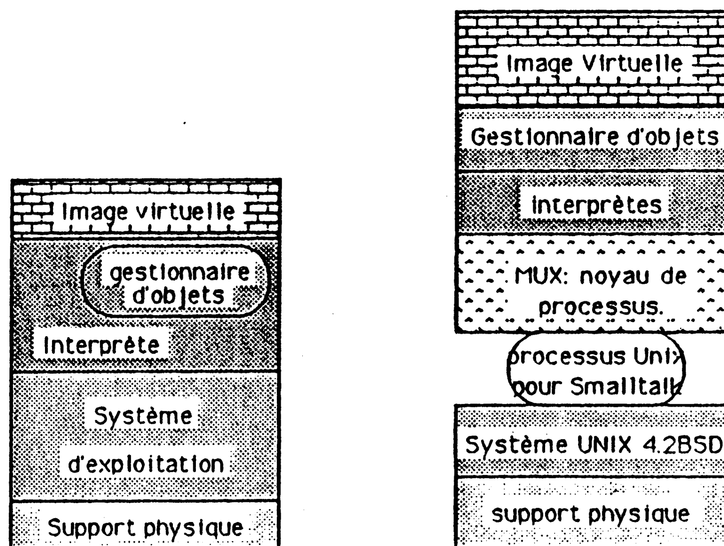
La mémoire virtuelle des objets est composée de deux gestionnaires gérant respectivement les objets en mémoire centrale et en mémoire de stockage. L'accès aux objets reste conforme à l'interface initiale du système Smalltalk-80. Tous les accès aux objets s'effectuent à l'aide de primitives définies dans l'interface qu'offre la mémoire des objets à la couche d'interprétation. Chaque nouvelle primitive d'accès a été parenthésée par deux fonctions qui résolvent les éventuels défauts d'objet et synchronisent les accès concurrents. La solution développée présente un mécanisme simple de gestion des objets qui se contente de résoudre les fautes sur les objets; les transferts d'objets en mémoire secondaire sont déclenchés uniquement lorsqu'il n'y a plus de place en mémoire centrale pour loger un nouvel objet. Ceci laisse une entière liberté sur le choix de futures politiques de transferts préventifs d'objets en mémoire secondaire, qui peuvent être élaborées d'après des mesures statistiques de l'évolution du système.

La deuxième partie de ce travail a consisté à définir la gestion des objets en milieu distribué [Decouchant86]. La conception de la mémoire répartie des objets est fondée sur l'extension des mécanismes mis en œuvre pour réaliser la mémoire virtuelle des objets en milieu centralisé. Pour cette raison, la mémoire répartie des objets compte un gestionnaire de plus : le gestionnaire du réseau. Ce gestionnaire est unique sur chaque site, et constitue le seul interlocuteur pour l'accès à des objets extérieurs au site ou pour toute manipulation d'objet inter-sites : migration d'objet, etc... Pour conserver l'homogénéité et la transparence des accès aux objets, un nouveau type d'objet a été défini : le représentant. L'introduction des objets-représentants peut être vue comme une transposition aux systèmes à objets du schéma de l'appel de procédure à distance. Un site qui doit accéder à un objet non local, doit posséder un représentant de cet objet. Un objet représentant se conporte pour son utilisateur comme l'objet réel, mais il n'est qu'une voie d'accès à cet objet. Tout accès à un objet représentant se traduit par un accès distant à l'objet réel qu'il représente. Les représentants sont en outre utilisés pour la gestion du partage des objets entre différents sites Smalltalk.

Pour gérer ces objets représentants, de nouveaux mécanismes ont été developpés. Des primitives ont été ajoutées dans l'interface de la mémoire des objets permettant de donner à un autre site l'accès à un objet et de faire migrer un objet vers un autre site. A partir de l'ajout de ces deux nouvelles fonctions, un certain nombre de problèmes sont apparus:

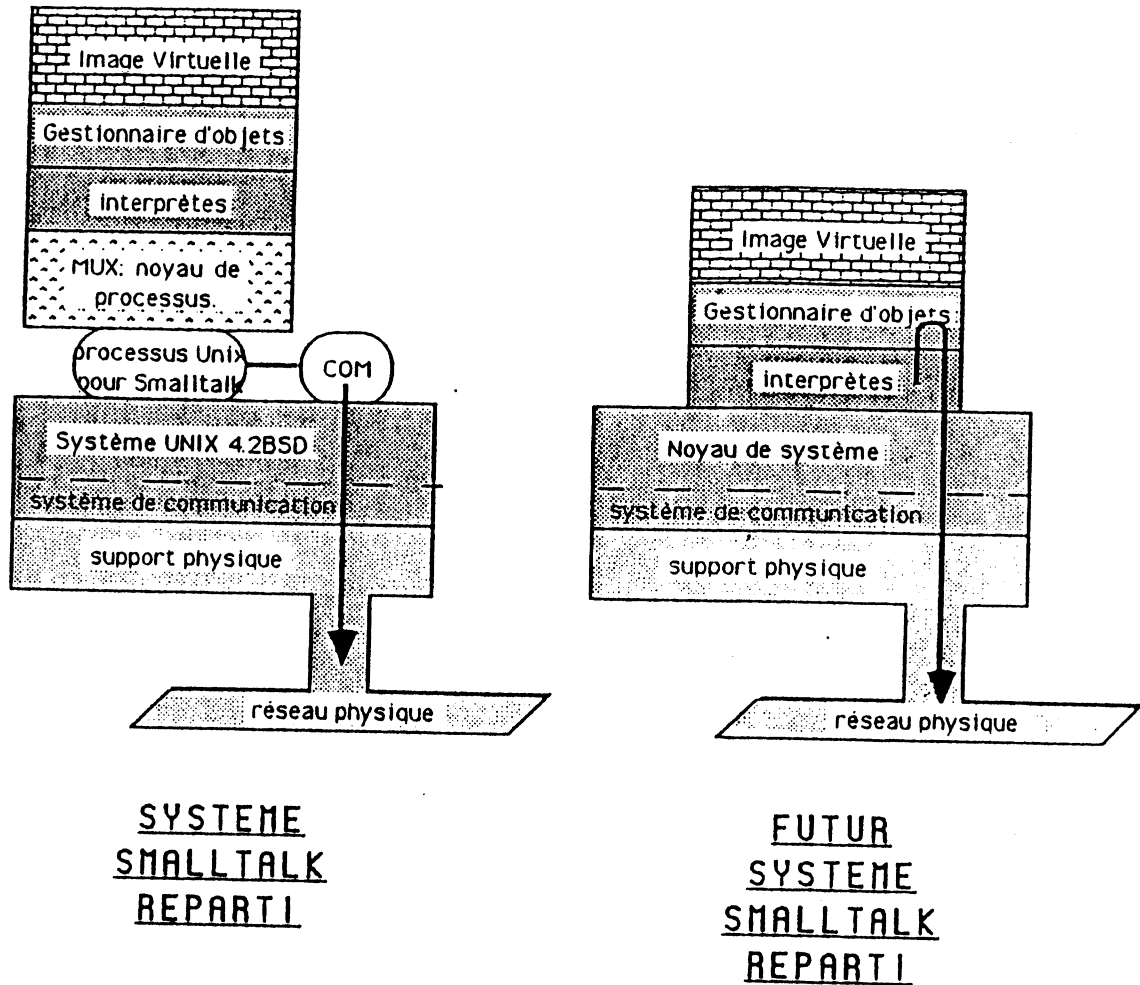
- la possibilité de faire migrer des objets a posé le problème de la gestion des objets-représentants et du contrôle de leur unicité sur chaque site,
- le partage d'objets a induit les problèmes liés à la protection et à la mise en place du contrôle des accès,
- le problème de la récupération de la mémoire des objets a été amplifié par la répartition des références inter-objets. Un circuit peut maintenant être réparti entre plusieurs sites. La résolution de ce problème ne constituait pas un objectif majeur de cette étude, mais une nécessité imposée par le système Smalltalk. Il était donc nécessaire d'apporter un solution minimale à ce problème. Une proposition d'algorithme réparti de récupération d'objets par marquage a donc été développée. Cet algorithme est une adaptation trés simple de l'algorithme défini en milieu centralisé; il est très coûteux car il monopolise tous les sites pendant son déroulement. Comme nous le verrons, d'autres algorithmes plus adaptés et plus efficaces pourraient être envisagés.
- enfin, comme nous sommes en présence d'un réseau de stations individuelles et autonomes, il convient de prendre en compte les problèmes de connexion et de déconnexion dynamiques des stations.

Pour chaque problème rencontré, nous nous sommes efforcé d'apporter une solution, qui mette bien en évidence les services et les protocoles qui doivent être offerts par la machine virtuelle, sur laquelle sont installés les systèmes Smalltalk. L'expérience acquise au cours de la construction du prototype au dessus d'Unix, nous a permis de proposer des améliorations au mécanisme de la mémoire d'objets répartie, qui pourraient être mises en œuvre dans une future réalisation sur une machine nue. La figure ci-après illustre la progression de la conception:



SYSTEME SMALLTALK BERKELEY

SYSTEME SMALLTALK CENTRALISE



La définition initiale du système Smalltalk-80 intègre les fonctions d'interprétation et de gestion des objets au sein d'une unique couche de système: la machine virtuelle. La version centralisée du système Smalltalk-80, développée dans ce travail, définit séparément les deux fonctions. C'est à partir de cette version, que la fonction de gestion des objets est développée pour permettre le partage d'objets entre les systèmes Smalltalk-80.

4. Plan de la thèse.

Le chapitre II présente le modèle général de programmation par objets. Cette présentation est illustrée par deux langages: Simula67 et Smalltalk-80. Simula67 est le premier langage a avoir introduit les concepts de base de la programmation par objets en étendant le langage Algol60. Le système Smalltalk-80 est un langage conçu dès le départ autour du modèle des objets. La notion d'objet y est complètement intégrée, ce qui en fait un système et un langage très homogènes. Smalltalk-80 est le support d'implémentation des idées développées dans cette thèse. Pour chacun de ces langages, nous présentons le modèle des objets tel qu'il a été developpé, les choix et les aspects novateurs.

Le chapitre III présente les problèmes de la mise en œuvre d'un modèle d'objets dans un système réparti. L'étude de trois systèmes expérimentaux existants (Eden, Emerald et SOS) sert de support à cette présentation. Chacun des problèmes inhérents aux systèmes répartis est replacé par rapport à ces systèmes.

Le chapitre IV propose une solution pour gérer en milieu centralisé la mémoire des objets du système Smalltalk-80, dans laquelle l'unité d'information échangée entre les deux niveaux de mémoire physique est l'objet. L'accès aux objets est assuré par un jeu de primitives composant l'interface de la mémoire virtuelle. Cette interface comprend des primitives de lecture et d'écriture, mais aussi des primitives de création d'objet et d'échange des valeurs de deux objets. Cette dernière primitive permet entre autres de réaliser les objets de taille variable, puisque la mémoire virtuelle ne le permet pas.

Le chapitre V étend le modèle du gestionnaire centralisé au milieu réparti. Cette extension, basée sur l'ajout et la gestion d'objets particuliers (les "représentants"), présente les divers problèmes induits par la répartition ainsi que leurs solutions. Cette proposition pour gérer des objets en milieu réparti reste conforme à la conception du gestionnaire centralisé malgré l'introduction de ces nouveaux objets artificiels: l'interface entre les couches d'interprétation et de gestion des objets n'est pas modifiée.

Le développement d'une première réalisation du gestionnaire des objets est présenté dans le chapitre VI. La structuration de cette réalisation, ainsi que les moyens mis en œuvre, dépendent fortement des limitations imposées par le système hôte (Unix 4.2BSD).

La conclusion de cette thèse présente les enseignements acquis au cours de cette étude. Une synthèse des apports et des problèmes introduits par chaque aspect nouveau est développée.

II. LE MODELE DES OBJETS ET SA MISE EN OEUVRE.

1. Le modèle général de la programmation par objets.

Avant tout, il convient de présenter en détail les concepts sur lesquels reposent les définitions des langages et systèmes à objets. Ces concepts y apparaissent avec de multiples variantes, chaque langage dénommant par un terme propre un concept commun [Stefik85]. Cette étude ayant comme support le système Smalltalk-80, sa terminologie constituera notre vocabulaire de référence. Certaines notions non présentes dans Smalltalk (ex: héritage multiple) sont replacées par rapport à la terminologie que nous présentons dans cette section.

objet:

composant élémentaire de tout système. Chaque objet se caractérise par un état et par un ensemble de procédures ("méthodes") permettant de consulter et de modifier cet état. L'état d'un objet est constitué d'un ensemble de données typées.

classe:

modèle qui décrit la structure et le comportement communs à un ensemble d'objets ("exemplaires") créés à partir de ce modèle.

exemplaire:

objet créé d'après le modèle décrit par sa classe.
Tout objet est un exemplaire d'une classe (et d'une seule).

message:

c'est le moyen de communication entre les objets. C'est une requête adressée à un objet, qui provoque l'exécution d'une procédure de son destinataire. Bien que ce terme puisse faire penser à une exécution parallèle des objets expéditeur et destinataire de la requête, dans Smalltalk-80 (et la plupart des langages à objets) le traitement d'un message est synchrone (appel de procédure). Cependant certains langages définissent un modèle asynchrone de traitement des messages.

méthode:

procédure d'accès à un objet, qui décrit la séquence des actions à entreprendre à la réception d'un message.

La correspondance "message → méthode" peut être déterminée statiquement (à la compilation) ou dynamiquement (par l'objet destinataire, lors de la réception du message). Smalltalk-80 réalise une correspondance dynamique.

sous-classe:

Une sous-classe est une classe dont la structure des exemplaires et les méthodes sont héritées de celle d'une autre classe. Certains langages permettent l'héritage multiple, et dans ce cas une classe peut être sous-classe de plusieurs classes. Au modèle d'exemplaires ainsi hérité, il peut être apporté des modifications de la structure des exemplaires et des procédures.

super-classe:

C'est la (resp. une) classe dont une sous-classe est dérivée suivant la relation d'héritage simple (resp. multiple).

métaclasse:

Cette notion a été introduite dans les langages qui considèrent les classes comme des objets. Puisque tout objet est un exemplaire d'une classe, chaque classe doit être un exemplaire d'une autre classe. Une telle classe est alors appelée "métaclasse". Cette définition est le résultat du désir d'homogénéiser le modèle des objets et le traitement des différentes entités du système.

Les concepts fondamentaux de la programmation par objets résident dans le principe de décomposition hiérarchique, le principe d'abstraction des données et dans la relation d'héritage [Booch86].

Le principe de décomposition consiste à découper une application en modules ou parties fonctionnellement indépendantes, et à définir leurs interactions. Les langages à objets constituent un progrès par rapport aux langages modulaires (ex: Modula-2) qui permettent de décomposer une application en modules, mais non de créer dynamiquement des modules.

Le principe d'abstraction des données consiste à décrire le comportement d'un objet par l'interface qu'il présente au reste du système. Cette interface se résume à un ensemble de méthodes par lesquelles on agit sur l'objet, et elle est totalement indépendante de la représentation interne de l'objet. Ici encore, les langages à objets sont plus souples que les langages modulaires grâce à la possibilité d'association dynamique entre message et méthode, qui permettent notamment la coexistence de réalisations multiples d'une même interface.

La relation d'héritage permet de définir une classe d'objets par une spécialisation d'une autre classe. Cette classe est alors une sous-classe de la classe spécialisée. Une sous-classe hérite de l'interface d'accès de sa super-classe, ainsi que de la définition de l'implémentation de l'état de ses exemplaires. Une sous-classe peut ajouter ou surcharger

des méthodes à l'interface qu'elle a héritée. De même, elle peut compléter l'implémentation héritée de l'état de ses exemplaires.

D'autres concepts moins fondamentaux pour le modèle mais néammoins très riches ont été introduits dans la programmation par objets tels que la généricité, qui permet de paramétrer un module, une procédure, un type ou une classe avec un autre type. La généricité peut être obtenue à partir de la propriété d'héritage, l'inverse n'est pas possible de manière simple [Meyer86].

Il convient de noter que beaucoup d'idées et de concepts de la programmation par objets et des langages et systèmes à base d'objets, proviennent du langage Simula67 [Dahl66, Dahl70]. En effet, Simula67 fut le premier langage à developper les notions de classe et d'exemplaires, car il était destiné à faire de la simulation c'est à dire à créer des classes ou des types d'objets représentant des entités réelles telles que des bateaux, des machines, etc... partageant une structure et un comportement communs. L'idée de représenter d'autre entités en termes de classes et d'exemplaires n'était pas encore apparue.

Ces notions ont été reprises et généralisées par le SCG (Software Concepts Group) de Xerox à Palo Alto lors de la définition du premier langage Smalltalk [Goldberg76]. Le système peut alors être vu comme un ensemble d'objets interagissant les uns avec les autres par des requêtes (messages). L'aspect interface-utilisateur de la programmation par objets s'est développé par la définition du système Smalltalk-80 [Goldberg83, Goldberg84, Krasner83]. Celui-ci intègre l'affichage et la manipulation des images et des textes comme des interactions entre des objets, ce qui rend le système plus homogène. La définition du système LOOPS [Bobrow83], a succédé à la définition de la dernière version de Smalltalk. Par rapport à ce dernier, LOOPS introduit un héritage multiple dans la hiérarchie des classes et une programmation avec le langage Interlisp [Xerox83].

Le système Smalltalk-80 utilise les objets comme unique entité de structuration, alors que d'autres systèmes introduisent des entités hétérogènes. Cette hétérogénéité provient souvent du fait que la notion d'objet a été rajoutée au dessus d'un langage ou d'un système existant. Par exemple, c'est le cas des langages C++ [Stroustrup86] et Objective-C [Cox86], qui sont des extensions du langage C [Kernighan78] au modèle des objets. De même on peut citer Ceyx [Hullot85], qui est une extension du langage Lisp pour la manipulation d'objets.

Ce chapitre présente quelques langages et systèmes à base d'objets. Tout d'abord nous présentons le langage Simula67 qui est le précurseur de l'ensemble des langages et des systèmes à base d'objets. Puis nous présentons Smalltalk-80 qui est le support de référence et de travail de cette thèse.

2. Le langage Simula67.

2.1. Introduction.

Le langage Simula a été développé, comme une extension du langage Algol60, à partir de l'année 1963 au "Norwegian Computing Center" (Oslo, Norvège). La dernière version du langage (Simula 67) introduit la notion de classe. Ce langage a été avant tout conçu pour permettre le développement des systèmes de simulation à événements discrets. Le but pousuivi était alors de pouvoir décrire le comportement et l'évolution du système en termes d'entités dont les valeurs pouvaient être modifiées par des actions associées à des événements datés. Cela revient donc à regrouper des actions et des comportements en *classes* et de décrire chaque classe par un ensemble unique d'instructions. Un souci majeur était d'offrir la possibilité au programmeur de concevoir son application de façon très modulaire, où chaque composant de programme est développé indépendamment des autres, avec très peu d'interactions entre eux.

2.2. La structure de bloc.

Cette notion est centrale dans le langage Simula. La raison principale pour laquelle Simula a été défini à partir du langage Algol, est que ce dernier offrait une structure de bloc similaire à celle envisagée dans la définition du langage.

La structure de blocs permet de concevoir une application de façon très modulaire en encapsulant un ensemble d'instructions et de données. Les blocs peuvent être nommés et dans ce cas, ils sont plus communément appelés des procédures.

Mais, ce qui est plus important est que cette structure peut posséder des données locales et éventuellement d'autres structures de blocs (procédures ou non) qui lui sont privées.

Le langage permet de faire la distinction entre un bloc composé d'un ensemble d'instructions, et des exemplaires d'exécution de ce même bloc. Dans le langage Algol60, ces exemplaires sont gérés en pile ("dernier entré, premier sorti"). Simula67 se distingue par le fait qu'un exemplaire de bloc peut survivre au bloc qui a initialisé son exécution. Il peut aussi survivre à la terminaison de l'application. Cette extension rend nécessaire la récupération des blocs inutilisés. Cette récupération utilise une procédure de marquage de ces blocs.

La notion de bloc ainsi introduite permet au programmeur de définir de nouvelles structures de contrôle qui autorisent une programmation plus souple. Le langage offre des structures de contrôle de base (Procédure, SiAlorsSinon, Tantque) qui peuvent être enrichies par la création d'activités parallèles pouvant se synchroniser par des primitives.

2.3. Les classes.

Les classes de Simula sont définies comme des procédures capable d'engendrer des exemplaires de bloc dont la durée de vie est indépendante de leurs appels. Ces blocs sont appelés des objets. En Simula, il y a donc une distinction entre les classes et les objets.

Les variables et les procédures locales à la classe constituent un modèle pour l'état et les méthodes d'accès aux futurs objets créés à partir de celle-ci.

La création d'un objet à partir d'une classe s'effectue par un appel à celle-ci par l'intermédiaire d'un opérateur de création (new). La syntaxe de création d'un objet est la suivante: *unObjet :- new uneClasse(paramètre_1, ..., paramètre_N).* L'opérateur de création "new" est chargé de réserver l'espace en mémoire pour l'objet créé, puis d'exécuter sur cet objet le corps principal de la procédure "uneClasse" qui doit contenir les instructions d'initialisation. De plus, le moyen d'accès à l'objet créé doit être mémorisé dans une variable de type *référence*, sinon l'objet est inaccessible et sera récupéré par le système.

Chaque objet peut être modifié directement par la notation pointée: si "unObjet" désigne un objet de la classe "uneClasse" et que "unChamp" soit le nom d'une variable locale de cette classe, alors le champ "unChamp" de cet objet peut être directement manipulé par une expression du type *unObjet.unChamp*. Cela ressemble tout à fait au mode d'accès à une variable de type "structure" dans les langages de haut niveau actuels. Cette possibilité est contestable car elle permet de manipuler l'objet en court-circuitant l'ensemble des procédures par lesquelles on est censé interagir avec l'objet. Il faut noter que les procédures de la classe peuvent être également appelées à l'aide de cette notation (exemple: *unObjet.uneFonction(paramètre_1, ..., paramètre_N);*). Cette dernière façon d'interagir avec l'objet est la seule utilisée dans les langages et les systèmes à objets actuels (Smalltalk, C++, Emerald, Eden, ...).

Il est important de noter que Simula effectue un contrôle strict sur les passages de paramètres et sur les affectations. Ainsi pour que l'expression de création précédemment exposée soit correcte, il est nécessaire que la variable "unObjet" ait été déclaré par l'expression: *ref (uneClasse) unObjet;*. Toutes les variables sont donc typées à l'aide des types prédéfinis (ex: integer x;) ou à l'aide du qualificatif *ref* qui permet d'exprimer que la variable pointe vers un objet d'un nouveau type (classe). Le choix d'effectuer le contrôle des types à la compilation a été fait pour augmenter l'efficacité de l'exécution et pour restreindre les problèmes des erreurs à l'exécution. La seule erreur (liée à la manipulation des classes et des objets) qui peut arriver lors de l'exécution est le fait qu'on tente d'accéder à un objet désigné par une variable de type "référence" qui ne désigne aucun objet (sa valeur est alors égale à *none*).

Pour illustrer et conclure cette section, nous pouvons par exemple définir une classe "pile" permettant de créer des piles d'entiers de tailles variables.

```
class pile(n); integer n;
begin
      integer borne_max;
      integer sommet;
      integer table[n];

      procedure empiler(valeur); integer valeur;
      begin
            if sommet < borne_max then
            begin
                  sommet := sommet + 1;
                  table[sommet] := valeur;
            end
            else erreur_pile_pleine();
      end

      integer procedure depiler();
      begin
            if sommet > 0 then
            begin
                  sommet := sommet - 1;
                  depiler := table[sommet+1];
            end
            else erreur_pile_vide();
      end


      borne_max := n - 1;
      sommet := 0;

end of pile;
```

La classe "pile" permet de créer des objets dont la taille maximale est fixée à la création (appel de la procédure "pile"). L'instruction *unePile :- new pile(10);* alloue un espace en mémoire composé de 2 entiers (*borne_max* et *sommet*) et d'un tableau de 10 entiers (*table*). Cet appel exécute également le corps principal de la classe "pile". Celui-ci contient les instructions d'initialisation de chaque nouvel exemplaire. Ces instructions initialisent la pile à l'état "vide" et fixent la borne supérieure de la pile pour le contrôle des empilements de valeurs. Les deux procédures *empiler* et *depiler* peuvent agir sur les objets de type "pile" de n'importe quelle taille. Remarquons que les expressions du type *unePile.sommet := 0* sont tout à fait permises par le langage. Ceci enfreint les règles de la programmation par les objets telle qu'elles sont actuellement définies: l'ensemble des procédures d'une classe est l'unique voie pour accéder à un objet.

2.4. Les coroutines.

Simula67 inclut un noyau de coroutines qui permet de commuter le contrôle d'exécution entre plusieurs objets de diverses classes, à l'aide de primitives de type "resume". La communication entre les différentes coroutines s'effectue par l'intermédiaire de variables globales au programme ou locales à une coroutine (dans ce cas les autres coroutines doivent accéder à la variable locale par accès direct ou par des procédures).

Lors de l'exécution, l'appel à une coroutine s'effectue à l'aide de la primitive *call* qui permet d'arrêter l'exécution et d'effectuer un branchement à l'instruction suivante de la coroutine. Au premier appel, le branchement s'effectue naturellement à la première instruction de la coroutine. La coroutine peut rendre le contrôle à l'appelant par la primitive *detach*. Par la suite, l'appelant et l'appelé vont s'échanger le contrôle par des successions d'appels *call/detach*. Il est à remarquer que ce mécanisme n'est pas symétrique puisqu'il fait intervenir un objet maître et un objet esclave.

Un mécanisme plus symétrique est offert entre deux objets esclaves. Ces deux objets peuvent par exemple s'échanger plusieurs fois le contrôle à l'aide d'une succession d'appels *resume/resume* avant que l'un d'eux ne repasse le contrôle à l'objet maître par un appel *detach*.

Cette mécanique est à rapprocher de celle introduite dans le système Smalltalk-80 (voir section suivante). Celui-ci met en place un noyau de processus (en termes d'objets) pour exprimer des applications parallèles. L'ajout de tels mécanismes est nécessaire lorsque les objets manipulés par les langages ne sont pas réellements actifs (l'appel d'une procédure sur un objet transfère le contrôle à celui-ci).

2.5. La structure d'héritage.

L'héritage de Simula ou "concaténation" permet de créer une classe à partir d'une autre classe en la spécialisant. La classe ainsi créée (sous-classe) hérite des paramètres d'appel et de leurs spécifications, de la structure des futurs objets, des procédures et des instructions d'initialisation. La spécialisation consiste à enrichir cette structure de classe héritée par l'ajout de nouveaux paramètres, de nouveaux attributs pour les futurs objets, de nouvelles procédures et de nouvelles instructions d'initialisation.

Si B est une classe créée par héritage de la classe A, et que ces deux classes sont définies comme suit:

```
class A(pA_1,pA_2,...,pA_n); < spécifications des pA >;
begin
        < attributs des objets de A >

        < procédures de A >

        < instructions d'initialisation de A >

end of A;
```

```
A class B(pB_1,pB_2,...,pB_n); < spécifications des pB >;
begin
        < attributs des objets de B >

        < procédures de B >

        < instructions d'initialisation de B >

end of B;
```

Avec l'héritage que l'on vient de définir, la déclaration de la classe B est équivalente à celle définie ci-après.

```
class B(pA_1,pA_2,...,pA_n,pB_1,pB_2,...,pB_n); < spécifications des pA et des pB >;
begin
        < attributs des objets de A >
        < attributs des objets de B >

        < procédures de A >
        < procédures de B >

        < instructions d'initialisation de A >
        < instructions d'initialisation de B >

end of B;
```

Si *p* désigne une référence sur un objet de classe A, alors l'affectation:
p :- *<objet de classe B, héritée de A>* est licite.

2.6. Conclusion.

Tous les concepts de la programmation par les objets sont déjà présents. Cela est d'autant plus remarquable que la version de Simula introduisant ces notions de classes, d'objets, de blocs, etc... est apparue en 1967. A l'époque, ces concepts étaient surtout exploités pour la simulation, et leur intérêt pour la programmation structurée de systèmes et d'applications avait été peu perçu. C'est seulement en 1972 qu'est apparue la première version du langage Smalltalk qui reprenait ces idées. Par la suite, ce système a évolué jusqu'à la version Smalltalk-80 qui est présentée dans la section suivante.

Par rapport à Simula, Smalltalk introduit un modèle uniforme, où toutes les entités, y compris les classes, sont des objets. De plus, les liaisons aux procédures sont devenues dynamiques. La structure d'héritage s'est affinée par l'introduction d'opérateurs qui permettent de déplacer les recherches des procédures à l'exécution. Le contrôle des paramètres a ainsi été reporté à l'exécution. D'autres langages ont essayé d'introduire l'héritage multiple [Bobrow83], une vérification des types des objets plus élaborée et plus fine que la simple égalité des types (conformité) [Black86], des objets réellement actifs dans lesquels on peut avoir du parallélisme [Almes85], etc ...

3. Le langage et le système Smalltalk-80.

3.1. Introduction.

Le langage Smalltalk a été développé à partir de 1970 au centre de recherches de XEROX à Palo Alto (Palo Alto Research Center) [Goldberg76]. Les concepteurs avaient pour but de créer un outil susceptible d'être utilisé aussi bien pour enseigner la programmation que pour développer des activités éducatives. Petit à petit les buts ont évolué : en 1976, D.H.Ingalls présentait le système "Smalltalk-76" [Ingalls78] comme un système simple et performant permettant à quiconque d'avoir un accès et un contrôle créatif sur des informations aussi variées que des nombres, des textes, des sons et des images. Depuis 1981, les recherches se sont réorientées jusqu'au point de devenir plus conceptuelles qu'éducatives. Le langage Smalltalk-80 est le noyau d'une entité informatique qui comprend un système d'exploitation et des utilitaires très évolués dont la souplesse facilite le travail de l'utilisateur. L'utilisateur conçoit et implémente donc son application de façon incrémentale: toute nouvelle définition de classes et de méthodes peut être immédiatement testée. Chaque entité de programme ajoutée au système est conservée (c.a.d. considérée comme persistante) jusqu'à ce que l'utilisateur la remette à jour ou la détruise.

Le support matériel du système Smalltalk-80 nécéssite un ordinateur possédant une vitesse d'exécution et une taille de mémoire élevées (plusieurs méga-octets). Le poste de travail du système Smalltalk est composé d'un écran graphique de type "bitmap" de très haute résolution d'environ 1000 sur 800 points, et une souris comme moyen de désignation. Cette dernière permet d'accéder aux différentes fenêtres de l'écran (sélectionner, désélectionner, cacher, mettre en premier plan, ...) et de sélectionner l'information texte par l'intermédiaire de menus (couper, coller, détruire, ...). L'utilisation du multifenêtrage permet de poursuivre plusieurs activités en alternance. L'activité en cours s'applique au contenu d'une fenêtre non recouverte par d'autres fenêtres. On peut comparer cette technique de travail à la disposition de feuilles sur un bureau: la feuille sur laquelle l'utilisateur travaille se trouve au dessus des autres feuilles. Le multifenêtrage permet le feuilletage du livre constitué par le système Smalltalk.

3.2. Objets et classes.

3.2.1. Les classes et leurs exemplaires.

Une classe est la description d'un ensemble particulier d'objets conformes à une description commune.

Chaque classe définit un ensemble de méthodes par lesquelles ses exemplaires peuvent modifier leur état. Chacune de ses méthodes peut engendrer des envois de messages vers d'autres objets. Il convient de rappeler que l'envoi d'un message est une opération synchrone, et qu'il n'y a donc aucun parallélisme entre l'expéditeur et le destinataire d'un message. A chaque modèle de message acceptable par ses exemplaires, la classe fait correspondre la version compilée de la méthode.

La composition de l'état de chaque exemplaire est décrite dans sa classe par des variables d'état. Les méthodes qui agissent sur un objet peuvent accéder aux variables d'état de cet objet et à des variables de portée supérieure:

- chaque classe peut déclarer des variables, dites de classe, accessibles par tous ses exemplaires. Ceci constitue un changement par rapport à Simula67 où une classe d'objets peut être locale à une autre classe.

- des ensemble de variables ("pool") partagées par plusieurs classes peuvent être déclarées. Par exemple, l'objet-table contenant les codes ASCII des objets qui sont des caractères, est partagé par l'ensemble des classes définissant des objets de type texte. Ceci n'existe pas dans Simula67 car il est possible de déclarer des classes locales à une autre classe, pour que ces dernières puissent accéder aux variables locales de la classe englobante. La structure de "pool" de Smalltalk est plus propre, car il est inutile de créer une classe qui englobe plusieurs autres classes, pour que ces dernières puissent partager des variables.

- un ensemble nommé "Smalltalk" contient toutes les variables globales au système. Les classes du système Smalltalk appartiennent automatiquement à cet ensemble. L'utilisateur peut ajouter des variables autres que les classes. Ces variables deviennent alors persistantes au niveau du système.

L'exemple ci-après illustre les différentes sortes de variables disponibles dans Smalltalk. Cet exemple est le source de la définition de la classe "FileDirectory".

```
Object subclass: #FileDirectory
    instanceVariableNames: 'directoryName closed '
    classVariableNames: 'ExternalReferences '
    poolDictionaries: 'FilePool '
    category: 'Files-Abstract'!
```

```
FileDirectory comment: 'A FileDirectory is uniquely identified by the device
or server that it refers to. A FileDirectory is a collection of Files. It may
also be found in some other dictionary or FileDirectory, though often this is
implicit'!


!FileDirectory methodsFor: 'testing'!

includes: aFile
    "Answer whether aFile is in the receiver's list of files."

    ↑(self find: aFile) notNil'
includesKey: aFileName
    "Answer whether a file whose name is aFileName is included in the receiver."
    | file |
    file ← self find: (self initFileName: aFileName).
    file notNil ifTrue:
        [file release. "close the IFS connection, if an IFS leaf file"
        ↑true].
    ↑false! !

    ...

!FileDirectory methodsFor: 'file status'!

open
    "Open the directory."

    closed ← false.
    (ExternalReferences includes: self)
        ifFalse: [ExternalReferences addLast: self]!

    ...
```

La classe "FileDirectory" décrit des objets dont l'état est composé de deux champs: un champ nommé "directoryName" qui contient le nom symbolique de chaque objet-catalogue, et un champ "closed" qui est une variable booléenne indiquant si l'objet-catalogue est fermé ou non.

Tous les exemplaires de cette classe partagent une variable nommée "ExternalReferences" qui mémorise l'ensemble des objets-catalogue qui sont ouverts. Par exemple, si on regarde la méthode nommée "open", celle-ci ouvre le catalogue (l'objet désigné par "self" qui est le destinataire du message). Cette ouverture revient à placer le nom du catalogue dans la liste "ExternalReferences" (instruction "ExternalReferences addLast: self").

La variable "FilePool" est accessible par les exemplaires de la classe "FileDirectory" et par les exemplaires d'autres classes. Par exemple, "FilePool" est accessible par les méthodes de la classe "File", dont nous donnons uniquement la partie spécifiant l'état des exemplaires.

```
Object subclass: #File
	instanceVariableNames: 'fileDirectory fileName pageCache serialNumber
			lastPageNumber binary readWrite error '
	classVariableNames: ''
	poolDictionaries: 'FilePool '
	category: 'Files-Abstract'!

.	...
```

Un petit sous-ensemble de méthodes du système Smalltalk-80 n'est pas exprimé dans le langage: les méthodes-primitives. Les méthodes-primitives font partie de la machine virtuelle et ne peuvent pas être modifiées par le programmeur. Parmi celles-ci on trouve les primitives qui permettent de réaliser les entrées/sorties physiques au niveau de la machine. Les primitives sont invoquées par des messages comme les autres méthodes.

3.2.2. Sous-classe / super-classe.

Smalltalk utilise une relation d'héritage simple: une classe ne peut hériter que d'une seule autre classe. Seule la classe "Object", qui est la racine du système Smalltalk, ne possède pas de super-classe.

Une sous-classe peut définir de nouvelles variables et de nouvelles méthodes. Si de nouvelles variables sont ajoutées dans la sous-classe, l'état de ses exemplaires est composé des variables d'exemplaires de sa super-classe auquelles sont rajoutées les nouvelles variables. L'exemple ci-après définit la classe "AltoFileDirectory" comme une sous-classe de la classe "FileDirectory".

```
FileDirectory subclass: #AltoFileDirectory
	instanceVariableNames: 'dirFile bitsFile diskPages totalPages nSectors '
	classVariableNames: ''
	poolDictionaries: 'AltoFilePool '
	category: 'Files-Xerox Alto'!

AltoFileDirectory comment:
'A concrete example of a FileDirectory class. Implements the Xerox Alto File
System. See the Alto Operating System Reference Manual section about Disks.

Note that the Alto file address format is
	bits
	0-3  sector number (0 - 8r15, i.e., 12 or 14 sectors)
	4-12 cylinder number (0 - 8r312, Model 31; 0-8r625, Model 44)
	13  head number (0-1)
	14  disk number (0-1)
	15  restore bit.

Instance Variables:
	dirFile  <File> or nil
	bitsFile  <File> or nil
```

```
        diskPages <Integer>
        totalPages <Integer>
        nSectors  <Integer>
        '!


!AltoFileDirectory methodsFor: 'adding'!

addNew: aFile
        | sn page |
        sn ← self allocateSN: aFile.
        aFile serialNumber: sn.

        "allocate a new page (more success after O.S. stuff, bittable etc.)"
        self allocate: (page ← aFile initPageNumber: 0) after: 800.

        "write 0th -- leader, in the process filling it in and then creating first page".
        page reinitialize.
        page serialNumber: sn.
        page size: page dataSize.
        aFile leader: page address.
        aFile readWrite: Write.
        aFile updateLeader: page.
        self addEntry: aFile.
        ↑aFile! !

        ...
```

Toute variable ajoutée dans la définition du modèle des exemplaires d'une classe doit être déclarée avec un nom différent de toutes les variables déclarées dans la super-classe (pas de surcharge sur les variables).

Par contre, une sous-classe peut définir une méthode dont le nom est le même que celui d'une méthode de sa superclasse (ou de l'une des classes qui lui sont "supérieures" hiérarchiquement par la relation de superclasse).

Toute classe du système Smalltalk appartient à une arborescence de classes dont la racine est la classe "Object" et telle qu'une classe peut avoir plusieurs sous-classes mais une seule super-classe.

3.2.3. Définition d'une super-classe abstraite.

Une super-classe abstraite est créée lorsque deux classes partagent une partie de leurs descriptions et qu'aucune des deux n'est véritablement une sous-classe de l'autre. Une classe est créée et contient les aspects communs aux deux classes. Les deux classes sont alors des sous-classes de cette classe abstraite. Cette super-classe est dite abstraite car elle n'est pas créée pour avoir des exemplaires.

```
SequenceableCollection subclass: #ArrayedCollection
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Collections-Abstract'!

ArrayedCollection comment:
'I am an abstract collection of elements with integers as external keys.  I
re-implement several messages inherited from SequenceableCollection in order
to obtain some performance gains.

            Subclasses are
        Array       elements are pointers
        ByteArray   elements are bytes
        RunArray    elements are typically runs of the same thing
        String      elements are characters
        Text        elements are characters with style
        WordArray   elements are words

        '!

!ArrayedCollection methodsFor: 'accessing'!

size
        "Answer the number of indexable fields in the receiver. This value is the
        same as the largest legal subscript. Primitive is specified here to override

        SequenceableCollection size. Essential. See Object documentation
        whatIsAPrimitive. "

        <primitive: 62>
        ↑self basicSize! !

        ...
```

Dans cet exemple, la classe abstraite se nomme "ArrayedCollection". Cette classe ne déclare pas de variables d'exemplaires. Elle admet comme sous-classes les classes "Array", "ByteArray", etc..., qui spéficient individuellement l'état de leurs exemplaires. Cette classe abstraite contient les méthodes communes à ses sous-classes (exemple: la méthode "size").

3.2.4. Les métaclasses.

Comme nous l'avons dit précédemment, Smalltalk étend le concept d'objet à l'ensemble des classes. La classe d'une classe est appelée sa métaclasse. La métaclasse d'une classe "X" ne possède pas de nom et est désignée par l'expression "X class".

L'arborescence des métaclasses est calquée sur l'arborescence des classes qui ne sont pas des métaclasses. L'arborescence des métaclasses a pour racine la classe "Object class".

Par définition, toute classe autre que la classe "Object" doit être une sous-classe d'une autre classe. Cela est atteint en greffant l'arborescence des métaclasses de racine "Object class" sous la classe "Class". La classe "Class" est une super-classe abstraite pour toutes les métaclasses et décrit la nature générale des classes. Chaque métaclasse ajoute la conduite spécifique à son unique exemplaire.

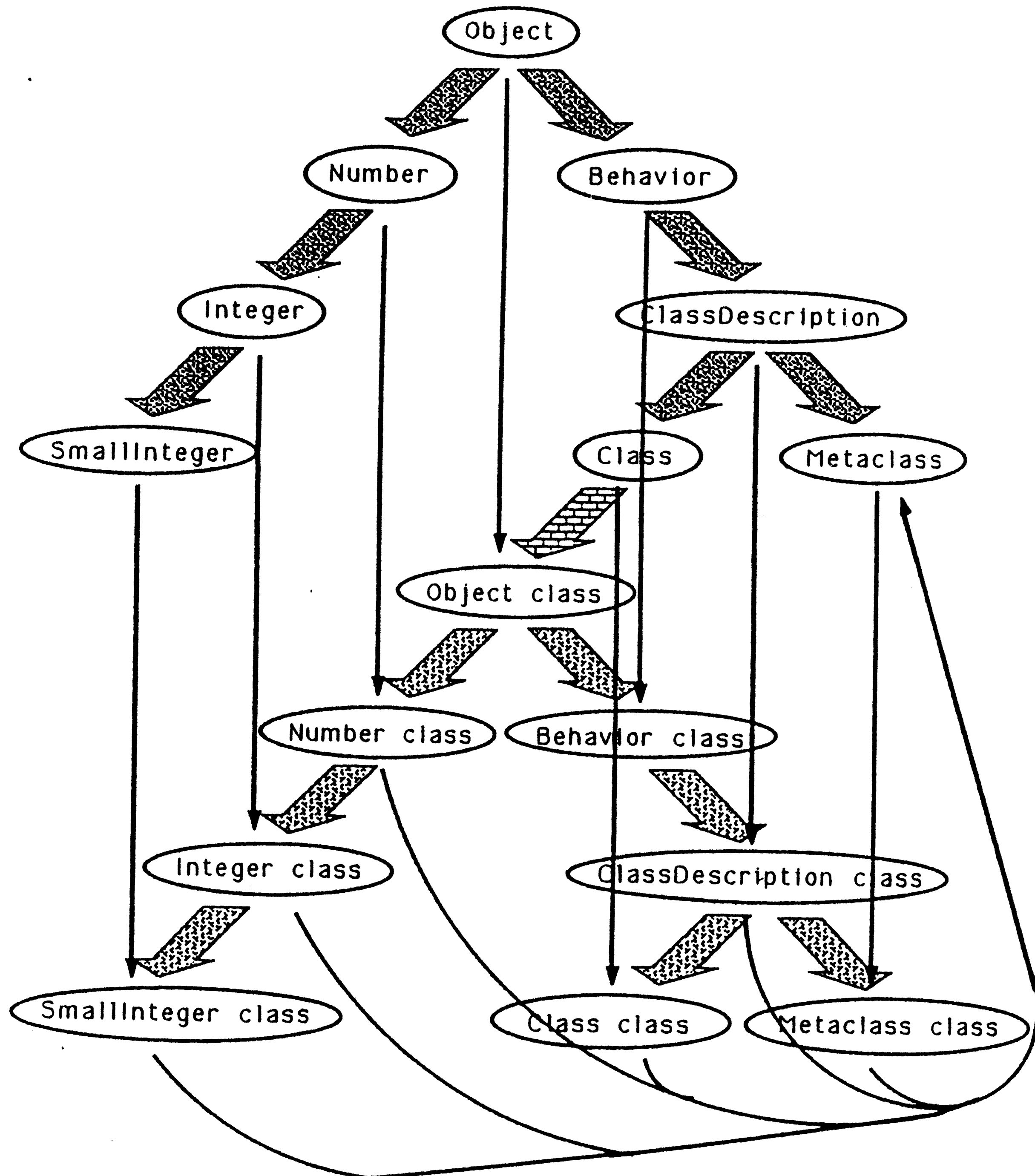
Les métaclasses peuvent redéfinir les méthodes de création par défaut "new" ou "new:". Le message "new:" permet de paramétrer la création d'un objet.

Toute classe est un exemplaire d'une autre classe, donc toute métaclasse doit être un exemplaire d'une classe. Toutes les métaclasses sont des exemplaires de la classe "Metaclass". Or la classe "Metaclass" est un exemplaire de sa métaclasse. Donc les deux classes "Metaclass" et "Metaclass class" sont des exemplaires et des classes l'une de l'autre. Par cet artifice, la récursion des classes de classes est stoppée.

Nous donnons la définition de la classe "FileDirectory class" qui est la métaclasse de la classe "FileDirectory" présentée dans le premier exemple.

```
FileDirectory class
    instanceVariableNames: ''!


!FileDirectory class methodsFor: 'class initialization'!

initialize
    self initializeExternalReferences
    "FileDirectory initialize"! !

!FileDirectory class methodsFor: 'instance creation'!

directory: aFileDirectory directoryName: aString
    "Answer an instance of me in directory aFileDirectory whose name is aString."

    ↑self new setDirectory: aFileDirectory directoryName: aString!

directoryFromName: fileDesignator setFileName: localNameBlock
    | directory stream |
    "Answer the file directory implied from the designator, presumably at the start of the
    designator, and evaluate the block with the rest of the designator (presumably the
    file name itself."

    directory ← Disk.
    stream ← ReadStream on: fileDesignator.
    localNameBlock value: (stream upTo: nil).
    ↑directory! !

    ...
```

Nous pouvons résumer les relations de sous-classe/super-classe et d'exemplaire de classe par une figure replaçant les classes "Number" , "Integer", "SmallInteger" et leurs métaclasses dans la hiérarchie des classes.



Légende:

X ——► Y : X est un exemplaire de Y.

3.3. Le modèle d'exécution.

3.3.1. Messages et méthodes.

Les messages représentent les interactions entre les composants du système Smalltalk-80. Un message requiert un traitement de la part de son destinataire. Ce dernier exécute une méthode qui appartient à sa classe ou à l'une des super-classes de sa hiérarchie de super-classes.

a- Les types de message:

Une expression message contient un destinataire, un modèle de message et éventuellement des arguments.

Message unaire: C'est un message sans argument.
exemple *" frame center "*

Message binaire:
Un message binaire possède un seul argument et un modèle de message qui est un ensemble de simple ou double caractères spéciaux appelés des "modèles de messages binaires".
exemples *" expenditures <= count " et " origin + offset "*

Message à mots-clés:
Un mot-clé est un identificateur suivi de ":".
Un message mot-clé a un ou plusieurs arguments précédés chacun par un mot-clé.

exemple *" list at: index put: element "*

Cette expression est équivalente à un appel d'une procédure "at:put:(list,index,element)" dans un langage algorithmique

b- Priorité entre messages:

Les messages unaires sont les plus prioritaires. Les messages à mots-clés sont les moins prioritaires.
Cela nous donne l'ordre suivant:

message unaire > message binaire > message mot-clé

L'ordre d'évaluation peut être modifié par l'utilisation de parenthèses.

exemple *" (frame + offset) center "*

c- Les méthodes:

Une méthode décrit la suite des opérations exécutées par un objet pour traiter un message émis par un autre objet. La notion de méthode est équivalente à celle de procédure pour les langages algorithmiques.
Une méthode est constituée de trois parties:
- Un modèle de message et les déclarations des arguments formels (paramètres formels de la procédure).
- Des déclarations de variables temporaires (variables locales de la procédure).
- Des expressions qui constituent le corps de la méthode (corps de la procédure).

d- Recherche de la méthode appropriée:

La détermination des actions à entreprendre, quand un message est reçu, inclut la recherche des méthodes dans la hiérarchie des classes du destinataire. La recherche commence dans la classe du destinataire et se continue dans la chaine des super-classes. Si aucune méthode n'est trouvée dans la dernière super-classe ("Object" ou "Object class") alors une erreur est signalée à l'utilisateur.

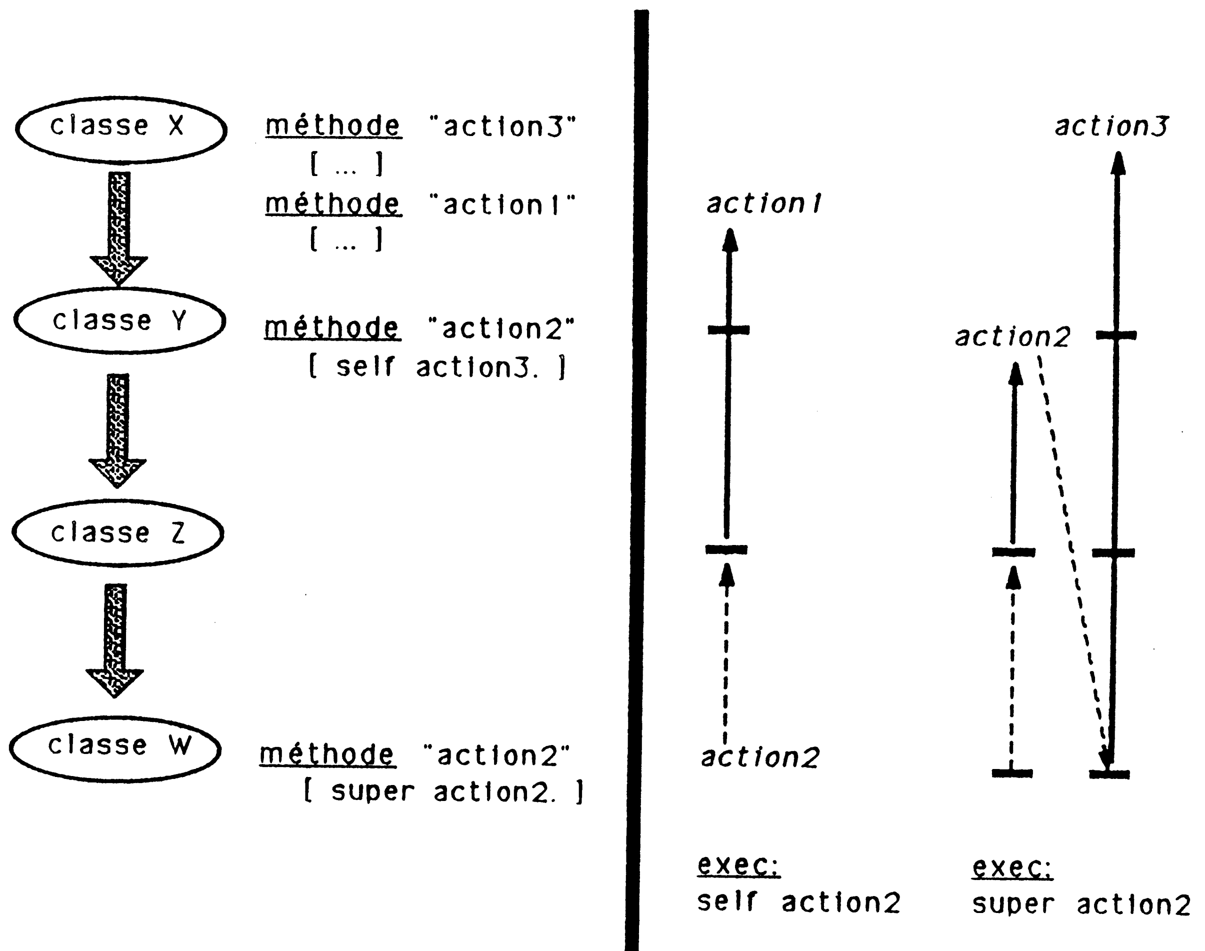
Si le destinataire est un objet classe, les recherches débutent à partir de sa classe qui est une métaclasse. Les messages auxquels une classe peut répondre s'appellent des "class methods"; ce sont les méthodes de création d'exemplaire de la classe. Si le destinataire n'est pas une classe, alors les messages auxquels il peut répondre s'appellent des "méthodes d'exemplaire"; ce sont les méthodes modifiant et testant l'état de l'exemplaire destinataire.

e- Deux destinataires particuliers: "self" et "super".

Self:
La pseudo-variable "self" référence le destinataire du message qui a nécessité l'exécution de la méthode courante.

Super:
La pseudo-variable "super" référence aussi le destinataire du message, mais la recherche pour une méthode commence dans la super-classe de la classe de la méthode qui s'exécute. C'est cette dernière méthode qui a envoyé un message avec "super".

Le schéma suivant illustre l'utilisation des pseudo-variables "self" et "super":



Légende:

———▶ Recherche de la méthode.

---------▶ Déplacement des recherches.

Dans l'annexe A, nous présentons en détail la structure des divers objets du système Smalltalk, ainsi que leurs durées de vie.

3.3.2. Les blocs.

Un bloc est un objet qui regroupe un ensemble d'instructions dont l'exécution peut être retardée. Cette notion provient directement du langage Simula67. Au niveau de la syntaxe, un bloc est un ensemble d'instructions encadré par un crochet ouvrant "[" et par un crochet fermant "]".

L'aspect nouveau introduit par Smalltalk réside dans le fait que la manipulation des blocs a été complètement homogénéisée avec celle des autres objets. Les instructions d'un bloc faisant partie d'une méthode, la création d'un objet destiné à contenir les instructions est inutile. Par contre, un bloc syntaxique donne lieu à la création d'un contexte de bloc en vue de sa future exécution. Ainsi une classe "BlockContext" a été créée pour définir le protocole de manipulation des objets-blocs.

- Un premier ensemble de méthodes que l'on peut effectuer sur un objet bloc regroupe les opérations permettant de fixer et d'accéder aux caractéristiques de l'objet (nombre d'arguments, nombre de variables temporaires nécessaires à une évaluation du bloc, etc...).

Exemple:
Soit la définition de la méthode "initBlocMultiplication" d'une classe "Calcul".

initBlocMultiplication

↑ *[:y :z | ↑y*z]*

L'expression *BlocMultiplication ← Calcul initBlocMultiplication.*
crée un exemplaire de bloc qui est affecté à la variable globale "BlocMultiplication".

L'expression *BlocMultiplication method.*
délivre le pointeur-objet (le nom dans l'image virtuelle) de la méthode compilée contenant les instructions du bloc "BlocMultiplication".

- Un deuxième groupe de méthodes permet de demander l'exécution d'un bloc en lui fournissant ou non des paramètres. Ce sont les méthodes "value", "value:", "value:value:", etc... Dans ce cas l'exécution est synchrone car elle est équivalente à l'appel d'une procédure.
L'exécution du bloc défini précédemment peut être demandée par une expression telle que:

```
z ← BlocMultiplication value: z value: z + 4.
```

Il est facile de voir que cette expression calcule la valeur de l'expression: z * (z + 4).

- Un troisième groupe de méthodes permettent d'exécuter le bloc de façon asynchrone, c'est à dire en créant une activité parallèle. Ce sont les méthodes "fork" ou "newProcess" qui intègre l'objet-bloc destinataire du message, dans un objet-processus comme contexte initial d'exécution.
On peut par exemple créé un processus "chien de garde" par l'exécution du groupe d'instructions:

```
unBloc ← [ [true] whileTrue: [] ].
ChienDeGarde ← unBloc fork
```

La notion de bloc est très importante dans le système Smalltalk car elle permet de définir de nouvelles structures de contrôle: les blocs étant des objets à part entière du système Smalltalk, ceux-ci peuvent apparaître comme destinataires ou paramètres de messages.
L'exemple ci-après définit deux blocs "eofBlock" et "retryBlock" qui traitent respectivement les événements "fin de fichier" et "fichier non ouvert" lors d'une opération de lecture sur un fichier.

```
eofBlock ← [ errFile put: 'Erreur: fin de fichier ].
retryBlock ← [ self open.
        ↑ self getCarIfEof: eofBlock ifErrOpen: [ errFile put: 'Erreur: problème d'ouverture' ]
```

Une expression telle que:

```
caractère ← unFichier getCarIfEof: eofBlock ifErrOpen: retryBlock.
```

essaie de lire un caractère du fichier "unFichier". Si la fin du fichier survient durant l'opération, le bloc "eofBlock" est exécuté. Si l'opération de lecture détecte une erreur sur l'ouverture du fichier, alors le bloc "retryBlock" est exécuté. Cette exécution fait une ouverture du fichier (représenté par la variable "self"), et renouvelle l'opération de lecture. Si de nouveau une erreur est détectée par rapport à l'ouverture du fichier alors un message d'erreur est fourni.

La notion de bloc étant centrale dans le système Smalltalk, la définition de la classe "BlockContext" est donnée en annexe B.

3.4. Les processus et le contrôle de l'exécution.

Le système Smalltalk fournit la possibilité de créer des processus indépendants et de synchroniser leurs exécutions à l'aide de trois classes: "Process", "ProcessorScheduler" et "Semaphore". Un processus représente une séquence d'actions qui peuvent être évaluées en parallèle avec les actions des autres processus.

Pour indication, nous donnons la définition de la classe "Process" dont les exemplaires sont les seuls objets du système qui permettent d'exprimer du parallélisme.

```
Link subclass: #Process
	instanceVariableNames: 'suspendedContext priority myList '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Kernel-Processes'!

Process comment: 'Instance Variables:
		suspendedContext <Context> activeContext at time of process suspension
		priority	<Integer> partial indication of relative scheduling
		myList	<LinkedList> on which I am suspended

I represent an independent path of control in the system. This path of control
may be stopped (by sending me the message suspend) in such a way that it can
later be restarted (by sending me the message restore). When any one of several
paths of control can be advanced, the single instance of ProcessorScheduler
named Processor determines which one will actually be advanced partly using
my priority.'!


!Process methodsFor: 'changing process state'!

resume
	"Allow the process that the receiver represents to continue. Put the receiver
	in line to become the activeProcess. Fail if the receiver is already waiting
	in a queue (in a Semaphore or ProcessScheduler). Essential. See Object
	documentation whatIsAPrimitive."

	<primitive: 87>
	self primitiveFailed!

suspend
	"Stop the process that the receiver represents in such a way that it can be

	restarted at a later time (by sending the receiver the message resume). If

	the receiver represents the activeProcess, suspend it. Otherwise fail and
	the code below will remove the receiver from the list of waiting
	processes. Essential. See Object documentation whatIsAPrimitive."

	<primitive: 88>
	Processor activeProcess == self
		ifTrue: [self primitiveFailed]
		ifFalse:
```

[Processor remove: self ifAbsent: [self error: 'This process was not active'].
myList ← nil]!

...

Le "scheduler" possède un objet qui représente les processus prêts à être activés; ce sont les processus en attente des ressources critiques que sont l'interprète et la machine virtuelle. L'interprète distribue son activité entre tous les processus Smalltalk prêts. A chaque cycle, le processus le plus prioritaire du système est déterminé. Si un processus prêt est plus prioritaire que le processus actif, il devient le processus actif.

Le cycle de l'interprète se résume en:

cycle:

1- changer éventuellement de processus.
2- aller chercher l'octet-code suivant dans la méthode compilée du contexte courant du processus élu.
3- incrémenter le pointeur d'instruction.
4- exécuter les actions spécifiées par l'octet-code.

fincycle:

4. Conclusion.

Le but de ce chapitre était de présenter les caractéristiques des langages et des systèmes à objets. Deux exemples ont été développés: le langage Simula67 et le système Smalltalk-80. Comme conclusion, il paraît intéressant de faire une synthèse des différents avantages et inconvénients introduits par ce mode de conception et de programmation.

Le tout premier avantage de la programmation par objets est la modularité. Du fait que l'application est conçue et réalisée en termes d'objets communiquants par des messages, où chaque objet reste maître des modifications de sa valeur, ceci établit un cloisonnement et un découpage automatique de l'application en modules (les classes) et en entités plus élémentaires (les objets). Chaque nouvelle définition de classe peut être mise au point indépendamment. Cette caractéristique centrale est directement liée à la possibilité de créer des classes.

Cependant celle-ci est mieux exploitée dans le système Smalltalk car la définition d'une classe particulière n'est faite qu'une seule fois: en effet, toute définition de classe donne lieu à la création d'un ensemble d'objets statiques et rémanents au sein du système. A l'opposé, une définition de classe dans le langage Simula67 est uniquement valable pour la durée d'une exécution. Si l'on veut modifier des procédures d'une classe Simula67 ou y rajouter de nouvelles procédures, il est nécessaire de recompiler entièrement la classe. Il est certain qu'un système comme Smalltalk, dans lequel on ne recompile que les parties

modifiées dans la dernière session d'édition, permet de faire du prototypage d'application plus rapidement.

Cette remarque montre que la programmation par objets peut être une programmation incrémentale à un degré plus ou moins important.

De plus, si le système sur lequel le langage évolue offre la dynamicité des associations *"message → méthode"*, comme c'est le cas pour le système Smalltalk-80, alors cette faculté de programmation incrémentale est accrue.

La plupart des langages et systèmes à objets offrent la possibilité de spécialiser une classe en créant une ou des sous-classes dérivées de celle-ci. Ce cheminement de la conception de l'application est descendant.

D'un autre coté la définition d'une structure d'héritage, permet une factorisation de la partie commune de la définition de plusieurs classes, cette partie commune étant implémentée par une classe qui est la super-classe de ces dernières. Cette factorisation est intéressante par l'économie de texte source réalisée. Ceci constitue plutôt un cheminement ascendant de la conception.

Mais ce qui peut être le plus intéressant dans la définition d'une classe est la possibilité de définir des classes génériques d'objets. Ceci revient à pouvoir paramétrer la définition d'une classe d'objets, éventuellement avec le nom d'une autre classe. Par exemple, on peut ainsi définir une classe implémentant une pile pouvant contenir des éléments dont le type est fourni en paramètre. Les expressions du genre: *unePile ← new pile(entier)* et *uneAutrePile ← new pile(caractère)* sont alors correctes. Smalltalk permet de définir des classes génériques en reportant le contrôle des types à l'exécution.

Les divers langages ou systèmes à objets peuvent être scindés en deux groupes: les langages et systèmes qui offrent une liaison dynamique à l'exécution et ceux qui imposent une liaison statique. Il serait intéressant de définir un langage qui permette à l'utilisateur de mettre au point une application en n'utilisant que des liaisons dynamiques, puis qui permette, dans un deuxième temps, de fixer un certain nombre de ces liaisons pour obtenir une version efficace.

Les caractéristiques de la programmation par objets sont bien adaptées à l'écriture de systèmes d'exploitation où la plupart des fonctions sont analogues à des applications qui utilisent les services d'un noyau existant. Ces fonctions peuvent alors être définies au moyen de classes, facilement modifiables et extensibles, ce qui permet notamment de définir des environnements d'exécution adaptés à des besoins divers. Néanmoins, les systèmes d'exploitation utilisant le modèle des objets sont encore à l'état de prototype de recherche. Le chapitre suivant présente quelques exemples de systèmes répartis mettant en œuvre le modèle des objets.

III. SYSTEMES REPARTIS A OBJETS.

Nous examinons dans ce chapitre l'apport de la programmation par objets à la conception de systèmes répartis. Nous rappelons d'abord brièvement les principaux problèmes posés par ces systèmes, avant d'examiner quelques systèmes répartis à objets représentatifs des tendances actuelles.

1. Systèmes répartis.

1.1. Motivations.

Jusqu'aux années 1970, les systèmes répartis étaient composés de gros ordinateurs reliés entre eux par des moyens de communication lents (ligne téléphonique, ...). La mise au point de supports de communication rapides, tels que les câbles coaxiaux ou les fibres optiques , alliée à la baisse des coûts des processeurs et des mémoires et à l'augmentation de leur puissance, a permis le développement d'une nouvelle catégorie de réseaux informatiques, les réseaux locaux, composés de postes de travail individuels et de machines interconnectés par un support à grand débit, limité à un ou deux kilomètres au plus. Ces nouveaux environnements permettent à chaque utilisateur de travailler de manière autonome et d'accéder à d'autres machines offrant des services spécifiques (serveur d'impression, serveur d'archivage, compilateur, ...).

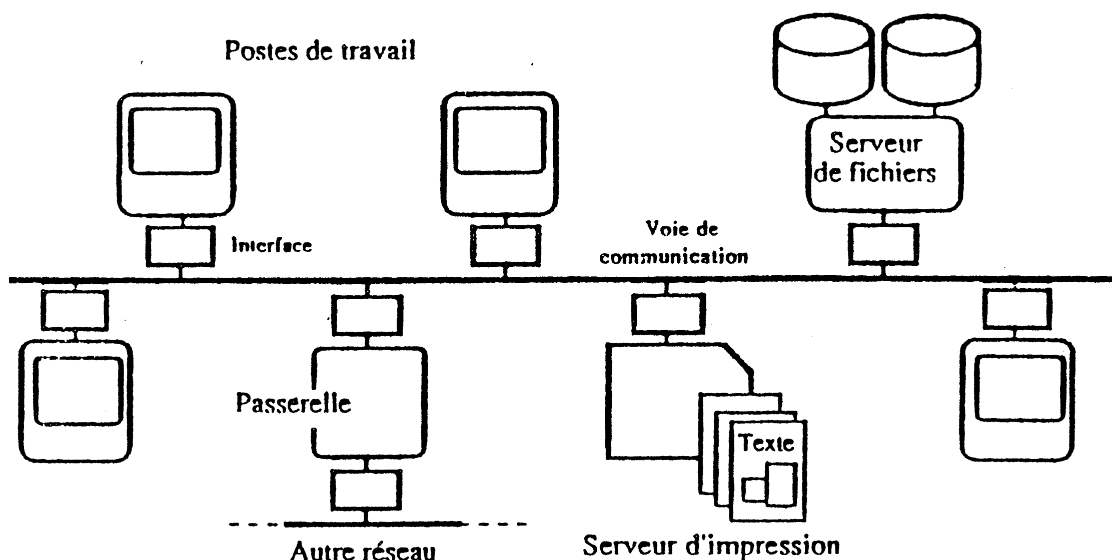
La définition d'une telle architecture a aussi été guidée par la nécessité d'une adaptation rapide à d'éventuelles modifications (ajout d'un site, déplacement d'une unité de stockage ou d'une imprimante, etc...) ou à d'éventuelles défaillances (panne d'un serveur, etc...).

On trouvera dans [Tanenbaum85, Balter86a, Krakowiak87] une synthèse des concepts et des problèmes introduits par les systèmes répartis. Chacun de ces articles fournit une bibliographie abondante représentative de l'état de l'art.

1.2. Architecture d'ensemble.

L'architecture des sytèmes répartis étudiés est composée de stations de travail possédant une mémoire de plusieurs méga-octets, un écran à points de haute résolution et un moyen de désignation appelé "souris". Chaque station de travail peut-être soit un poste individuel, soit un poste multi-utilisateurs. Un support de stockage local (disque) est facultatif, les ressources disques pouvant être gérées par des stations serveurs. Il est important de noter que les stations dépourvues de disque doivent pouvoir importer les supports systèmes nécessaires à leur démarrage et à leurs traitements.

Des stations serveurs sont présentes sur le réseau et assurent divers services. Les ressources offertes par ces dernières sont partagées par l'ensemble des systèmes. Le partage est géré sous contrôle des sites serveurs suivant des règles connues de tous les sites.



1.3. Principes de conception.

Une application conçue en milieu distribué requiert un certain nombre de nouvelles fonctions ou caractéristiques de la part du support système du réseau. Celles-ci proviennent surtout du fait que l'application doit évoluer dans un milieu hétérogène non centralisé.

La transparence des accès est la première nécessité induite par la répartition. L'utilisateur, bien que travaillant sur un nœud du système réparti, désire accèder avec les mêmes facilités aux ressources locales à sa machine et aux entités distantes. De cette façon, le programmeur conçoit son application distribuée comme si elle devait évoluer en environnement centralisé. Cette transparence est réalisée par la mise en place d'un mode de désignation des entités indépendant de leur localisation physique, et de fonctions d'accès qui permettent d'atteindre indifféremment des entités locales et distantes.

Néanmoins, dans certains cas, l'utilisateur a besoin d'avoir accès à l'information de localisation (mises au point, utilisation de ressources à localisation spécifique telles qu'une imprimante,...).

La notion de "serveur" est une application du principe de la transparence pour implémenter un service (impression, stockage, ...). Un serveur se présente comme un ensemble de primitives qui permettent d'obtenir les services qu'il gère. Un service peut être réalisé par un serveur localisé, ou par un ensemble de serveurs coopérants.

Une autre caractéristique recherchée par les concepteurs d'un système réparti est l'adaptabilité et l'extensibilité. Ainsi le système doit pouvoir s'adapter à des modifications de l'architecture matérielle (ajout de nouveaux périphériques ou de nouveaux postes de travail) ou de la réalisation des services.

1.4. Problèmes liés à la répartition.

1.4.1. Hétérogénéité.

Lorsqu'on désire faire communiquer des systèmes centralisés pour réaliser un système réparti qui permette de partager de l'information, il se pose le problème de l'hétérogénéité des sites. Cette hétérogénéité peut être matérielle (différence de la représentation des entiers en mémoire, par exemple) ou logicielle (les systèmes n'offrent pas les mêmes protocoles ou les mêmes fonctions de communication). Ce problème n'est pas envisagé dans cette étude car chaque site supportera le système Smalltalk-80.

1.4.2. Désignation des entités.

Le premier problème, qui se pose lorsqu'on désire offrir le principe de transparence d'accès et de localisation des objets dans un système réparti, est la réalisation du mécanisme de désignation. Ce mécanisme permet aux usagers d'accéder aux entités présentes dans le système à l'aide d'un nom symbolique (adresse logique). La correspondance entre la désignation logique et la localisation physique de chaque entité est maintenue par le service de désignation. L'implémentation de ce service (ou de ce serveur) constitue une partie centrale de la définition et de la réalisation d'un système réparti.

1.4.3. Allocation de ressources.

La difficulté de ce problème en milieu réparti est introduite par l'impossibilité de définir un état global d'allocation des ressources. Des solutions à ce problème sont principalement développées pour l'allocation d'un processeur du réseau à une activité,

l'ordonnancement des activités, la répartition de charge et la détection des inter-blocages [Tanenbaum85].

1.4.4. Protection.

Le système réparti doit assurer la protection des ressources mises en commun par les différents sites. Cette fonction de protection présente deux aspects:
- la vérification des accès aux ressources manipulées (vérification des droits, impossibilité de court-circuiter les appels systèmes pour accéder aux informations, impossibilité de récupérer ou de réquisitionner des droits, etc...).
- la possibilité pour chaque site de diminuer ou de retirer les ressources mis en commun avec les autres sites.

1.4.5. Tolérance aux pannes.

Les risques de défaillance des systèmes répartis sont accrus par la multiplicité et la diversité des composants mis en œuvre, ainsi que par la relative fiabilité des moyens de communication. Le concept de transaction atomique est communément utilisé pour prendre en compte ce problème. Les techniques mises en œuvre permettent de conserver la cohérence des informations partagées par l'exécution de procédures de reprises après panne, qui ramènent les informations dans un état antérieur cohérent (point de reprise).

1.5. Apport de la structuration en objets.

La caractéristique principale des systèmes et langages à base d'objets réside dans le mode de conception et de réalisation des applications. Dans un premier temps, la réalisation détaillée des différents traitements qui seront associés aux messages n'est pas définie par l'utilisateur. A ce stade, on peut dire que l'application a déjà été formulée suivant un mode très proche de la décomposition du problème en modules. Cette façon de concevoir une application est donc une démarche descendante où l'on spécifie d'abord l'interface des objets, puis on raffine l'implémentation jusqu'au plus bas niveau. D'autre part, il est souvent intéressant de définir des objets comme des compositions d'autres objets plus élémentaires déjà existants dont l'implémentation est cachée. Dans ce cas, l'application est conçue d'une façon ascendante. L'intérêt premier de la programmation par objets est donc de permettre une conception bidirectionnelle très cloisonnée, donc très propre.

D'autre part, la programmation par objets homogénéise les notions de donnée et d'entité calculatoire (le processus) en une seule entité: l'objet, ce qui apparaît très intéressant pour la réalisation de systèmes répartis puisque la création d'une activité distante est alors réduite à la migration d'au moins un objet vers le site considéré. Les mécanismes de gestion du système réparti n'ont plus qu'une sorte d'entité à gérer: l'objet.

Le modèle de la programmation par objets est également bien adapté aux exigences des systèmes de base de données qui gèrent des entités comme des compositions ou agrégats d'autres objets [Gibbs83, Nierstrasz85, Barbedette87]. Chaque objet ne peut manipuler les objets composants de son état que par des invocations à ceux-ci, et non pas par des accès directs.

Cependant cette homogénéisation ne résoud pas tous les problèmes liés à l'hétérogénéité de la structure du réseau sous-jacent. La différence de représentation de la structure des objets sur différentes machines peut amener à introduire une certaine hétérogénéité au sein du système réparti. Par exemple deux machines ayant des représentations différentes des objets d'un même type doivent mettre en place des mécanismes de traduction. Certains systèmes répartis peuvent définir pour ces raisons une machine à objets virtuelle commune à tous les sites.

2. Quelques expériences.

Il existe aujourd'hui quelques exemples de systèmes répartis essayant d'intégrer les concepts de la programmation par objets. Parmi les expériences les plus significatives, nous avons choisi de présenter les projets Eden, Emerald et SOS.

2.1. Eden.

Eden [Almes85, Black85] est un système réparti développé à partir de 1980 à l'Université de Washington (Seattle USA). L'objectif du projet était d'étudier l'apport d'une structuration en objets à la solution des problèmes posés par la répartition. Le support matériel de Eden est composé de Vax et de Sun supportant Unix 4.2BSD et reliés par une liaison Ethernet. Le langage EPL (Eden Programming Language) permettant une programmation à base d'objets a été défini à partir du langage Concurrent Euclid [Holt83]. Eden doit être vu comme une expérimentation qui était surtout destinée à mettre en évidence les avantages et les problèmes induits par l'intégration du modèle des objets à un environnement réparti. Ce projet a constitué un terrain d'expérimentation pour la définition d'un nouveau système réparti (Emerald). Celui-ci est présenté dans la section suivante.

Une application Eden est composée d'un ensemble d'*Ejects* (Eden objects). Les objets communiquent par des messages qui sont des demandes d'exécution d'un traitement sur l'objet destinataire. Chaque objet est désigné dans le système distribué par l'intermédiaire d'un nom unique. Un objet qui doit envoyer un message à un autre objet doit posséder une capacité d'accès sur ce dernier. Cette capacité contient les droits d'accès sur l'objet qu'elle permet d'adresser, ainsi que son nom unique. Les objets sont mobiles sur le réseau, et une capacité permet de désigner un objet quelle que soit sa localisation.

Le système Eden permet de définir des types d'objets (Edentype) qui sont composés d'une description de l'état des objets et d'un ensemble de méthodes qui peuvent modifier cet état. La définition d'un type Eden ressemble tout à fait à la définition d'une classe Smalltalk, excepté qu'il n'y a ni héritage ni généricité. Comme dans Smalltalk, la liaison "*message* → *méthode*" est dynamique.

Un objet est une entité active qui est constituée d'un ensemble d'activités qui communiquent à l'aide de variables partagées et se synchronisent à l'aide de moniteurs. Chaque activité au sein d'un objet peut envoyer un message à un autre objet. L'activité émettrice est bloquée jusqu'à ce que le traitement associé au message soit terminé. Les autres activités de l'objet permettent à celui-ci de pouvoir répondre à des invocations ou de pouvoir émettre d'autres messages. Ceci permet à un objet de mener plusieurs activités en parallèle sur un même ensemble de données (son état).

Tout objet peut initialiser des points de reprise par l'exécution d'une primitive système qui sauvegarde une représentation passive de l'objet en mémoire permanente. Lors d'une reprise après panne, un objet est capable de reconstruire son état. De plus, un objet qui a une représentation passive en mémoire permanente peut se désactiver, c'est à dire qu'il n'existe plus désormais pour celui-ci d'activité au sein du système. La représentation passive de l'objet pourra être réactivée si un autre objet lui envoie un message. Cette faculté d'activation et de désactivation permet de conserver beaucoup d'objets susceptibles d'avoir une activité future dans le système, tout en améliorant les performances de celui-ci. Le mécanisme souffre d'un manque d'efficacité qui provient surtout de sa réalisation au dessus du système Unix. En effet, chaque objet est implémenté par un processus Unix dans lequel sont gérées des activités parallèles. La création d'un ensemble trop important de processus est catastrophique pour le système Unix. On ne peut raisonnablement conserver sous forme de processus Unix des objets passifs. Cependant la désactivation des objets doit être explicitement précisée par l'utilisateur. Si un utilisateur crée beaucoup d'objets et qu'il ne déclare pas ceux qui peuvent passer sous une forme passive, alors il peut dégrader le système et ainsi pénaliser les autres usagers.

Malgré cette limitation, Eden présente une forme macroscopique et active de la notion d'objet. La structure d'exécution interne à chaque objet comprend un noyau de processus qui lui permet de multiplexer son activité. Chaque sous-activité est créée pour traiter un message. Le traitement d'un message est synchrone (appel de procédure local ou distant). Cela permet à la sous-activité, qui reçoit le résultat de son appel, de posséder l'historique et les conditions qui ont eu pour effet la génération du message.

Eden est en fait une expérience qui a permis de montrer que le modèle des objets pouvait apporter des solutions aux problèmes des systèmes répartis. Bien qu'une seule entité,l'Eject, représente des notions aussi différentes que des fichiers, des valeurs ou des programmes, la composition du système Eden n'est pas uniforme par le fait qu'il y existe deux types d'objets: les Ejects qui sont des objets actifs et les objets locaux du langage EPL. La désignation des Ejects est uniforme: objets, disques, sites, ... De plus, le fait de permettre à une activité de déplacer des objets pour faire des accès locaux ou de les accéder

à distance, engendre à la fois souplesse et efficacité au niveau du système. Par contre, la possibilité pour un objet de pouvoir sauvegarder un état de reprise par une procédure atomique, n'est pas satisfaisante pour implémenter efficacement des procédures de reprises après panne. Le problème vient du fait qu'une procédure de reprise met en œuvre beaucoup d'objets, et qu'un état de reprise s'exprime par un ensemble de relations entre ceux-ci.

2.2. Emerald.

Emerald est un projet en cours de réalisation à l'Université de Washington (Seattle USA) par une partie de l'équipe qui a réalisé le projet Eden. Etant donnée l'expérience acquise avec le projet Eden, il s'agit de définir un nouveau noyau de système réparti et un nouveau langage de programmation par objets permettant d'exprimer des applications distribuées.

Un net progrès a été fait par rapport à Eden, car ce dernier permettait de programmer une application à l'aide des expressions du langage EPL, mais aussi à l'aide des expressions de Concurrent Euclid qui est le langage support de EPL. Cette hétérogénéité au niveau du langage de programmation était des plus contraignantes pour l'utilisateur, car s'il voulait transformer une entité Concurrent Euclid en un objet Eden alors il devait modifier sa programmation.

Les objets communiquent par des messages. La réception d'un message donne lieu à une détermination dynamique de la méthode à exécuter. Les objets dits locaux appartiennent à un objet global et s'exécutent au sein de celui-ci. Les communications entre ces objets donnent lieu à une édition de liens statique mise en place lors de la compilation. Cette séparation a été guidée par un souci d'efficacité: les liaisons "message → méthode" dynamiques permettent une programmation souple mais coûteuse en temps d'exécution. Pour replacer ces définitions par rapport au système Eden, on peut dire que les objets globaux sont équivalents aux Ejects d'Eden, et que les objets locaux sont équivalents aux entités temporaires manipulées par les expressions Concurrent Euclid des méthodes de l'Eject.

Le typage des objets d'Emerald est basé sur la notion de type abstrait et de type concret.

Un type abstrait définit un ensemble d'opérations applicables aux objets de ce type. Chacune de ces opérations est spécifiée par sa "signature", qui est en fait la spécification du nombre et des types des paramètres.

Un type concret (classe) décrit une réalisation particulière d'un type abstrait, ainsi qu'un mécanisme de création d'exemplaires structurés selon cette description. Une classe doit donc spécifier:
- la structure des données qui constituent l'état des exemplaires.
- le code des opérations applicables aux exemplaires.

Il est important de remarquer qu'un même type abstrait peut donner lieu à plusieurs implémentations.

Le langage Emerald ne met pas en œuvre de relation d'héritage, mais introduit une relation de "conformité" entre classes, pour définir avec précision dans quelles conditions un objet de type T1 peut être utilisable à la place d'un autre de type T2. La relation de conformité entre classes fournit une relation de compatibilité entre types plus riche que la simple égalité des types. Les contrôles de types, suivant cette relation de conformité, s'effectuent à la compilation.

Dans un système tel que Smalltalk, la notion de type abstrait n'existe pas. Une relation de compatibilité entre types est fournie par la structure de hièrarchie des classes. Celle-ci permet de pouvoir manipuler un objet d'une classe avec les méthodes d'une de ses super-classes. La relation d'héritage exprime une compatibilité entre deux classes qui sont liés par le fait qu'une des deux est une spécialisation de l'autre. La relation de conformité de Emerald généralise celle introduite par l'héritage, et elle est définie comme suit:

Un type abstrait S est conforme à un type abstrait I si et seulement si:

- *toute opération de I est fournie par S.*
- *les opérations de I et leurs homologues de S ont le même nombre de paramètres et de résultats.*
- *les types abstraits des paramètres d'appel des opérations de I sont conformes aux types abstraits des paramètres des opérations de S.*
- *les types abstraits des résultats de chaque opération de S sont conformes aux types ab traits des résultats des opérations correspondantes dans I.*

Les objets du système Emerald sont manipulés par des invocations qui masquent la localisation des objets, car c'est au moment de l'exécution que l'objet est localisé. Lors d'une référence distante à l'objet, celui-ci est déplacé sur le site d'où émane la demande d'accès: l'accès est donc ramené à un accès local. Cependant cette transparence de localisation des objets n'est pas totale car Emerald fournit la possibilité de pouvoir manipuler la localisation des objets. Ainsi on peut, par programmation, *localiser* un objet sur le réseau, *fixer* un objet sur un site donné puis *re-autoriser son déplacement* et enfin *forcer la migration* d'un objet. Dans le cas où un objet est fixé sur un site, Emerald fournit quand même la possibilité de pouvoir réaliser un véritable accès à distance.

Emerald n'implémente pas réellement de politique de migration systématique. Quelques cas triviaux sont cependant pris en compte par le compilateur. Par exemple, les objets en lecture seule sont systématiquement dupliqués sur les sites qui les utilisent. Le but d'Emerald est de fournir un moyen de programmer des applications distribuées d'une façon homogène et efficace. Son aspect novateur est d'unifier les concepts des systèmes répartis autour d'une seule entité: l'objet. Emerald fournit un principe de conformité beaucoup plus riche qu'une simple relation de compatibilité de types offerte dans la plupart des langages.

2.3. SOS.

SOS (Somiw Operating System) est le système d'exploitation conçu pour un poste de travail et développé dans le cadre d'un projet ESPRIT (Secure Open Multimedia Integrated Workstation). Ce projet consiste à concevoir un environnement bureautique pour des stations de travail reliées par un réseau de communication. Le système utilise le modèle des objets afin que sa structure soit très modulaire et par là même modifiable dynamiquement. La modularité et l'encapsulation des entités introduites au niveau du système sont également développées en milieu distribué.

Le support langage de SOS est C+ + qui est une extension vers les objets du langage C. C+ + permet de définir des classes, puis de créer des exemplaires de celles-ci. Il offre également la possibilité de créer une classe par dérivation ou spécialisation d'une autre classe. Dans SOS, la correspondance "appel → procédure" peut être faite à l'exécution. Un éditeur de liens dynamique a donc été développé (il est cependant toujours possible d'utiliser des liaisons statiques). Il faut néanmoins noter que la liaison est dynamique uniquement pour le premier appel, et n'est plus modifiée ensuite. Cette édition de liens est tout à fait analogue à celle fournie par le système Multics [Organick72]. Cette dynamicité est bien inférieure à celle du système Smalltalk où chaque appel de procédure fait l'objet d'une liaison dynamique. Chaque appel peut donner lieu à l'exécution de procédures différentes déterminées par le contexte courant d'exécution.

Pour exprimer et gérer le partage des objets entre les différents sites SOS a introduit la notion de *mandataire* ("proxy"). Un mandataire est un objet qui représente sur son site de résidence un objet distant. L'objet distant est appelé ***objet principal***. Un objet mandataire est utilisé par l'application comme s'il était réellement l'objet principal. Toute invocation d'opération sur cet objet est automatiquement et transparemment répercuté sur l'objet principal via un accès à distance. Cette définition a l'avantage d'homogénéiser la notion et l'utilisation des objets en milieu réparti. Une application est donc conçue comme un ensemble d'objets communiquants sans qu'on ait besoin de savoir où se trouvent réellement les objets avec lesquels on veut communiquer.

3. Conclusion et critiques des diverses réalisations.

Plusieurs aspects intéressants ont été évoqués dans ce bref tour d'horizon des systèmes répartis à base d'objets. Eden a introduit la notion de traitement parallèle de plusieurs requêtes au sein d'un même objet. Ceci présente un net progrès par rapport à Smalltalk qui ne permet de faire que dû traitement synchrone (appel procédural) entre objets. En fait, les *Ejects* d'Eden ressemblent fortement à des serveurs: la communication entre ces objets suit le modèle "client-serveur". C'est de même pour pallier ce manque d'asynchronisme, que Smalltalk a défini une classe d'objets "Process" qui est destinée à fournir les programmes des activités parallèles. Cependant cette notion s'intègre mal dans le système.

Emerald a repris cet aspect de traitement parallèle au sein d'un objet, tout en homogénéisant le modèle réparti des objets. Ceci constitue un progrès considérable puisqu'il suffit de déclarer un objet local comme global pour le rendre effectivement actif. Le fait qu'un objet soit actif ou non ne s'exprime plus par une différence de programmation (Concurrent Euclid ou EPL), mais c'est simplement une caractéristique de l'objet qui peut être modifiée à tout instant.

La définition de la relation de conformité entre les types abstraits de Emerald permet d'étendre les possibilités de vérifications statiques à la compilation tout en permettant une grande souplesse de programmation. Si le type abstrait d'un objet X est conforme au type abstrait d'un objet Y alors on pourra par exemple écrire l'expression: Y ← X. On est sûr que X peut interpréter les mêmes messages que Y puisque son interface d'accès contient au moins les procédures de l'interface de Y.

SOS a introduit la notion d'objet mandataire ou "proxy" qui permet de ramener toute communication distante à une communication locale au site. C'est à dire que l'application se décrit toujours comme un ensemble de communications locales au site, le système se chargeant de traduire un accès local à un mandataire en un accès distant. Cette structuration du système engendre une grande souplesse au niveau de la conception d'une application similaire à celle offerte par l'utilitaire NFS [Sun84a, Sun84b] qui permet d'accéder sous Unix à des espaces de mémoire secondaire appartenant à d'autres sites comme si ces espaces de mémoire secondaire étaient supportés par des disques locaux au site. Une notion analogue a été développée lors de la conception du gestionnaire réparti pour Smalltalk-80, proposée dans le chapitre V.

IV. PRINCIPE D'UNE MEMOIRE VIRTUELLE D'OBJETS POUR SMALLTALK-80.

1. Introduction.

Ce chapitre présente une architecture pour la conception d'une mémoire virtuelle gérant des objets. Cependant, on ne peut raisonnablement concevoir une gestion d'objets indépendamment du système auquel elle est destinée. En effet, celle-ci dépend fortement des fonctions que le système nécessite. Cette proposition n'a pas pour objectif de définir une mémoire virtuelle d'objets pouvant s'adapter à n'importe quel système à objets, mais de proposer un modèle pour un système particulier dont l'étude est susceptible de dégager un certains nombres de problèmes spécifiques de la gestion des objets. Afin de mettre le plus facilement en évidence les problèmes des systèmes à base d'objets, il convient de gérer les objets en tant que tels. La conception proposée ignore donc tous les moyens et artifices existants qui seraient sensés "faciliter l'implémentation" comme l'existence d'une mémoire paginée au sein du système support ou d'un système de "mémoire cache". On se place donc comme si on voulait définir une mémoire virtuelle d'objets spécifique à Smalltalk-80 sur une machine nue.

Ce chapitre est organisé comme suit:

La deuxième section expose les motivations et les intérêts de cette conception, ainsi que les spécifications des fonctions de la gestion des objets attendues par la couche d'interprétation du système Smalltalk-80. De plus, une première solution (LOOM) de mémoire virtuelle pour le système Smalltalk-80, développée par les chercheurs de Xerox, est également présentée et discutée.

Les concepts généraux de la solution proposée sont exposés dans la troisième section. Nous présentons le découpage logique en modules du gestionnaire d'objets et précisons les différents protocoles de communication entre les modules pour le traitement des accès aux objets, des défauts d'objets et des créations d'objets.

La gestion de la mémoire centrale et de la mémoire secondaire est détaillée dans les quatrième et cinquième sections: translation d'adresse, désignation, mécanismes, choix des startégies de gestion, etc...

La sixième section expose un certain nombre d'améliorations, qui peuvent être apportées à une conception initiale où l'aspect de performances a été délibéremment mis de côté.

La conclusion de ce chapitre essaie d'établir un bilan des recherches et des solutions apportées. Cependant il faut noter qu'aucun choix de stratégie telle que celle de pré-chargement d'objets n'a été proposée. Les choix de telles stratégies doivent être guidées par des mesures statistiques de la composition et de l'évolution à l'exécution du système Smalltalk.

2. Spécification d'une mémoire virtuelle d'objets.

2.1. Motivation.

Le système Smalltalk-80 tel qu'il est actuellement défini peut adresser 32 K objets, car chaque "nom" d'objet (son pointeur) est codé sur 16 bits. Si on suppose que les objets ont une taille moyenne de 40 octets, on doit disposer d'une mémoire centrale de 1,28 méga-octet. Une mémoire de cette taille est tout à fait envisageable. Mais la limite des 32 K objets a déjà été atteinte par les concepteurs et utilisateurs du système. Il faut dés lors envisager un espace d'adressage des objets sur 24 ou 32 bits. Or, si pour des pointeurs-objet codés sur 16 bits, il est possible de réaliser une mémoire centrale qui puisse contenir 32 K objets, cela est impossible pour des pointeurs-objets sur 24 ou 32 bits. En effet avec des pointeurs-objet codés sur 24 bits, on peut adresser 8192 K objets. Ce qui donnerait une taille de la mémoire centrale de 327,68 méga-octets matériellement irréalisable. Pour cela nous nous intéressons à la réalisation d'une mémoire virtuelle des objets, réalisée à l'aide de deux types de mémoire physique: une mémoire centrale et une mémoire de masse (par exemple un disque).

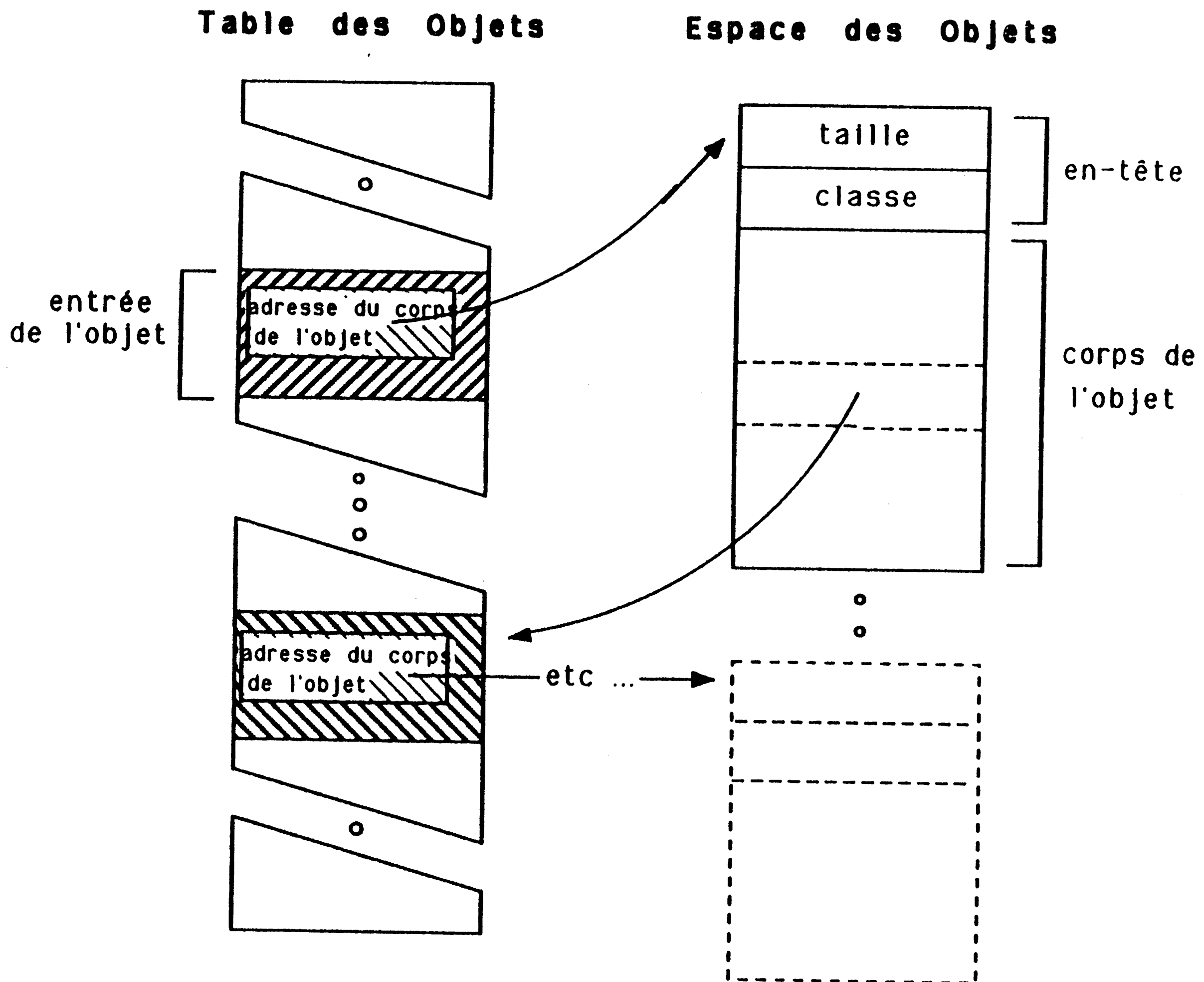
2.2. La mémoire des objets et son interface avec l'interprète.

2.2.1. Implémentation standard de la mémoire des objets.

Dans l'implémentation standard du système Smalltalk-80, tous les objets du système sont présents dans la mémoire centrale du calculateur.
Tout objet est composé de deux parties:

- Une en-tête qui contient sa taille en mots et le pointeur vers sa classe.
- Un corps qui contient les informations de l'objet, c'est à dire soit des valeurs immédiates soit des pointeurs vers d'autres objets.

Dans le système, tout objet est désigné sans ambiguïté par son pointeur-objet, qui est un index d'entrée dans une table appelée "table des objets", qui localise tous les objets du système en mémoire. Cette table contient l'adresse physique de l'objet dans la mémoire, et réalise ainsi une indirection d'accès aux objets. Cette indirection permet le déplacement des objets en mémoire centrale et le remplacement de la valeur de chaque objet, sans engendrer de modification de désignation.
Tous les accès aux objets passent par cette table, ainsi que les allocations et les libérations de pointeur-objet; l'allocation (resp. la libération) d'un pointeur-objet pour un nouvel objet correspond à l'allocation (resp. la libération) d'une entrée de cette table.



Organisation de la mémoire des objets.

Chaque entrée de la table des objets est constituée de deux mots de mémoire et par conséquent, tout pointeur-objet est une valeur paire codée sur un mot. Nous détaillons maintenant les informations contenues dans une entrée.

compteur	O	P	F	no. de segment
adresse du corps de l'objet				

Constitution d'une entrée.

2.2.2. L'interface interprète / mémoire des objets.

2.2.2.1. L'interprète (rappels).

L'interprète a pour fonction d'exécuter les processus Smalltalk. Pour cela, il entretient la liste des processus prêts à être poursuivis. De cette liste, il extrait un processus qui devient le processus actif; puis de ce processus, il charge les six registres qui constituent le contexte courant d'exécution. Le cycle de l'interprète a été précédemment exposé dans la section II.3.4.

Pour la détermination et le traitement d'un octet-code, l'interprète doit accéder à des objets à l'aide des procédures et fonctions de son interface avec la mémoire.

En résumé, l'interprète a deux fonctions:

- entreprendre les actions nécessaires pour chaque octet-code.
- distribuer l'exécution entre les divers processus Smalltalk d'après l'état global du système.

On peut remarquer que l'interprète déroule un algorithme séquentiel non bloquant; le choix du processus actif n'est pas remis en cause au cours du traitement de l'octet-code courant. Nous allons préciser maintenant les procédures et les fonctions qui constituent l'interface "Interprète/Mémoire".

2.2.2.2. Les procédures et fonctions d'accès aux objets.

La mémoire des objets offre un ensemble de primitives à la couche d'interprètation du système Smalltalk pour lui permettre d'accéder aux objets. Ces primitives n'accèdent pas toutes aux mêmes sortes d'informations de l'objet. En effet, les informations inhérentes

à chaque objet peuvent être divisées en deux ensembles d'informations fonctionnellement différents.

- Une première catégorie rassemble les informations qui décrivent la nature et l'utilisation de l'objet (sa classe, sa taille, son compteur d'utilisation, sa composition, etc...). Ces informations sont celles qui étaient présentes dans l'en-tête de l'objet, et dans la table d'indirection présentées dans la section IV.2.2.1. On appellera cet ensemble le *descripteur de l'objet*.

- L'ensemble des informations qui décrivent la valeur de l'objet constitue la deuxième catégorie d'informations. Cet ensemble sera appelé le *bloc-valeur de l'objet*. Dans la section IV.2.2.1, cet ensemble d'informations avait été appelé le corps de l'objet.

Ces deux définitions répercutées au niveau de l'implémentation du système Smalltalk constituent un changement de la structure d'un objet par rapport à la définition initiale. En effet dans la version initiale la classe et la taille faisaient partie du bloc-valeur de l'objet. Maintenant ces informations font parties de la partie descripteur de l'objet. Cette décision découle logiquement de la sémantique de ces informations.

Après avoir préciser cette modification de la structure de l'objet, nous pouvons classer les procédures et les fonctions de l'interface de la mémoire des objets suivant six types d'accès. Ces types d'accès se différencient par la nature des informations manipulées. Un accès élémentaire à un objet peut se ramener soit à:

- une écriture dans le descripteur de l'objet.
- une lecture dans le descripteur de l'objet.
- une écriture dans le bloc-valeur de l'objet.
- une lecture dans le bloc-valeur de l'objet.
- une création de l'objet (dans ce cas l'accès se ramène à une création du descripteur et du bloc-valeur).
- une énumération des exemplaires de l'objet si celui-ci est une classe. En fait, comme on le verra plus loin, les fonctions qui énumèrent les exemplaires d'une classe se résument à des accès en lecture sur les descripteurs de plusieurs objets.

2.3. LOOM: Une première mémoire virtuelle pour Smalltalk.

La nécessité de réaliser une mémoire virtuelle pour Smalltalk-80 a, dés le début, été ressentie par les premiers utilisateurs du système du centre de recherche de Xerox à Palo Alto [Kaehler82, Kaehler86]. En effet, ils se sont vite aperçus que le développement de la moindre application sur le système Smalltalk engendrait une consommation importante de ressources mémoires.

De plus, la conception initiale de la mémoire des objets telle que nous l'avons présentée précédemment avait pour inconvénient majeur de limiter l'espace mémoire des objets à la taille de la mémoire centrale du calculateur support. Pour ces raisons, les chercheurs de Xerox ont conçu et développé une mémoire virtuelle des objets nommée LOOM (Large Object-Oriented Memory). LOOM est un système de mémoire virtuelle mono-utilisateur qui gère et échange les objets entre deux niveaux de mémoire physique. L'unité d'échange entre les divers niveaux de mémoire est l'objet; les chercheurs de Xerox et plus particulièrement Ted Kaeler et Glenn Krasner étaient déjà convaincus du fait qu'il n'est pas naturel d'appliquer des méthodes de pagination sur les systèmes à base d'objets, telles qu'on les implémente sur les systèmes multi-utilisateurs traditionnels. La structure d'une mémoire virtuelle dépend de la granularité des informations échangées entre les différentes mémoires. La plupart des systèmes gèrent des segments de programme, les systèmes paginés transfèrent des pages-disque (il est à noter qu'un système peut être à la fois segmenté et paginé), tandis qu'une minorité de systèmes tels que OOZE[1] [Kaehler81], et LOOM gèrent et transfèrent des objets.

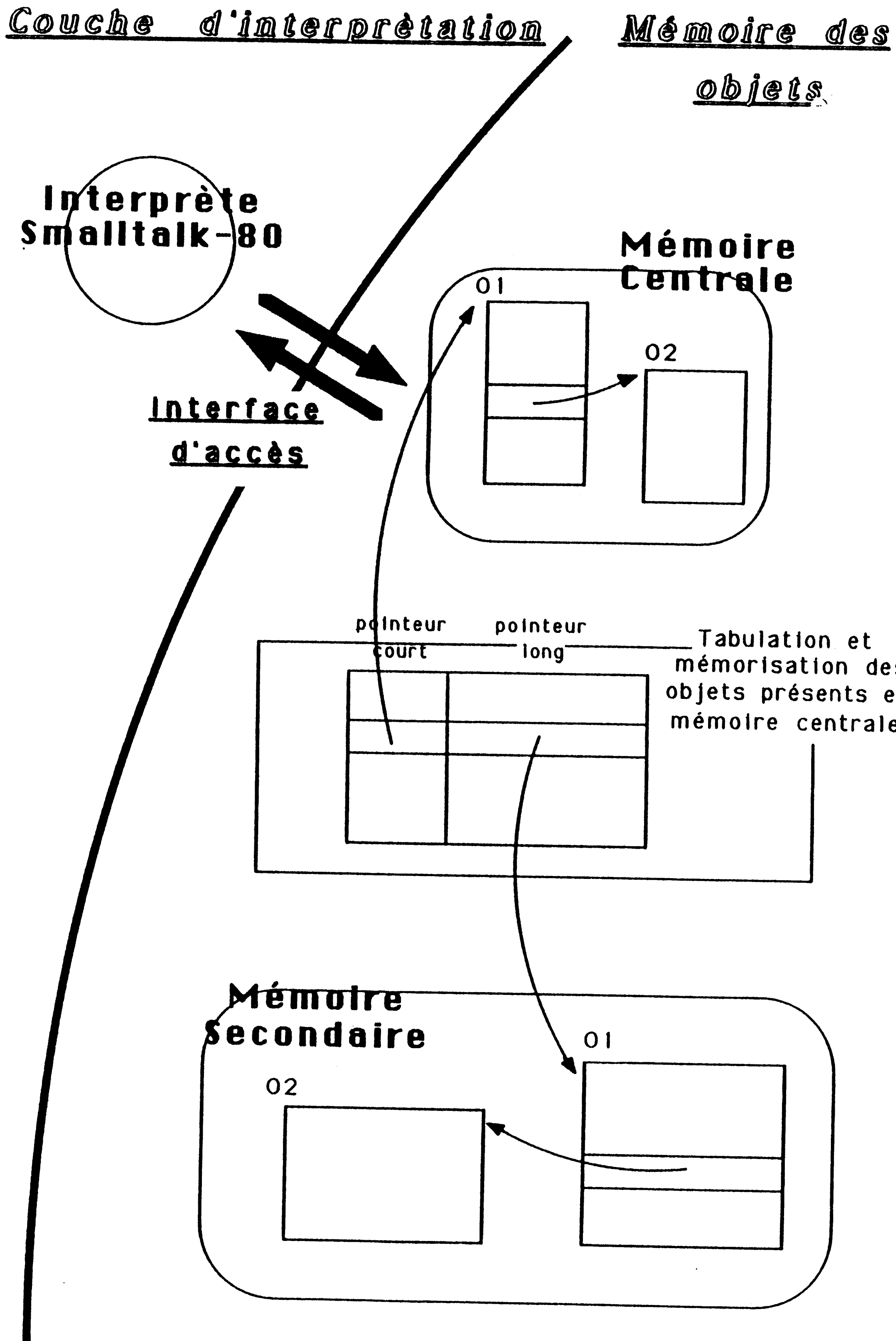
Les concepts.

LOOM échange individuellement les objets entre la mémoire principale et la mémoire secondaire de façon à ne maintenir en mémoire principale que les objets nécessaires à un instant donné. Contrairement aux techniques de paginations appliquées sur des systèmes à objets [Stamos84], LOOM a pour effet de concentrer au maximum les objets requis en mémoire centrale. L'interprète Smalltalk accède aux objets en mémoire centrale de la même façon qu'il est fait dans le système Smalltalk à mémoire résidente (version initiale de Smalltalk-80).

Pour rester cohérent avec la version initiale de Smalltalk, LOOM nomme les objets en mémoire principale avec des pointeurs codés sur 16 bits et accède aux objets en mémoire secondaire avec des pointeurs codés sur 32 bits. Sachant que la valeur de la majorité des objets Smalltalk est composée de pointeurs, il faut alors traduire chaque objet lorsqu'il passe d'un niveau de mémoire à un autre; par exemple, lorsqu'un objet est amené en mémoire principale, un pointeur de désignation en mémoire centrale lui est alloué et sa valeur est traduite sous forme de pointeurs de désignation en mémoire centrale. Un pointeur de désignation en mémoire centrale correspond par une relation bijective à son homologue de la mémoire secondaire pour désigner sans ambiguïté l'objet.

L'accès à un objet par l'interprète peut être résumé par le schéma ci-après.

[1] OOZE est le système de mémoire virtuelle qui a été développé pour "Smalltalk-76".


Couche d'interprètation
Mémoire des objets
Interprète Smalltalk-80
Mémoire Centrale
O1
O2
Interface d'accès
pointeur court
pointeur long
Tabulation et mémorisation des objets présents en mémoire centrale
Mémoire Secondaire
O1
O2

Conclusion.

Dans l'ensemble, cette solution est un bon départ pour essayer de réaliser une mémoire virtuelle à partir des réalités et des caractéristiques du système Smalltalk. Cependant cette solution engendre des inconvénients. Par exemple, il a été choisi d'augmenter les capacités d'adressage du système Smalltalk en augmentant à 32 bits l'adressage de l'image virtuelle de la mémoire secondaire; ceci est une décision tout à fait correcte. Cependant, pourquoi avoir gardé un adressage sur 16 bits à l'intérieur de la mémoire centrale, ce qui engendre la nécessité de traduction précédemment exposée?

La réponse est simple : les concepteurs de cette mémoire virtuelle ne voulaient pas modifier l'interprète initial qui travaillait avec des objets adressables sur 16 bits. Ainsi, tous les problèmes étudiés et toutes les solutions développées pour ceux-ci proviennent directement de ce choix initial qui n'est pas très heureux. Cette première implémentation d'une mémoire virtuelle pour Smalltalk est donc complètement faussée par ces contraintes. Une solution beaucoup plus raisonnable est de représenter de la même façon les objets dans les deux niveaux de mémoire, et d'y accéder à l'aide du même moyen. On peut alors mettre en évidence les vrais problèmes inhérents au système Smalltalk.

3. Principe d'une réalisation centralisée de la mémoire virtuelle d'objets.

Dés qu'on suppose que le nombre des objets manipulables n'est pas limité à la capacité de la mémoire centrale, se pose le problème de la réalisation d'une mémoire des objets à l'aide de deux niveaux de mémoire physique. Du point de vue de l'utilisateur, il n'y aura qu'une seule mémoire: "la mémoire des objets du système Smalltalk-80". Des échanges plus ou moins fréquents entre les deux niveaux de mémoire sont nécessaires pour réaliser cette mémoire des objets, mais chacun de ces échanges est totalement transparent à tout processus-interprète.

A tout moment, il n'y a qu'un sous-ensemble des objets du système présent en mémoire centrale. Il faut donc que le gestionnaire de la mémoire des objets soit capable de savoir si un objet est présent et si non, d'être à même de l'y amener. Pour amener un objet en mémoire, il faut qu'il dispose de la place nécessaire ou qu'il libère de la place pour le loger. Cette place peut être fabriquée en enlevant un ou plusieurs objets de la mémoire centrale ou en retassant l'espace des objets en mémoire centrale. Si l'on décide d'enlever des objets, il se pose alors le problème de la détermination des critères selon lesquels on les choisit (algorithme de remplacement). De même si on veut retasser l'espace des objets en mémoire centrale, il faut fixer un algorithme de retassement et les critères permettant de déclencher son exécution.

Ces problèmes sont inhérents à toute mémoire segmentée ou paginée. L'aspect nouveau réside dans ses caractéristiques quantitatives. Dans les systèmes segmentés

classiques, les segments sont en petit nombre et de grandes tailles, alors que dans le système Smalltalk-80, les segments (objets) sont très nombreux et de petites tailles.

3.1. Gestion d'une mémoire à deux niveaux.

3.1.1. Présentation du problème.

L'organisation de la mémoire des objets du système Smalltalk-80 est une organisation de mémoire segmentée. Dés que la taille de la mémoire centrale d'une machine est insuffisante pour contenir l'ensemble des segments d'un système, il se pose le problème de la réalisation de cette mémoire segmentée à l'aide de deux niveaux de mémoire physique:

- le niveau mémoire centrale où sont lus, modifiés ou exécutés "à la demande" les segments.
- le niveau mémoire d'archivage où sont sauvegardés et restaurés les segments.

Dans une telle mémoire, à tout moment, un objet référencé peut ne pas être présent en mémoire centrale. Ceci s'appelle un "défaut de segment"; dans le système Smalltalk-80, on appellera cela un "défaut d'objet". Dans le cas où un défaut d'objet survient lors du traitement de l'octet-code, l'interprète doit interrompre le déroulement du "processus-Smalltalk" courant. Cette interruption doit s'accompagner d'une sauvegarde des registres de l'interprète, qui contiennent son état d'avancement pour le traitement de l'octet-code.

Si l'on fait une analogie entre l'interprète et un processeur, les registres Smalltalk (respectivement les registres privés) de l'interprète ont pour équivalent les registres (respectivement les registres internes) du processeur. Un défaut d'objet a pour conséquence de faire perdre à l'interprète son caractère séquentiel: le choix du processus courant peut être remis en cause au cours du traitement d'un octet-code.

3.1.2. Le nouveau cycle de l'interprète.

Comme nous ne désirons pas, dans un premier temps, redéfinir un interprète Smalltalk-80, une solution pour lui conserver son aspect séquentiel est de supposer que l'on n'a plus un interprète unique mais autant d'interprètes que de "processus-Smalltalk" à interprèter. Nous faisons donc l'hypothèse de l'existence au niveau de la machine cible d'un noyau de synchronisation qui gère l'exécution des divers processus-interprète. Cette approche nous décharge des problèmes de gestion des "processus-Smalltalk". Elle nous rend également indépendant du nombre de processeurs physiques dont nous disposerons pour la réalisation du système avec la mémoire virtuelle des objets.

Chaque algorithme pour le traitement d'un octet-code particulier comprend désormais des points où un processus-interprète peut se bloquer sur un éventuel défaut d'objet. Avant qu'un processus-interprète actif ne passe au traitement de l'octet-code suivant, sa légitimité comme processus actif peut être remise en cause par l'intermédiaire du noyau de système. Le cas échéant, le processus-interprète actif peut être interrompu et un autre processus-interprète est activé.

Nous faisons l'hypothèse que ce noyau de processus nous permet:

a- la création et la destruction dynamique des processus.
b- la suspension et la réactivation des processus. Ceci est nécessaire pour implémenter les primitives de synchronisation du système Smalltalk-80.
c- la synchronisation des processus à l'aide des opérations classiques sur des sémaphores.
d- la communication entre les processus via un mécanisme de boîtes aux lettres.

Ces deux dernières facilités nous sont nécessaires pour l'implantation du système gestionnaire de la mémoire à deux niveaux. En outre, les processus seront gérés par priorité pour rendre compte de la politique du système à cet égard.

Le nouveau cycle de l'interprète est alors:

cycle
1- aller chercher l'octet-code suivant dans la méthode compilée à l'emplacement indiqué par le pointeur d'instruction.
2- incrémenter le pointeur d'instruction.
3- exécuter les actions spécifiées par l'octet-code.
fincycle

Désormais, chaque processus-interprète peut être suspendu à tout moment puisqu'il possède un jeu propre de registres pour mémoriser son contexte d'exécution.

3.1.3. L'accès aux objets.

Pour accéder aux informations d'un objet, un processus-interprète utilise les procédures d'accès aux objets. Or il est possible que l'objet désigné ne soit pas présent en mémoire centrale (défaut d'objet) d'où la nécessité de parenthéser les procédures d'accès avec deux procédures qui garantissent la présence de l'objet en mémoire centrale (verrouillage/déverrouillage) durant toute la durée de l'accès.

La procédure *début-accès* signale la volonté d'un processus-interprète d'accéder à un objet.
Lorsque cette procédure est exécutée, elle assure au processus-interprète que l'objet est

présent en mémoire centrale et que le type d'accès demandé sur cet objet est possible (verrouillage). Elle bloque un processus-interprète sur détection d'un défaut d'objet.

La procédure *fin-accès* est exécutée après la fin de l'accès à un objet par le processus-interprète.
Cette procédure est toujours passante et a pour effet de libérer l'objet des contraintes d'accès dues au type d'accès effectué par le processus-interprète (déverrouillage).

Ces deux procédures constituent l'interface entre les processus-interprète et la gestion de mémoire à deux niveaux. Plutôt que de réaliser deux procédures pour parenthéser les accès, qui demandent en paramètre d'entrée le type de l'accès désiré, il est préférable de réaliser autant de procédures "début-accès" et "fin-accès" qu'il y a de types d'accès possibles.

Nous donnons ci-après les procédures de parenthésage pour les différents types d'accès possibles:

- Accès en lecture sur le descripteur.

procédure "début-accès-lecture-descripteur" (pointeur-objet)
procédure "fin-accès-lecture-descripteur" (pointeur-objet)

- Accès en écriture sur le descripteur.

procédure "début-accès-écriture-descripteur" (pointeur-objet)
procédure "fin-accès-écriture-descripteur" (pointeur-objet)

- Accès en lecture sur le bloc-valeur.

procédure "début-accès-lecture-bloc-valeur" (pointeur-objet)
procédure "fin-accès-lecture-bloc-valeur" (pointeur-objet)

- Accès en écriture sur le bloc-valeur.

procédure "début-accès-écriture-bloc-valeur" (pointeur-objet)
procédure "fin-accès-écriture-bloc-valeur" (pointeur-objet)

- Accès pour création d'un objet.

fonction "début-accès-sur-création-d'objet" → pointeur-objet

Cette fonction délivre un pointeur pour le nouvel objet. Elle correspond à l'allocation d'un nom pour pouvoir désigner l'objet par la suite. Cette opération génère toujours un

défaut d'objet. Ce défaut d'objet est un peu spécial dans le sens où ce n'est pas un défaut d'objet sur l'accès au contenu d'un objet, mais sur la faculté de pouvoir le désigner.

procédure "fin-accès-sur-création-d'objet" (pointeur-objet)

Il est à remarquer que:

1- Les procédures et fonctions composant l'interface de la mémoire des objets, définie dans l'annexe C, peuvent faire successivement des accès à des objets différents. Chacun de ces accès est parenthésé par les procédures "début-accès" et "fin-accès" adéquates.

2- Nous ne traitons pas le type d'accès "énumération des exemplaires d'une classe" qui, nous le verrons plus tard, se réduit à un accès en lecture sur le bloc-valeur d'un objet qui est une classe.

3.1.4. Les fonctions gestionnaire de la mémoire.

**** topographie:**

Il est nécessaire que le gestionnaire de la mémoire soit capable de détecter la présence ou l'absence d'un objet en mémoire centrale. Pour ceci, il doit posséder une structure de topographie qui:
- d'après un pointeur-objet détermine si un objet est présent en mémoire centrale et, si c'est le cas fournit sa localisation.
- admette qu'un objet puisse être déplacé en mémoire sans aucune gêne pour les processus-interprète qui l'utilisent.

Pour des raisons d'efficacité évidente, il faut que cette fonction de topographie, qui joue un rôle similaire aux "mémoires associatives" des systèmes classiques, soit le plus possible économe en espace mémoire et en temps de calcul du fait de sa fréquence élevée d'utilisation pour le traitement des octets-code. Une particularité du système Smalltalk-80 est due à l'existence d'un grand nombre de petits objets de petites tailles en mémoire. Cette structure de topographie est donc un point central dans l'étude d'une mémoire objet virtuelle.

**** gestion de la mémoire libre:**

Tout objet pouvant être déplacé, chargé ou déchargé de la mémoire centrale, il faut gérer l'implantation des objets en mémoire. Pour cela, il faut pouvoir:
- allouer et libérer des blocs de mémoire de tailles variables.
- récupérer et reformer des zones de mémoire libre soit par recollage, soit par compactage.

Ce problème comprend deux aspects:

- un aspect "algorithmique": définition des procédures qui allouent, libèrent et retassent la mémoire.

- un aspect "stratégique": Quelle politique de gestion de mémoire choisir ? (Buddy System, Best-Fit, préformattage en blocs de taille fixée qui serait statistiquement déterminée, ...)

**** algorithme de remplacement:**

Dans l'hypothèse où aucune place n'est trouvée pour charger un objet, il faut en créer en déchargeant un ou plusieurs objets présents en mémoire. Ces objets sont choisis par l'algorithme de remplacement. Comment décider du ou des objets à sortir de la mémoire ?

- selon une liste LRU:
  .soit pour tous les objets présents en mémoire.
  .soit pour tous les objets que chaque processus peut atteindre.

- selon une connaissance statistique de l'évolution d'un processus Smalltalk.

- selon une méta-connaissance des objets manipulés et de leurs rôles dans le langage Smalltalk-80.

  - méthodes
  - contextes de méthode
  - contextes de bloc
  - etc...

Bien d'autres critères peuvent être envisagés.

Dés maintenant, on peut affirmer que ce choix ne pourra être fondé que sur des mesures du fonctionnement du système. Les diverses solutions devront être analysées et comparées. La conception de cet algorithme fait partie des points clefs de la réalisation d'une mémoire objet virtuelle.

**** algorithme de récupération d'objets:**

La récupération d'objets ("garbage collection") est le traitement qui consiste à récupérer l'espace de mémoire (centrale et secondaire) occupé par un ou plusieurs objets détruits. Ce problème provient du fait que les langages et systèmes à objets engendre une grande consommation d'objets, et que les objets récupérables ne sont pas signalés aux systèmes. Tout objet est susceptible de devenir "persistant" ou "volatile" [Thatte86]. Ce problème constitue une préoccupation majeure des concepteurs de systèmes à objets.

Comme nous avons deux niveaux de mémoire, nous avons deux niveaux de récupération d'objets. Les traitements de récupération d'objets de chacune des deux mémoires ne sont pas indépendants puisqu'un même objet peut se trouver présent dans les deux niveaux de mémoire. Dans ce cas, l'objet n'est peut être pas dans le même état de modification dans l'un et l'autre niveau de mémoire. Un protocole de récupération d'objets doit être établi entre les deux niveaux.

La récupération des objets dans une mémoire à un seul niveau est mise en œuvre par deux procédures:

- une procédure qui récupère un objet lorsque son compteur d'utilisation devient nul.
- une procédure qui met en œuvre un marquage des objets pour détecter et détruire les circuits non accessibles à partir de la racine du système.
Par exemple, A et B sont deux objets qui se réfèrencent mutuellement et dont les compteurs d'utilisation valent "1". Ces deux objets doivent être détruits, malgré la non nullité de leurs compteurs d'utilisation, car ils forment un circuit non accessible de l'extérieur.

Pour une mémoire objet à deux niveaux de mémoire, la procédure de récupération d'objets par marquage peut être faite sur la mémoire de masse (mémoire permanente des objets du système). Pour des raisons d'efficacité, elle est exécutée peu fréquemment (saturation de l'espace disque, démarrage ou arrêt du système). La répartition, entre les deux niveaux de mémoire, de la procédure de récupération d'objets par compteur d'utilisation est moins nette. La réalisation de cette procédure constitue un futur point de discussion.

3.2. Organisation générale du système à mémoire segmentée.

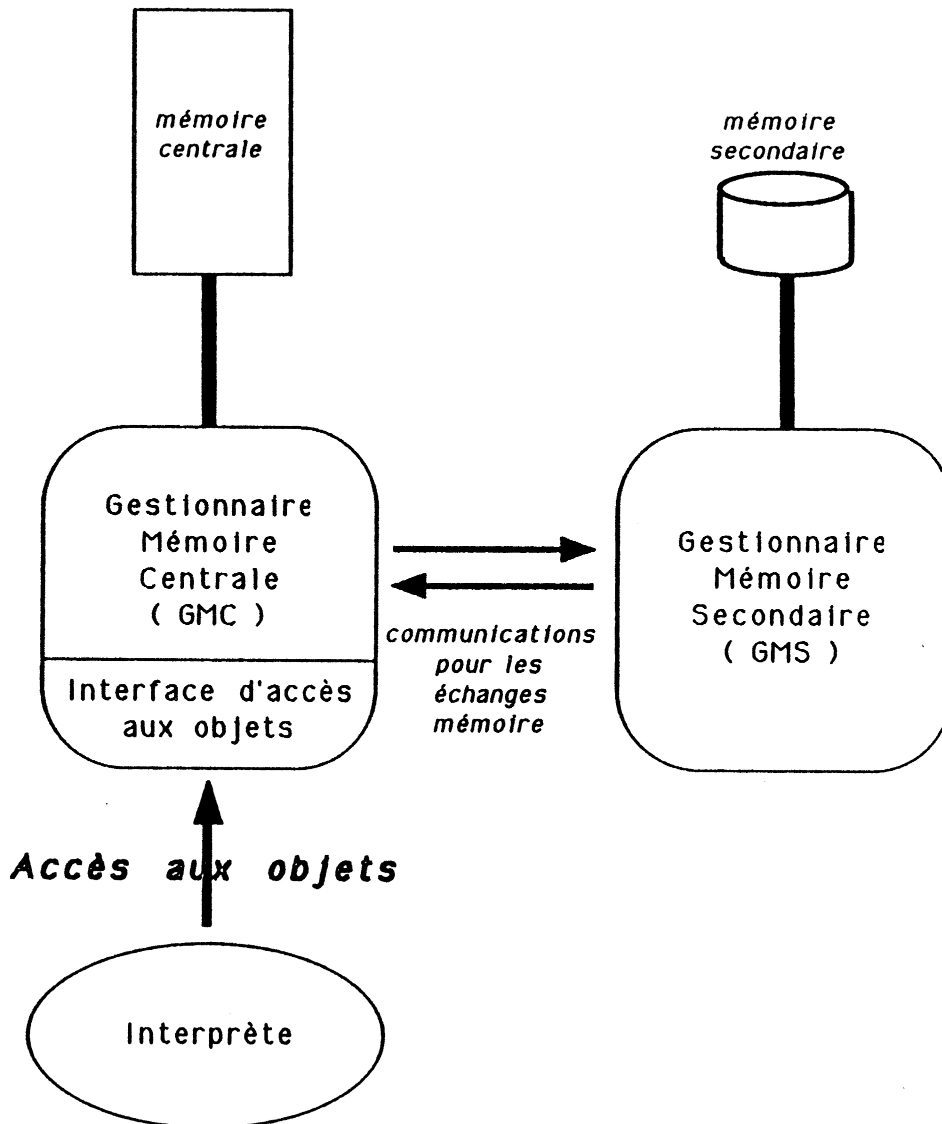
3.2.1. Découpage en modules et processus.

Le système est découpé en trois modules:

- le module interprète (MI).
- le module de gestion de la mémoire centrale (GMC).
- le module de gestion de la mémoire secondaire (GMS).

Chaque module est exécuté par un ou plusieurs processus. Les processus-interprète s'exécutent dans le module interprète (MI) et utilisent les procédures *début-accès* et *fin-accès* de l'interface du GMC pour accéder aux objets. Le travail gestionnaire de la mémoire centrale permettant de satisfaire ces accès est assuré par un processus particulier (PGMC) qui s'exécute dans le module GMC. Celui-ci utilise l'interface du module GMS pour charger et décharger les objets depuis et vers la mémoire de stockage.
Le module GMS se charge de la gestion en mémoire de masse des objets du système, ainsi

que de la gestion des noms (pointeur-objet) au sein du système. C'est donc lui qui détermine et délivre le nom de tout nouvel objet créé.



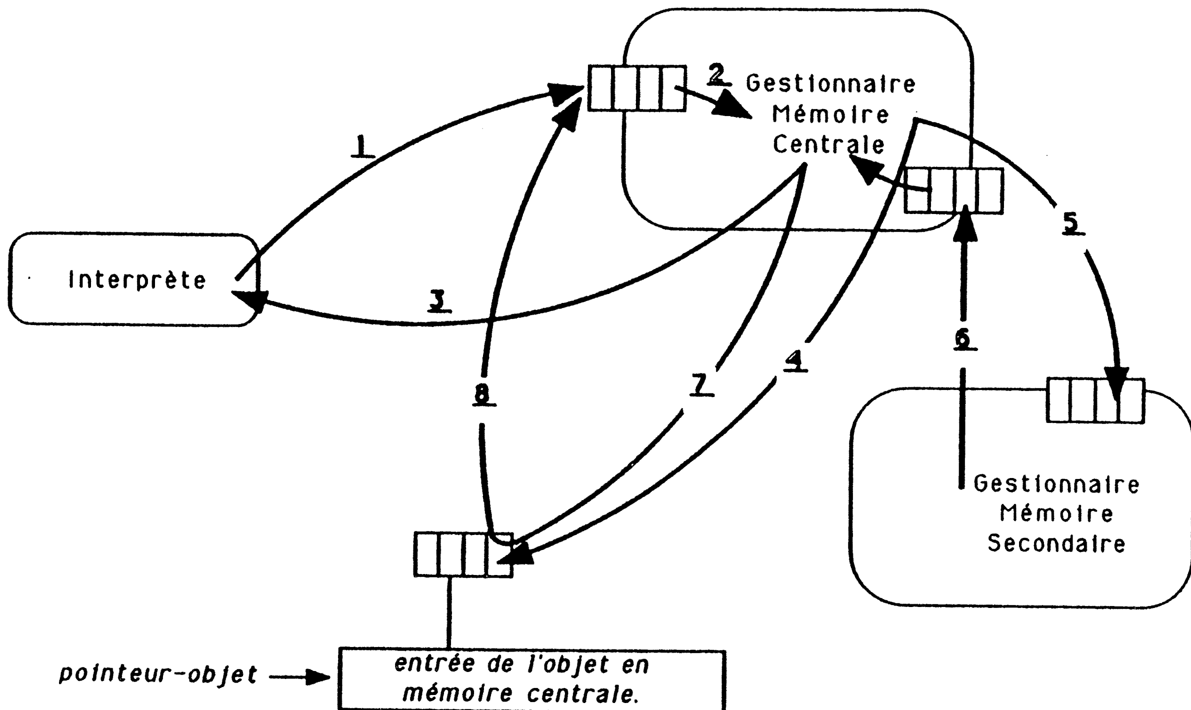
Les modules du système.

3.2.2. Le protocole de résolution des défauts d'accès aux objets.

Le module GMC constitue l'interface avec les processus-interprète pour la résolution des défauts d'objets. Cette interface est réalisée par des communications entre les PI et le PGMC. Les relations de communication entre PI et PGMC sont implantées dans ces procédures. Le mécanisme de synchronisation utilisé est la communication par messages via des boîtes aux lettres. L'utilisation de ces boîtes aux lettres est donc réservée aux procédures de l'interface de gestion de la mémoire (plus précisément aux procédures "début-accès") et par là sont transparentes à l'utilisateur. Le GMC possède une boîte aux lettres devant recueillir les défauts d'objet.

Tout processus-interprète qui, par l'intermédiaire des procédures de l'interface et des procédures "début-accès", fait une requête au PGMC doit attendre la fin de traitement de celle-ci. Par conséquent, chaque processus-interprète possède une boîte aux lettres personnelle sur laquelle il attend le message du PGMC lui indiquant la fin de traitement de sa requête .

La figure suivante représente un exemple de fonctionnement de l'interface d'accès aux objets.



Légende:

1 Un processus interprète fait un défaut d'objet et envoie un message au gestionnaire de la mémoire centrale en précisant le type de l'accès tenté.

2 Le gestionnaire de la mémoire centrale retire un message de défaut d'objet.
Le gestionnaire re-essaie l'accès.

3 L'accès est possible car entre-temps le traitement d'un message d'un autre interprète a levé ce défaut. Le processus interprète est alors ré-activé.

4 L'accés n'est pas possible. Le gestionnaire stocke le message dans la liste des messages de l'entrée associée à l'objet en mémoire.

5 Le gestionnaire envoie une requête de chargement au gestionnaire de la mémoire secondaire.

6 Au bout d'un certain temps, le gestionnaire de la mémoire centrale récupère un message de chargement pour un objet donné.

7 Le gestionnaire de la mémoire centrale met à jour les indicateurs de présence de l'objet concerné

8 Le gestionnaire de la mémoire centrale retraite tous les messages contenus dans la file de l'entrée associée à l'objet.

Accès à un objet et résolution d'un éventuel défaut d'objet.

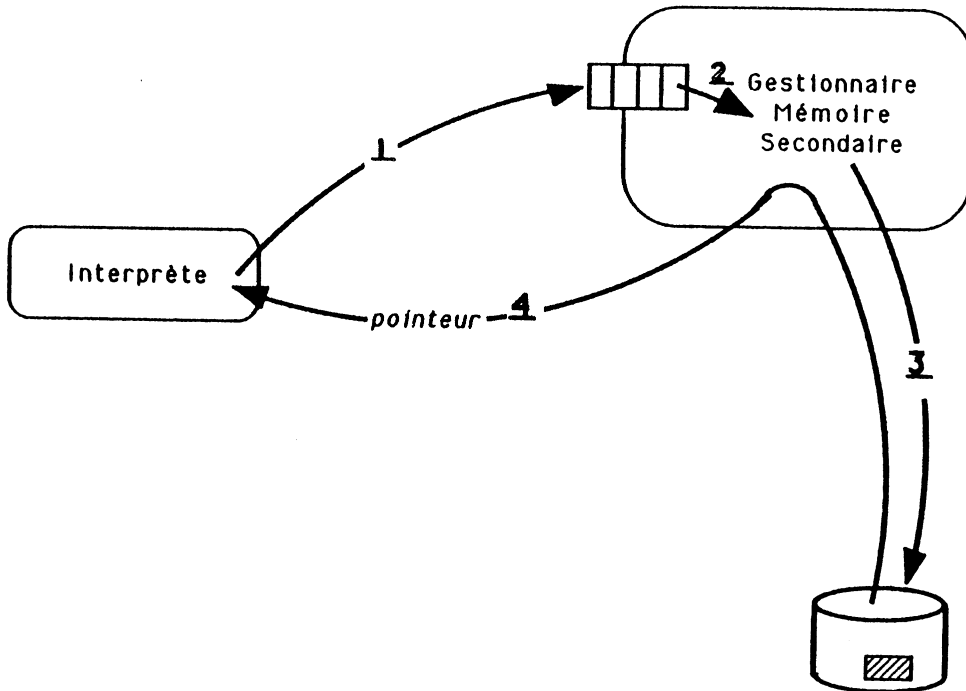
3.2.3. Le protocole GMC / GMS.

Une tentative d'accès à un objet non présent en mémoire centrale provoque un déroutement du cours logique de l'exécution d'un processus Smalltalk. L'exécution reprend lorsque cet objet est chargé en mémoire. Le processus gestionnaire de la mémoire (PGMC) voit le module de gestion des objets en mémoire secondaire (PGMS) comme un serveur auquel il peut demander quelques travaux:

- charger le descripteur d'un objet depuis le disque.
- recopier le descripteur d'un objet sur le disque.
- charger le bloc-valeur d'un objet depuis le disque.
- recopier le bloc-valeur d'un objet sur le disque.

Le protocole GMC / GMS est également réalisé à l'aide de messages. Chacun de ces deux modules possède une boîte aux lettres par type d'échange. Les boîtes aux lettres du module GMS reçoivent les demandes du module GMC et celles du module GMC reçoivent les réponses du module GMS.
Nous préciserons ultérieurement les relations GMC / GMS induites par le protocole de récupération d'objets.

3.2.4. Le protocole de création d'objet.



Légende:

1 Un processus interprète envoie un message de demande de création d'objet au gestionnaire de la mémoire secondaire.

2 Le gestionnaire de la mémoire centrale retire un message et crée un nom pour le nouvel objet.

3 Le gestionnaire de la mémoire secondaire réserve la place de l'objet en mémoire de masse pour ses futures recopies.

4 Puis il réveille le processus interprète auquel il fournit le nom de l'objet créé. Le processus interprète dispose alors d'un nouvel objet.

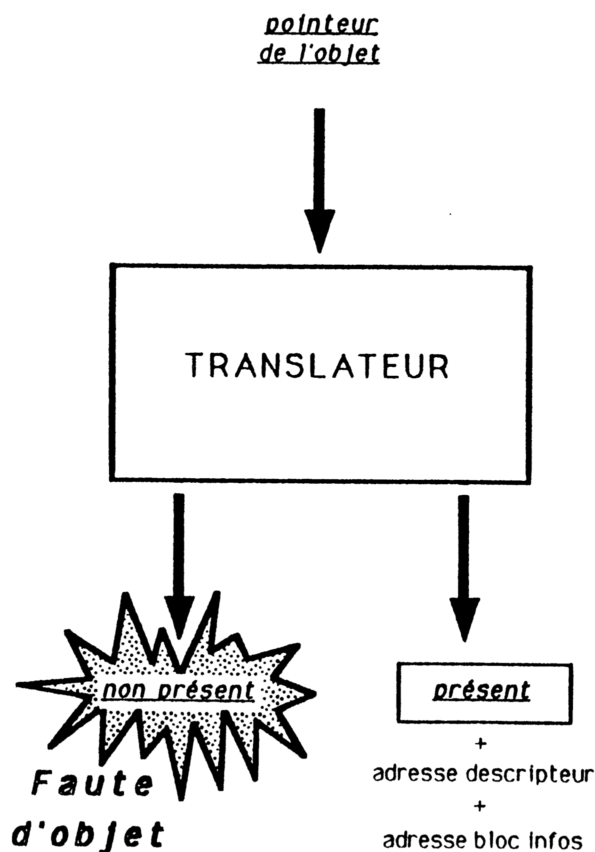
Protocole de création d'un objet.

4. Le module gestionnaire de la mémoire centrale.

4.1. Le translateur.

4.1.1. Fonction du translateur.

Tous les objets du système Smalltalk ne pouvant tous tenir en mémoire, le PGMC doit être capable d'y maintenir un sous ensemble de ceux-ci, et de les déplacer afin d'optimiser l'occupation mémoire. Nous avons donc besoin d'un translateur qui, à partir d'un pointeur-objet, puisse dire si l'objet est présent en mémoire centrale et si oui, retourne l'adresse de son descripteur et de son bloc-valeur. Le translateur est constitué d'une structure de topographie prolongée par une structure d'indirection qui permet de déplacer l'objet en mémoire centrale sans avoir à modifier la topographie.

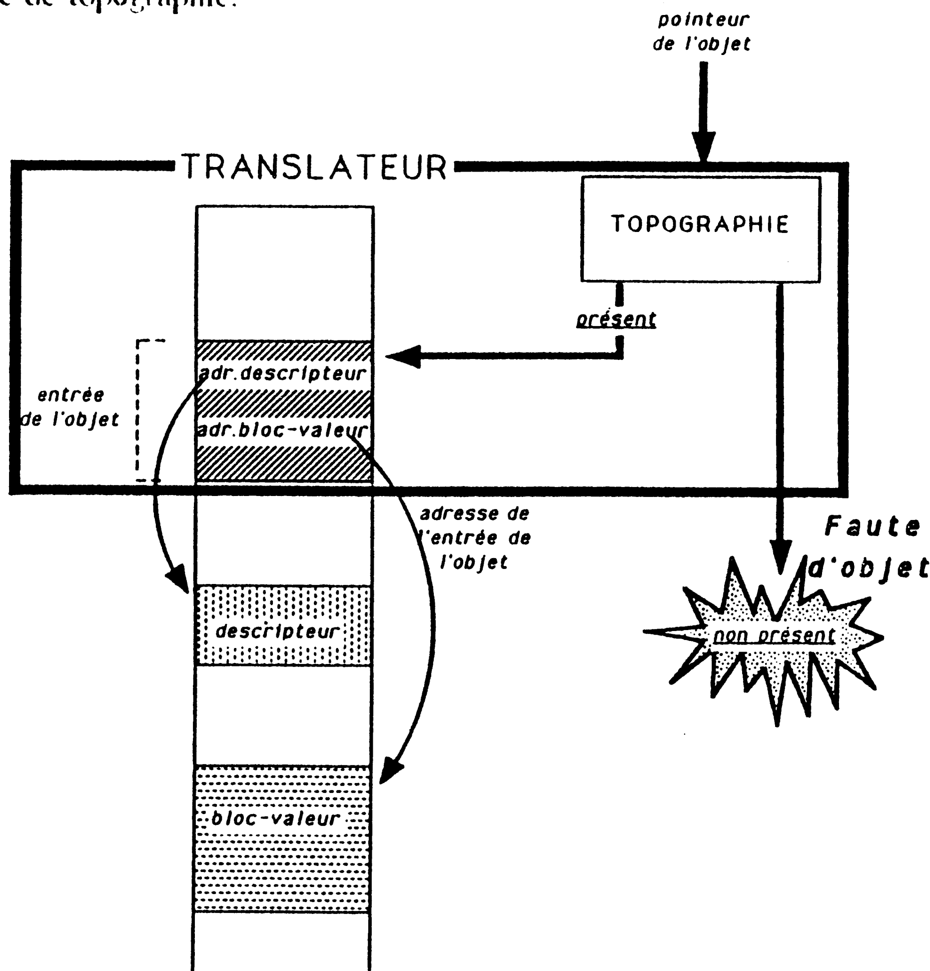


Fonctionnement du translateur.

Il est à noter qu'un objet est repéré de la même façon en mémoire centrale et en mémoire de masse; toute référence se fait par l'intermédiaire de son pointeur-objet. Par conséquent, il n'y aucune conversion des informations de l'objet lors du chargement ou du déchargement de celui-ci d'un niveau de mémoire vers l'autre.

Si le translateur se résume en une structure de topographie qui réalise la correspondance: *pointeur-objet → (adresse-descripteur, adresse bloc-valeur)*, il faut recalculer cette correspondance chaque fois que le descripteur ou le bloc-valeur est déplacé en mémoire centrale. Pour résoudre ce problème, nous introduisons la notion "d'entrée d'un objet". L'entrée d'un objet devient son pointeur interne dans la mémoire centrale.

Le translateur est constitué d'une structure de topographie qui réalise la correspondance: *pointeur-objet → adresse-entrée-objet*, et d'une fonction de gestion de l'ensemble des entrées des objets présents en mémoire centrale. Une entrée correspond à un objet unique et contient l'adresse du descripteur et du bloc-valeur de l'objet en mémoire centrale. De cette façon, lorsque le descripteur ou le bloc-valeur sont déplacés, les valeurs des adresses physiques sont modifiées dans l'entrée de l'objet, ce qui évite de modifier la structure de topographie.



Représentation et accès à un objet en mémoire centrale.

D'autres informations dans l'entrée d'un objet sont nécessaires. Elle sont exposées dans la section suivante.

4.1.2. Synchronisation sur l'accès au translateur.

Le translateur est utilisé par les processus-interprète et par le processus gestionnaire de la mémoire centrale. Les processus-interprète utilisent la structure de topographie pour obtenir, à partir d'un pointeur-objet, les adresses en mémoire du descripteur et du bloc-valeur d'un objet. Le gestionnaire de la mémoire centrale élabore la structure de données nécessaire à la réalisation de cette fonction de topographie. Il faut donc résoudre les conflits d'accès entre ces divers processus. Les protocoles de type lecteurs/rédacteurs ou d'exclusion mutuelle, paraissent suffisants pour régler cette synchronisation.

4.1.3. Composition d'une entrée.

L'entrée d'un objet en mémoire centrale contient toutes les informations nécessaires pour sa gestion. Ces dernières sont détaillées ci-après en précisant la fonction de chacune d'entre elles:

- Pour pouvoir accéder aux informations de l'objet, l'entrée contient l'adresse du descripteur et du bloc-valeur en mémoire.

- Les demandes de chargement ou de déchargement du descripteur ou du bloc-valeur ne doivent pas être dépendantes de l'implantation de ceux-ci en mémoire de masse. Ceci veut dire que toute demande de chargement/déchargement s'effectue en fournissant au GMS le pointeur-objet de l'objet. Pour cette raison, l'entrée d'un objet contient la valeur de son pointeur-objet. C'est un moyen simple de réaliser la fonction inverse de la structure de topographie (c'est à dire la relation *entrée-objet* → *pointeur-objet*).

- Dans chaque entrée, des variables d'état associées au descripteur et au bloc-valeur permettent de décider si l'accès par un processus-interprète est possible.

- De même , des indicateurs de modification du descripteur et du bloc-valeur indiquent si ces entités ont subi des modifications depuis leur chargement en mémoire centrale. Ces indicateurs permettent de minimiser le volume des recopies en mémoire de masse.

- Associée à chaque entrée, on trouve une file d'attente de requêtes. Cette file contient tous les messages de "défaut-d'objet", émis par des processus-interprète, qui ne peuvent pas être traités dans l'immédiat ou qui sont en cours de traitement.
  Par exemple, si la première requête a eu pour effet de lancer les opérations requises (chargement ou déchargement), les requêtes suivantes sont mémorisées dans la file. Toutes ces requêtes sont retraitées après la fin du chargement ou du déchargement qui les a bloqué.

Toutes les informations que nous venons d'introduire constituent les informations nécessaires à la description d'un objet en mémoire centrale et au fonctionnement du processus gestionnaire de la mémoire. La figure suivante résume les diverses informations de l'entrée d'un objet.

Champ	Description
pointeur-objet	nom Smalltalk de l'objet décrit par cette entrée.
file-messages	file des messages des processus interprète en défaut sur cet objet.
état-D	état du descripteur de l'objet (présent, absent ou chargement en cours).
m-D	indicateur de modification du descripteur.
adr-descripteur	adresse en mémoire centrale du descripteur.
état-B	indicateur de modification du bloc-valeur.
m-B	état du bloc-valeur de l'objet (présent, absent ou chargement en cours).
adr-bloc-valeur	adresse en mémoire centrale du bloc-valeur.

Les différentes informations de l'entrée d'un objet.

4.1.4. Ajout d'informations dans le descripteur d'un objet.

Des informations supplémentaires ont été ajoutées dans le descripteur de chaque objet pour permettre de réaliser efficacement les primitives d'énumération des exemplaires d'une classe.

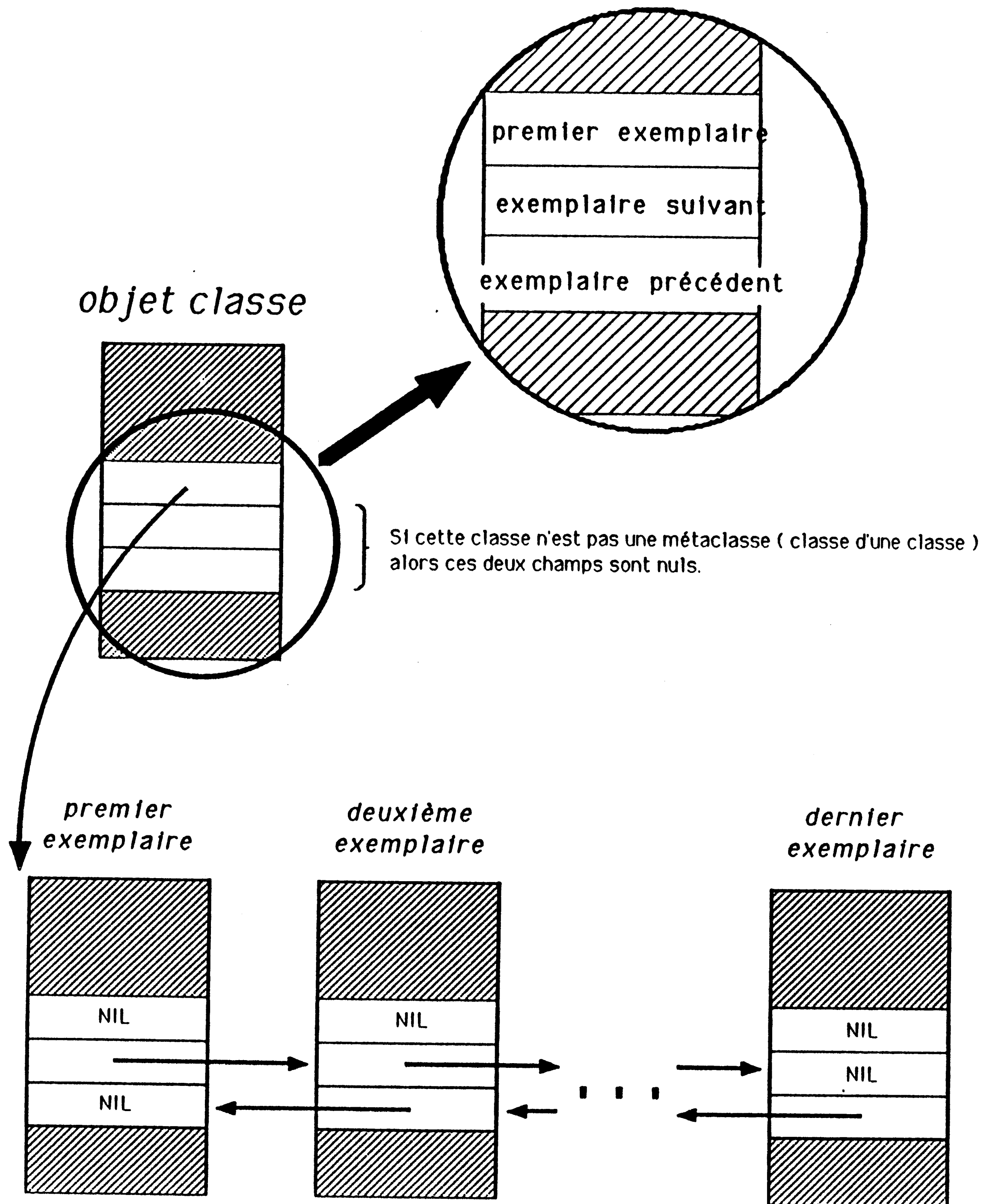
On maintient la liste des exemplaires d'une classe par leur chaînage dans une liste bi-directionnelle dont l'en-tête est détenue par la classe. Cette solution, qui est une solution temporaire, est coûteuse car elle nécessite l'ajout d'espace dans chaque descripteur d'objet pour le maintien des chaînages. En effet, la solution propre aurait été de rajouter un objet à l'information détenue par chaque objet-classe. Cet objet contiendrait la liste des exemplaires. Cependant, il serait nécessaire de modifier toute les classes de l'image virtuelle et de créer une classe *ListeDesExemplaires* pour ces objets.

On a donc rajouté trois nouveaux champs dans le descripteur de chaque objet:

premier_exemplaire: *dans le cas où l'objet est une classe, ce champ contient le pointeur de son premier exemplaire.*

exemplaire_précédent: *ce champ contient l'exemplaire précédent dans la liste des exemplaires.*

exemplaire_suivant: *ce champ contient l'exemplaire suivant dans la liste des exemplaires.*



Implémentation de la liste des exemplaires d'une classe.

4.1.5. Synchronisation sur l'entrée d'un objet.

En résumé, les informations d'une entrée sont de trois sortes:

- les informations fonctionnelles: ce sont les informations nécessaires au processus gestionnaire de la mémoire pour ses traitements (pointeur-objet, file des messages en attente).
- les informations relatives au descripteur de l'objet.
- les informations relatives au bloc-valeur.

Nous nous intéressons maintenant à la synchronisation de l'accès aux entrées.

Les différentes informations contenues dans une entrée sont consultées ou modifiées par les processus-interprète et le PGMC. Il est nécessaire de garantir leur cohérence et donc de synchroniser les processus demandeurs. Pour ce faire, il faut distinguer les types d'accès effectués par ces processus.

* les processus-interprète accèdent en lecture à: état-D, état-B, adr-descripteur et adr-bloc-valeur.

* les processus-interprète accèdent en écriture à: m-D et m-B.

* le PGMC accède en lecture et écriture à: pointeur-objet, file-messages, m-D, m-B, état-D, état-B, adr-descripteur et adr-bloc-valeur.

Un mécanisme identique à celui qui permet de contrôler l'accès au translateur semble adéquat. Notons que, si la portée du mécanisme du translateur est étendue à l'accès aux entrées, la synchronisation requise est atteinte. Toutefois, il est souhaitable d'offrir une synchronisation plus fine au niveau de chaque entrée.

4.2. Protocole d'accès à un objet.

L'accès à un objet est réalisé uniquement par un processus-interprète. Cet accès est "possible" dés que le descripteur et/ou le bloc-valeur de l'objet sont en mémoire centrale. Le conflit d'accès à l'objet peut porter sur la valeur de l'objet (bloc-valeur) ou sur son descripteur. Par exemple, il est nécessaire de garantir la cohérence du descripteur de l'objet relative aux modifications du compteur d'utilisation. Cependant, aucune hypothèse de synchronisation n'est faite dans le système Smalltalk-80 sur l'accès au bloc-valeur (la cohérence du bloc-valeur repose sur une synchronisation explicite entre les différents processus-interprète au niveau de l'application).

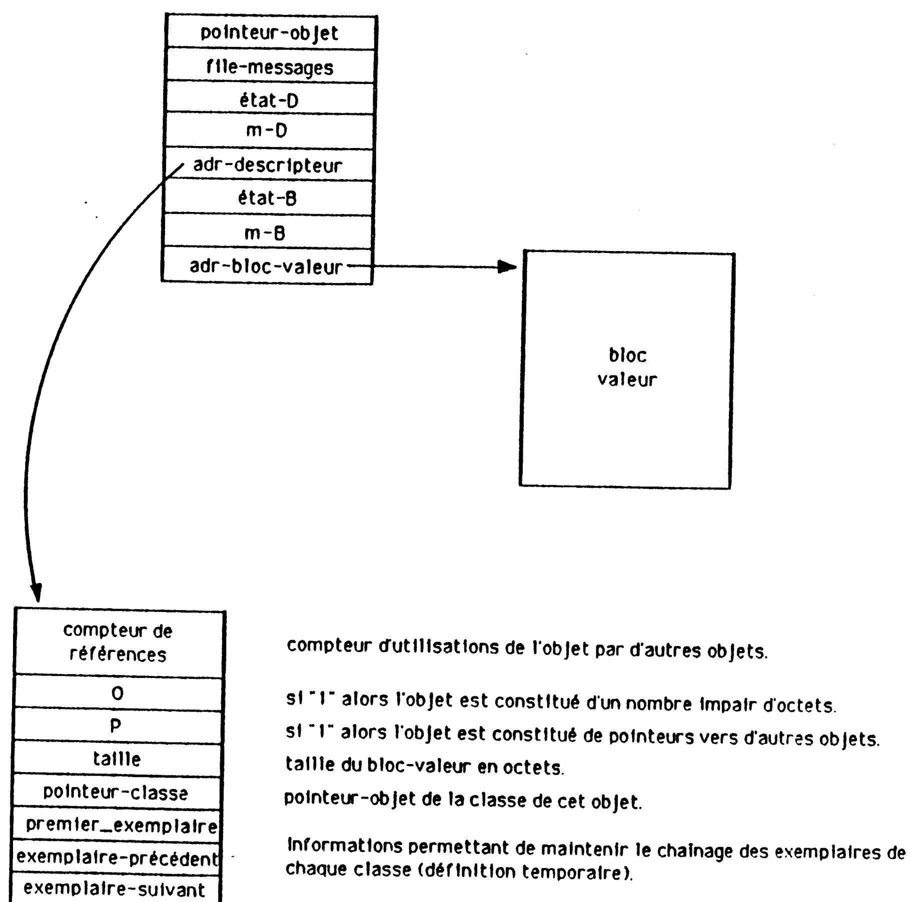
Il reste à examiner le problème de la variation éventuelle de la taille d'un objet au cours de son existence. Au niveau qui nous intéresse les objets sont de taille fixe, car Smalltalk implémente les objets dynamiques à l'aide de la primitive "swapPointersOf"

définie dans l'interface d'accès aux objets exposée dans l'annexe C. Celle-ci effectue l'échange des bloc-valeurs et des descripteurs des deux objets passés en paramètres. Cette dynamicité existe uniquement au niveau utilisateur, et non au niveau gestion des objets.

Pour faire "grandir" ou "diminuer" un objet, des méthodes sont prévues en conséquence. Si l'objet n'est pas assez grand, la méthode alloue un nouvel exemplaire un peu plus grand puis recopie les valeurs de l'objet dans le nouvel exemplaire et permute les exemplaires. L'ancienne valeur de l'objet sera récupérée par la procédure de récupération d'objets. Ce qui devrait être la modification d'un objet se ramène donc à la modification d'une entrée et par là, le problème de la cohérence d'un objet et de sa taille se ramène à celui de la cohérence d'une entrée.

4.2.1. Représentation d'un objet.

Après avoir donné toutes les caractéristiques des entités qui permettent un accès cohérent à un objet en mémoire centrale, on peut donner la forme générale de son implémentation (figure ci-après).



Composition logique d'un objet en mémoire.

4.2.2. Modifications des procédures d'accès aux objets.

Comme tout objet en mémoire centrale est accessible par un nom interne (son entrée), on suppose dorénavant que les fonctions "début-accès" retournent l'adresse de l'entrée de l'objet. Cette modification permet au processus qui a exécuté une fonction "début-accès" de manipuler l'objet en mémoire directement avec son nom interne.

Les types "pointeur-objet" et "entrée-objet" sont par exemple définis dans le langage C, avec une syntaxe très approximative, comme:

```
typedef struct
{
        unsigned reference-count : 10;
        unsigned O : 1;
        unsigned P : 1;
        unsigned long size;
        unsigned long class;
        unsigned long location;
        unsigned long premier_exemplaire;
        unsigned long exemplaire_suivant;
        unsigned long exemplaire_precedent;
}
entrée-objet;

typedef unsigned long pointeur-objet
```

Nous pouvons redéfinir les caractéristiques des fonctions "début-accès et "fin-accès":

- Accès en lecture dans le descripteur.

fonction "début-accès-lecture-descripteur" (pto : pointeur-objet) : entrée-objet
procédure "fin-accès-lecture-descripteur" (entrée : entrée-objet)

- Accès en écriture dans le descripteur.

fonction "début-accès-écriture-descripteur" (pto : pointeur-objet) : entrée-objet
procédure "fin-accès-écriture-descripteur" (entrée : entrée-objet)

- Accès en lecture dans le bloc-valeur.

fonction "début-accès-lecture-bloc-valeur" (pto : pointeur-objet) : entrée-objet
procédure "fin-accès-lecture-bloc-valeur" (entrée : entrée-objet)

- Accès en écriture dans le bloc-valeur.

fonction "début-accès-écriture-bloc-valeur" (pto : pointeur-objet) : entrée-objet
procédure "fin-accès-écriture-bloc-valeur" (entrée : entrée-objet)

- Création d'objet.

fonction "début-accès-sur-création-d'objet : (entrée-objet,pointeur-objet)
procédure "fin-accès-sur-création-d'objet" (entrée : entrée-objet)

- Accès en écriture sur le descripteur et le bloc-valeur de deux objets.

fonction "début-accès-total-en-écriture-sur-deux-objets" (pto1, pto2 : pointeur-objet) : (entrée-objet, entrée-objet)

procédure "fin-accès-total-en-écriture-sur-deux-objets" (entrée1, entrée2 : entrée-objet)

Les deux dernières procédures ont été créées pour assurer la cohérence et le bon fonctionnement de la procédure *swapPointersOf: firstPointer and: secondPointer*. Celle-ci a pour effet d'échanger les objets pointés par les pointeurs-objet *firstPointer* et *secondPointer*, et ainsi permet entre autres de réaliser la dynamicité des objets Smalltalk. Comme tout objet en mémoire centrale est représenté sous la forme d'une entrée, qui permet l'accès au descripteur et au bloc-valeur, il est facile d'effectuer la permutation des descripteurs et des blocs-informations. Après cet échange, les indicateurs de modification des descripteurs et des bloc-valeur des deux objets sont positionnés.

Nous venons de terminer l'exposé des modifications apportées aux procédures de parenthésage des accès aux objets. Celles-ci assurent la synchronisation des accès aux objets par les processus-interprète et règlent en cas de défaut le protocole d'échange PI / PGMC.
Ce sont les algorithmes déroulés par ces procédures que nous allons voir plus en détail maintenant.

4.2.3. Les algorithmes des procédures "début-accès" et "fin-accès".

Nous supposons que nous réalisons une synchronisation en exclusion mutuelle sur le descripteur et le bloc-valeur de tout objet en mémoire centrale. On considère toujours ces deux entités comme indépendantes. Pour réaliser ces exclusions mutuelles, nous disposons de deux sémaphores d'exclusion mutuelle "mutex-descripteur" et "mutex-bloc-valeur" dans chaque entrée d'objet.
- Le sémaphore "mutex-descripteur" permet d'assurer une manipulation cohérente des informations de l'entrée relatives au descripteur (état-D, m-D et adr-descripteur).

- Le sémaphore "mutex-bloc-valeur" permet d'assurer une manipulation cohérente des informations de l'entrée relatives au bloc-valeur (état-B, m-B et adr-bloc-valeur).

Toutes les entrées des objets présents en mémoire centrale sont gérées par le processus gestionnaire de la mémoire centrale (PGMC). Si celui-ci invalide l'entrée d'un objet après qu'un processus-interprète soit passé par la structure de topographie pour récupérer l'adresse de cette entrée, alors le processus-interprète ne dispose plus d'un moyen d'accès cohérent pour atteindre l'objet.

Il faut donc garantir à un processus-interprète qui exécute un accès sur un objet, que l'adresse de l'entrée de cet objet demeure valide pendant toute la durée de son accès. Ceci n'est pas une grosse contrainte si on suppose que tout processus-interprète ne peut accéder à plus de deux objets en même temps. Si on a N processus-interprète, il y aura au maximum 2N entrées que le gestionnaire de la mémoire centrale ne pourra détruire ou déplacer en mémoire centrale.

Pour réaliser cela, nous introduisons deux procédures *bloquer-entrée* et *débloquer-entrée*. La fonction *bloquer-entrée* enlève la possibilité au PGMC de pouvoir déplacer l'entrée associée à l'objet dont le pointeur-objet est passé en paramètre. Si cette l'entrée pour l'objet n'existe pas alors la fonction la construit, la bloque et renvoie son adresse. La procédure *débloquer-entrée* permet au processus-interprète qui l'exécute de relâcher le blocage qu'il a mis en place sur l'objet.

Il est à noter que si plusieurs processus-interprète bloquent la même entrée (accès simultanés aux même objet), alors le PGMC doit attendre que tous les processus la débloquent avant de pouvoir la détruire ou la déplacer.

Ces deux procédures sont spécifiées comme suit:

fonction "bloquer-entrée" (pto : pointeur-objet) : entrée-objet

procédure "débloquer-entrée" (entrée : entrée-objet)

Pour illustrer l'utilisation de ces deux procédures, nous donnons les algorithmes des procédures "début_accès_sur_création_d'objet" et "fin_accès_sur_création_d'objet" précédemment présentées.

```
fonction " debut-accès-sur-création-d'objet "
    envoyer-message-création-d'objet ( processus-courant ) ;
    message ← retirer-message ;
    message ← message.pointeur-objet ;
    entrée-objet ← bloquer-entrée ( pointeur-objet ) ;
    retourner ( entrée-objet, pointeur-objet ) ;
ffonction
```

4.3. Le processus gestionnaire de la mémoire.

4.3.1. Les fonctions du processus gestionnaire de la mémoire.

Dans cette section nous examinons plus en détail le rôle de chacune des fonctions du gestionnaire des objets en mémoire centrale (topographie, gestion de la mémoire libre, algorithme de remplacement et récupération d'objets). et, nous justifions notre choix pour la conception d'une fonction.

a) Topographie.

Le but premier de la structure de topographie est de maintenir les correspondances (pointeur-objet, adresse entrée) pour tous les objets en mémoire centrale. Cette structure de topographie subit un ensemble d'actions, de la part des processus-interprète et du processus gestionnaire de la mémoire, qui modifient son état.
Les diverses actions possibles sont:
- recherche d'une correspondance.
- création d'une correspondance.
- destruction d'une correspondance.
- modification d'une correspondance.

Outre cette gestion des correspondances, la structure de topographie doit satisfaire à d'autres exigences de la part du processus gestionnaire de la mémoire.

1- Une mémorisation des références aux objets qui est la base de travail de l'algorithme de remplacement.

2- Une mémorisation de l'état d'une correspondance. On a vu précédemment que la fonction "bloquer-entrée" assure au processus-interprète qui l'exécute que le chemin d'accès à l'objet qui sera établi ne pourra pas être modifié ou détruit.

3- Une mémorisation inverse de tout chemin d'accès. Lorsque le processus gestionnaire de la mémoire décide de recopier un objet en mémoire de masse, il est nécessaire qu'il connaisse le nom de l'objet qui occupe une zone de mémoire donnée.

4- Une mémorisation des messages de défaut d'objet que le processus gestionnaire de la mémoire ne peut satisfaire immédiatement.

De même, les processus-interprète et le processus gestionnaire de la mémoire imposent à la structure de topographie de détenir les structures de synchronisation propres à l'objet. Ces structures sont réduites à deux sémaphores d'exclusion mutuelle "mutex-descripteur" et "mutex-bloc-valeur".

b) Gestion de mémoire libre et allocation de mémoire.

Dans ce paragraphe, on présente quelques solutions possibles pour la gestion de la mémoire•centrale. Aucune solution ne s'impose a priori car nous ne disposons pas de mesures et de données statistiques du système Smalltalk-80 pour justifier un choix. Nous nous limitons donc à faire un tour d'horizon des algorithmes existants à ce jour en précisant leurs principes. Cependant, en faisant quelques hypothèses sur les objets Smalltalk, nous présentons plus particulièrement l'un d'entre-eux.

Une première classe d'algorithmes réunit toutes les stratégies qui parcourent une liste chainée des zones de mémoire libre pour satisfaire les demandes. Parmi ces stratégies, nous pouvons citer les plus connues:

- "FIRST-FIT"
La liste est parcourue jusqu'à ce que l'on trouve une zone libre dont la taille est supérieure ou égale à la taille requise. La première zone satisfaisant cette condition est fractionnée en une zone dont la taille est égale à la taille demandée et en une zone-fragment qui est rechainée dans la liste.

- "BEST-FIT"
On parcourt la liste des zones libres afin de déterminer la zone qui produira la plus petite zone-fragment. Si une zone dont la taille est égale à la taille demandée est trouvée alors la recherche est terminée.

- "WORST-FIT"
On parcourt la liste afin de déterminer la zone qui produira la plus grande zone-fragment. Si une zone de taille appropriée est découverte, on la choisit.

Tous ces algorithmes ont été développés et enrichis, et ainsi sont apparues des stratégies voisines: "BEST-FIT-LAST", "BEST-FIT-FIRST", "RANDOM-FIT", "MODIFIED-FIRST-FIT", etc...

Des synthèses des techniques de gestion de mémoire sont présentées dans [Nicholls75, Pratt75, CROCUS75, Bozman84]. Une comparaison de l'efficacité de chacun de ces algorithmes est fournie par [Bays77].

Une deuxième classe d'algorithmes réunit les stratégies qui partitionnent la mémoire libre en sous-ensembles de zones de taille fixée. Chaque sous ensemble contient toutes les zones de mémoire libre d'une taille donnée. Cet éclatement de l'ensemble des zones de mémoire libre ne constitue pas à lui seul un grand progrés par rapport à la première classe d'algorithmes. On peut tout à fait concevoir un algorithme "FIRST-FIT" avec une gestion par sous ensembles de zones libres, mais seule la vitesse d'exécution de la procédure d'allocation d'une zone serait améliorée. Cette propriété devient intéressante lorsqu'elle est accompagnée d'un accroissement de la rapidité de reformation des zones de mémoire libre. L'algorithme "BUDDY-SYSTEM" possède cette propriété. Des mesures réalisées sur les systèmes à mémoire segmentée montrent que cet algorithme est plus efficace que les algorithmes présentés précédemment [Peterson77].

Son fonctionnement est le suivant:
On recherche une zone libre dont la taille est supérieure ou égale à la taille requise. Cette recherche s'effectue dans des sous ensembles ("pool") dont les éventuelles zones libres sont de plus en plus grandes. Lorsqu'on a trouvé cette zone, elle est successivement fractionnée en zones-sœurs ("buddies"), jusqu'à ce qu'on obtienne une zone de taille égale ou immédiatement supérieure dans la fonction de fragmentation.
Une relation de la forme: $L(i) = L(i-1) + L(i-k)$ régit le fractionnement des zones.
$L(i)$ est la taille de la zone fragmentée.
$L(i-1)$ et $L(i-k)$ sont les tailles des "buddies".

Il est très important de remarquer que par la fonction de fragmentation, la décomposition d'une zone en "buddies" est unique. Par conséquent, la recomposition de la zone initiale à partir des "buddies" est possible et unique.

Nous présentons maintenant quelques exemples de "BUDDY-SYSTEM":

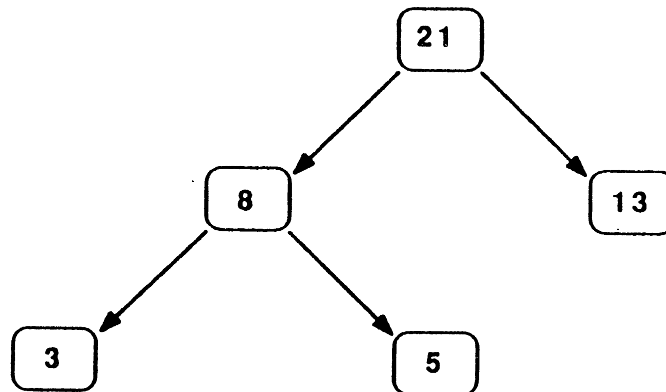
k=1 "BINARY-BUDDY-SYSTEM". $L(i) = 2 * L(i-1)$

Cela revient à fractionner les zones en deux "buddies" de taille moitié.
Les tailles disponibles avec cette loi de fragmentation sont:
1 2 4 8 16 32 64 128 256 512 ......

k=2 "FIBONACCI-BUDDY-SYSTEM". $L(i) = L(i-1) + L(i-2)$

Les zones sont fractionnées suivant les valeurs des nombres de Fibonacci.
Les premières valeurs sont:
1 1 2 3 5 8 13 21 34 55 79 ......

Exemple de fractionnement par l'algorithme:
On désire allouer une zone de taille 4. La plus petite des tailles des zones libres supérieure ou égale à 4 est de 21.
Cette zone est fractionnée comme suit:



La zone de taille 5 est allouée.

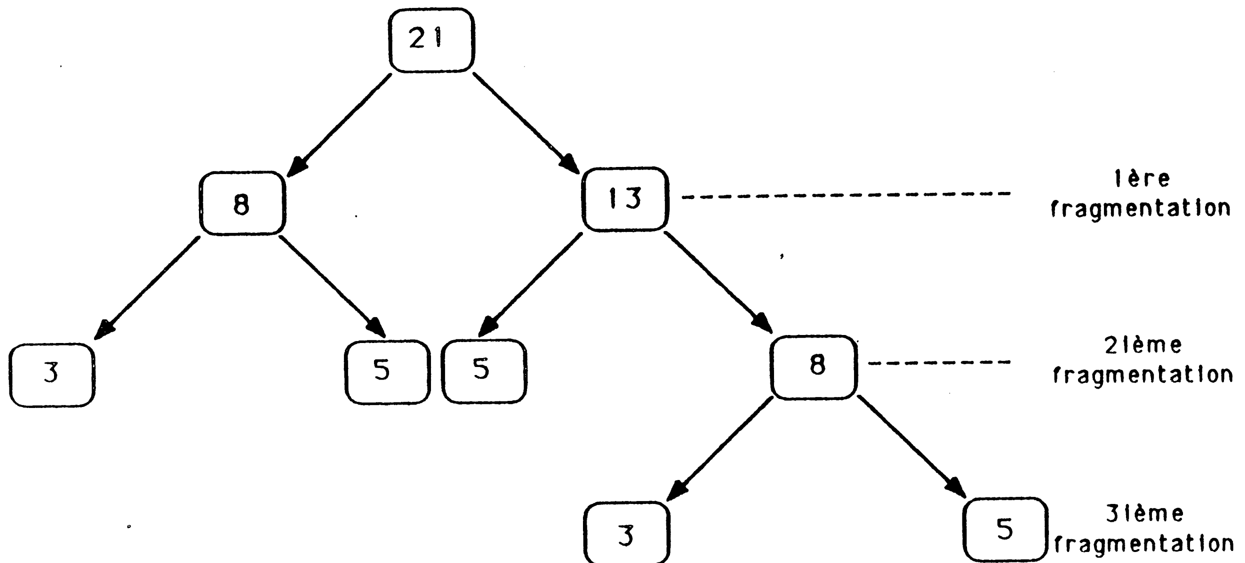
On voit très bien que la recomposition des zones est une fonction déterministe (pas de choix pour recomposer les zones).

Loi de recomposition:
Des zones contiguës de tailles A et B (A<B) sont recomposables si et seulement si il existe L (i) tel que:

$$L(i) = A + B \text{ et } A = L(i-1) \text{ et } B = L(i-2)$$

Par conséquent, des zones voisines de tailles respectives L (i-2) et L (i-1) ne sont pas recomposables.

exemple:



Les deux zones consécutives de tailles (5,3) ne sont pas recomposables. Le couple recomposable est (3,5).

L'efficacité d'un algorithme d'allocation de mémoire est évalué à l'aide de la mesure de deux facteurs:
- La fragmentation externe qui caractérise l'incapacité à satisfaire des demandes de zones plus importantes du fait que la mémoire libre est constituée d'un ensemble de zones de tailles insuffisantes.
- La fragmentation interne qui caractérise l'inaccessibilité de la mémoire inutlisée appartenant à des zones allouées de tailles supérieures aux tailles qui étaient demandées.

Si la fragmentation externe peut être évitée par des tassages plus ou moins fréquents de la mémoire centrale, la fragmentation interne ne peut être évitée. La fragmentation est implicitement liée à la stratégie d'allocation des zones de mémoire libre.

Exemple.
La suite des L(i) étant (en multiples de la taille du grain):
1 1 2 3 5 8 13 21 34 55 89 ......

Si on veut satisfaire une demande de taille X, on allouera une zone de mémoire telle que sa taille L (i) vérifie: L (i-1) < X <= L (i)
La différence L (i - X) est inutilisable.

Remarque.
On s'aperçoit que plus l'ordre k d'un "Buddy-system" est grand, plus les termes engendrés par la relation sont voisins les uns des autres. La suite des termes fournit donc une plus grande palette de tailles pour des ordres de plus en plus grands.

k=1: 1 2 4 8 16 32 64 128 256 ....
k=2: 1 1 2 3 5 8 13 21 34 ....
k=3: 1 1 1 2 3 4 6 9 13 ....
k=4: 1 1 1 1 2 3 4 5 7 ....
et ainsi de suite...

proposition:
Comme les objets Smalltalk sont de petite taille [Bézivin84], (environ 60% des objets Smalltalk ont une taille comprise entre 3 et 15 mots mémoire), un "Buddy-system" d'ordre relativement élevé (k= 10 ou 15) fournirait une palette assez complète des tailles de façon à minimiser la fragmentation interne.

Dans les conditions usuelles d'utilisation d'un système Smallatlk-80, un "Buddy-system" est un bon choix pour l'algorithme d'allocation de mémoire, car il cumule plusieurs avantages:
- fragmentation interne réduite (ordre k élevé).
- exécution plus rapide que les autres algorithmes (First-fit, Best-fit, ...) pour l'allocation et pour la libération (reformation éventuelle de blocs).
- fragmentation externe peu importante.

L'algorithme d'allocation de mémoire suivant le principe du "Buddy System" a été énoncé par [Knowlton65]. Le "Fibonacci Buddy System" a été étudié par [Hirschberg73]. La famille des algorithmes de gestion de mémoire à l'aide de "Buddy System" fait l'objet d'une bonne synthèse dans [Shen74, Peterson77].

c) Algorithme de remplacement.

Dans le système Smalltalk-80, l'algorithme de remplacement a pour objectif la désignation du ou des objets qui sont à recopier en mémoire de masse afin de libérer de la place en mémoire centrale. Ces objets peuvent être ou non des objets appartenant à l'environnement du processus-interprète pour lequel il faut libérer de la place en mémoire centrale.

Comme nous manquons de données statistiques sur le fonctionnement du système Smalltalk-80, nous nous contentons d'énumérer quelques algorithmes en précisant leurs aspects intéressants. L'algorithme de remplacement doit choisir un objet qui "ne risque pas d'être référencé dans un délai raisonnable". Malheureusement, le mieux que l'on puisse faire est d'extrapoler le futur en fonction du passé. Parmi les divers algorithmes de remplacement, on distingue les algorithmes dits "à pile" et ceux qui ne le sont pas. Un

algorithme est dit "à pile" si après toute référence, tout objet présent dans une mémoire pouvant contenir M objets est présent dans une mémoire pouvant contenir M' objets (M'>M). Les algorithmes de remplacement sont présentés et étudiés dans [Denning70, Knuth68, Denning73].

1- Les algorithmes "ordinaires".

Ces algorithmes n'utilisent pas d'information sur l'usage des objets.

" FIFO ". (First in, first out)
On remplace l'objet le plus anciennement chargé. Le surcoût provient de l'enregistrement de l'ordre de chargement des objets, mais il ignore la possibilité que l'objet le plus ancien puisse être aussi le plus utilisé.

" RANDOM ".
On choisit aléatoirement l'objet à décharger. Cette solution n'induit pas un surcoût, mais c'est un algorithme peu intéressant.

2- Les algorithmes à pile.

L'avantage des algorithmes à pile est de permettre la prévision du comportement d'un processus (nombre de défauts d'objets) dans une mémoire de taille M' si l'on connait son comportement dans un mémoire de taille M (M'>M).
Avec un tel algorithme, un même processus provoque de moins en moins de défauts d'objets, si son espace est de plus en plus grand.

" LRU " (Least Recently Used)
On remplace l'objet le moins récemment utilisé. Le surcoût de cette solution résulte de l'enregistrement de la succession des accès aux objets.

" LFU " (Least Frequently Used)
On remplace l'objet le moins fréquemment utilisé. Le surcoût de cette solution résulte également de l'enregistrement de la succession des accès aux objets.
Il faut noter que pour implémenter un algorithme "LFU", il faut éviter l'erreur qui consiste à remplacer à tort un objet récemment chargé car il possède un faible taux d'utilisation.

Ayant fait le tour des algorithmes de remplacement "de base", il se pose un problème: sur quel ensemble de mémoire, cet algorithme doit-il s'appliquer ?

- sur toute la mémoire centrale: l'algorithme de remplacement s'appliquerait à tous les objets présents en mémoire quels que soient les processus qui les référencent. Ceci est sans nul doute la solution la plus simple à réaliser.

- sur les objets référencés par chaque processus: dans ce cas, comment peut-on décider du ou des objets à retirer de la mémoire centrale ? Doit-on les retirer au processus qui a fait le défaut d'objet ou à un des autres processus du système ? Il faut introduire un algorithme de choix qui détermine ce processus.
L'algorithme suivant choisit un processus en fonction de sa fréquence de défaut d'objet:

3- L'algorithme de remplacement " PFF " [Chu76]: (Page Fault Frequency):

Sur un défaut d'objet, si la fréquence de défaut d'objet est supérieure à une fréquence critique alors la place nécessaire au chargement de l'objet est allouée en plus de la place déjà occupée par les autres objets de ce processus. La place est prise soit dans la zone de mémoire libre, soit à un autre processus en lui retirant des objets.
Si la fréquence de défaut d'objet est inférieure à la fréquence critique alors on enlève des objets au processus qui a fait la faute. La fréquence critique est déterminée statistiquement.

Cet algorithme n'est pas intéressant pour le système Smalltalk-80 car le niveau de partage des objets est apparemment important. De plus, rien n'est actuellement fait sur la protection des objets dans le système. Un progrès souhaitable est justement la possibilité de protéger des objets du système et des processus. Cependant, une protection "minimum" est assurée par la définition même de Smalltalk-80: (message, classe, exemplaire de classe).

On devra concevoir un algorithme de remplacement travaillant sur l'ensemble des objets en mémoire centrale sans distinction d'appartenance à tel ou tel processus. Plusieurs études devront être faites pour déterminer le meilleur algorithme; des algorithmes peuvent être bons, voire très bons pour certaines classes de problèmes et se révéler mauvais pour d'autres classes de problèmes.

d) Algorithmes de récupération d'objets.

Dans le système Smalltalk-80 à un seul niveau de mémoire, l'algorithme de récupération d'objets est réalisé par deux algorithmes distincts:
- un algorithme de récupération d'objets par compteur d'utilisations.
- un algorithme de récupération d'objets par marquage .
Le problème est de reformuler ces deux algorithmes pour une mémoire à deux niveaux.

1- Récupération d'objets par compteur d'utilisations.

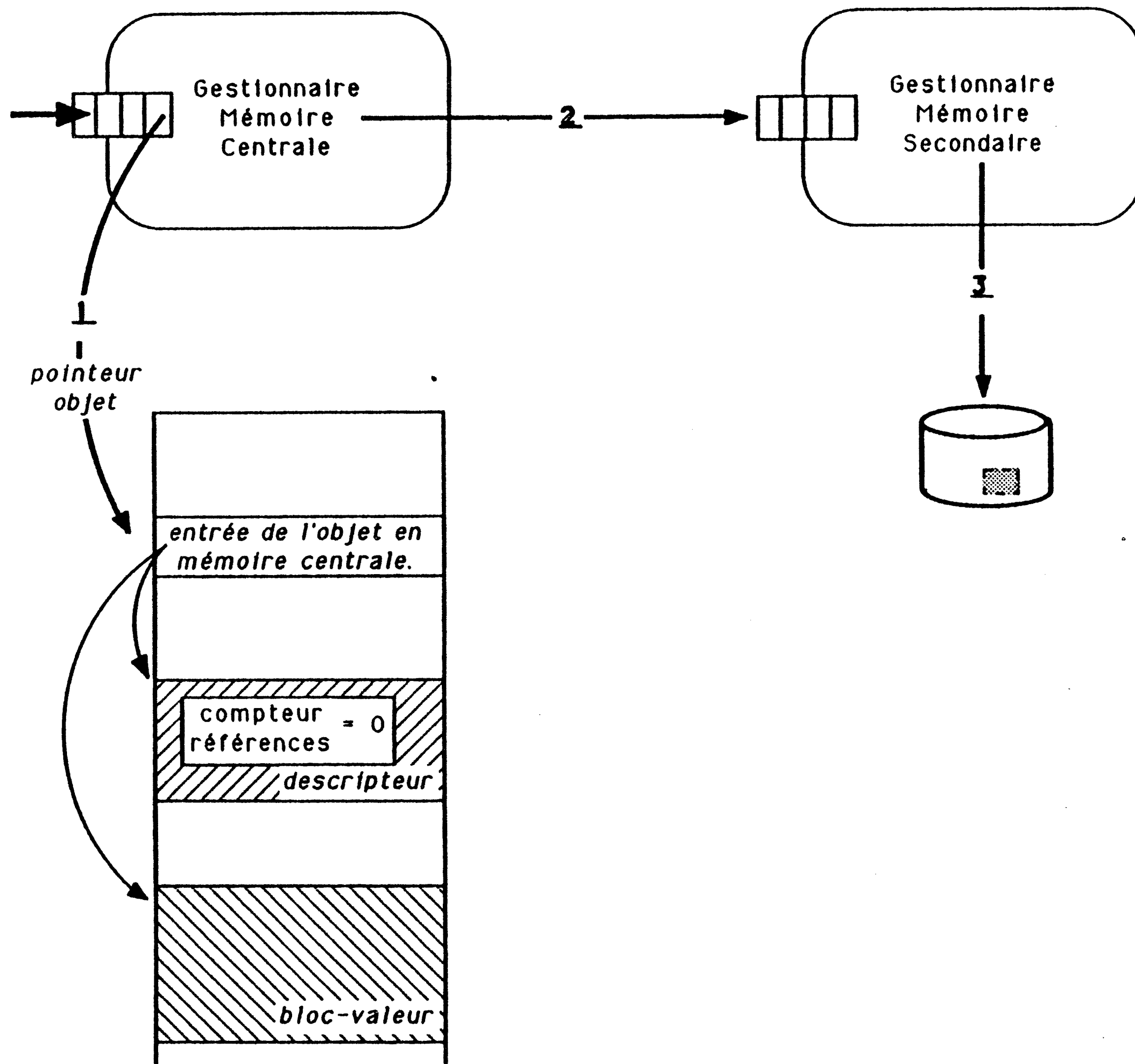
principe:

Chaque objet possède un compteur d'utilisations qui exprime le nombre des objets qui référencent cet objet. Lorsque ce compteur devient nul, l'objet n'est plus référencé et peut donc être récupéré [Bobrow80, Christophe84, Goldberg83].

La nullité du compteur d'utilisations d'un objet est détectée par la procédure "countDown:" qui effectue l'opération de décrémentation du compteur d'un objet. Lors de l'affectation d'une valeur à un champ-pointeur d'un objet, cette décrémentation s'effectue sur l'objet désigné par l'ancienne valeur du champ, alors qu'une incrémentation s'effectue sur l'objet désigné par la nouvelle valeur.

Si le compteur d'utilisation devient nul, il faut décrémenter tous les compteurs d'utilisation des objets atteignables à partir de cet objet. Tous les objets parcourus dont le compteur de références devient nul sont récupérés. La procédure "countDown" est donc une procédure récursive qui, le cas échéant, a pour effet de récupérer l'espace mémoire occupé par un sous-graphe d'objets.

Pour récupérer un objet, le processus gestionnaire de la mémoire rend libre son espace mémoire centrale et renvoie au gestionnaire des objets du système la valeur du pointeur-objet qui est désormais libre. Celui-ci répercute la destruction de l'objet sur l'espace permanent des objets. Le gestionnaire des objets du système possède une boîte aux lettres pour recevoir les messages de récupération de pointeurs-objet.



Légende:

1 Après qu'un processus-interprète ait détecté la nullité du compteur de références d'un objet et qu'il l'ait signalé au gestionnaire mémoire centrale, celui-ci procède à sa récupération. Il détruit la représentation de l'objet en mémoire centrale.

2 Puis il envoie un message au gestionnaire mémoire secondaire afin qu'il répercute aussi la récupération sur la mémoire de stockage des objets.

3 Le gestionnaire de la mémoire secondaire libère l'espace de mémoire secondaire qu'occupait l'objet détruit. C'est sur l'espace de mémoire secondaire que sont reportés le problème de récupération des trous créé par les destructions d'objets.

Protocole de récupération d'objets par compteur d'utilisations.

2- Récupération d'objets par marquage.

Principe:

Cett récupération d'objets s'effectue sur la mémoire de masse, une fois que tous les objets y ont été éventuellement mis à jour. On propage une marque à partir de la racine du système. Tous les objets que l'on peut atteindre à partir de la racine sont marqués et les objets non marqués sont récupérables [Thorelli72, Goldberg83, Thatte86]. Cette opération doit être réalisée par le gestionnaire des objets du système (PGMS) car il s'agit de détruire les circuits qui n'ont pu être détectés par l'algorithme basé sur la nullité des compteurs d'utilisations.

L'exécution de cet algorithme peut être déclenchée en deux circonstances très différentes:
- En fin de session de travail.
- En cours de session si une ressource du MGMS est en train de se tarir (espace mémoire de masse, ensemble des pointeurs-objets allouables). Dans ce deuxième cas, nous devons interrompre le déroulement de la session en cours et sauvegarder l'état de la mémoire centrale avant de purger la mémoire de masse.

Nous pouvons supposer qu'il sera mis en place un dispositif d'arrêt des processus-interprète et du processus gestionnaire de la mémoire qui pourra être actionné par le gestionnaire des objets du système. A partir du moment où l'arrêt des activités est effectif (recopies des objets modifiés et sauvegarde de la liste des objets présents en mémoire centrale au moment de l'arrêt), nous nous retrouvons dans le cas où il n'y a plus d'activité qui modifie les objets. L'ensemble du système en mémoire de masse est donc dans un état stable.
Après l'exécution de l'algorithme par marquage, la mémoire centrale doit être rechargée à l'aide de la liste des objets qu'on a précédemment sauvegardée. Enfin toutes les activités qui ont été suspendues sont réactivées.

Conclusion:

Cet algorithme peut paraître lourd, mais normalement, il ne doit être exécuté que très rarement. Les algorithmes de récupération d'objets par compteur d'utilisation et par marquage de fin de session de travail contribuent à rendre l'exécution d'une récupération d'objets par marquage en cours de session aussi rare que possible. Le déclenchement de cette dernière peut provenir de la nature même des programmes exécutés par l'utilisateur: un programme qui manipule beaucoup de listes circulaires d'objets est plus disposé à en provoquer l'exécution qu'un programme qui n'en manipule pas. Malgré tout, on peut admettre que l'exécution d'une session de marquage sur l'espace des objets, en cours de session, est très peu probable.

Quelques algorithmes se distinguent des deux algorithmes présentés. Par exemple, c'est le cas d'un algorithme fondé sur une gestion des objets par "génération" (sous-ensemble d'objets qui ont résisté à plus ou moins de phases de récupération) [Ungar84a]. Cet algorithme est présenté dans le chapitre suivant.

On peut également citer d'autres algorithmes tels que ceux développés dans [Dijkstra78, Deutsch76, Lieberman83].

Une bonne synthèse de ces deux algorithmes, ainsi que quelques variantes de ceux-ci, a été réalisée à l'Université de Rennes [Michel86]. Cette étude présente en outre les algorithmes de récupération d'objets en milieu réparti.

4.3.2. Structures de données et mécanismes de base.

Dans cette section, nous présentons les structures de données et les quelques procédures qui permettent de représenter les objets en mémoire centrale et de résoudre les défauts d'objets.

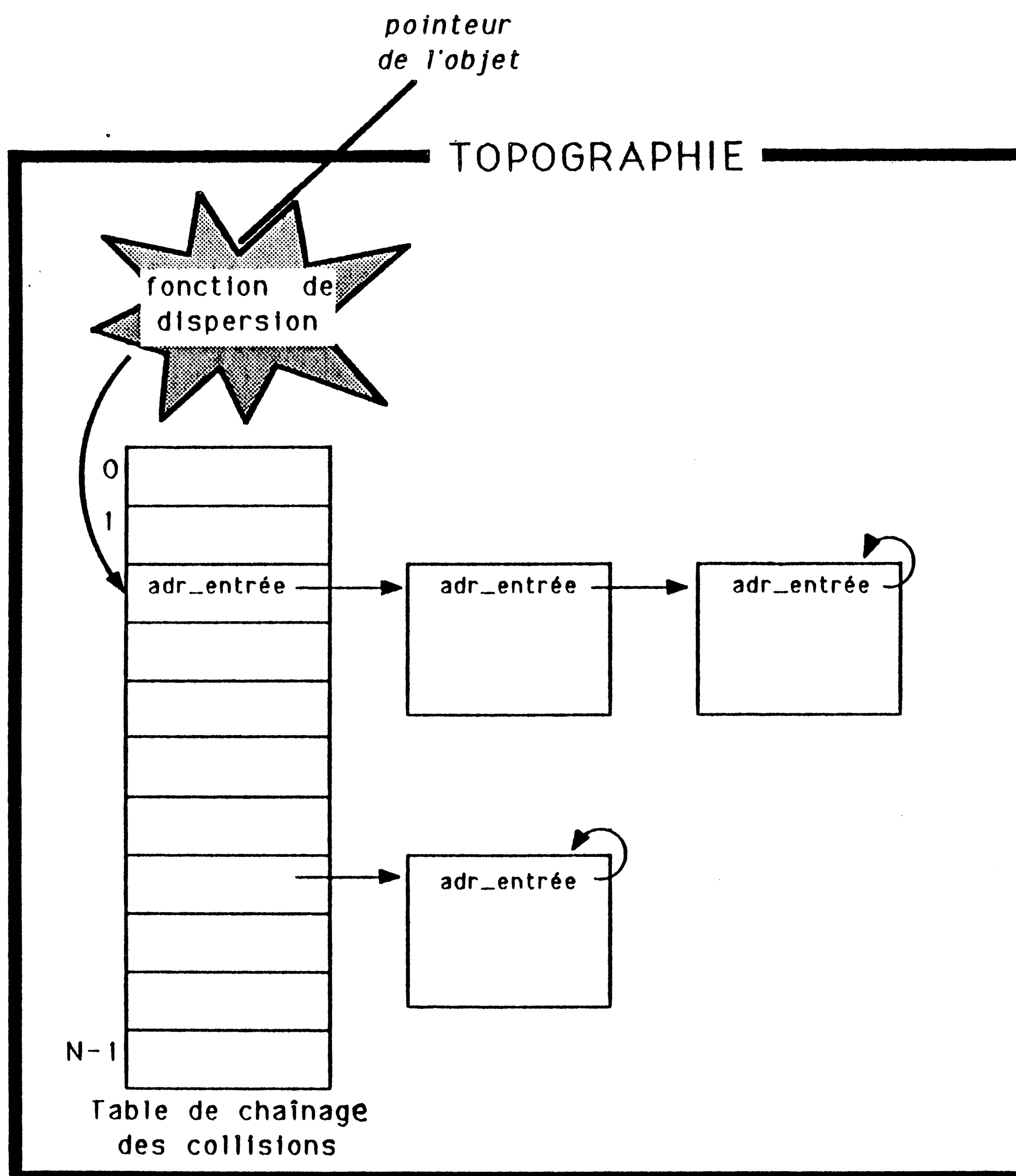
1- La structure de topographie.

La topographie est la structure de données qui maintient les correspondances (pointeur-objet → adresse mémoire) via des entrées.

Nous proposons une mise en œuvre de cette structure. Cette solution est seulement donnée à titre d'exemple et ne s'appuie en aucun cas sur des mesures du système . On réalise la structure de topographie par une fonction de hachage suivant la valeur du pointeur-objet. On dispose d'une table à N entrées et la valeur de la fonction de dispersion est calculée comme suit:

hachage (pointeur-objet) = (val (pointeur-objet) mod N) + 1

On résout les collisions par un chaînage des entrées des objets.



La structure de topographie.

L'indice d'entrée dans la table ne suffit pas pour repérer un objet dans la topographie et on est obligé de répéter la valeur du pointeur-objet dans l'entrée qui est allouée à l'objet. Cette information est également nécessaire au gestionnaire de la mémoire pour connaitre l'identité de l'objet d'une entrée donnée.

Si le processus gestionnaire de la mémoire désire récupérer une entrée, il doit remettre le chainage à jour. Un chainage arrière est donc obligatoire. Les entrées sont dorénavant doublement chainées dans la structure de données représentant la topographie.

2- Structure complète d'un objet en mémoire centrale.

En plus des informations servant à localiser l'objet en mémoire centrale d'autres informations sont nécessaires pour assurer la gestion des données propres à l'objet.

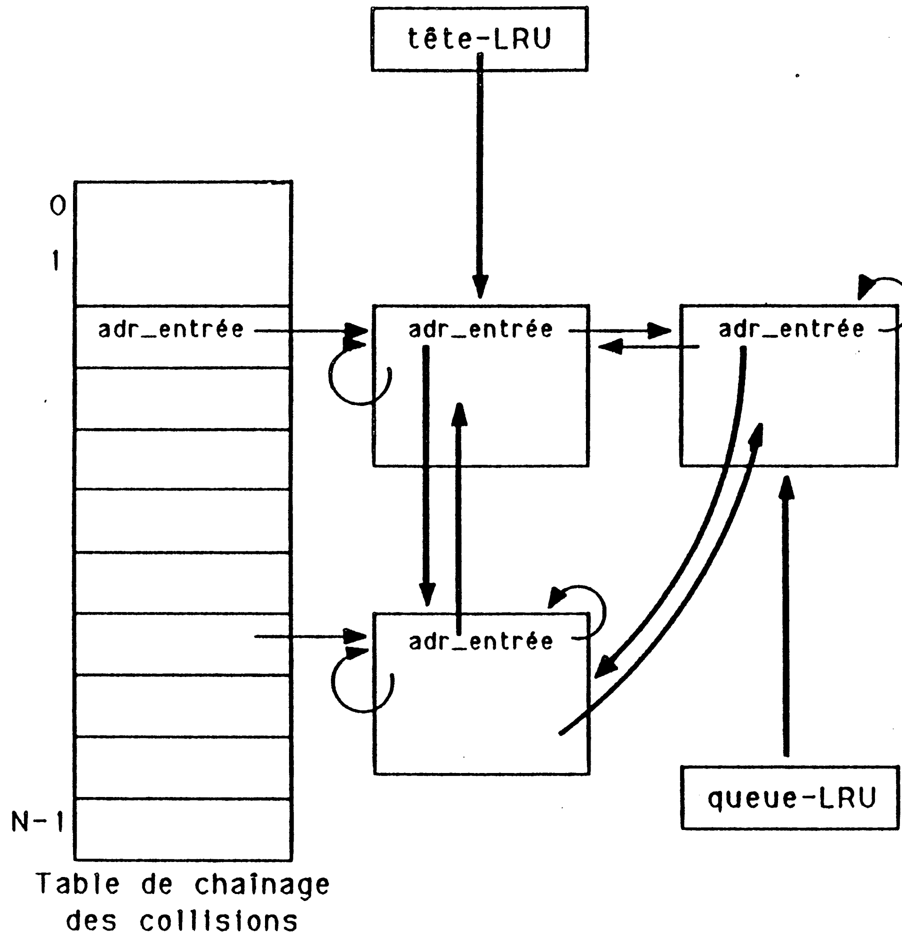
* Deux sémaphores: "mutex-descripteur" et "mutex-bloc-valeur". (rappel)
Ces deux sémaphores assurent l'accès en exclusion mutuelle aux informations de l'objet.

* Une file des messages en attente sur l'objet: "file-messages". (rappel)

* Une variable de blocage: "blocage"
Cette variable représente le nombre de processus-interprète qui ont bloqué l'objet. Un objet bloqué ne peut être détruit par le gestionnaire de la mémoire, ainsi qu'un objet qui est en cours de transfert.

* De plus, on introduit la structure de données nécessaire à l'algorithme de remplacement. On choisit un algorithme "LRU". Il suffit d'ajouter un double chaînage qui ordonne totalement les entrées.
Une entrée référencée est remontée en tête de "LRU". Les entrées que le gestionnaire peut éventuellement récupérer sont situées en queue de la liste.

La version finale de la structure de topographie et du classement "LRU" est donnée ci-après.



Topographie et classement "LRU".

Les informations nécessaires à une bonne gestion des objets en mémoire centrale peuvent être classées suivant leurs fonctions en deux groupes distincts. Ces informations comprennent toutes celles permettant de localiser l'objet, de gérer ses accès et de connaître son état de modification par rapport à sa copie en mémoire permanente.

i- Les informations de gestion de la mémoire.

chaînage des collisions	Informations permettant de chainer les objets dans la structure de topographie.
chaînage LRU	Informations permettant de chainer les objets dans la liste LRU.
pointeur de l'objet	Nom de l'objet représenté par cette entrée.
file des messages en attente	Liste des messages de processus interprète en défaut sur cet objet.
variable de blocage	Variable permettant de bloquer un objet en mémoire centrale. Un objet bloqué n'est plus ni relogeable, ni déchargeable.
mutex-descripteur	Informations permettant de manipuler correctement le descripteur de l'objet.
mutex-bloc-valeur	Informations permettant de manipuler correctement le bloc-valeur de l'objet.

ii- Les informations d'état de l'objet en mémoire centrale.

état-D	*m-D*	*adr-descripteur*	*état-B*	*m-B*	*adr-bloc-valeur*

Un sémaphore d'exclusion mutuelle "mutex-topographie" permet de contrôler l'accès à l'ensemble de la structure: topographie + LRU + entrées.

```
procédure debut-synchronisation-topographie
    P ( mutex-topographie ) ;
fprocédure

procédure fin-synchronisation-topographie
    V ( mutex-topographie ) ;
fprocédure
```

Nous donnons maintenant les algorithmes de la fonction "bloquer-entrée" et de la procédure "débloquer-entrée" qui sont utilisées par les fonctions et procédures "début-accès" et "fin-accès".

```
fonction bloquer-entrée ( pointeur-objet )
   iterer
      debut-synchronisation-topographie ;
      ( cle, entree-objet ) ← topographie (pointeur-objet ) ;
   jusqua ( entree-objet ≠ entree-nil )
      fin-synchronisation-topographie ;
      envoyer-message-defaut-sur-objet (pointeur-objet, lecture-descripteur) ;
      message ← retirer-message ;
   fiterer
   mettre-objet-en-tête-LRU (entree-objet) ;
   entree-objet.blocage ← entree-objet.blocage + 1 ;
   fin-synchronisation-topographie ;
   retourner ( entree-objet ) ;
ffonction

procédure debloquer-entrée (entree-objet)
   debut-synchronisation-topographie ;
   entree-objet.blocage ← entree-objet.blocage - 1 ;
   fin-synchronisation-topographie ;
fprocédure
```

Ayant précisé la structure de topographie, nous donnons pour indication les algorithmes des fonctions "topographie" et "test-et-création-topographie". La fonction "topographie" (pointeur-objet) → (clé, entrée-objet) retourne la clé de la fonction de hachage et l'adresse de l'entrée associée à l'objet dont le pointeur-objet est passé en paramètre. Si l'objet n'est pas présent en mémoire, la valeur entrée-nil est retournée.

```
fonction topographie (pointeur-objet)
    cle ← ( val (pointeur-objet) mod N ) + 1 ;
    entree ← table[cle] ;
    tantque ( entree ≠ entree-nil ) faire
        si ( entree.pointeur-objet = pointeur-objet )
            alors retourner ( cle, entree ) ;
            sinon entree ← entree.suivant ;
        fsi
    ftantque
    retourner ( cle, entree-nil ) ;
ffonction
```

La fonction test-et-création-topographie (pointeur-objet) → (entrée-objet) teste l'existence de l'entrée associée à l'objet dont le pointeur-objet est passé en paramètre. Si cette entrée existe alors cette fonction retourne son adresse, sinon elle essaie de la construire et retourne son adresse, ou entrée-nil si elle n'a pas pu la construire.

```
fonction test-et-creation-topographie (pointeur-objet)
    ( cle, entree ) ← topographie (pointeur-objet) ;
    si ( entree = entree-nil )
        alors
        entree ← allouer-zone-memoire ( taille-entree ) ;
        si ( entree ≠ entree-nil )
            alors
            chaîner-entree-dans-topgraphie ( entree, cle, pointeur-objet ) ;
            initialiser-chaînage-LRU ( entree ) ;
            entree.pointeur-objet ← pointeur-objet ;
            entree.blocage ← 0 ;
            entree.mutex-descripteur ← 1 ;
            entree.mutex-bloc-valeur ← 1 ;
            entree.etat-D ← absent ;
            entree.etat-B ← absent ;
        fsi
    fsi
    retourner ( entree ) ;
ffonction
```

4.3.3. Stratégie de gestion mémoire: "un gestionnaire plus adapté".

Au lieu de considérer le processus gestionnaire de la mémoire comme un mécanisme qui se contente de déclencher des actions sur demande, on peut le concevoir comme pouvant anticiper l'action des processus-interprète d'après des stratégies prédéfinies. De cette façon, le processus gestionnaire de la mémoire peut entreprendre certaines actions pendant qu'il n'a pas de messages à traiter, ou il peut même délaisser le traitement des messages pour ramener la mémoire dans un état qu'il juge plus satisfaisant.

Ces actions peuvent être entreprises selon le degré d'encombrement de la mémoire. On a alors un compromis tassage-déchargements à établir pour permettre de maintenir un taux d'espace mémoire libre à peu près constant. Des actions de rafraichissement périodique des objets en mémoire centrale peuvent aussi être envisagées de façon à gagner du temps au moment de la récupération d'objets [Thatte86].

De même, le processus gestionnaire de la mémoire pourrait faire du préchargement d'objets. La connaissance sur laquelle s'appuierait cette action est des plus floue. Cette action pourrait être mis en œuvre à partir de la connaissance fine de l'évolution des références ou d'une connaissance statistique du système.

Le processus gestionnaire de la mémoire pourrait également faire un choix sur les requêtes à traiter. Un processus-interprète qui ferait trop de fautes d'objet pourrait être délaissé pendant un certain temps.

5. Améliorations possibles.

Cette section présente trois aspects suivants lesquels les performances du module de gestion d'objets centralisé développé peuvent être améliorées. En effet, s'il est incontestable que le gestionnaire d'objets développé permet de se libérer de la contrainte imposée par la limitation de l'espace disponible en mémoire centrale, il est néanmoins à déplorer que les performances des accès aux objets sont sérieusement dégradées. L'ensemble des trois améliorations proposées constitue une évolution progressive de la conception initiale vers une gestion des objets plus efficace.

La première solution développée essaie de généraliser le principe de blocage d'objet, mis en place sur chaque accès élémentaire. Un objet bloqué ne peut pas être ni enlevé de la mémoire centrale, ni déplacé Ceci revient à conserver dans un état bloqué tous les objets constituant l'environnement d'exécution immédiat de chaque processus-interprète.

La deuxième amélioration consiste à mettre en place des court-circuits pour accélérer l'accès aux objets constituant l'environnement immédiat de chaque processus interprète. Ceci revient non seulement à bloquer ces objets, mais aussi à les accéder par leur adresse physique, et non plus par l'intermédiaire d'une désignation logique (le pointeur-objet).

Enfin, la troisième proposition qui est la plus complète, permet de conserver un accès rapide à tout objet tant que celui-ci n'a pas été déplacé en mémoire centrale par le gestionnaire. Cette solution augmente fortement la durée pendant laquelle un objet est accessible par l'intermédiaire de son adresse physique en mémoire centrale. Cette proposition est doublement intéressante puisqu'elle permet d'implémenter, avec la même information, le chemin d'accès à l'objet et une de ses protections élémentaires (lecture et écriture).

5.1. Fixation d'un objet.

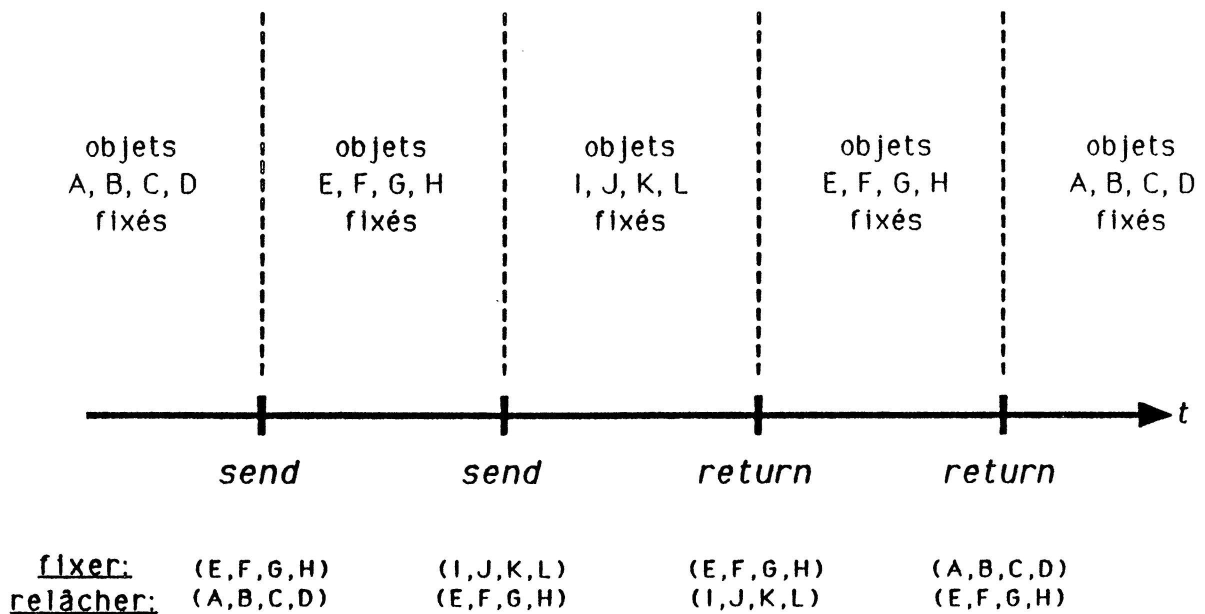
L'entrée d'un objet est bloquée en mémoire centrale pendant toute la durée de l'accès à l'objet par les processus-interprète qui utilisent les procédures de l'interface interprète/gestionnaire de la mémoire centrale. Le temps pendant lequel l'entrée de l'objet est bloquée est relativement court car il correspond à un accès élémentaire à l'objet telle que la modification d'un champ du bloc-valeur ou du descripteur de l'objet.

Si le déchargement d'un objet en mémoire de masse, entre deux accès, est trop fréquent, cela peut coûter fort cher en temps d'attente de service disque ("thrashing"). Pour limiter ce problème, on augmente le temps de résidence de l'objet en mémoire centrale. L'objet est dit **fixé**.

La fixation d'un objet en mémoire centrale n'a de sens que pour les objets fréquemment accédés. Un objet fixé ne peut être vidé de la mémoire centrale. Par conséquent il faut prévenir l'engorgement de la mémoire centrale, sous peine de priver le processus gestionnaire de la mémoire de toute liberté de gestion. La solution consiste à n'autoriser le droit de fixation que sur des objets sujets à de nombreuses références et tels que le nombre de ceux-ci soit restreint. Par exemple, on peut décider de fixer les objets qui composent l'environnement immédiat d'exécution des processus-interprète.
Nous fixons donc les objets pointés par les registres Smalltalk:

- activeContext
- homeContext
- method
- receiver

Ces objets sont fréquemment utilisés au cours du temps, et d'autre part leur nombre est tout à fait raisonnable (quatre par processus-interprète). A chaque traitement d'un octet-code, "send" ou "return", les objets pointés par les registres sont relâchés et quatre nouveaux objets sont fixés. La figure suivante illustre les modifications des registres et l'évolution de l'ensemble des objets fixés.



Exemple de fixation d'objets lors d'envois de messages.

Nous pouvons mettre en évidence les modifications préconisées et les gains obtenus par l'ajout des procédures "fixer" et "relâcher", en comparant une série de N accès sans fixation de l'objet et une série de N accès où l'objet est fixé au premier accès.

1er cas: N accès consécutifs sans fixation de l'objet.

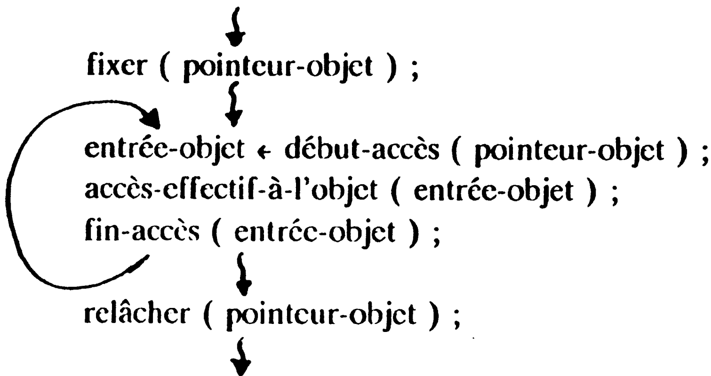
Tous les accès sont réalisés à l'aide des procédures "début-accès" et "fin-accès" suivant le schéma:

entrée-objet ← début-accès (pointeur-objet) ;
accès-effectif-à-l'objet (entrée-objet) ;
fin-accès (entrée-objet) ;

Si on réalise N accès consécutifs au même objet, on exécute N fois la procédure "début-accès" et cela peut même entraîner N fautes d'objets successives.
Il faut pourtant remarquer que la procédure "début-accès" possède deux fonctions:
- synchroniser l'accès à un objet avec d'autres accès extérieurs.
- fixer l'objet pendant l'accès.(temps court)

Dans l'exemple suivant, on introduit une procédure "fixer" qui a pour rôle de fixer l'objet pour un temps plus long. La fonction "début-accès" n'a plus qu'une fonction de synchronisation de l'accès avec d'éventuels accès concurrents.

2ème cas: N accès consécutifs avec fixation de l'objet.

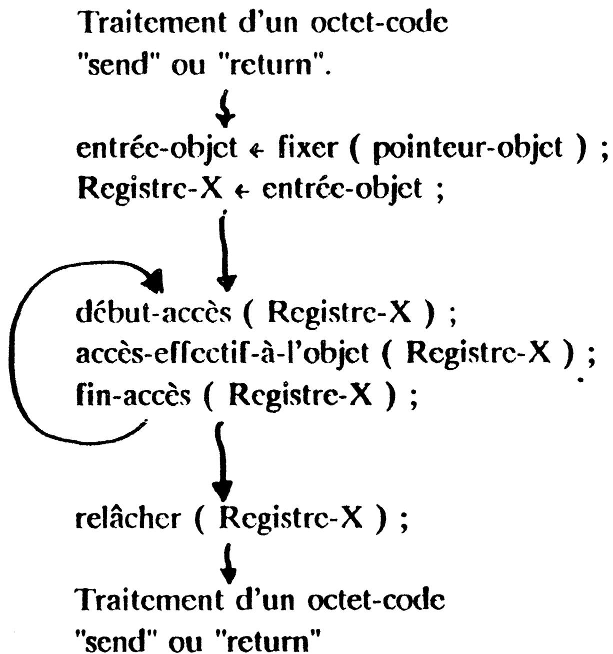


Pour N accès consécutifs, on n'exécute qu'une seule fois la procédure "fixer". Cela ne peut entraîner qu'une seule faute d'objet. On peut toutefois éviter les N passages par la structure de topographie par l'exécution répétée de la procédure "début-accès", en manipulant l'objet par son entrée après qu'il ait été fixé.

5.2. Court-circuit d'accès aux objets par les registres.

Le principe consiste à ne plus accéder à un objet fixé à l'aide de son pointeur-objet (adresse virtuelle), mais à l'accéder par son adresse physique. Ces accès sont cohérents puisqu'un objet fixé ne peut pas être déplacé en mémoire centrale; ceci nous assure que l'adresse de l'entrée reste valide pendant toute la durée de la fixation.

Nous pouvons donner le schéma d'exécution des N accès successifs:



Un registre Smalltalk ne contiendra plus un pointeur-objet, mais l'adresse en mémoire centrale de l'entrée de l'objet. Le passage par la structure de topographie est alors inutile.

Gains obtenus pour N accès consécutifs:

- un seul accès à la structure de topographie.
- un seul défaut d'objet possible.
- une faible quantité d'objets fixés en mémoire centrale.
- l'environnement immédiat de tout processus-interprète est constamment présent en mémoire centrale par fixation des objets référencés par les registres constituant son contexte d'exécution courant. Un processus en attente depuis longtemps est assuré de conserver son environnement immédiat en mémoire centrale ce qui induit un gain de temps au redémarrage du processus.

Inconvénients:

- fixations et relâchements inutiles d'objets dans le cas d'un échange fréquent du code à exécuter entre deux méthodes.

5.3. Proposition pour une solution plus efficace.

Les deux solutions que l'on vient d'exposer constituent des améliorations très limitées de l'accès aux objets. On a vu que la fixation d'un objet ne peut être envisagée que sur un ensemble restreint d'objets si l'on ne veut pas enlever toute liberté d'action au gestionnaire de la mémoire des objets. Pour cette raison, le champ d'application de cette solution a été limitée au contexte immédiat de chaque interprète Smalltalk. La deuxième solution propose d'ajouter des court-circuits d'accès aux objets pointés par les registres de chaque interprète. Cette solution complémente la première. Le fait que le champ d'application de ces solutions soit très limité provient du fait que la décision de fixer un objet échappe au contrôle du gestionnaire des objets, puisque la fixation d'un objet est effectuée par l'intermédiaire de primitives définies comme faisant partie de l'interface d'accès aux objets.

Le principe de base est donc d'incorporer le mécanisme de fixation des objets à celui du gestionnaire d'objets qui a été précédemment exposé. Le nouveau gestionnaire d'objets initialise et annule les court-circuits d'accès aux objets indépendamment des accès effectués par les interprètes. Jusqu'alors les accès aux objets se faisaient par l'intermédiaire d'une adresse virtuelle (le pointeur de l'objet). En addition à cette adresse virtuelle, on ajoute un champ adresse physique pour le descripteur et pour le bloc-valeur dont la validité est maintenue par le gestionnaire des objets. Si ces champs ont pour valeur "adr_nulle", alors le court-circuit n'est pas valable et l'accès doit se faire par la méthode classique (passage par la structure de topographie, résolution d'un éventuel défaut d'objet, ...). La composition de chaque moyen de désignation peut être décrit en langage C par la définition suivante:

```
typedef struct
{
        unsigned long pointeur_objet;
        unsigned long adr_physique_descripteur;
        unsigned long adr_physique_bloc_valeur;
}
```

Chaque interprète dispose d'un ensemble fini de ces structures d'adressage. Cet ensemble contient les structures d'adressage des derniers objets manipulés par l'interprète. Lors du premier accès à un objet, un accès classique est effectué et les court-circuits d'accès sont automatiquement mis en place pour les accès suivants.

Si le gestionnaire des objets déplace quelques objets en mémoire centrale ou s'il en enlève quelques uns (récupération d'espace), alors les court-circuits des objets concernés sont invalidés.

Une façon efficace, par laquelle le gestionnaire des objets peut valider et invalider les court-circuits des espaces d'adressage des interprètes, consiste à dater ces espaces d'adressage par rapport à sa dernière session de modifications de la mémoire des objets. Plus précisément, si la date de la structure d'adressage d'un interprète est antérieure à la dernière date de modification de la mémoire des objets, celui-ci ne peut se servir des

court-circuits et doit les rendre nuls (il ré-initialise toutes les adresses physiques à la valeur "adr_nulle"). Puis il affecte la date de son espace d'adressage avec la date de la dernière session de travail du gestionnaire d'objets.

Il est important de remarquer que chaque interprète est la seule activité à modifier son espace d'adressage. Par conséquent, il n'y a pas de structure de synchronisation nécessaire pour la réalisation de cette solution, ce qui constitue un net progrés pour la gestion des accès aux objets. Le gain attendu par cette solution est assez conséquent.

6. Conclusion.

Ce chapitre avait pour objectif de présenter les problèmes liés à la réalisation d'une mémoire des objets segmentée pour le système Smalltalk-80. La solution proposée est fondée sur le filtrage de tous les accès aux objets, et à leur interprétation par une machine virtuelle. Cette solution a l'avantage de rendre la conception de la mémoire virtuelle des objets indépendante des caractéristiques du système hôte et de son implémentation. Ce choix a été pris avec le dessein de concevoir un gestionnaire d'objets centralisé, comme si on désirait l'implémenter pour une machine nue, en utilisant les possibilités (topographie cablée, etc...) de la partie gestion de la mémoire de cette machine virtuelle.

La mémoire des objets est architecturée en deux modules: le gestionnaire des objets en mémoire centrale et le gestionnaire des objets en mémoire de stockage. Ceux-ci gèrent respectivement les ressources mémoire centrale et mémoire secondaire de la station de travail. Il est à remarquer, que contrairement aux systèmes classiques, la conception proposée ne fait pas intervenir un niveau de mémoire temporaire communément appelé mémoire de "swap" ou de pagination (pour les mémoires virtuelles paginées). Celle-ci est habituellement implémentée à l'aide d'une zone réservée de mémoire secondaire. Sa fonction est de stocker l'ensemble des données manipulées par les activités que sont les processus. Ces données ne sont en fait que temporaires, n'ont pas d'image en mémoire permanente, et par conséquent ne doivent pas survivre à la fin de l'activité qui les manipule.

Au lieu de concevoir la mémoire des objets comme une mémoire segmentée, on aurait pu la concevoir comme une mémoire paginée sur l'espace des objets qui se trouve en mémoire de masse [Stamos84]. Cette solution a été écartée car elle ne tient pas compte de la caractéristique segmentée de la mémoire des systèmes à base d'objets, qui sont composés de beaucoup de petits objets. Ceci constitue également une donnée nouvelle pour la conception de systèmes de gestion de mémoire segmentée et de leurs supports physiques.

On peut cependant citer les travaux réalisés sur la machine Burroughs B5500 [Batson77], pour la définition d'une mémoire virtuelle adaptée aux caractéristiques du langage Algol 60. Cette mémoire virtuelle était une mémoire segmentée où l'unité de segmentation était le bloc du langage Algol. Une partie matérielle avait été conçue, qui pouvait gérée des segments limités à 1023 mots. L'adaptation de cette mécanique à la

gestion des blocs Algol s'était avérée tout à fait acceptable. Cependant, cette mémoire virtuelle ne proposait qu'un adressage segmenté des blocs en mémoire centrale, et non en mémoire secondaire où ces blocs étaient compactés dans un fichier-texte exécutable.

Des définitions d'architecture de machine plus appropriée au système Smalltalk-80 ont également été réalisées [Ungar84b, Ungar87]. Ces développements ont porté sur la définition d'une couche d'interprétation plus efficace, mais ne propose pas de définition d'un système de gestion de mémoire approprié. Quelques éléments pour concevoir une gestion de mémoire plus appropriée aux systèmes à objets sont également présentés dans [Deutsch83, Suzuki83].

La proposition développée est basée sur un découpage logique en modules gérant les divers supports physiques des objets du système. Seuls les mécanismes de base ont été présentés en détail, et les diverses stratégies évoquées n'ont pas fait l'objet de développements précis. Ces dernières constituent cependant une partie toute aussi importante que la partie mécanisme, pour la définition d'une mémoire virtuelle des objets efficace. La recherche de cette efficacité commencerait, dans un premier temps, par une évaluation du système. Ceci devrait permettre de déduire des caractéristiques et des contraintes qui constitueraient le cahier des charges pour la définition de stratégies de gestion de la mémoire virtuelle.

- allocation mémoire libre (taille moyenne des objets).
- préchargement (connaissance de l'évolution du système).
- exploitation de diverses localités à l'exécution.

On obtiendrait vraisemblablement autant de mesures différentes que de familles d'applications (graphique, gestion de base de données, ...). D'après l'implémentation des objets en mémoire que l'on a proposé, il apparait déjà que la taille des informations nécessaires à la gestion est de l'ordre de celle des informations propres à l'objet.

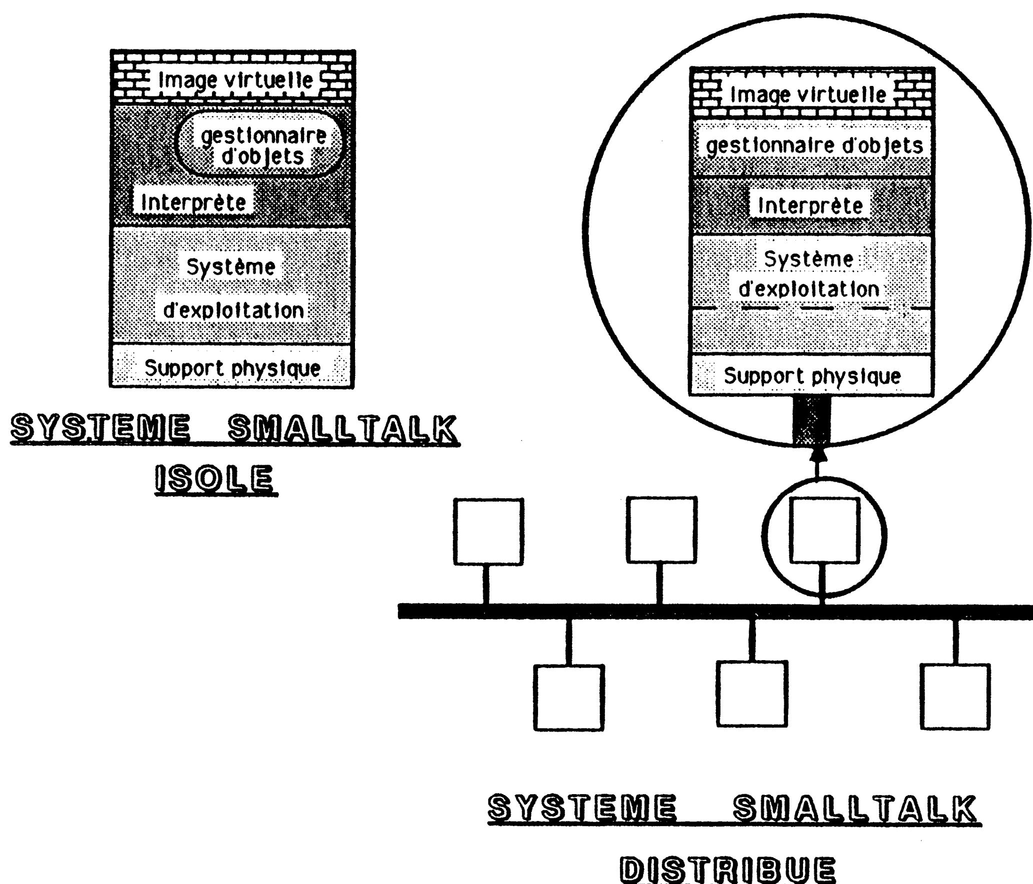
Le modèle de mémoire virtuelle développé dans ce chapitre n'a pas fait l'objet d'une implémentation particulière, mais il a été testé et validé en même temps que les concepts de répartition et de partage d'objets développés dans la mémoire répartie d'objets. La conception de la mémoire répartie d'objets est présentée dans le chapitre suivant.

V. CONCEPTION & REALISATION D'UN SYSTEME SMALLTALK-80 REPARTI.

1. Introduction

Le chapitre précédent a présenté en détail la conception d'un gestionnaire d'objets centralisé pour le système Smalltalk-80. Cette conception a été définie suivant la volonté de gérer les objets en tant que tels. L'objet est l'entité d'allocation et d'échange entre les divers niveaux de mémoires physiques.

On rappelle que l'objectif final de ce travail est la conception et la réalisation d'un gestionnaire distribué d'objets pour le système Smalltalk-80. Ainsi on veut définir le support de gestion d'objets pour des stations de travail Smalltalk-80 pouvant être reliées les unes aux autres par un réseau physique. Ceci constitue une avancée majeure à l'environnement de travail du système smalltalk-80 qui est initialement un environnement centralisé où il n'existe pas de mécanisme de protection sur les objets.



Organisation du système Smalltalk distribué.

Plusieurs avantages sont attendus de cette extension.

Le premier avantage des systèmes répartis réside dans la possibilité de pouvoir faire communiquer des utilisateurs travaillant sur des postes de travail distincts.

Le second avantage attendu est le partage d'informations ou de ressources, qui constitue un objectif majeur de la conception des systèmes répartis.

Les systèmes répartis se différencient par la nature des entités partagées et par la façon de réaliser ce partage. Par exemple, le système NFS [Sandberg86], ne permet de partager que des systèmes de fichiers Unix entre plusieurs sites. Chaque site perçoit ces fichiers comme locaux. Comme dans NFS, la plupart des systèmes répartis essayent de développer un partage transparent à l'utilisateur: les ressources sont alors banalisées et leurs localisations masquées par le système. De même, les aspects de désignation, de fiabilité, de protection, etc... font partis des soucis majeurs des concepteurs des mécanismes de partage.

Pour les systèmes à objets, partager des informations se ramène à partager des objets car toute entité du système est un objet. Cela constitue un progrès puisque avec la mise en place d'un mécanisme unique, on offre la possibilité de tout partager. Le travail développé dans ce chapitre est donc principalement motivé par cet aspect novateur pour l'expression de la répartition et du partage d'informations.

On peut prendre avantage de la communication et du partage pour réaliser un "serveur" capable de répartir, entre les divers sites, la charge globale du système réparti. Cet aspect est généralement très peu traité dans les systèmes traditionnels. La principale caractéristique qui rend la réalisation de ce but difficile est la différence conceptuelle qu'il existe entre l'entité manipulée pour réaliser la répartition de charge (processus), et l'entité que l'on manipule lorsqu'on réalise un quelconque partage d'informations (données). Pour le système l'une est une entité active, et l'autre est une entité passive. Les systèmes à objets homogénéisent ces deux notions en une seule entité: l'objet. Répartir la charge du système réparti revient alors à déplacer des objets entre les divers sites (migration d'objets). Ces déplacements peuvent être fait automatiquement par la couche système ou peuvent être exprimés par les utilisateurs pour mettre en œuvre un parallélisme effectif au sein de leur application.

Si l'on veut rester fidèle à l'aspect individuel du système Smalltalk-80, chaque site doit pouvoir conserver son autonomie vis à vis des autres sites. Individuellement chaque utilisateur peut faire évoluer son système Smalltalk et développer des applications locales sans aucune gêne pour les autres systèmes distants.

Des règles de partage doivent permettre à l'utilisateur de contrôler la participation de son poste de travail Smalltalk-80 à l'ensemble des ressources partageables. Par exemple, celui-ci peut contrôler les immigrations d'objets étrangers sur son site, qui si elles sont trop importantes peuvent éventuellement saturer les ressources physiques du site (mémoire permanente des objets) ou peuvent dégrader l'efficacité de sa station de travail (taux de charge trop important).

Le gestionnaire distribué des objets doit gérer les déconnexions des sites. Une station particulière peut à tout moment se retirer du réseau (arrêt ou isolation du site). Il est à noter que la déconnexion envisagée est une déconnexion volontaire, ne résultant pas d'une panne matérielle ou logicielle. Lors d'une déconnexion d'un site, il faut assurer que tous les objets prêtés sont retournés à leurs sites propriétaires, ceci afin de préserver la cohérence du réseau. Le partage d'objets est grandement compliqué par la possibilité qu'ont les sites de pouvoir se connecter et se déconnecter dynamiquement au réseau.

Le gestionnaire d'objets doit également mettre en œuvre un mécanisme distribué de récupération d'objets qui prenne en compte la répartition des objets et les connexions/ déconnexions dynamiques des stations.

Jusqu'à présent, toutes les implémentations de Smalltalk sont des versions centralisées [Krasner83]. Dans le système Smalltalk-80 original, le gestionnaire des objets fait partie intégrante de la couche d'interprétation. Un des desseins poursuivis est

d'implémenter le gestionnaire distribué d'objets comme une couche à part entière du système, tout en préservant l'interface entre l'interprète et l'image virtuelle. Le gestionnaire d'objets est dés lors réparti sur les différents sites qui composent le réseau. Cependant, chaque site est autonome et peut donc évoluer indépendamment s'il n'utilise pas des ressources localisées sur d'autres sites. Le gestionnaire distribué d'objets est conçu pour remplacer la couche originelle d'accès aux objets du système. Celle-ci doit être vue comme une extension logique de la "mémoire virtuelle d'objets" présentée au chapitre IV et non comme une "verrue" sur cette dernière. Les mécanismes du gestionnaire centralisé d'objets permettent d'accéder un objet comme un segment d'une mémoire segmentée. Le gestionnaire distribué d'objets reprend les mêmes mécanismes en les complétant pour qu'ils permettent d'accéder transparemment à un objet distant comme à un objet local. Par contre, d'autres fonctions telles que la migration d'objets sont ajoutées. Par cette conception d'une mémoire virtuelle distribuée, on rejoint les mêmes objectifs que ceux présents dans la conception de la mémoire virtuelle du système Apollo/Domain [Leach83]. L'évolution de la structure du poste de travail Smalltalk-80 est résumée par la figure 1 de cette thèse. Cette figure met bien en évidence l'extension du gestionnaire d'objets du système Smalltalk isolé pour la définition de celui du système Smalltalk distribué.

2. Les problèmes d'un gestionnaire distribué d'objets.

Le développement des systèmes distribués a en partie été motivé par le désir d'étendre l'ensemble limité des ressources partageables de chaque système, en un ensemble de ressources accessibles par tous les sites. Il existe un certain nombre de systèmes conçus à partir de cette motivation tels que NFS (partage de systèmes de fichiers) ou la "Newcastle Connection" [Brownbridge82], qui unifie un ensemble de systèmes Unix indépendants en un seul système réparti où tous les systèmes Unix ont leur racine (/) greffée sous une racine virtuelle distribuée. Dans ce sens, cette réunion des ressources individuelles en un ensemble de ressources communes pose les problèmes de désignation, de protection et de cohérence. Ces problèmes sont ceux des systèmes répartis classiques auxquels ont été apportés un certain nombre de solutions. Ceux-ci n'ont été que peu traités pour les systèmes à objets tels que Smalltalk-80.

Dans le système Smalltalk, toute entité est un objet, donc la seule ressource élémentaire qui peut être partagée est l'objet. En conséquence, le problème du partage de ressources sur un réseau de sites Smalltalk se réduit au problème de la gestion distribuée des objets sur ce réseau. Cette gestion distribuée des objets peut être réalisée par la coopération des divers gestionnaires d'objets locaux. Chaque gestionnaire d'objets local tourne sur un système unique, se charge des objets locaux de ce site et coopère avec les gestionnaires d'objets des autres sites pour assurer un partage correct des objets.

Les problèmes qui résultent de la distribution et du partage des objets, tels que la désignation, la protection et la cohérence, sont amplifiés par les fonctions inhérentes à la répartition de ressources sur un réseau de systèmes autonomes. En particulier, des

fonctions telles que l'accès à distance, la migration dynamique des objets et les connexions/ déconnexions des divers systèmes accroissent énormément la complexité de la gestion distribuée des objets. Dans les paragraphes suivants, on s'intéresse aux divers problèmes liés à cette gestion.

2.1. La désignation des objets distribués.

La fonction de nommage dans un environnement distribué permet de désigner des objets sans spécifier leurs sites de résidence. Tout objet est connu sur chaque site sous un **nom local,** à partir duquel la "machine à objets" localise l'objet sur le réseau. Un tel type de nommage est un nommage virtuel par rapport au nommage physique.

L'accès à un objet distant doit s'effectuer de la même façon qu'un objet local et doit donc rester complètement transparent à un utilisateur. Bien que l'invisibilité de la localisation des objets soit souhaitable, la fonction de nommage doit quand même, dans certains cas, permettre aux utilisateurs de contrôler directement la localisation de leurs objets; par exemple, un utilisateur peut vouloir forcer la migration d'un objet particulier vers un site donné pour effectuer un transfert explicite, ou pour tirer avantage d'un parallélisme effectif.

En résumé, un mécanisme de nommage approprié doit fournir une localisation des objets transparente tout en fournissant aux utilisateurs la possibilité de contrôler la localisation d'objets quand cela s'avère nécessaire.

2.2. Les accès aux objets.

Les accès aux objets peuvent être locaux à un site ou distants. Un accès à un objet distant peut être géré de deux façons différentes:
- soit il donne lieu à un accès à distance effectif et par conséquent l'objet ne bouge pas,
- soit l'objet migre vers le site de l'objet appelant, où ce dernier effectue donc un accès local.

Les allées et venues d'objets peuvent être évitées en fournissant des copies des objets désirés (duplication d'objets) aux sites qui veulent y faire des accès. Cependant, la gestion des copies multiples d'objet induit un coût supplémentaire (en espace et en temps calcul), et un accroissement de la complexité de la gestion des objets (nécessité de synchroniser les mises à jour). Les coûts de gestion de copies d'objets peuvent s'avérer être plus élevés que ceux de migration. En particulier, les objets Smalltalk étant de petites tailles, il n'est certainement pas raisonnable d'en gérer des copies.

Le choix entre l'accès distant ou l'accès local après migration de l'objet dépend de la charge du réseau à l'instant donné, de la taille de l'objet et de la nature des requêtes de l'application. La charge du réseau est un paramètre qui dépend de l'activité à un instant

précis, et la taille de l'objet accédé est une caractéristique propre à chaque objet et donc non prévisible avec précision. Par contre, dans certains cas on peut présager des demandes de l'application: par exemple, il peut apparaître dans le code de l'application qu'un objet va être utilisé un certain nombre de fois et qu'il est donc intéressant de l'amener sur le site de l'application. Cependant l'objet déplacé peut donner lieu à des accès à distance de la part d'autres sites. Même dans le cas où un objet-code est partagé, celui-ci peut être mofifié à tout moment par son site propriétaire, ou utilisé concurremment par d'autres sites.

2.3. La protection des objets.

Le problème de protection consiste essentiellement en la détection des accès interdits ou erronés. En milieu distribué, le phénomène est amplifié par la migration des informations. En particulier, une ressource peut migrer vers un nouveau site qui doit alors être capable de la protéger.

Le problème de protection dans les systèmes à vocation générale tels que les systèmes à temps partagé, est difficile à mettre en œuvre. Dans ces systèmes, les entités manipulées sont composées d'un grand ensemble d'informations intégrées dans une entité unique: le processus. Deux solutions sont communément utilisés.

La première consiste à adjoindre des droits d'accès à chaque objet. Un accès est licite si le propriétaire de l'activité fait partie de l'ensemble des personnes qui possède le droit requis.

- Par exemple, le système de fichiers d'Unix [Ritchie74], est organisé suivant ce principe: chaque fichier possède trois indicateurs de protection (lecture, écriture, protection) pour trois classes d'utilisateurs (propriétaire, membre du même groupe que le propriétaire, étrangers).
- La protection des segments de Multics [Organick72], est également fondée sur ce principe: à chaque segment est associé une liste d'accès qui spécifie les utilisateurs qui possède plus ou moins de droits sur le segment. Un élément de cette liste est un couple: <identification-utilisateur><droits>. Pour chaque accès, on vérifie si l'utilisateur fait partie de la liste et si les droits requis par son opération sont compatibles avec ceux du segment.

La deuxième solution consiste non pas à attacher la vérification sur l'objet, mais sur le moyen d'y accéder (désignation). Les systèmes qui utilisent ce principe de protection sont appelés des systèmes à capacités [Ferrie76, Cosserat75]. Une capacité est un moyen de désignation qui contient à la fois les informations de localisation de l'objet et les opérations permises. La transmission et la modification des capacités sont évidemment réalisées par des mécanismes de protection matériel et logiciel. Une capacité d'une activité peut être transmise à une autre activité. Dans ce cas, les permissions transmises sont obligatoirement inférieures ou identiques à celles que possède l'activité donatrice. Ce système de protection pose un problème pour la révoquation de droits: le propriétaire peut désirer diminuer les

droits d'accès d'autres usagers à un objet, ce qui n'est pas facile à résoudre car il faudrait pouvoir modifier les permissions de toutes les capacités qui permettent d'accéder l'objet.

L'intégration du contrôle des accès aux objets (locaux ou distants) à la couche de gestion des objets doit permettre de déplacer des objets d'un site vers un autre, et de mettre en place des droits d'accès sur ceux-ci. Ces droits peuvent être éventuellement variables suivant les sites. Smalltalk ressemble fortement aux systèmes à capacité: les objets (segments) sont composés de capacités vers d'autres objets. D'autre part, comme chaque site est mono-utilisateur, et que le moyen de désignation d'un objet distant ("représentant") sera unique à chaque site (voir section 3.2.2 ci-après), le problème de la révoquation des droits par recherche et modification des éventuelles capacités des autres sites sera facilement résolu.

Le choix de transposer le système Smalltalk sur le modèle des systèmes à capacités nécessite de contrôler les transmissions et les modifications des capacités (pointeur-objet). Cette fonction de protection des capacités est réalisée par chaque couche gestionnaire d'objets et par chaque couche d'interprétation des différents sites. Ces deux couches, qui représentent la machine virtuelle Smalltalk, ne sont pas modifiables par l'utilisateur.

Un deuxième aspect de protection existe: un utilisateur doit pouvoir spécifier dans quelle mesure ses ressources personnelles peuvent être utilisées par les utilisateurs extérieurs.

A tout moment, chaque utilisateur doit pouvoir restreindre ou augmenter le degré de partage de ses ressources (temps calcul, mémoire secondaire, ...). L'interface de la couche de gestion des objets doit offrir des primitives pour que l'utilisateur puisse régler les quotas des différentes ressources empruntables par les utilisateurs des autres sites. Il faut donc spécifier l'interface de programmation utilisateur par laquelle l'utilisateur propriétaire peut contrôler le partage de son système. Ce problème n'a pas reçu de solution et constitue un futur point de développement.

Ce problème est d'autant plus important qu'il est directement engendré par la volonté de répartition de la charge globale du réseau. Les activités (ensemble d'objets) peuvent être déplacées de façon non prévisible sur n'importe quel site (soit automatiquement ou soit explicitement par un utilisateur), et peuvent ainsi le dégrader.

2.4. Allocation de mémoire et récupération d'objets.

Dans les paragraphes précédents, nous avons dit que les systèmes à base d'objets sont bien adaptés pour la gestion du partage de ressources entre plusieurs sites. Ce partage s'exprime naturellement en terme d'objets partagés.

Cependant, ces systèmes sont de grands producteurs et de grands consommateurs d'objets: les objets sont alloués avec une grande fréquence et ne sont pas explicitement détruits par les applications, comme c'est le cas dans des langages tels que Pascal ou C avec

les primitives "dispose" ou "free". De plus, ces systèmes sont organisés en une arborescence d'objets possédant un objet-racine qui pré-existe, et dont les liaisons "*objetA → objetB*" signifie que l'objet A utilise (référence) l'objet B. L'espace des objets de tels systèmes peut être rapidement encombré par les objets inutilisés si ceux-ci ne sont pas récupérés périodiquement.

La récupération d'objets consiste en la détection des objets inutilisés et en la récupération de l'espace mémoire qui leur est associé. Ceci engendre un parcourt de graphe coûteux qui peut devenir inacceptable si on ne l'optimise pas un tant soit peu.

En milieu centralisé, de nombreuses solutions ont été apportées à ce problème [Almes80, Bobrow80, Christophe84, Deutsch76, Dijkstra78, Lieberman83, Ungar84a]. Le développement de tels algorithmes en milieu réparti est plus récent, car les systèmes, qui rende la résolution du problème de récupération d'objets cruciale, sont principalement les systèmes à objets et les environnements Lisp. Le développement en environnement réparti des systèmes à objets est très récent (voir chapitre III). Cependant, il existe des travaux de développements de procédure de récupération d'objets en environnements répartis tels que ceux développés dans [Hughes85, Wiseman85, Shin85, Halstead85]. Une bonne synthèse de présentation et d'évaluation de ces algorithmes de récupération distribué d'objets est fournie dans [Michel86].

Le problème de la récupération des objets dans un système Smalltalk-80 distribué est un problème très important. La section 3.4 ci-après développe un algorithme de récupération d'objets distribué par marquage. Celui-ci est simplement une adaptation à un environnement réparti de l'algorithme centralisé. Comme tous les algorithmes de marquage, il est contraignant car il impose l'arrêt total des activités des utilisateurs de tous les sites. D'autres algorithmes, notamment ceux développés dans les références précédentes, peuvent être envisagés.

3. Les principes du gestionnaire distribué d'objets.

Le gestionnaire d'objets est réalisé comme un ensemble de gestionnaires locaux d'objets coopérants. Chaque gestionnaire local évolue sur une station de travail particulière et fournit un ensemble de primitives à l'utilisateur. Ces primitives permettent de nommer et de partager des objets entre différents sites sans avoir à s'inquiéter de leurs localisations sur le réseau. Le gestionnaire d'objets est également capable d'accepter des modifications dynamiques de l'état de répartition des objets suite à des migrations, de gérer des récupérations réparties d'objets et de prendre en compte des connexions et des déconnexions aléatoires de sites.

La section 1 explicite la façon selon laquelle des objets peuvent être partagés. Etant donné la granularité des objets Smalltalk, il est nécessaire d'introduire des fonctions qui permettent de ramener l'expression du partage sur des entités facilement manipulables par l'utilisateur: classe, arborescence des objets d'un objet composé, ...

La section 2 introduit la notion d'objet "représentant" nécessaire à la mise en œuvre de la répartition et aux partages des objets. Cet concept constitue la notion de base de l'architecture de répartition développée.

Les sections 3,4,5 et 6 présentent les mécanismes (migration d'objet, récupération d'objets, connexions et de déconnexions dynamiques et protection des accès aux objets) relativement à cette définition.

3.1. Partage d'objets.

L'objet est l'unité de partage. Cependant des utilitaires peuvent être aménagés de façon à ce qu'un système puisse donner des droits d'accès sur un ensemble d'objets à tous les autres systèmes. Ces utilitaires doivent faire partie d'une interface utilisateur, qui ressemblera certainement à une application de type "browser".

Ainsi par cette interface, l'utilisateur d'un site pourra spécifier les objets partageables avec d'autres sites (boîte aux lettres, compilateur, ...). Ce partage pourra se faire par:

- l'accès à une ou plusieurs classes.
Tous les systèmes peuvent créer des exemplaires de cette classe, mais ne peuvent pas modifier sa définition. De même, ils peuvent utiliser toutes les méthodes possédées par cette classe.

- l'accès à une sous-hiérarchie de classes.

- don du droit d'accès à un objet-exemplaire.
Dans le cas où l'objet partagé est un objet-processus, cela donne la possibilité à une activité de créer une activité à distance ou d'agir sur une autre activité (opérations "fork", "resume", ...).

Remarque: Dans le cas où un objet est partagé, sa destruction par son site propriétaire revient à en effectuer sa migration vers un autre site qui possède un objet représentant pour cet objet. Ce nouveau site de résidence devient également le nouveau propriétaire de l'objet (voir section 3.4.1 ci-après).

3.2. Les références distantes et les objets "représentant".

Le but principal poursuivi dans la conception du gestionnaire d'objets est la transparence de la localisation physique de tout objet. Un nom d'objet est toujours un pointeur désignant un objet local. Plus précisément, si un objet *O1* détient un nom *pointeur-objet* pour référencer un autre objet *O2*, *pointeur-objet* identifie toujours un objet situé sur le même site que l'objet *O1*.

Deux choix pour la désignation des objets distants étaient possibles:
- soit on définit un système de nommage unique à l'ensemble des sites, et il est nécessaire de définir un service de localisation qui a partir d'un nom unique localise l'objet sur le réseau.
- soit chaque site possède un service de nommage propre, et dans ce cas il faut définir une structure (descripteur) qui permette de contenir et de décrire la localisation d'un objet.

C'est cette deuxième solution que nous avons retenue, car elle permet à un objet de désigner un objet distant par un nom local, sans qu'il soit nécessaire de définir un mécanisme de nommage universel entre les sites. Ceci permet de définir un système Smalltalk réparti dont la structure de l'image virtuelle reste conforme à celle du système centralisé.

Pour les références distantes, nous introduisons un type particulier d'objet *("le représentant")* qui représente localement un objet distant.

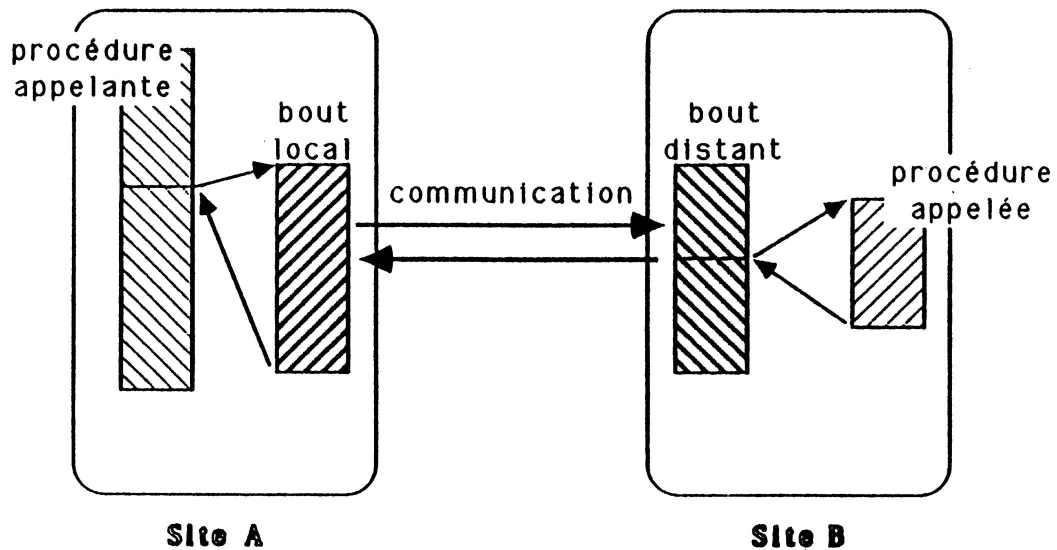
Un objet représentant est fonctionnellement équivalent à un lien Unix[1] excepté qu'un objet représentant n'est pas visible par le programmeur. C'est une donnée privée au gestionnaire d'objets semblable à n'importe quel objet Smalltalk vis à vis de l'échange mémoire ("swapping"), de la récupération d'objets et du relogement des objets en mémoire.

L'objet représentant est similaire à la notion de bout ("stub") [Birrell84], des mécanismes d'appel de procédure à distance. Un bout est une procédure, qui est créée par le système d'appel de procédure à distance, et qui apparaît à l'utilisateur comme la procédure distante.

Ainsi la programmation d'un appel à distance se résume à l'appel de cette procédure locale à laquelle il faut fournir quelques paramètres (nom de la procédure que l'on veut exécuter). Cette procédure bout n'est en fait qu'une procédure qui établit une communication avec un bout homologue sur le site de résidence de la procédure. Cet homologue fait alors localement l'appel à la procédure désirée. La communication sur le réseau est transparente à l'utilisateur.

---

[1] Unix est une marque déposée par les laboratoires AT&T-Bell.



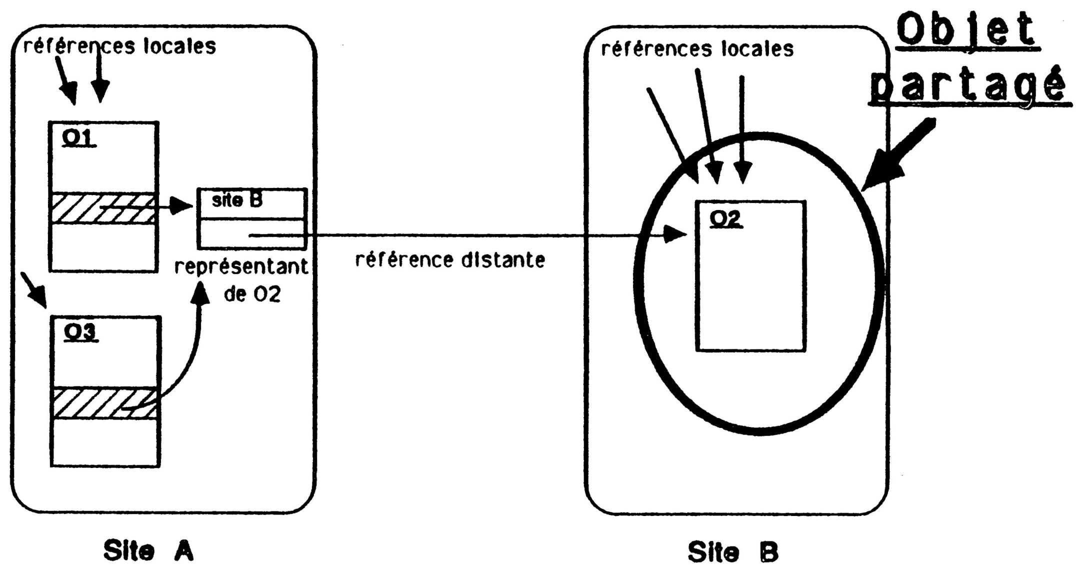
Modèle général de l'appel de procédure à distance.

La notion de représentant reprend le mécanisme de "stub" en le développant. Un objet représentant joue le rôle de l'objet réel, comme le "stub" joue le rôle de la procédure distante. L'intégration au système de l'objet représentant est beaucoup plus forte car à tout moment l'objet réel peut venir prendre la place du représentant et vice versa. Ces échanges entre objets représentants et objets réels restent complètement invisible à l'utilisateur. Les objets représentants sont donc complètement intégrés à l'ensemble des objets réels du système Smalltalk. Cette solution a été adoptée pour assurer la correction et pour conserver l'homogénéité de l'image virtuelle du système Smalltalk-80. Le principe de la migration d'un objet, détaillé dans la section 3.3, vérifie aussi cette propriété.

Un objet représentant est un objet privé au gestionnaire d'objet, c'est à dire qu'il existe en tant qu'objet dans l'image virtuelle mais il n'est visible que pour le gestionnaire d'objet. Tout accès par une activité utilisateur à un objet représentant, équivaut à un accès effectif et transparent à l'objet distant qu'il représente.

Les objets représentants sont créés et détruits par le gestionnaire d'objets lors de l'accès à un objet (voir l'exemple développé dans l'annexe D), de migrations ou de partages d'objets.

Un objet représentant est composé de deux champs qui localisent l'objet réel sur le réseau; le premier champ est le nom du site de résidence de l'objet réel, et le second est le nom local de cet objet sur son site. Comme il est représenté dans la figure suivante, *O2* appartient au site *B* et un objet représentant est créé sur le site *A* pour servir de représentant local à l'objet distant *O2*.
De même si *O3*, appartenant au site *A*, référence aussi *O2*, alors il utilise le même objet représentant que *O1* pour référencer *O2*.



Utilisation d'un objet représentant pour des accès distants à un objet réel.

Quand l'objet réel change de site, les références sur l'objet qui a migré sont toujours correctes et n'ont donc pas à être modifiées: l'objet représentant est mis à jour par le gestionnaire distribué d'objets. Outre la transparence de la localisation physique de l'objet réel sur le réseau, l'objet représentant garantit la cohérence de l'espace des objets du site qui le possède. En effet, un objet représentant s'insère parfaitement dans l'édifice des objets composés du système Smalltalk (un objet n'existe que s'il est référencé par au moins un autre objet, et tout objet réel ne peut référencer qu'un objet *local*).

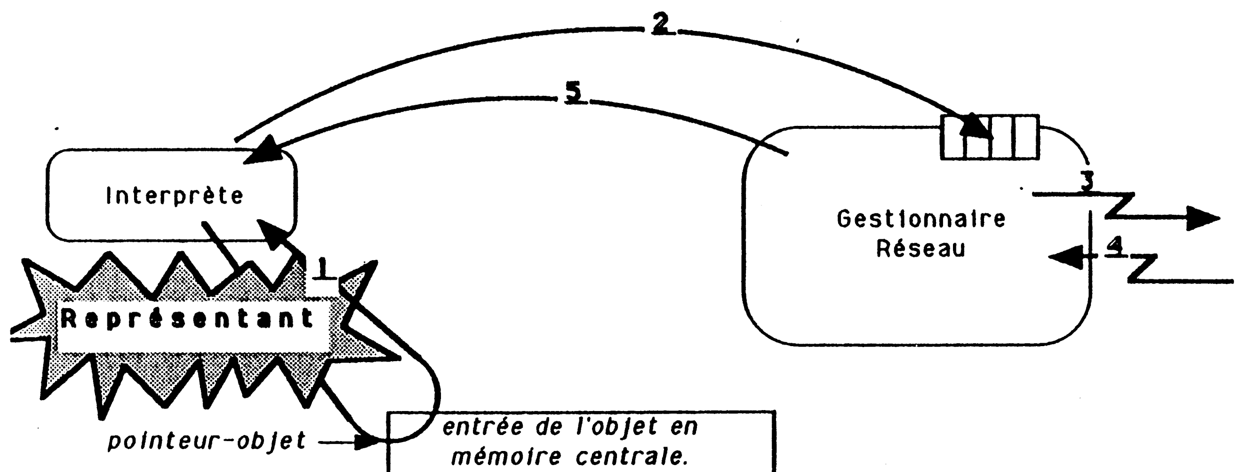
Ayant introduit la notion d'objet représentant comme principe pour l'expression de la répartition des objets sur le réseau, nous examinons maintenant les différents problèmes inhérents à la répartition.

1) Référence à un objet représentant par un autre objet représentant.

2) Migration d'un objet sur le réseau.
3) Récupération des éventuelles structures circulaires distribuées. C'est un problème délicat des systèmes centralisés et qui est amplifié par la répartition.
4) Cohérence des accès aux objets partagés et privilèges des différents systèmes sur ceux ci.
5) Protocole de connexion et déconnexion des sites, et cohérence des autres systèmes connectés au réseau.

3.2.1. L'accès à un objet sur le réseau.

Le protocole d'accès à un objet distant est représenté dans la figure suivante:



Légende:

1 Un processus-interprète essaie d'accéder à un objet.
L'objet est le représentant d'un objet distant. L'accès local engendre un accès distant.

2 Le processus interprète demande au gestionnaire réseau de lui faire l'accès.

3 Le gestionnaire du réseau transmet la demande d'accès à son homologue du site possédant l'objet réel.

4 Le gestionnaire du réseau du site de l'objet réalise l'accès souhaité et en retourne le résultat.

5 Le gestionnaire du réseau réveille l'interprète, qui reçoit en même temps le résultat de son accès comme si celui-ci avait été local. Il peut dés lors progresser dans l'exécution du code qu'il interprète.

Protocole d'accès à un objet distant.

La communication entre les sites du réseau s'effectue par l'intermédiaire des gestionnaire du réseau de chaque site. Chaque gestionnaire du réseau assure la communication du système auquel il appartient avec les autres systèmes (communication

fiable). Le gestionnaire réseau traite les demandes d'accès à des objets distants des processus interprètes et exécute des accès à des objets locaux pour le compte d'activités extérieures au site. Celui-ci assure de plus la correction des opérations inter-sites telles que:

migration d'objet
duplication d'objet
destruction d'objet
récupération répartie d'objets

Remarques:

1- La délivrance d'un résultat de l'accès, nécessaire dans le cas d'une demande de lecture, peut être facultative dans le cas d'une demande d'écriture. Cependant il paraît plus judicieux que le processus-interprète attende un compte rendu d'écriture, qui indique si l'opération s'est correctement effectuée ou non (erreur lors de l'accès, accès illicite, ...).

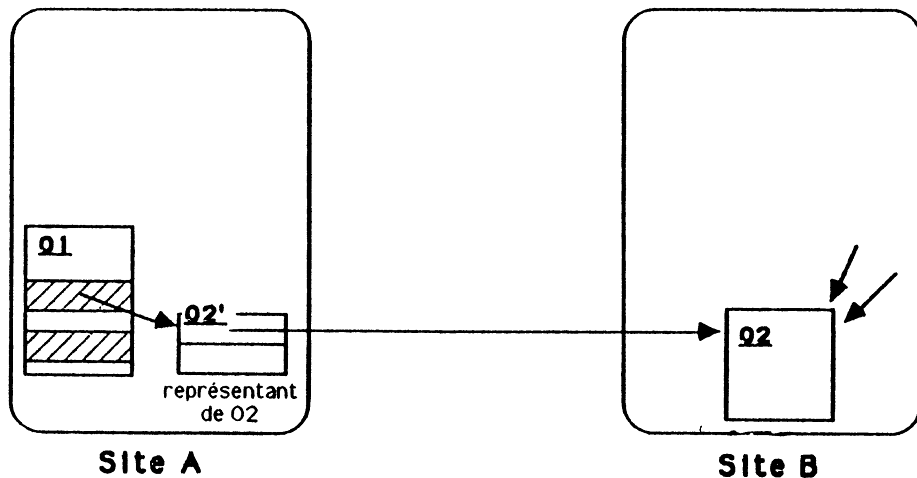
2- Un objet *O1* qui désire référencer un objet (établissement d'un lien de référence) *O2* doit incrémenter le compteur de références de ce dernier. Mais dans le cas où l'objet *O2* est distant, l'objet va indirectement le référencer via un objet représentant *O2'*. C'est le compteur de références de *O2'* qui est incrémenté et non pas celui de *O2*. Ceci est également valable pour une décrémentation.

Le processus gestionnaire du réseau de chaque site exécute les accès locaux à son site, pour le compte des processus-interprète distants. Cependant il ne se bloque pas en cas de défaut d'objet, car il est l'unique interlocuteur des autres sites. Ainsi, chaque processus gestionnaire du réseau a la possibilité de se différencier, vis à vis du gestionnaire de la mémoire, des processus-interprète: il ne se bloque pas en attente d'accès à un objet, mais il demande éventuellement le chargement d'un objet au processus gestionnaire de la mémoire. Dés que l'accès est possible, c'est à dire lorsque le gestionnaire de la mémoire le lui signale, il l'effectue et retourne le résultat au processus-interprète. Pendant le temps nécessaire au chargement, il a pu traiter d'autres requêtes d'accès à distance pour d'autres processus-interprète et émettre d'autres messages.

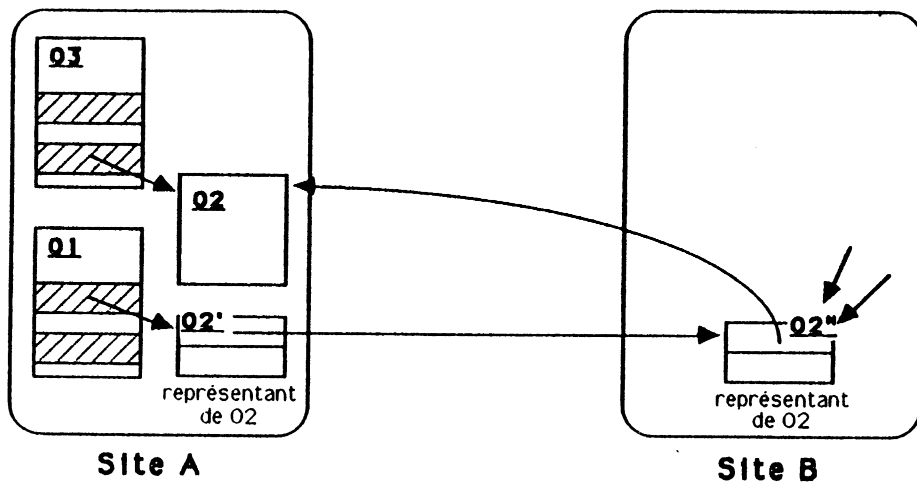
3.2.2. Unicité des objets représentants sur les sites.

Un objet représentant permet de représenter localement un objet distant. La question qui se pose est de savoir si, sur un site donné, il peut exister plusieurs représentants pour un même objet réel distant. Pour répondre à cette question, examinons le cas où un objet *O1* d'un site A référence un objet *O2* situé sur un site B. Pour ce faire, le système du site A crée un objet représentant *O2'* pour représenter localement l'objet distant. L'objet *O1* référence l'objet représentant *O2'* comme s'il s'agissait de l'objet-réel *O2*. Maintenant, supposons que l'objet *O2* doive être déplacé sur le site A. Sur le site B, l'objet *O2* est désormais désigné par un objet représentant *O2"*. Ainsi, on arrive à la situation où l'objet

O1 référence l'objet représentant *O2'*, qui référence à son tour l'objet représentant *O2"* de l'objet-réel *O2*.



Avant la migration de O2 sur le site A.



Aprés la migration de O2 sur le site A.

Problème de désignation du à la non-unicité des objets représentants.

Une telle situation est inacceptable: sur le site A, il est possible d'accéder de deux façons différentes à l'objet *O2*, soit par l'intermédiaire de l'objet représentant *O2'* via le site B ou soit par référence directe à l'objet.

exemple:

- *O1* référence *O2* via *O2'* sur le site A, puis via *O2"* sur le site B.
- *O3* référence directement *O2*.

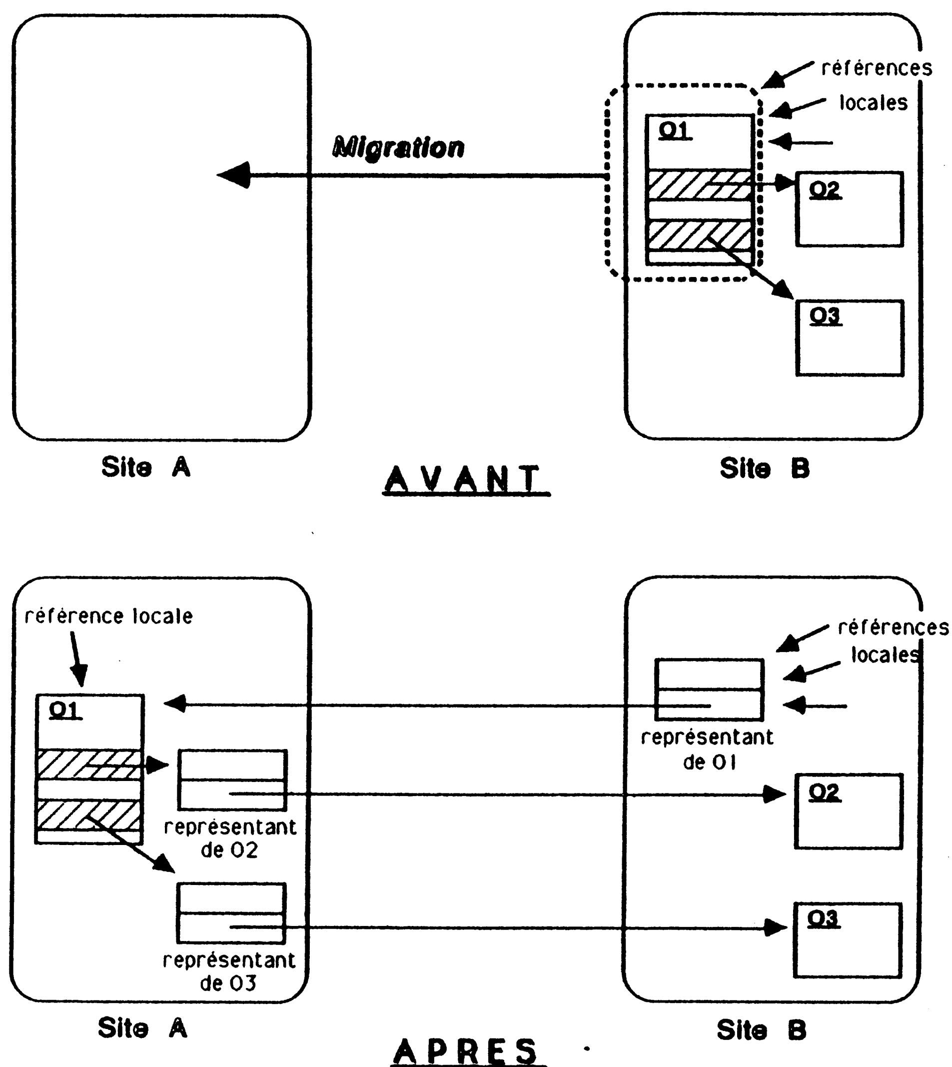
Le mieux est de ne conserver qu'une seule possibilité d'accès à l'objet: sur chaque site, on ne garde que l'objet réel ou qu'un objet représentant unique si l'objet réel est distant. Pour cela, lorsqu'on déplace un objet d'un site vers un autre site, on doit d'abord regarder sur le site destinataire s'il existe un objet représentant pour cet objet; dans le cas où il en existe déjà un les contenus des deux objets (réel et représentant) sont échangés .

3.3. Migration d'un objet.

Plusieurs systèmes Smalltalk coopérants peuvent vouloir partager des objets pour mettre en place une application distribuée (par exemple deux sites doivent au minimum partager un objet "échiquier" composé d'un ensemble d'objets-pièce pour pouvoir implémenter un jeu d'échecs répartis). Le partage de ces objets est possible par des accès locaux ou distants. Les accès distants sont réalisés par l'intermédiaire des objets représentants. Pour des raisons d'efficacité, des accès à un objet distant peuvent être transformés en des accès locaux.par copie ou migration de l'objet désiré.

La migration d'un objet consiste à déplacer un objet réel d'une station de travail vers une autre. Après la migration d'un objet, celui-ci est remplacé par un objet représentant sur le site de départ. Un objet réel peut revenir sur son site d'origine, puis repartir, et ceci plusieurs fois de suite... Pour cette raison, lorsqu'on déplace un objet, il est nécessaire de vérifier l'éventuelle existence sur le futur site de résidence d'un objet représentant. Si un objet représentant existe, l'objet réel et cet objet représentant sont échangés. L'objet réel est déplacé sur le nouveau site et prend le nom de l'objet représentant et un objet représentant remplace l'objet réel, sur le site origine, et localise l'objet réel sur son nouveau site de résidence.

Si un objet est partagé par plus de deux sites, tous les objets représentant de ces systèmes doivent être mis à jour. La mise à jour se fait automatiquement lors d'un accès distant qui aboutit sur un objet représentant. Le principe de cette mise à jour est présentée dans la section 3.5 qui suit.



Principe de migration d'un objet.

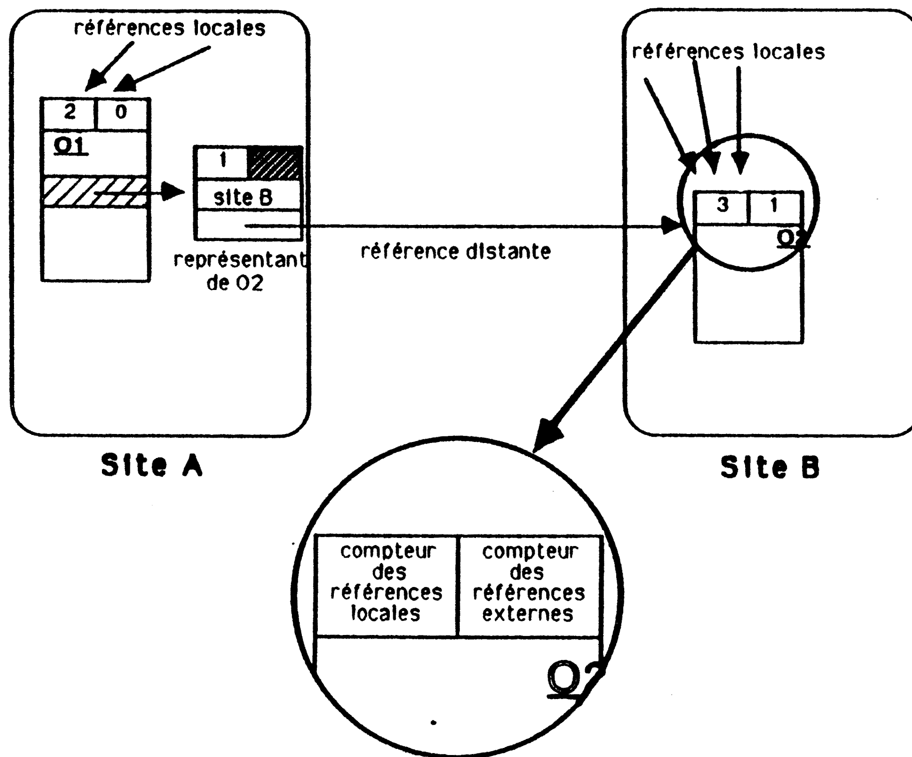
Le principe de migration d'un objet qui vient d'être présenté consiste uniquement en la définition d'une mécanique qui effectue le changement de localisation d'un objet tout en maintenant la cohérence des espaces d'objets de tous les sites du réseau. Les aspects "stratégiques" qui déterminent à quel moment et selon quels critères un objet doit être déplacé n'ont pas été abordés.

3.4. Gestion de l'espace des objets et récupération d'objets.

Nous rappelons que dans le système Smalltalk, un objet existe si son compteur de références (c.a.d. le nombre d'objets qui l'utilisent) est différent de zéro. Dans un système Smalltalk distribué, un objet peut être utilisé par des objets locaux ou distants.

Dans le but de conserver le mécanisme des références pour déterminer si un objet doit être conservé ou récupéré, il est nécessaire d'introduire un compteur de références supplémentaire pour chaque objet réel: "le compteur des références externes".

La valeur de ce compteur est égale au nombre d'objets représentants qui référencent l'objet réel à travers le réseau. Il est important de remarquer que sa valeur est inférieure ou égale au nombre des objet réels qui référencent son détenteur. Cela vient du fait que sur un site donné, il n'y a qu'un seul objet représentant pour désigner un objet distant.
La figure suivante illustre cette propriété:



Les différents compteurs de références d'un objet.

En résumé, tout objet possède deux compteurs:

- Un compteur local qui est le nombre d'objets locaux au site qui le référencent.

- Un compteur extérieur qui est le nombre d'objets représentants distants qui le référencent à travers le réseau.

Notre approche introduit tout naturellement deux sortes de récupération d'objets:
- une récupération d'objets locale à chaque site.
- une récupération d'objets globale à l'ensemble des sites.

Dans sa version initiale, le système Smalltalk récupère des objets selon deux méthodes (voir également section IV.4.3.1.d):
- selon la nullité de compteurs de références.
- avec une procédure de marquage qui permet de récupérer les cycles d'objets inaccessible à partir de la racine de Smalltalk.

On va donc examiner l'adéquation de ces deux méthodes à un environnement réparti. Puis nous présenterons une méthode de récupération fondée sur sur le vieillissement qui, en milieu centralisé, est mieux adaptée aux systèmes à base d'objets que les deux méthodes précédentes.
Chacune de ces méthodes est discutée selon le milieu où elle est développée: centralisé ou réparti.

3.4.1. Récupération d'objets par compteur de références.

Le principe de récupération d'objets par compteur de références [Bobrow80, Christophe84]. est basé sur la détection de l'inutilisation d'un objet mis en évidence par la nullité de son compteur de références ou plutôt d'utilisations. L'objet peut donc être récupéré.

L'algorithme présenté au chapitre précédent doit être modifié pour prendre en considération le dénombrement des références externes. Nous nous plaçons dans le cas où le compteur local d'un objet (non représentant) devient nul (compteur-références-locales(objet) = 0).

```
si compteur-références-extérieures(objet) = 0
    alors
    /* L'objet n'est plus référencé localement et n'a jamais été référencé à distance: */
    /* on peut donc le détruire. */
    détruire-objet(objet)
    sinon
    /* L'objet est utilisé par d'autres sites, mais l'actuel site de résidence ne s'en sert plus.
    Sa destruction locale revient à le faire migrer sur un nouveau site qui l'utilise. Il est à
    remarquer que si le site de résidence de l'objet est aussi son site propriétaire, alors le
    nouveau site de résidence devient aussi le nouveau propriétaire de l'objet. */
    nouveau-site-de-résidence ← trouver-un-nouveau-site-de-résidence(objet);
    transférer-l'objet-sur-nouveau-site(objet,nouveau-site-de-résidence);
fsi
```

Cet algorithme de récupération des objets par compteurs de références local et extérieur fonctionne si aucun message d'incrémentation n'est perdu. Ceci peut être assuré par un protocole de diffusion fiable tel que celui développé dans [Decitre83, David87].

Pour déterminer le nouveau site de résidence de l'objet, on peut appliquer l'algorithme suivant:

```
si site-de-résidence(objet) ≠ site-propriétaire(objet)
    alors
    nouveau-site-de-résidence ← site-propriètaire(objet);
    sinon
    /* Un nouveau site de résidence doit être choisi. Pour cela, le site de résidence de l'objet
    interroge tour à tour tous les autres sites jusqu'à ce que soit découvert un site qui possède
    un objet représentant pour cet objet. Ce site devient du même coup le nouveau
    propriétaire de l'objet.
    nouveau-site-de-résidence ← élection-d'un-nouveau-site-de-résidence(objet);
    site-propriétaire(objet) ← nouveau-site-de-résidence;
fsi
```

3.4.2. Récupération d'objets par marquage.

Le principe consiste à propager une marque depuis la racine du système et à détruire tous les objets non marqués (voir également section IV.4.3.1.d). La mise en œuvre de cet algorithme est facile en milieu centralisé: on arrête toutes les activités en un point correct et on propage une marque à partie de la ou les racines du système. L'algorithme diffuse alors à travers le graphe des références. Chaque exploration de branche du graphe des référence s'arrête sur les objets-nœuds déjà marqués. Toutes les activités interrompues sont alors redémarrées.

L'inconvénient majeur de cette procédure est justement l'interruption des activités qui est perceptible par les utilisateurs et souvent inacceptable.

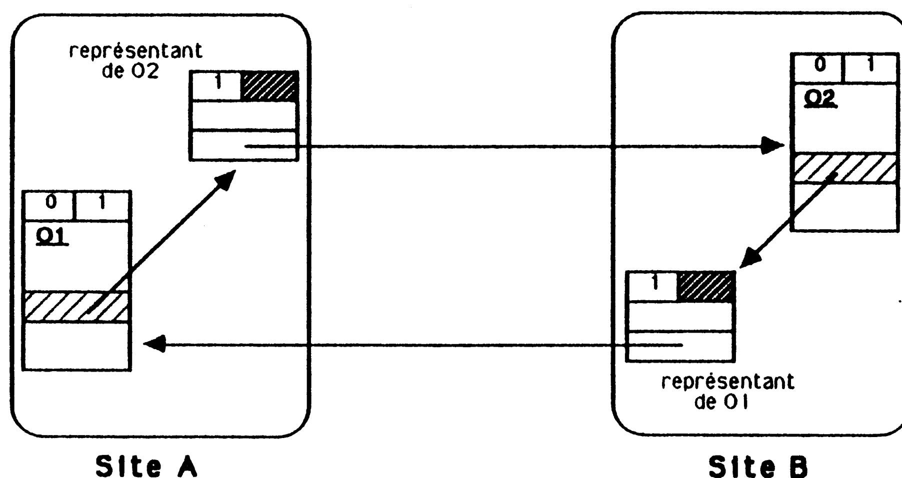
On sera amené à distinguer deux sortes de procédure de marquage:
- Une procédure de marquage locale à un site.
- Une procédure de marquage global à tous les sites.

Marquage local à un site.

Un site peut décider de faire un marquage local en propageant une marque depuis sa racine et à partir de tous les objets dont les compteurs de références extérieures sont non nuls. Ainsi chaque site peut éliminer ses "cycles fantômes locaux" sans aucune interaction avec les autres sites. Cette procédure ne met donc à jour que les compteurs locaux des objets de ce site et permet de récupérer les objets dont les compteurs locaux sont devenus nuls. Cet algorithme, défini dans l'implémentation initiale de Smalltalk, peut être acceptable dans la mesure où il n'est pas exécuté trop fréquemment.

Marquage global à l'ensemble des sites.

Une procédure de marquage global est une généralisation de la procédure de marquage local, Un marquage global a pour but l'élimination des "cycles fantômes" répartis sur le réseau. Pour l'exécuter tous les sites doivent se concerter.



Exemple de structure circulaire répartie.

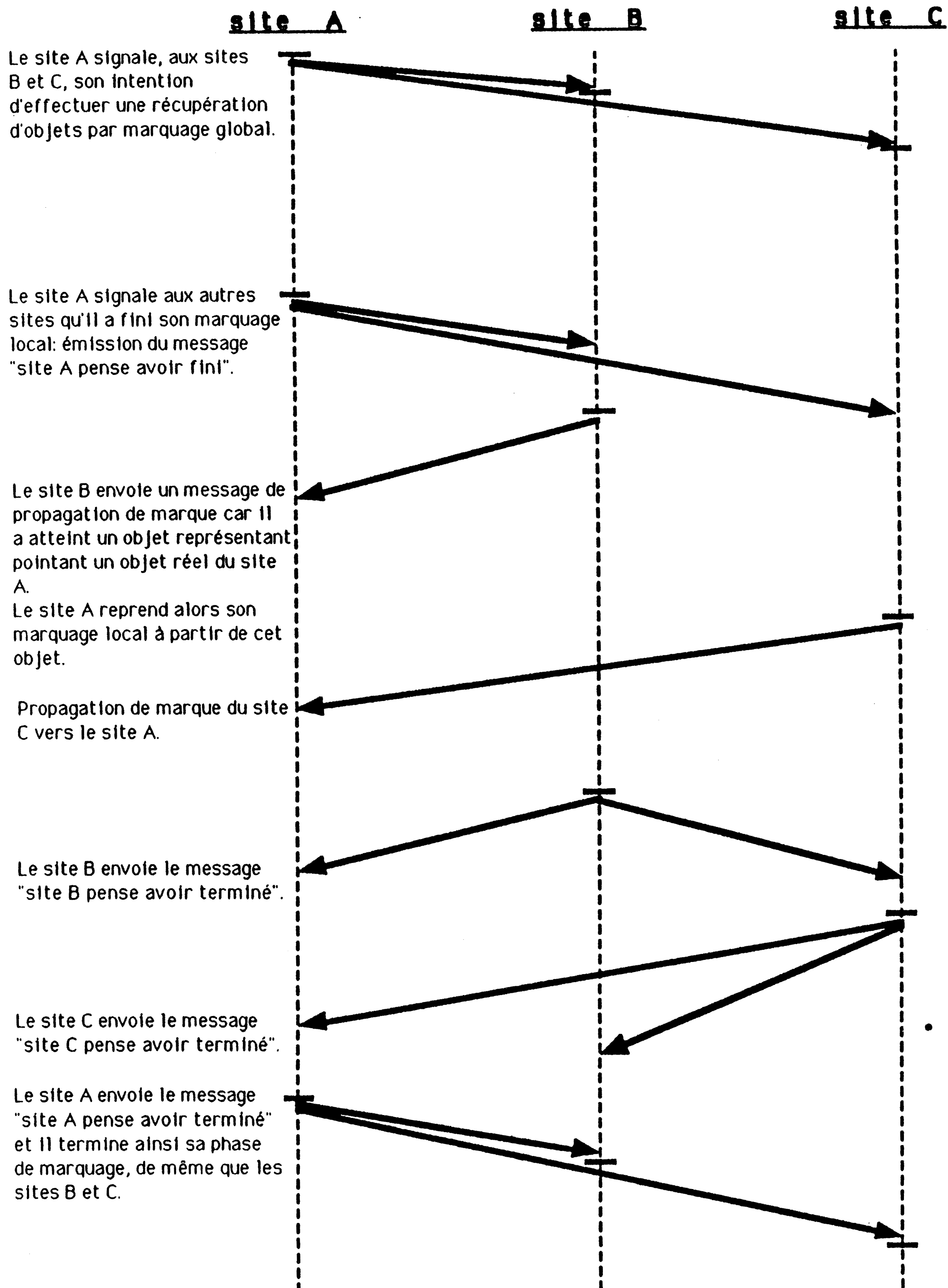
Il est important de signaler tout d'abord qu'actuellement, on ne peut envisager une récupération d'objets par marquage réparti que si toutes les activités des différents sites sont interrompues. Il convient donc de mettre en place un protocole de lancement et d'interruption de la procédure de marquage réparti qui assure la cohérence de tous les

objets manipulés sur le réseau. Le protocole de lancement peut être réduit à l'émission d'un message, et à s'assurer que tous les sites l'ont reçu. Là encore, un protocole de diffusion fiable est suffisant.

Fonctionnement: Chaque site émet une marque à partir de sa racine. Cette marque indique si on a ou non déjà traité l'objet. Lorsque chaque site arrive sur un objet représentant, il émet un message de propagation de marque vers un autre site à partir de l'objet réel qui est désigné par l'objet représentant (sauf si cet objet représentant est déjà marqué !). Cet objet réel devient alors une nouvelle racine pour l'autre site, qui propage la marque à partir de cet objet.

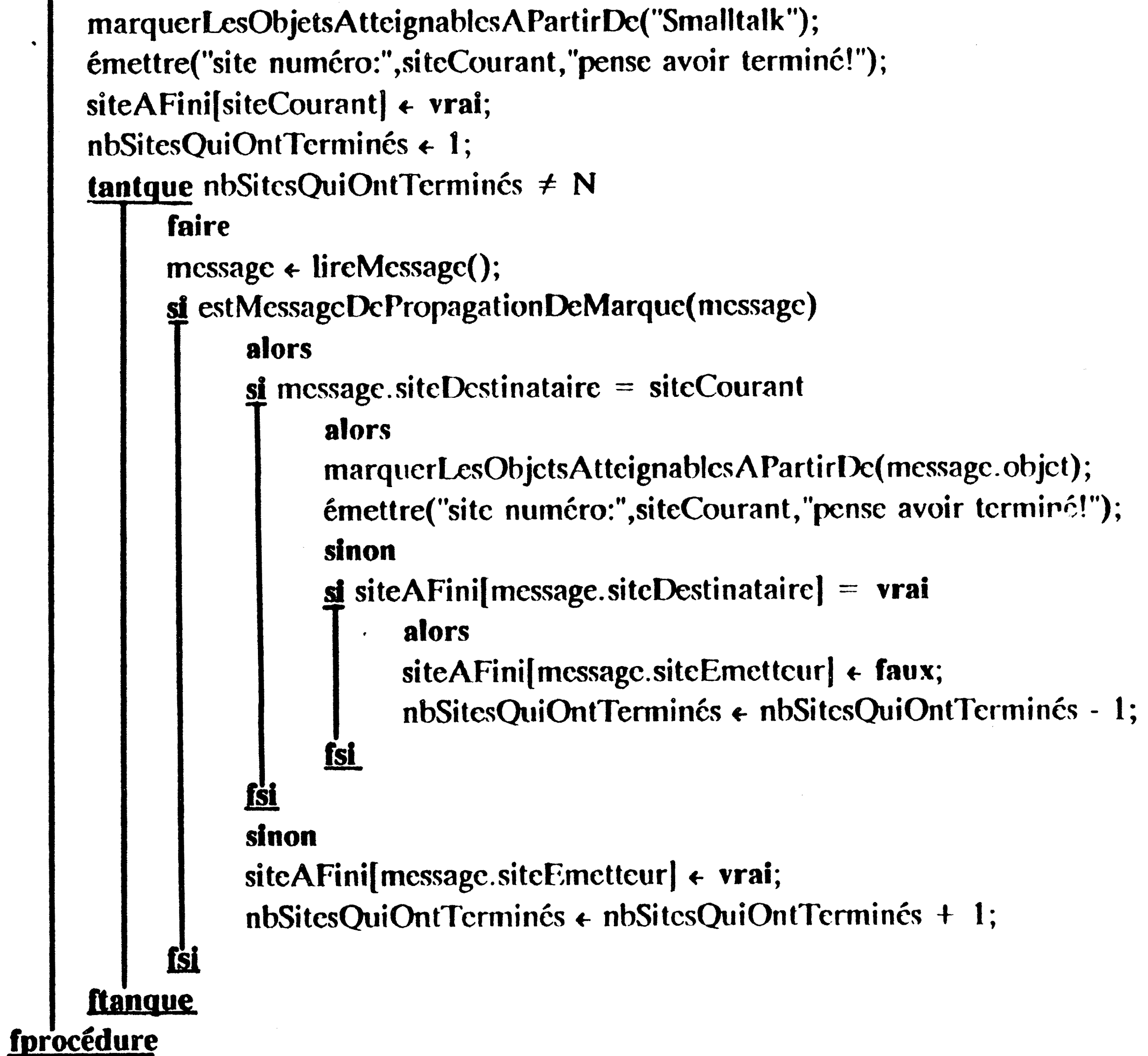
Il se pose le problème de savoir comment cet algorithme s'arrête. On peut faire l'hypothèse que ces messages sont échangés en diffusion (tous les sites les reçoivent). L'algorithme s'arrête lorsque tous les sites ont reçu successivement de la part des autres sites le message **(site i pense avoir terminé!).**
Il faudra assurer impérativement que tous les messages diffusés arrivent dans l'ordre où ils ont été émis par les différents sites. Ceci est nécessaire car si un message de propagation de marque sur le site "i" est émis après une séquence de messages (site i pense avoir terminé!), alors le message de terminaison du site "i" est invalidé et les sites doivent attendre jusqu'à ce que la séquence complète soit obtenue.
Le réseau doit donc garantir le non-entrelacement des messages en diffusion.

Encore une fois, si on dispose d'un mécanisme de diffusion fiable [Decitre83, David87], le protocole de terminaison est facile à mettre en œuvre. Dans le cas où l'on ne dispose pas de cet propriété du réseau de communication, on peut utiliser un algorithme présenté dans [Dijkstra83]. Cet algorithme permet de détecter la terminaison d'un calcul distribué entre N machines. Lorsqu'une machine a fini sa part de calcul, elle passe de l'état actif à l'état passif, jusqu'à ce qu'éventuellement un message arrive et la réacti pour une session de calcul suivante. Le problème consiste à faire détecter par toutes les machines la terminaison globale du calcul. La technique mise en œuvre met en place un circuit virtuel entre les machines sur lequel circule un jeton à deux états. Une présentation de cet algorithme est également fourni dans [Courtiat87].



Exemple du déroulement d'un marquage global entre trois sites.

L'algorithme de marquage réparti est présenté ci-après.



```
procédure marquageGlobal()
    marquerLesObjetsAtteignablesAPartirDe("Smalltalk");
    émettre("site numéro:",siteCourant,"pense avoir terminé!");
    siteAFini[siteCourant] ← vrai;
    nbSitesQuiOntTerminés ← 1;
    tantque nbSitesQuiOntTerminés ≠ N
        faire
        message ← lireMessage();
        si estMessageDePropagationDeMarque(message)
            alors
            si message.siteDestinataire = siteCourant
                alors
                marquerLesObjetsAtteignablesAPartirDe(message.objet);
                émettre("site numéro:",siteCourant,"pense avoir terminé!");
                sinon
                si siteAFini[message.siteDestinataire] = vrai
                    alors
                    siteAFini[message.siteEmetteur] ← faux;
                    nbSitesQuiOntTerminés ← nbSitesQuiOntTerminés - 1;
                fsi
            fsi
            sinon
            siteAFini[message.siteEmetteur] ← vrai;
            nbSitesQuiOntTerminés ← nbSitesQuiOntTerminés + 1;
        fsi
    ftanque
fprocédure
```

```
procédure marquerLesObjetsAtteignablesAPartirDe(objet)
    si objetNonMarqué(objet)
        alors
        marquer(objet);
        si estReprésentant(objet)
            alors
            émettrePropagationDeMarqueVers(objet.site,objet.objetDistant);
            sinon
            I ← 1;
            tantque I ≤ taille(objet)
                faire
                marquerLesObjetsAtteignablesAPartirDe(objet.champ[I]);
                I ← I + 1;
            ftanque
        fsi
    fsi
fprocédure
```

3.4.3. Récupération d'objets fondé sur le vieillissement.

Cet algorithme a été développé comme système de récupération d'objets d'un système Smalltalk centralisé [Ungar84a]. Cette méthode est fondée sur l'observation du système Smalltalk, qui a mis en évidence le fait que, soit les objets meurent rapidement, soit ils sont utilisés "à vie". A partir de cette observation, cet algorithme gère les objets en terme de "générations". Lors de sa création, un objet est positionné dans la génération la plus jeune, et au fur et à mesure qu'il persiste il passe de génération en génération: si un objet survit à une session de récupération, il est déclaré plus vieux et il est placé dans la génération immédiatement plus agée. Il n'est plus sujet au processus de récupération s'il atteint la plus vieille génération.

Cet algorithme se révèle être bien adapté à l'évolution des objets à l'intérieur des systèmes qui sont de gros consommateurs de ressources mémoire, comme le sont les systèmes à base d'objets. Cet algorithme a été développé et testé sur la version du système Smalltalk (BS) developpée à l'université de Berkeley.

Avec l'organisation de la mémoire telle que nous l'avons décrite, la récupération d'objet en mémoire principale n'est plus un problème crucial. En effet, un objet qui n'est plus utilisé est à plus ou moins long terme relégué en mémoire secondaire, et n'encombre donc plus la mémoire principale.

3.5. Connexions et déconnexions dynamiques.

Une des caractéristiques de notre système distribué est d'accepter le fait que les stations puissent individuellement se connecter et se déconnecter dynamiquement au réseau. Pour assurer la cohérence globale du système, des opérations spécifiques doivent être exécuter lors des connexions et lors des déconnexions.

Lorsqu'un site se connecte, il diffuse le fait qu'il est présent pour pouvoir communiquer ultérieurement avec les autres sites du réseau. Ainsi le protocole est simple et facile à mettre en œuvre, si on fait abstraction des risques de pannes ou d'erreurs lors de la connexion. La déconnexion d'un site est plus difficile à mettre en œuvre pour deux raisons principales:

1) Il se peut que le système qui se déconnecte possède des objets partagés par les autres systèmes. Ces objets doivent migrer vers d'autres sites, si l'on veut qu'ils puissent être utiliser après la déconnexion du site. Si un objet n'appartient pas au site qui veut se déconnecter, il revient alors sur son site propriétaire (si celui-ci est connecté, sinon il doit migrer sur un autre site). Si un objet appartient au site, les sites qui veulent s'en servir doivent le rapatrier chez eux. La migration des objets partagés est nécessaire pour assurer une déconnexion propre. Pour les déconnexions brutales telles que des pannes logicielles ou matérielles, il n'existe pas encore de solutions adéquates. On peut même dire que les références inter-sites induites par la définition des objets représentant augmentent considérablement les conséquences d'une panne d'un site.

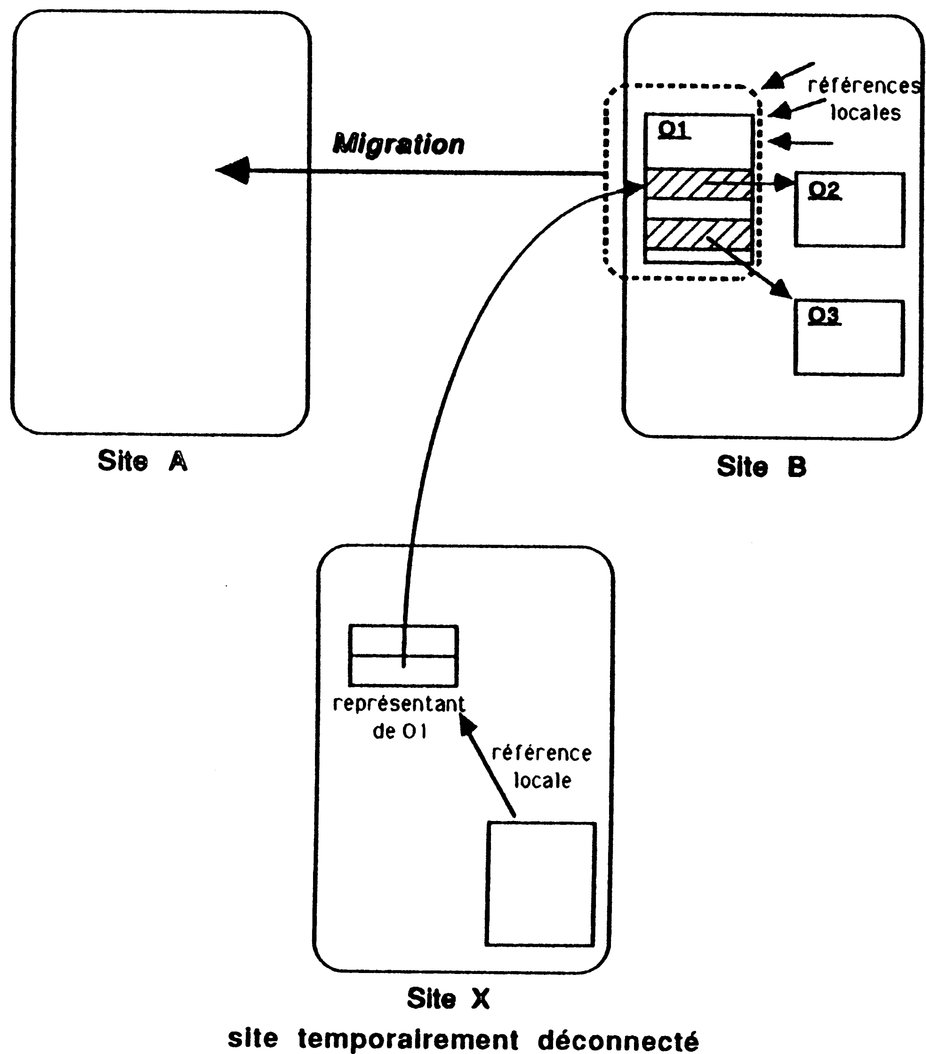
2) De même, le système qui veut se déconnecter peut posséder des objets sur d'autres sites. Doit-il les laisser sur ces sites ou doit-il les rapatrier? S'il laisse les objets sur les sites, ce qui peut apparaître intéressant dans un premier temps, ceux-ci peuvent de même se déconnecter. Si le site se reconnecte alors que ces sites sont toujours non atteignables, il peut alors se trouver dans un état complétement incohérent car il ne dispose plus d'objets qui peuvent lui être critiques. Ceci est le problème général où un site possède une ressource utilisée par d'autres sites. Si le site tombe en panne, les autres sites sont dans l'impossibilité d'accéder la ressource. Par exemple, si par NFS deux sites se partagent un même système de fichiers, et si le site où est localisé le système de fichiers tombe en panne alors le second site est dans un état incohérent. Des solutions pourraient être introduites par la mise en œuvre d'un système de gestion de copies et de versions.

Un site déconnecté possédant un objet représentant pour désigner un objet distant ne peut être averti si ce dernier change de site de résidence. Il faut donc qu'un site qui se reconnecte ait la possibilité de mettre à jour ses objets représentant. Pour assurer ceci, on peut imaginer deux solutions:

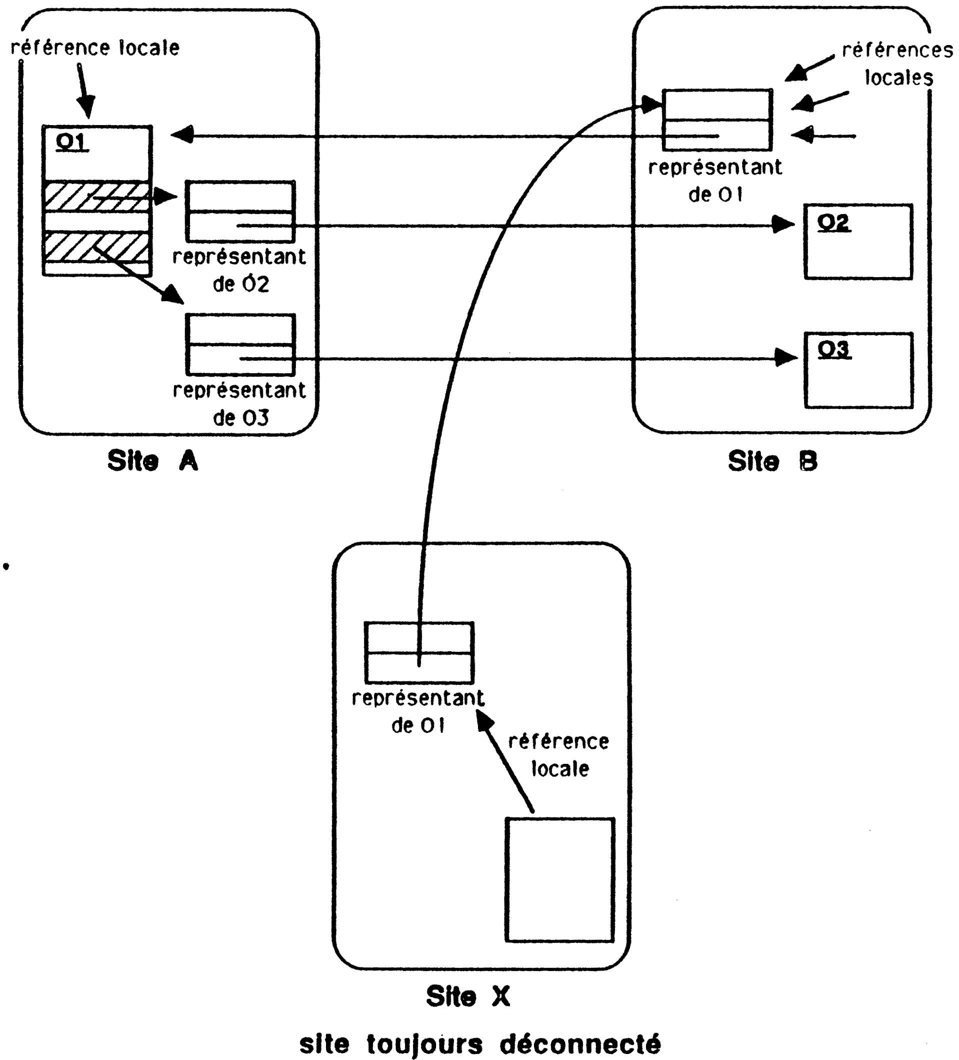
a- Soit toute migration d'objet donne lieu à la mise à jour automatique des objets représentant des autres sites. Pour cette raison, lorsqu'un objet partagé migre, des messages doivent être envoyés à tous les sites pour la mise à jour des différents objets représentant. Les messages pour les sites déconnectés doivent ainsi être mémorisés jusqu'à

ce qu'ils se reconnectent. Cette méthode peut se révéler très coûteuse en stockage des messages différés. Cette méthode n'est pas retenue.

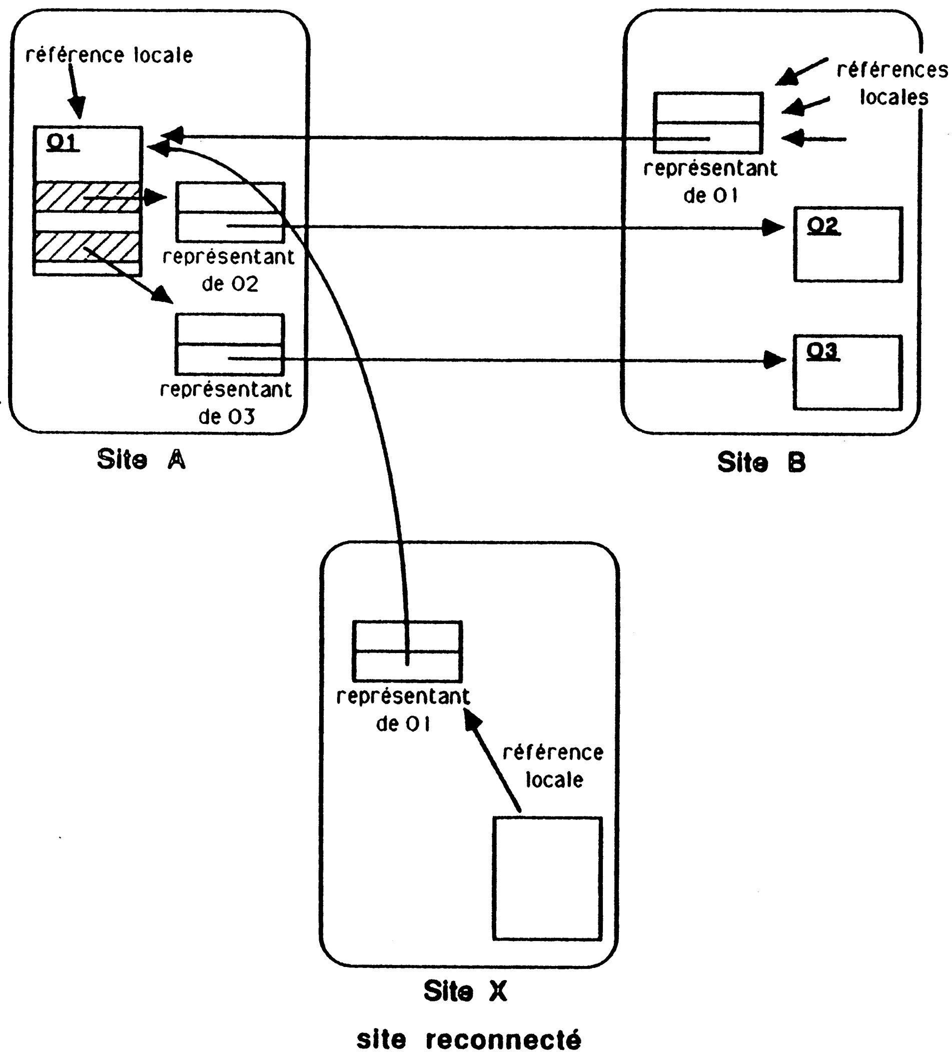
b- soit la modification de résidence d'un objet réel dans le réseau n'est pas répercutée immédiatement. La mise à jour se fera automatiquement lors des tentative d'accès à l'objet réel par l'intermédiaire d'objet représentant non à jour. Cet algorithme est illustré par les trois figures suivantes qui montre les différentes phases dans la remise à jour d'un objet représentant d'un site qui se reconnecte au réseau.



Etats des sites avant la migration de O1 (le site X est déconnecté).



Etats des sites après la migration de O1 (le site X est toujours déconnecté).



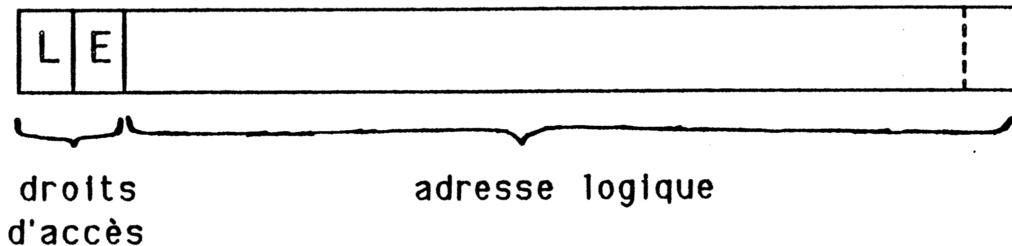
Reconnexion du site X, et mise à jour à la suite d'un accès à l'objet représentant.

Il est important de remarquer qu'à la suite de ce choix, un site ne peut plus à lui seul décider de la récupération d'un objet représentant qui ne lui sert plus. En effet, un objet représentant fait implicitement partie de l'historique de migration de l'objet réel. La destruction des objets représentants ne peut être faite que par une phase de concertation inter-sites.

3.6. Protection des accès.

Comme nous l'avons dit précédemment, la structure du système Smalltalk réparti justifie le choix de gérer la protection suivant le modèle des systèmes à capacités [Ferric76]:
- l'objet représentant est un moyen de désignation propre à un site (où il n'y a qu'un utilisateur) pour accéder à un objet réel distant. Cette notion s'adapte tout à fait à la notion de capacité des systèmes centralisés qui est un moyen de désignation local à une activité pour pouvoir accéder à une entité (programme, donnée, ...).
- la solution de gestion de la protection à l'aide de capacité permet une grande souplesse pour le passage de droits aux autres sites. L'utilisateur propriétaire de l'objet peut uniquement fournir des moyens de désignation à un sous-ensemble d'utilisateurs.

Cependant ce mécanisme autorise un utilisateur qui a reçu une capacité a la transmettre à son tour à un autre utilisateur. La capacité transmise ne peut que contenir des droits inférieurs ou égaux pour l'accès à l'objet. Cette vérification est automatiquement faite par le gestionnaire d'objets de chaque site: une capacité (pointeur-objet) vers un objet distant est incluse dans un objet représentant qui n'est manipulable que par le gestionnaire d'objets.



Nouveau format d'une capacité vers un objet (pointeur-objet).

Le mécanisme de protection défini n'a pas encore été mis en œuvre et testé dans la version courante du gestionnaire réparti des objets.

4. Composition de la machine virtuelle Smalltalk.

Nous appelons "machine virtuelle" du système Smalltalk l'ensemble des structures de données et des logiciels utilisé par le système utilisateur Smalltalk (image virtuelle). Dans la présentation de la mémoire virtuelle des objets, nous avons décomposé la machine virtuelle en deux entités:

1- Un noyau de synchronisation.

2- Les processus de gestion des ressources du système qui se composaient d'un gestionnaire des objets en mémoire secondaire et d'un gestionnaire des objets en mémoire centrale.

Ce qui nous permet de distinguer les deux fonctions de la machine virtuelle:

1- Exécution et ordonnancement des divers processus Smalltalk.
A chaque processus Smalltalk est associé un processus-interprète, qui interprète le code du processus Smalltalk.

2- Gestion des objets du système.

Dans le but de faire communiquer des systèmes Smalltalk distants, nous devons revoir et compléter la définition de chaque entité composant la machine virtuelle.

4.1. Le noyau de synchronisation.

Nous faisons donc l'hypothèse de l'existence au niveau de la machine cible d'un noyau de synchronisation qui gère l'exécution des processus-interprète et des processus de gestion des objets du système. Cette approche nous décharge des problèmes de gestion des processus. Elle nous rend également indépendant du nombre de processeurs physiques dont nous disposerons.

Nous faisons l'hypothèse que ce noyau de processus nous permet de:

1- créer et détruire dynamiquement des processus.

2- les suspendre et les réactiver, ce qui est nécessaire pour implémenter les primitives de synchronisation du système Smalltalk-80. Le noyau gère deux types de processus: des processus système et des processus-interprète. Un processus serveur du réseau contrôle les activités de chaque site: "Butler" [Dannenberg82].

3- synchroniser ces processus à l'aide des opérations classiques sur des sémaphores.

4- gérer la communication entre les processus (stockage et distribution des messages suivant des priorités établies).

4.2. Organisation du gestionnaire des objets.

Trois processus systèmes coopèrent pour réaliser la fonction de gestion des objets: le gestionnaire réseau, le gestionnaire de la mémoire centrale et le gestionnaire de la mémoire secondaire (voir figure suivante).

En plus de ces processus systèmes, plusieurs autres processus peuvent être présents sur un site. Ce sont les processus-interprète qui accèdent aux objets pour exécuter des actions Smalltalk. Chacun de ces processus représente au sein du noyau du système l'activité d'un objet-processus Smalltalk. Ces processus peuvent être vus comme des clients des processus de la gestion des objets. Au niveau de l'exécution, les processus système possèderont des priorités plus élevées que les processus-interprète.

a- Le gestionnaire du réseau.

Le gestionnaire du réseau est le processus-maître du site Smalltalk, il assure la supervision de l'interface avec les autres sites et l'ordonnancement interne des activités locales. Toute activité (d'interprétation ou système) lui est soumise.

Nous donnons ici quelques caractéristiques du gestionnaire réseau:

1- Il offre un moyen de communication (en passant par le noyau) avec les autres sites.

2- Il contrôle et exécute les demandes d'accès à des objets locaux pour le compte de processus-interprète distants.

3- Il transmet les demandes d'accès à distance, requis par les processus de son site, au gestionnaire du réseau du site où se trouvent les objets à accéder.

4- Sur le lancement d'une phase de récupération d'objets, il arrète les activités de son site pour effectuer de la récupération d'objets.

La communication entre les différents processus (système ou interprète) se fait par messages et peut être locale ou distante. S'il s'agit d'une communication à distance, seules les communications "gestionnaire du réseau <=> gestionnaire du réseau" seront autorisées. Toutes les exécutions à distance sont interdites. Par conséquent la tâche du noyau est réduite puisque sur chaque site il n'y a qu'un seul processus qui peut demander une communication avec un processus distant. Le noyau de processus n'a pas à gérer les noms sur le réseau puisque sur chaque site le serveur réseau possède le même nom. Si un processus-interprète veut communiquer avec un processus-interprète d'un autre site alors la communication transite obligatoirement par les serveurs réseau des deux sites concernés.

Puisque le serveur réseau gère les objets sur le réseau, il a une vue plus générale de l'ensemble des accès aux objets et notamment c'est lui qui négocie des accès avec ses

homologues des autres sites. Il a également le pouvoir de bloquer ou de ralentir les gestionnaires de la mémoire centrale et de la mémoire secondaire.

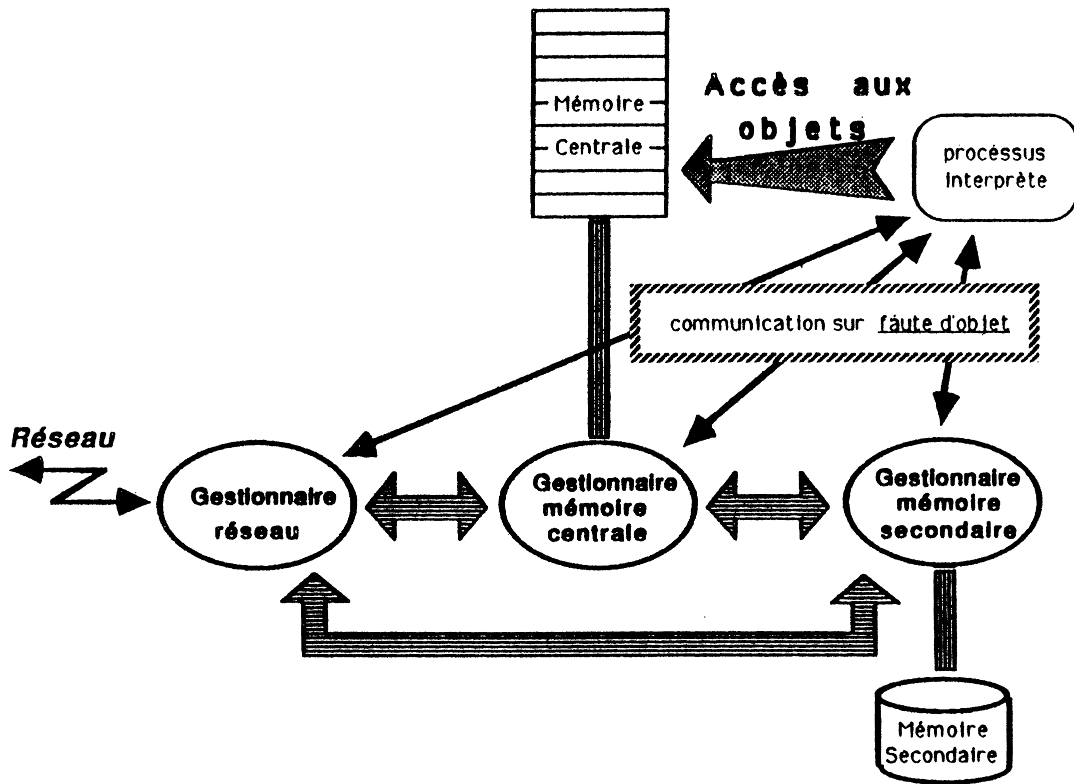
La conception des gestionnaires de mémoire centrale et de mémoire secondaire reste fidèle à celle précédemment décrite pour le gestionnaire d'objets centralisé.

b- Le gestionnaire de la mémoire centrale.

Le gestionnaire de mémoire centrale gère les objets en mémoire centrale (relogement des objets en mémoire centrale, résolution des défauts d'objets locaux au site, choix des objets à vider en mémoire secondaire, gestion de l'espace libre, recopies préventives d'objets, ...). Il résoud les défauts d'objets en allouant l'espace mémoire centrale nécessaire à l'objet manquant, et envoie une demande de chargement d'objet au gestionnaire de la mémoire secondaire. Après que l'objet est été amené en mémoire centrale, il réveille le processus-interprète ayant provoqué la faute.

c- Le gestionnaire de la mémoire secondaire.

Ce gestionnaire s'occupe de la gestion des objets en mémoire de stockage. Ce stockage est actuellement réalisé dans deux fichiers: l'un contient la table des objets du site et l'autre contient l'ensemble des objets. Comme nous l'avons présenté auparavant, la table des objets est une structure qui permet de donner un nom (pointeur-objet) à chaque objet: Un nom est un indice dans cette table. Cette table permet de réaliser une indirection entre ce nom virtuel et l'adresse physique dans l'espace des objets. Ceci permet entre autre de pouvoir modifier l'adresse physique d'un objet (par exemple, pour augmenter sa taille) tout en conservant le même moyen de désignation logique.



Le modèle de fonctionnement du système.

VI. IMPLEMENTATION DU SYSTEME SMALLTALK-80 REPARTI.

1. La réalisation développée.

Dans la conception du gestionnaire d'objets précédemment exposée, les objets en mémoire centrale doivent être accessibles par tous les processus. L'implémentation a été développée au dessus d'Unix 4.2 BSD. Ce système ne permet pas à plusieurs processus de partager de la mémoire. En effet, une règle essentielle de la conception du système Unix est le cloisonnement des espaces d'adressage des activités. Tout au plus, des processus peuvent-ils partager un segment de code si celui-ci a été déclaré comme ré-entrant. En conséquence, il était impossible d'implémenter la maquette en allouant un processus Unix pour chaque processus Smalltalk.

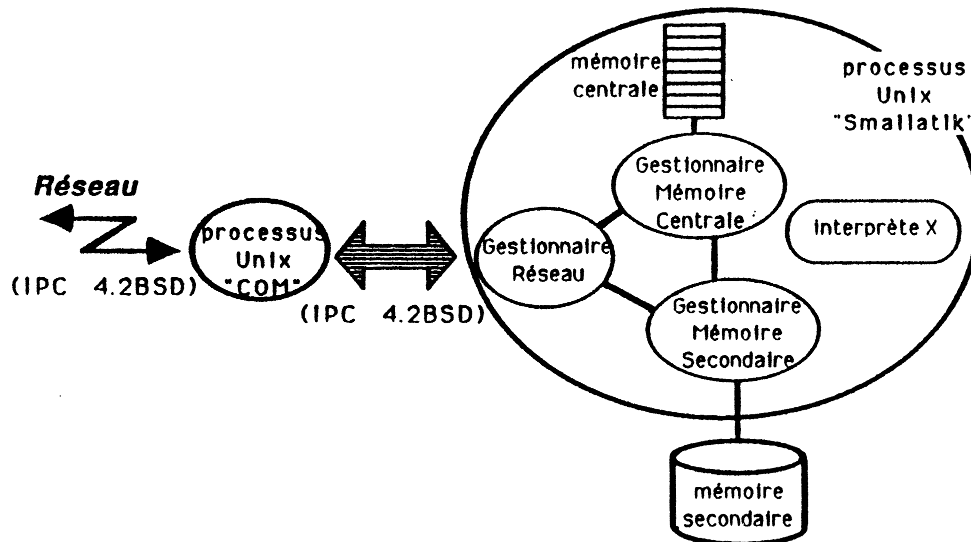
La solution adoptée consiste à gérer des processus dits "légers" à l'intérieur d'un processus Unix. De cette façon ces processus "légers" peuvent partager une mémoire qui est celle du processus Unix englobant. Du point de vue conceptuel, cette solution n'est pas entièrement satisfaisante puisque chaque processus "léger" peut accéder, en plus de la zone de donnée représentant le mémoire centrale, à tout l'espace du processus Unix. Cet espace contient toutes les boîtes aux lettres supportant la communication entre les divers processus "légers", les données servant à l'activation des diverses activités (ordonnancement des processus "légers"), l'état des flots d'entrée/sortie du processus Unix, etc...

Ce multiplexeur d'activités au sein d'un unique processus Unix se nomme Mux [Tricot87], et offre les services suivants:

- création d'un processus "léger".
- destruction d'un processus "léger".
- interruption d'un processus "léger".
- ré-activation d'un processus "léger".

Cet utilitaire permet en outre d'effectuer des entrées/sorties non bloquantes: si à l'intérieur du processus Unix englobant, un processus "léger" effectue une opération de lecture/écriture par l'intermédiaire des primitives Unix standard, ceci a pour effet de bloquer tout le processus Unix en attente de la terminaison de cette entrée/sortie. Pour cette raison, Mux gère les entrées/sorties et les redistribue sur les différents processus "léger"

De la même façon, il est apparu nécessaire de gérer les entrées/sorties sur le réseau de telle manière que la machine virtuelle soit non bloquante: un deuxième processus Unix "COM" a donc été introduit pour effectuer les communications entre le réseau et le processus Unix représentant le système gestionnaire d'objets. Ceci est également une contrainte imposée par les moyens de communication inter-activités du système Unix.

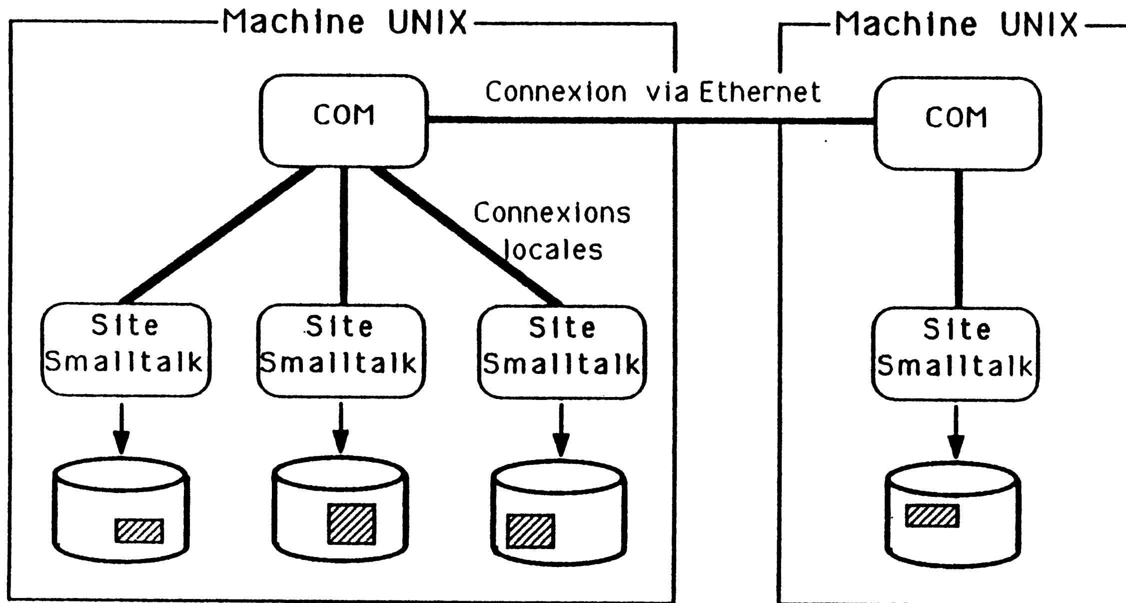


Implémentation du système.

Les deux processus Unix communiquent au moyen de connexions par "sockets". Nous expliquons plus précisément, dans le paragraphe suivant, la nature et les possibilités offertes par une connexion "sockets" du système Unix 4.2 BSD. Pour l'instant, nous pouvons assimiler une "socket" à un port de communication pour un processus Unix. Une "socket" peut être connectée à une autre "socket" d'un autre processus Unix. La connection ainsi établie est bidirectionnelle. De ce point de vue, il est intéressant de noter que les tubes (pipe) standards d'Unix sont mis en œuvre par des connexions sockets dans lesquelles on ne se sert que d'un sens de communication.

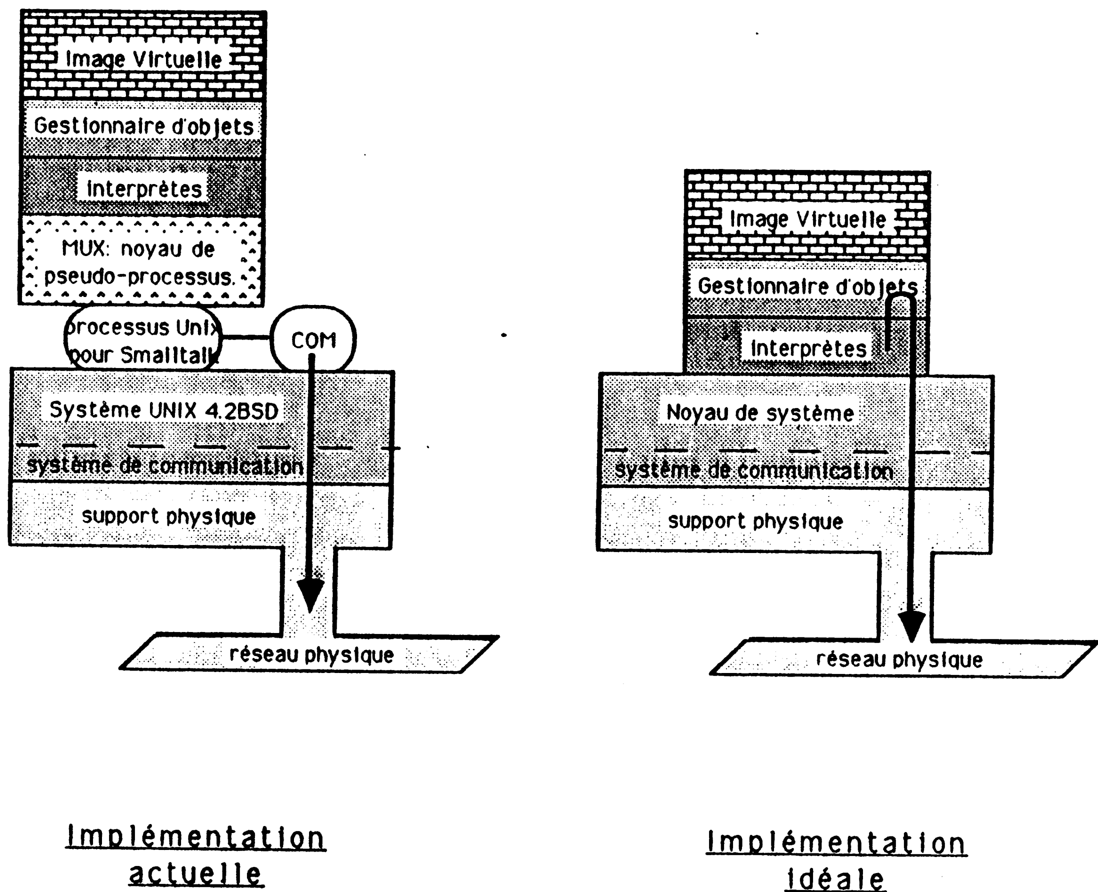
Le processus COM communique avec le processus "mutiplexeur" à l'aide d'une connexion par "socket" interne au système Unix. Ceci nous a permis, dans un premier temps, de nous abstraire des problèmes de communications réseau pour nous concentrer sur les problèmes de mise au point de l'implémentation du gestionnaire d'objets, en permettant à plusieurs maquettes de pouvoir se connecter à un même processus COM. Ces connexions sont dynamiques et permettent à plusieurs implémentations Smalltalk de tourner sur un même calculateur, bien qu'au niveau conceptuel ils représentent des sites complètement indépendants. Ainsi, les communications réseau ont été simulées par des communications par tube, et les connexions/déconnexions des stations de travail sont

simulées par des connexions/déconnexions par socket au processus COM. Ainsi, les tests de la maquette ont pu s'effectuer sur deux niveaux comme le montre la figure suivante:



Répartition géographique des systèmes coopérants.

La figure suivante illustre la structuration en couches du gestionnaire d'objets, afin d'obtenir une réalisation aussi indépendante que possible des supports matériels et logiciels. Il est évident qu'il aurait été préférable de développer un noyau de système directement sur une machine nue, car cet empilement de couches nuit énormément à l'efficacité de l'implémentation. Cependant, cette implémentation aussi inefficace soit elle, ne remet pas en cause la validité du modèle.



Structure en couches de la station de travail Smalltalk-80.

La réalisation a été effectuée et testée sur un Sun 2/50 dont le micro-processeur est un Motorola 68010.

La validité des mécanismes testés en milieu centralisé, à l'aide d'une simulation d'un réseau de sites Smalltalk par des processus d'un même système Unix, n'est pas remise en question par un futur développement en milieu réparti. Pour cette future implémentation, les connexions se seront plus internes, mais passeront par des connexions IPC (InterProcess Communication) distribuées [Leffler83a,Leffler83b]. Ces connexions distribuées s'utilisent couramment sur des protocoles tels que TCP/IP établi au dessus d'Ethernet.

Cependant le choix d'Unix 4.2 BSD pour le développement d'une maquette présente le désavantage d'interdire à plusieurs processus de partager une zone de mémoire commune; cette mémoire commune représentant la mémoire des objets de chaque site. Ce défaut est donc la principale lacune d'Unix 4.2 BSD pour pouvoir en faire un système de simulation d'autres systèmes.
Cette lacune pourra être levée par transport de la maquette sur Unix système V [Thomas86], car celui-ci offre la mémoire partagée entre les activités. A ce propos, il est à

noter que la tendance actuelle des constructeurs de station de travail, comme SUN, consiste à intégrer dans un même système les fonctions du système Unix 4.2 BSD

- *communications par sockets.*
- *connexion dynamique.*
- *possibilité d'attente multiple sur plusieurs connexions.*

et les fonctions du système Unix système V.

- *mémoire partagée.*
- *sémaphores.*
- *gestion de messages.*
- *tubes nommés.*

Ce nouveau système à l'avantage de rassembler les possibilités des deux systèmes Unix en un système unique, qui dés lors peut devenir non seulement le standard Unix, mais aussi un puissant outil de développement d'autres systèmes. Par exemple, ce système offre la mémoire partagée entre processus ainsi que des moyens puissants de communication, mais de plus effectue un cloisonnement rigoureux des espaces privés de ces processus.

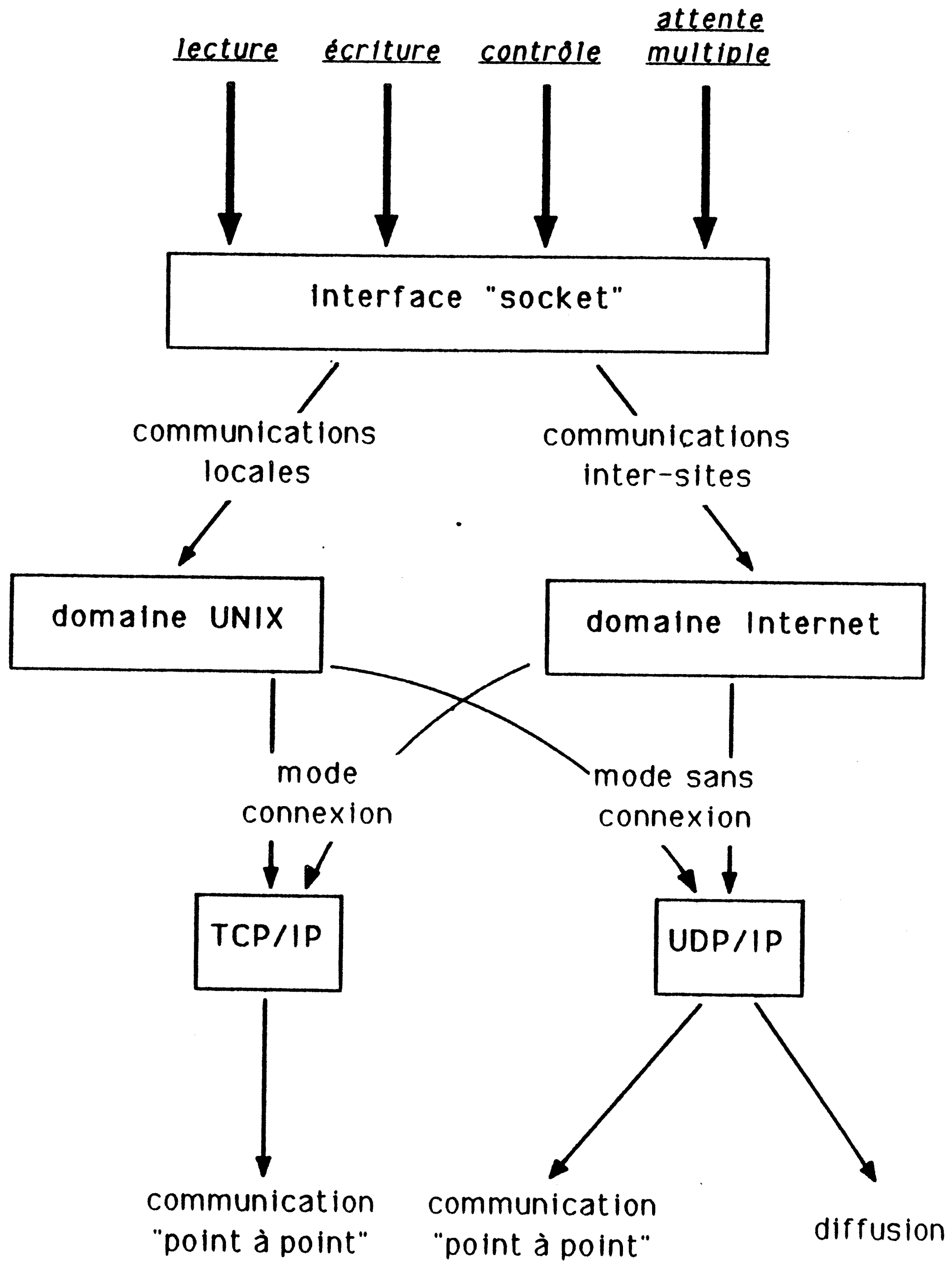
2. Eléments pour une réalisation future.

A titre d'indication, il est intéressant de montrer ce que pourrait être une implémentation efficace et surtout plus propre du gestionnaire distribué d'objets, et du système Smalltalk-80. Il convient avant tout de présenter les possibilités offertes par les sockets d'Unix 4.2 BSD.

Une "socket" est donc un port de communication d'un processus Unix. Sur ce port de communication, on peut effectuer des opérations de lecture et d'écriture, mais aussi des opérations de contrôle:

- connexion dynamique.
- déconnexion dynamique.
- test de la possibilité de lecture/écriture non bloquante.
- attente multiple d'un processus sur plusieurs sockets.

Cette attente multiple peut se faire en mode bloquant, non bloquant ou être limitée dans le temps. Plus généralement, les communications par sockets peuvent se font dans un domaine qui peut être par exemple UNIX, Internet, ...



Les types de communications intéressantes pour l'implémentation future.

Dans le domaine Unix les communications s'effectuent entre deux processus locaux d'un même système.

Internet est le domaine utilisé par le réseau américain ARPA. Dans ce domaine, on peut effectuer des communications, en mode connecté (TCP) ou en mode déconnecté (UDP), avec deux sockets locales à un même site ou réparties sur deux sites différents.

En mode connecté, la communication s'effectue, comme on l'a vu précédemment, entre deux sockets locales à un même site ou réparties sur deux sites différents.

En mode déconnecté, la communication s'effectue soit en point à point soit en diffusion. Toutes les informations transmises sont distribuées sur toutes les sockets des sites possédant le même nom qui est un codage Unix.

On peut ainsi concevoir la structure de la future implémentation comme ci-après:

1- Tous les processus du gestionnaire d'objets et les processus-interprète sont implémentés par autant de processus Unix.

2- Le processus COM disparaît car maintenant ce sont les processus réseau qui se lient les uns aux autres directement.

3- Chaque processus peut communiquer avec n'importe lequel des autres par l'intermédiaire de messages qui font partie des fonctionnalités spécifiques à Unix system V.

4- Les processus réseau communiquent à l'aide de deux sockets du domaine Internet en mode UDP:
   - L'une fonctionne en diffusion pour véhiculer tous les messages relatifs à la gestion des objets représentants et pour les protocoles de récupération d'objets répartis.
   - L'autre fonctionne en point à point pour les communications relatives par exemple à la migration d'objets, et plus généralement pour toutes les transactions entre deux sites.

Les communications entre les différents gestionnaires du réseau sont asynchrones et non bloquantes, ce qui leurs permet de rester à l'écoute des requêtes des autres gestionnaires centralisés d'objets (gestionnaires des mémoires centrale et secondaire).

Il est évident que cette implémentation serait beaucoup plus efficace que celle qui a été développée. Cependant, celle-ci peut encore être amélioré par l'implémentation d'une partie de la gestion des objets non pas par une couche de logiciel, mais par un support matériel micro-programmé: topographie, translation d'adresse, génération d'interruption en cas d'une faute d'adressage, ...

VII. CONCLUSION.

L'objectif de ce travail était de concevoir un mécanisme de gestion d'objets pour un système réparti composé de postes de travail Smalltalk-80. La conception du système s'est déroulé en deux phases: une première phase au cours de laquelle un gestionnaire d'objets a été développé pour la réalisation d'une mémoire virtuelle d'objets centralisée, et une deuxième phase au cours de laquelle le gestionnaire centralisé d'objets a été étendu pour permettre l'expression du partage et de la répartition des objets. Plusieurs considérations ont influencé les choix d'architecture faits pour la réalisation de cette mémoire virtuelle distribuée d'objets.

Tout d'abord, l'organisation en objets du système Smalltalk-80 centralisé a logiquement imposé l'objet comme unité d'échange entre les niveaux de mémoire.

Smalltalk est un système supporté par une couche d'interprétation qui simule une machine virtuelle capable d'interprèter le code machine Smalltalk-80. Ce code fait partie de l'image virtuelle et est engendré par un compilateur décrit sous forme d'objets et de classes. On aurait pu développer et tester cette mémoire des objets en la réalisant également sous forme de classes Smalltalk-80. Pour cela, il aurait fallu modifier le compilateur en plus de la couche d'interprétation. Ce travail représentant un investissement trop important en temps et en volume de travail, il a paru plus raisonnable de ne modifier que la partie "gestion d'objets" qui initialement appartenait à la couche d'interprétation. Ce choix a permis de conserver la définition actuelle du système Smalltalk-80, et de concevoir un système Smalltalk-80 distribué compatible avec la version centralisée dans lequel la fonction de gestion d'objets a été isolée de la couche d'interprétation. Pour assurer cette compatibilité, l'interface entre la couche d'interprétation et le gestionnaire d'objets doit donc rester conforme à celle de la définition initiale de l'interprète.

A partir de la définition centralisée de la mémoire virtuelle d'objets, une extension en milieu réparti du gestionnaire d'objets a été conçue.

Smalltalk-80 est un système très homogène, aussi il était important de conserver cette homogénéité dans l'expression de la répartition. Pour cela, un nouveau type d'objet a été défini: "le représentant". Un représentant est un objet-descripteur localisant sur le réseau l'objet réel qu'il représente. La notion de représentant peut être considérée comme une extension au modèle des objets des mécanismes de l'appel de procédure à distance, qui permet en outre la mobilité des objets. Un objet représentant s'intègre parfaitement dans la structure de l'image virtuelle. Chaque objet Smalltalk est en fait une entité composée de références locales vers d'autres objets. Pour conserver cette composition, l'objet

représentant offre une référence locale (pointeur-objet par lequel on désigne l'objet) pour désigner un objet distant. L'accès à un objet représentant s'effectue de façon automatique et transparente aux utilisateurs. Cependant, l'homogénéité n'a pas été poussée jusqu'à faire apparaître les objets représentants sous forme d'exemplaires d'une classe du système Smalltalk. Ceux-ci sont uniquement des objets internes au gestionnaire des objets.
Ce choix résulte des considérations suivantes:

- rendre les objets représentants invisibles à l'utilisateur,
- assurer la protection des objets partagés par les différents sites.

Sur la base du choix des objets représentants pour exprimer le partage et la répartition des objets, divers aspects ont été traités: l'unicité ou non sur un site d'un représentant pour un objet reél donné, l'élaboration d'un mécanisme de migration d'objets, la protection des objets partagés et l'élaboration d'un mécanisme de connexion/ déconnexion dynamique d'une station au système réparti. La réalisation de ces mécanismes a nécessité l'introduction de primitives supplémentaires dans l'interface d'accès aux objets.

Le problème de la récupération des objets a été soulevé, mais ne constituait pas un aspect central de ce travail. Une adaptation en milieu réparti, d'un algorithme de marquage défini en centralisé a été proposée. Celui-ci est facile à mettre en œuvre, mais s'avère coûteux. Il faut cependant noter que la structure du gestionnaire d'objets que nous avons défini a permis de diminuer l'importance de ces mécanismes de récupération: ceux-ci ne doivent plus s'appliquer à la mémoire de traitement, mais au support permanent des objets (mémoire secondaire). Cela provient du fait que les objets en mémoire centrale, qui n'ont pas été touchés pendant un intervalle de temps fixé, sont ramenés dans la mémoire secondaire. Par le même mécanisme, les objets récupérables disparaissent tôt ou tard de la mémoire centrale. Le traitement associé à cette récupération d'objets n'a plus lieu d'être immédiat et il peut être reporté à un moment où il est moins pénalisant pour l'utilisateur (taux de charge réduit, fin de session de travail, ...).

Une première maquette a été développé sur Unix 4.2BSD, au dessus d'un noyau de processus (Mux), qui offre la possibilité de créer des activités parallèles au sein d'un même processus Unix. Mux a été nécessaire pour permettre aux différentes activités réalisant la fonction de gestion d'objets de partager la mémoire centrale des objets où sont chargés et accédés les objets. L'addition de cette couche de multiplexage a dégradé les performances de la maquette. Un portage de la maquette sur Unix système V devrait permettre une évaluation plus précise des coûts réels introduits par les mécanismes de gestion d'objets. De même, il faudra attendre la possibilité de pouvoir développer de véritables applications avec ce Smalltalk-80 distribué pour juger de la viabilité d'une telle conception.

Il faut noter que les concepts développés proviennent directement du choix qui a consisté à désigner un objet distant par un nom local à chaque site. Un autre choix aurait été de fournir à l'ensemble des sites un moyen de désignation global a tous les sites. Dans ce cas, la notion de représentant disparait et la définition d'un serveur de désignation global devient nécessaire. Celui-ci doit alors être capable de localiser un objet à partir d'un nom unique.

Actuellement un certain nombre de systèmes à objets sont en cours de développement. Cependant, l'absence de système réparti à objets réellement opérationnel ne nous permet pas de conclure sur l'adéquation de ce modèle pour l'expression de la répartition. Ce problème est encore ouvert.

Plusieurs aspects du système n'ont pas été développés et pourraient faire l'objet de travaux ultérieurs:

Le problème important des reprises après panne n'a pas été traité. Un site qui possède des objets partagés et qui tombe en panne, rend incohérentes les applications des autres sites qui utilisent ces objets. De plus, comme ces objets Smalltalk peuvent être des classes ou plus généralement des objets centraux pour les autres sites, un site qui tombe en panne peut entraîner la défaillance de tous les autres sites. De même, le problème du contrôle des accès concurrents à un objet n'a pas été traité.

Le gestionnaire développé offre des primitives pour exprimer le partage et la répartition au niveau de l'objet. Ces primitives permettent l'expression du partage dans une application. On sent la nécessité de développer, au niveau du système Smalltalk, des outils qui permettent à l'usager d'un poste de travail de partager des ressources avec d'autres usagers. De ce point de vue, il serait utile d'étendre à un système réparti les fonctions d'un outil graphique tel que le "Browser", qui permet à l'usager d'inspecter et de modifier son système Smalltalk.

Il aurait été intéressant de développer un mécanisme de transactions dans le gestionnaire d'objets. Ce mécanisme serait défini sur la base de la définition de deux nouvelles primitives: "debut-transaction" et "fin-transaction", qui permettraient de différer et de rendre atomique la mise à jour, en mémoire permanente, d'un ensemble d'objets manipulés par l'application. Cet ensemble d'objets pourrait être déterminé par l'introduction d'un attribut supplémentaire au niveau de chaque objet, initialisé par le concepteur de l'application.

La maquette développée a montré la faisabilité des mécanismes de gestion du partage et de la migration de l'information, dans un système réparti à objets tel que Smalltalk-80. Cependant, l'implémentation de l'interprète Smalltalk-80, qu'il s'est avéré nécessaire de refaire, n'a pas été achevée. Pour cette raison, nous ne pouvons conclure sur la validité de ces mécanismes pour la gestion des objets du système Smalltalk.

Ce travail n'était qu'une pré-étude, qui a pour prolongement le projet Guide [Balter86b], qui vient de démarrer au sein du Laboratoire de Génie Informatique de Grenoble en association avec le Centre de Recherche Bull. Sa conception est de même fondée sur le modèle de structuration en objets. Il a pour but le développement d'un langage et d'un système d'exploitation pour des postes de travail individuels et des serveurs interconnectés par un réseau local, utilisés pour des applications générales.

Dans un premier temps, un langage et un système expérimental doivent être définis. Ceux-ci ont pour buts principaux la validation de nouvelles méthodes de conception et la mise en œuvre d'outils pour l'expression et le développement d'applications réparties.

Un langage utilisant le modèle des objets est en cours de développement et essaie d'intégrer la relation de conformité telle qu'elle est définie dans Emerald, et la relation d'héritage définie dans Smalltalk-80. La relation de conformité permet d'effectuer des contrôles de types à la compilation, et ainsi assure une programmation correcte et efficace. La relation d'héritage permet de reporter les vérifications de types à l'exécution, ce qui autorise une programmation réellement incrémentale. L'usager peut alors doser comme il le veut les degrés de vérifications statiques et dynamiques.

Parmi les aspects développés par le système Guide, la gestion de *domaines* (environnement d'exécution) est un aspect central. Celui-ci provient directement du fait que le système Guide doit être un système multi-usagers offrant un niveau de cloisonnement pour l'adressage des objets entre activités. Un domaine pourra être réparti entre plusieurs sites.

La gestion des objets constitue un point de recherche central. Cependant, celle-ci diffère de celle développée dans cette thèse par le fait qu'elle doit gérer un espace de noms global à tous les sites: un serveur de nom doit être ainsi conçu. De plus, la gestion des objets en mémoire permanente, qui n'a pas été traitée dans ce travail, est également un point de recherche important. Cette gestion doit de plus offrir des primitives qui permettent de mettre en place des mécanismes de transactions et de reprises après panne.

BIBLIOGRAPHIE.

[Albano85] **Albano A., Cardelli L., Orsini R.** : *Galileo: A Strongly Typed, Interactive Conceptual Language ;* ACM TODS, vol. 10, no. 2, 1985. *(pp.230-260).*

[Almes80] **Almes G.T.** : *Garbage Collection in an object-oriented system ;* Ph.D.Thesis. CMU-CS-80-128, Carnegie-Mellon University, Pittsburgh (Pensylvania), June 1980.

[Almes85] **Almes G.T., Black A.P., Lazowska E.D., Noe J.D.** : *The Eden System: A Technical Review ;* IEEE Transactions on Software Engineering, no. SE-11,1, IEEE, January 1985. *(pp.43-59).*

[Balter86a] **Balter R., Vandôme G., Donnelly A., Finn E., Horn C.** : *Systèmes Distribués sur réseau local: Analyse et classification ;* Projet ESPRIT 834, Rapport Interne DSG/CRG/86003, BULL, Décembre 1986.

[Balter86b] **Balter R., Krakowiak S., Meysembourg M., Roisin C., Rousset de Pina X., Scioville R., Vandôme G.** : *Principes de conception du système d'exploitation réparti GUIDE ;* BIGRE+Globule, no. 52, Décembre 1986.

[Barbedette87] **Barbedette G., Richard P.** : *TOOL: Un langage typé orienté objet pour les bases de données ;* Actes des troisièmes journées Bases de Données Avancées, INRIA, Port-Camargue (FRANCE), 20-22 Mai 1987. *(pp.47-67).*

[Batson77] **Batson A.P., Brundage R.E.** : *Segment Sizes and Lifetimes in Algol 60 Programs ;* Communications of the ACM, vol. 20, no. 1, University of Virginia, Charlottesville, January 1977. *(pp.36-44).*

[Bays77] **Bays C.** : *A comparison of next-fit, first-fit, and best-fit ;* Communications of the ACM, vol. 20, no. 3, March 1977. *(pp.191-192).*

[Bézivin84] **Bézivin J.** : *Simulation et langages orientés objets. ;* Deuxièmes Journées dEtude sur les Langages Orientés Objets., no. 41, BIGRE+Globule, BREST, 22-23 Novembre 1984.

[Birrell84] **Birrell A.D., Nelson B.J. :** *Implementing Remote Procedure Calls ;* ACM Transactions on Computer Systems, vol. 2, no. 1, February 1984.

[Black85] **Black A.P. :** *Supporting distributed applications: experience with Eden. ;* Proceedings of the 10th ACM SIGOPS Conference, December 1985.

[Black86] **Black A.,Hutchinson N., Jul E., Levy H. :** *Object Structure in the Emerald System ;* OOPSLA86. Special issue of the SIGPLAN notices., vol. 21, no. 11, ACM, Portland, Oregon, September 29 - October 2, 1986. *(pp.78-86).*

[Bobrow80] **Bobrow D.G. :** *Managing reentrant structures using reference counts ;* ACM Transactions on Programming Languages & Systems, vol. 2, no. 3, July 1980. *(pp.269-273).*

[Bobrow83] **Bobrow D.G., Stefik M. :** *The LOOPS Manual ;* XEROX, Palo Alto Research Center, December 1983.

[Booch86] **Booch G. :** *Object-Oriented Development ;* IEEE Transactions on Software Engineering, vol. SE-12, no. 2, IEEE, February 1986.

[Bozman84] **Bozman G., Buco W., Daly T.P., Tetzlaff W.H. :** *Analysis of free storage algorithms ;* IBM Systems Journal, vol. 23, no. 1, 1984.

[Brownbridge82] **Brownbridge D.R, Marshall L.F., Randell B. :** *The Newcastle Connection - or UNIXes of the World Unite! ;* Software Practice and Experience, vol. 12, no. 12, December 1982., *(pp.1147-1162).*

[CCA83] **CCA :** *ADAPLEX: Rationale and Reference Manual ;* Technical Report CCA-83-03, Computer Corporation of America, 1983.

[CROCUS75] **CROCUS :** *Système d'exploitation des ordinateurs ;* Dunod 1975.

[Christophe84] **Christopher T.W. :** *Reference Count garbage collection ;* Software Practice & Experience, vol. 14, no. 6, June 1984. *(pp.503-507).*

[Chu76] **Chu W.W., Opderbeck H. :** *Analysis of the PFF replacement algorithm via a semi-Markov model ;* Communications of the ACM, vol. 19, no. 5, University of California, Los Angeles, May 1976.

[Copeland84] **Copeland G., Maier D. :** *Making Smalltalk a Database System ;* Proceedings of the ACM SIGMOD Conference, Boston (US), June 18-21 1984. *(pp.316-325).*

[Cosserat75] **Cosserat D.** : *A data model based on the capability protection mechanism ;* RAIRO-Informatique(AFCET), vol. 9, Septembre 1975. *(pp.63-78).*

[Courtiat87] **Courtiat J-P., Dembinski P., Groz R., Jard C.** : *ESTELLE: un langage ISO pour les algorithmes distribués et les protocoles ;* Technique et Science Informatiques, vol. 6, no. 2, 1987. *(pp.89-102).*

[Coutaz87] **Coutaz J.** : *The Construction of User Interfaces and the Object Paradigm ;* European Conference on Object Oriented Programming, AFCET, Paris, June 15-17 1987.

[Cox86] **Cox B.J.** : *Object Oriented Programming: An Evolutionary Approach. ;* Addison-Wesley Publishing Company., 1986.

[Dahl66] **Dahl O.J., Nygaard K.** : *SIMULA. An Algol-based simulation language. ;* Communications of the ACM, no. 9, 1966. *(pp.671-678).*

[Dahl70] **Dahl O-J, Myhrhaug B., Nygaard K.** : *The SIMULA 67 Common Base Languages. ;* Norwegian Computing Center S-22, Oslo, Norway, 1970.

[Dannenberg82] **Dannenberg R.B.** : *Resource sharing in a network of personal computers ;* Department of Computer Science, Carnegie Mellon University, December 1982.

[David87] **David J-P., Legendre M., Natkin S.** : *Un protocole de diffusion fiable pour les réseaux locaux industriels ;* Technique et Science Informatiques, vol. 6, no. 2, 1987. *(pp.187-190).*

[Decitre83] **Decitre P., Khider A., Roisin C., Sanchez V., Vandôme G., Vatton I.** : *Spécification d'un protocole de conversation fiable sur le réseau DANUBE ;* Rapport de recherche IMAG, no. 368, IMAG BP 68, Grenoble, Mars 1983.

[Decouchant84] **Decouchant D.** : *Une mémoire objet pour Smalltalk-80 ;* BIGRE+Globule: Actes des deuxièmes journées d'étude du groupe de travail AFCET-Informatique sur les Langages Orientés Objets., no. 41, Brest, Novembre 1984.

[Decouchant86] **Decouchant D.** : *Design of a Distributed Object Manager For The Smalltalk-80 System. ;* OOPSLA86. Special issue of the SIGPLAN notices., vol. 21, no. 11, ACM, Portland, Oregon, September 29 - October 2, 1986. *(pp.444-452).*

[Denning70] **Denning P.J. :** *Virtual memory ;* Computing Surveys, vol. 2, no. 3, September 1970.

[Denning73] **Denning P.J., Coffman E.G. :** *Operating system theory ;* Prentice Hall, 1973.

[Deutsch76] **Deutsch L.P., Bobrow D.G. :** *An efficient, incremental, automatic garbage collector ;* Communications of the ACM, vol. 19, no. 9, September 1976. *(pp.522-526).*

[Deutsch83] **Deutsch L.P., Schiffman A.M. :** *Efficient implementation of the Smalltalk-80 system ;* 11th Principles of Programming Languages Conference, ACM, Salt Lake City, Utah, 1984. *(pp.297-302).*

[Dijkstra78] **Dijkstra E.W., Lamport L., Martin A.J., Scholten C.S., Steffens E.F.M. :** *On-the-Fly Garbage Collection: An Exercise in Cooperation. ;* Communications of the ACM, vol. 21, no. 11, ACM, November 1978. *(pp.966-975).*

[Dijkstra83] **Dijkstra E.W., Feijen W.H.J., Van Gasteren :** *Derivation of a termination detection algorithm for distributed computations ;* IPL 16, 1983. *(pp.217-219).*

[Ferrié76] **Ferrié J., Lanciaux D. :** *Le systeme Plessey 250 ;* Rapport de recherche INRIA, no. 168, Avril 1976.

[Fikes85] **Fikes R., Kehler T. :** *The Role of frame-based representation in reasoning ;* Communications of the ACM, vol. 28, no. 9, 1985. *(pp.904-920).*

[Gibbs83] **Gibbs S. :** *An object-oriented office data model. ;* Doctoral dissertation, Dept. of Computer Science, University of Toronto, 1983.

[Goldberg76] **Goldberg A., Kay A. :** *Smalltalk-72 Instruction Manual. ;* Xerox Palo Alto Research Center, Palo Alto, California., March 1976.

[Goldberg83] **Goldberg A., Robson D. :** *Smalltalk-80: The language and its implementation ;* Addison-Wesley Publishing Company, 1983.

[Goldberg84] **Goldberg A. :** *Smalltalk-80: The interactive programming environment ;* Addison-Wesley Publishing Company, 1984.

[Halstead85] **Halstead R.H.** : *Multilisp: a language for concurrent sumbolic computation ;* ACM Transactions on Programming Languages and Systems, vol. 7, no. 4, October 1985. *(pp.501-538).*

[Hirschberg73] **Hirschberg D.S.** : *A class of dynamic memory allocation ;* Communications of the ACM, vol. 16, no. 10, October 1973. *(pp.615-618).*

[Holt83] **Holt R.C.** : *Concurrent Euclid, the Unix System, and Tunis. ;* Addison-Wesley Publishing Company., 1983.

[Hughes85] **Hughes J.** : *A Distributed Garbage Collection Algorithm ;* Lecture Notes in Computer Science. Functional Programming Languages and Computer Architecture, Springer-Verlag éditeur, Nancy (FRANCE), Septembre 1985. *(pp.256-272).*

[Hullot85] **Hullot J.M.** : *CEYX: Programmer en Ceyx. ;* Rapport Technique, no. 45, INRIA, Centre de Rocquencourt, Février 1985.

[Hullot86] **Hullot J.M.** : *SOS Interface, un Générateur d'Interfaces Homme-Machine ;* Actes des Journées Sfcet-Informatique sur les Langages Orientés Objet, no. 48, Bigre+Globule, Paris, 8-10 Janvier 1986. *(pp.69-78).*

[Ingalls78] **Ingalls D.H.** : *The Smalltalk-76 Programming System Design and Implementation. ;* 5th Annual ACM Symposium on Principles of Programming Languages., January 1978. *(pp.9-15).*

[Kaehler81] **Kaehler T.** : *Virtual Memory for an Object-Oriented Language. ;* BYTE, vol. 6, no. 8, August 1981.

[Kaehler82] **Kaehler T., Krasner G.** : *LOOM - Large Object-Oriented Memory for Smalltalk-80 systems. ;* Smalltalk-80. Bits of History, Words of Advice., Addison-Wesley Publishing Company., 1982.

[Kaehler86] **Kaehler T.** : *Virtual Memory on a Narrow Machine for an Object-Oriented Language. ;* OOPSLA86. Special issue of the SIGPLAN notices., vol. 21, no. 11, ACM, Portland, Oregon, September 29 - October 2, 1986. *(pp.87-106).*

[Kernighan78] **Kernighan B.W., Ritchie D.M.** : *The C Programming Language ;* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Knowlton65] **Knowlton K.C. :** *A fast storage allocator ;* Communications of the ACM, vol. 8, no. 10, October 1965. *(pp.623-625).*

[Knuth68] **Knuth D.E. :** *The Art of Computer Programming Vol I: Fundamental Algorithms ;* Addison-Wesley, Reading, Massachusetts, 1968.

[Krakowiak87] **Krakowiak S. :** *Systèmes d'exploitation répartis ;* Technique et Science Informatiques, vol. 6, no. 2, 1987. *(pp.151-161).*

[Krasner83] **Krasner G. :** *Smalltalk-80: bits of history, words of advice ;* Addison-Wesley Publishing Company, 1983.

[Leach83] **Leach P.J., Levine P.H., Douros B.P., Hamilton J.A., Nelson D.L., Stumpf B.L. :** *The architecture of an integrated local network ;* IEEE Journal on Selected Areas in Communication, November 1983. *(pp.842-856).*

[Leffler83a] **Leffler S.J., Fabry R.S., Joy W.N. :** *A 4.2 BSD Interprocess Communication Primer ;* Computer Systems Research Group., Department of Electrical Engineering and Computer Science., University of California, Berkeley, CA 94720., July 27, 1983.

[Leffler83b] **Leffler S.J., Fabry R.S., Joy W.N. :** *4.2 BSD Networking Implementation Notes. ;* Computer Systems Research Group., Department of Electrical Engineering and Computer Science., University of California, Berkeley, CA 94720., July 1983.

[Lieberman83] **Lieberman H., Hewitt C. :** *A Real-Time Garbage Collector Based on the Lifetimes of Objects ;* Communications of the ACM, vol. 26, no. 6, June 1983.

[Meyer86] **Meyer B. :** *Genericity versus Inheritance ;* OOPSLA86. Special issue of the SIGPLAN notices., vol. 21, no. 11, ACM, Portland, Oregon, September 29 - October 2, 1986. *(pp.391-405).*

[Michel86] **Michel B. :** *Ramasse-miettes Distribués ;* Rapport de DEA, Université de Rennes. IRISA (FRANCE), Juin 1986.

[Mylopoulos80] **Mylopoulos J., Bernstein P.A., Wong H.K.T. :** *A Language Facility for Designing Database-Intensive Aplications ;* ACM TODS, vol. 5, no. 2, 1980. *(pp.185-207).*

[Nelson83] **Nelson D.L. :** *DOMAIN: Informatique Distribuée dans le Système "DOMAIN" d'Apollo. ;* Actes des Conférences du Printemps Convention Informatique, 1983. *(pp.192-195).*

[Nicholls75] **Nicholls J.E.** : *The Structure and Design of Programming Languages ;* Addison-Wesley, Reading, Massachusets, 1975.

[Nierstrasz85] **Nierstrasz O.M., Tsichritzis D.C.** : *An Object-Oriented Environment for OIS Applications ;* Proceedings of Very Large Data Bases, Stockholm, 1985. *(pp.335-345).*

[Organick72] **Organick E.I.** : *The Multics System. An Examination of Its Structure ;* MIT Press, Massachusetts Institute of Technology, 1972.

[Peterson77] **Peterson J.L., Norman T.A.** : *Buddy Systems ;* Communications of the ACM, vol. 20, no. 6, June 1977. *(pp.421-431).*

[Pratt75] **Pratt T.W.** : *Programming Languages: Design and Implementation ;* Prentice-Hall, Englewood Cliffs, 1975.

[Ritchie74] **Ritchie D.M., Thompson K.** : *The Unix time-sharing system. ;* Communications of the ACM, vol. 17, no. 7, ACM, July 1974.

[Sandberg86] **Sandberg R.** : *The Sun Network File System: design, implementation and experience ;* EUUG: European UNIX systems User Group, Florence, April 1986.

[Shapiro85] **Shapiro M., Habert S., Dumeur R., Fekete J-D.** : *SOMIW Operating System Design ;* Esprit project 367, Technical report., December 1985.

[Shapiro86] **Shapiro M.** : *Structure and Encapsulation in Distributed Systems. ;* Proceedings of 6th International Conference on Distributed Computer Systems., Boston MA, May 1986.

[Shen74] **Shen K.K., Peterson J.L.** : *A Weighted Buddy System for Dynamic Storage Allocation ;* Communications of the ACM, vol. 17, no. 10, October 1974. *(pp.558-562).*

[Shin85] **Shin H., Malek M.** : *Parallel Garbage Collection with Associative Tag ;* Proceedings of the 1985 International Conference on Parallel Processing, IEEE, August 20-23 1985. *(pp.369-375).*

[Sibert86] **Sibert J.L., Hurley W.D., Bleser T.W.** : *An Object Oriented User Interface Management System ;* SIGGRAPH'86, vol. 20, no. 4, 1986. *(pp.259-268).*

[Stamos84] **Stamos J.W.** : *Static grouping of small objects to enhance performance of a paged virtual memory ;* ACM Transactions on computer systems, vol. 2, no. 2, May 1984.

[Stefik85] **Stefik M., Bobrow D.** : *Object-Oriented Programming: Themes and Variations. ;* The AI Magazine, 1985.

[Stroustrup86] **Stroustrup B.** : *The C+ + Programming Language ;* Addison-Wesley Publishing Company, 1986.

[Sun84a] **Sun** : *Remote Procedure Call Protocol Specification ;* Sun Microsystems Inc., October 1984.

[Sun84b] **Sun** : *Remote Procedure Call Reference Manual ;* Sun Microsystems Inc., October 1984.

[Suzuki83] **Suzuki N., Terada M.** : *Creating efficient systems for object-oriented languages ;* 11th Principles of Programming Languages Conference, ACM, Salt Lake City, Utah, 1984. *(pp.290-296).*

[Tanenbaum85] **Tanenbaum A.S., Van Renesse R.** : *Distributed Operating Systems ;* ACM Computing Surveys, vol. 17, no. 4, December 1985. *(pp.419-470).*

[Thatte86] **Thatte S.M.** : *Persistent Memory: A Storage Architecture for Object-oriented Database Systems. ;* Proceedings of the International Workshop on Object-Oriented Database systems, IEEE, Pacific Grove (US), September 23-26 1986. *(pp.148-159).*

[Thomas86] **Thomas R., Rogers L.R., Yates J.L.** : *Advanced Programmer's Guide To UNIX SYSTEM V. ;* Osborne McGraw Hill publisher, Berkeley, California 94710, 1986.

[Thorelli72] **Thorelli L.E.** : *Marking algorithms ;* BIT, vol. 12, no. 4, 1972. *(pp.555-568).*

[Tricot87] **Tricot C.** : *Mux: A lightweight Multiprocessor Subsystem Under Unix ;* EUUG: European UNIX systems User Group, Helsinki and Stockholm, May 12-14 1987.

[Ungar84a] **Ungar D.** : *Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm ;* First Symposium on Practical Software Development Environments, ACM Software Engineering, Pittsburgh, April 1984. *(pp.157-167).*

[Ungar84b] **Ungar D., Blau R., Foley P., Samples D., Patterson D.** : *Architecture of SOAR: Smalltalk on a RISC.* ; IEEE, Ann Arbor, MI, June 1984. *(pp.188-197).*

[Ungar87] **Ungar D., Patterson D.** : *What Price Smalltalk?* ; Computer, vol. 20, no. 1, IEEE, January 1987.

[Wiseman85] **Wiseman S.R.** : *A Garbage Collector for a large distributed address space* ; RSRE Report 85009, June 1985.

[Xerox83] **Xerox** : *Interlisp Reference Manual* ; XEROX Parc., Palo Alto Research Center, October 1983.

ANNEXE A: LES OBJETS COMPOSANT L'ENVIRONNEMENT D'EXECUTION DU SYSTEME SMALLTALK-80.

Nous présentons en détail les différents objets, qui interviennent dans le système Smalltalk-80 pendant l'exécution. Chacun d'eux est caractérisé par sa fonction, sa durée de vie, sa portée et son implémentation.

1. Les méthodes.

Durée de vie:
Les méthodes sont des objets statiques.
Ce sont des objets qui ne peuvent varier que par destruction ou modification explicite de la part de l'utilisateur. On peut donc les classer parmi les objets constants du système.

Portée:
Chaque méthode ne peut être atteinte que pour le traitement d'un objet exemplaire de sa classe ou de l'une de ses sous-classes.

Implémentation:
Une méthode étant un objet constant, sa taille et son contenu ne varient pas au cours du temps.

en-tête
zone des littéraux
code de la méthode

Légende:

en-tête: zone contenant des informations relatives à l'interprètation de la méthode.
Cétte zone contient par exemple, le nombre d'arguments de la méthode.

zone des litéraux: zone contenant des références à d'autres objets référencés par le code de la méthode.
Celle-ci inclut:
- les modèles des messages envoyés par la méthode.
- les variables temporaires de la méthode.
- les constantes de la méthode (ex: le nom d'une classe).

zone du code: zone contenant les instructions de la méthode.

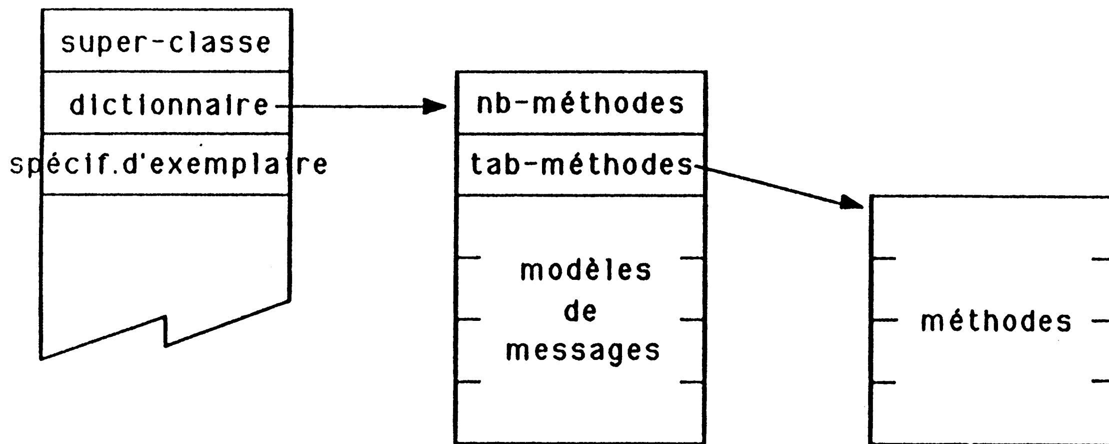
2. Les classes.

Pour traiter un message, l'interprète recherche la méthode compilée à exécuter à l'aide des dictionnaires de messages des classes sur lesquelles s'effectuent les recherches. Le dictionnaire de messages est trouvé dans la classe du destinataire ou dans l'une de ses super-classes. L'objet classe doit donc contenir les informations permettant de remonter l'arbre des classes du système.
De même, chaque objet-classe contient les informations structurelles qui décrivent la composition (spécification de l'état) et le comportement de chacune des ses

exemplaires (dictionnaire de méthodes qui associe une méthode compilée à chaque modèle de message).

Durée de vie:
Une classe est un objet statique. Son existence dépend de l'utilisateur.

Portée:
Une classe est un objet global du système.



Légende:

super-classe: pointeur vers l'objet super-classe.

dictionnaire: pointeur vers la table des modèles de messages.

spécif.d'exemplaire: spécification des exemplaires de la classe.

nb.méthodes: nombre de paires (modèle, méthode) dans le dictionnaire.

tab.méthodes: pointeur vers la table des méthodes associées aux modèles de messages.

3. Les contextes de méthodes et de blocs.

L'interprète utilise des contextes pour représenter l'état d'exécution des méthodes compilées et des blocs. On distingue deux sortes de contextes:

- Les contextes de méthodes

  Un contexte de méthode représente l'état d'exécution d'une méthode compilée par l'interprète.

- Les contextes de blocs

  Un contexte de bloc représente l'état d'exécution d'un bloc rencontré dans une méthode compilée. Un contexte de bloc référence le contexte de la méthode qui contient le bloc. Le corps d'un bloc est un sous ensemble des instructions d'une méthode, dont l'exécution peut être différée. Les blocs sont utilisés pour implémenter des structures de contrôle.

Durée de vie:

Un contexte de méthode ou de bloc n'est que l'état de l'interprète à un stade particulier de l'exécution d'une méthode ou d'un bloc. Cet état est tout à fait fugitif. Lors de l'envoi d'un message par une méthode, le contexte de la méthode courante est suspendu et un nouveau contexte est créé pour traiter le message. Une fois la méthode du message terminée, l'exécution reprend avec le contexte initial. Les contextes sont des objets gérés en pile: il n'y a pas de pile explicite, mais chaque contexte contient un pointeur vers le contexte qui l'a créé pour traiter un message. Chaque contexte contient une pile d'exécution qui est de taille réduite.
Les contextes sont des objets créés et détruits par l'interprète.

Portée:
L'objet contexte n'a de relations qu'avec d'autres objets contextes.

Implémentation:
A ce stade, pour caractériser convenablement l'implémentation des contextes, il convient d'énumérer les ressources qui lui sont nécessaires.

1- La méthode compilée dont le code est exécuté.

2- L'emplacement de l'octet-code qui doit être exécuté au cycle suivant de l'interprète.
C'est le pointeur d'instruction (instruction pointer).

3- Le destinataire et les arguments du message qui a invoqué l'exécution de la méthode compilée.

4- Toutes les variables temporaires nécessaires pour l'exécution de la méthode compilée.

5- Une pile d'exécution.

Le cycle de l'interprète se résume comme suit:

<u>*cycle:*</u>

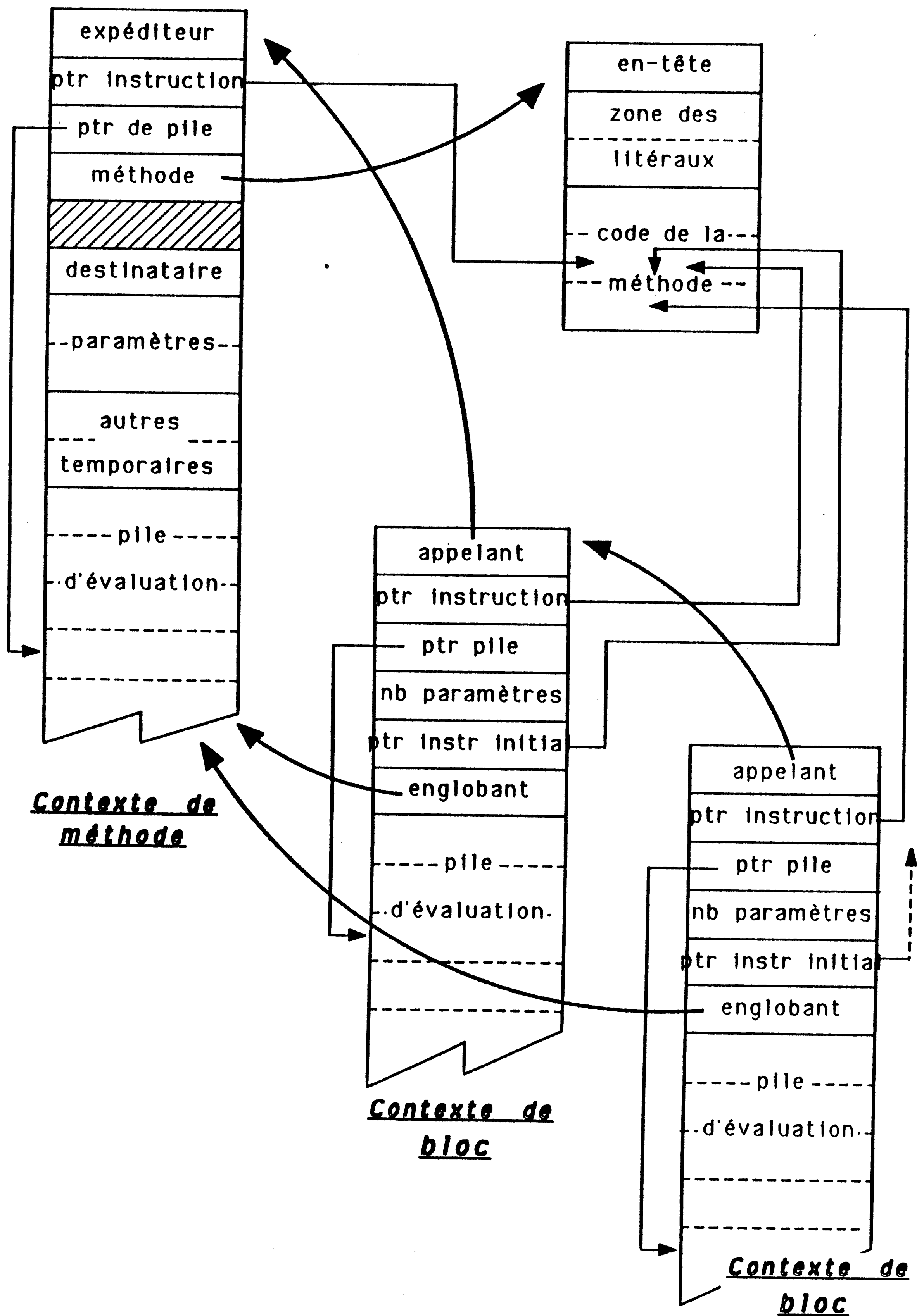
1- aller chercher l'octet-code suivant dans la méthode compilée à l'emplacement indiqué par le pointeur d'instruction.

2- incrémenter le pointeur d'instruction.

3- exécuter les actions spécifiées par l'octet-code.

<u>*fincycle:*</u>

Implémentation d'un contexte de méthode et d'un contexte de bloc. Leur dépendance est également précisée.



expéditeur
ptr instruction
ptr de pile
méthode
destinataire
paramètres
autres
temporaires
pile
d'évaluation
Contexte de méthode
en-tête
zone des
litéraux
code de la
méthode
appelant
ptr instruction
ptr pile
nb paramètres
ptr instr initial
englobant
pile
d'évaluation
Contexte de bloc
appelant
ptr instruction
ptr pile
nb paramètres
ptr instr initial
englobant
pile
d'évaluation
Contexte de bloc

Légende:

Contexte de méthode:

<u>expéditeur:</u> pointeur vers l'objet-contexte qui a envoyé le message nécéssitant l'exécution de la méthode.

<u>ptr instruction:</u> compteur instruction du contexte: désigne la prochaine instruction à exécuter.

<u>ptr pile:</u> sommet de la pile d'exécution du contexte.

<u>méthode:</u> pointeur sur l'objet méthode compilée.

<u>destinataire:</u> pointeur sur l'objet destinataire du message qui a invoqué l'exécution de la méthode compilée.

<u>paramètres:</u> paramètres d'appel de la méthode compilée.

Contexte de bloc:

<u>appelant:</u> pointeur vers l'objet contexte (de bloc ou de méthode) qui a créé ce contexte de bloc.

<u>ptr instruction:</u> compteur instruction du contexte: désigne la prochaine instruction à exécuter.

<u>ptr pile:</u> sommet de la pile d'exécution du contexte.

<u>nb paramètres:</u> nombre de paramètres du bloc.

<u>ptr instr initial:</u> valeur initiale du pointeur d'instruction: cette valeur est déterminée par le compilateur.

<u>englobant:</u> pointeur sur le contexte de la méthode qui contient le bloc.

ANNEXE B: DEFINITION DE LA CLASSE

"BLOCKCONTEXT".

```
ContextPart variableSubclass: #BlockContext
	instanceVariableNames: 'nargs startpc home '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Kernel-Methods'!

BlockContext comment:
'My instances function similarly to instances of MethodContext, but they hold
the dynamic state for execution of a block in Smalltalk. They access all temporary
variables and the method sender via their home pointer, so that those values
are effectively shared. Their indexable part is used to store their independent
value stack during execution.

My instance must hold onto its home in order to work. This can cause circularities
if the home is also pointing (via a temp, perhaps) to the instance. In the
rare event that this happens (as in SortedCollection sortBlock:) the message
fixTemps will replace home with a copy of home, thus defeating the sharing of
temps but, nonetheless, eliminating the circularity.

Instance Variables: *indexed*
	nargs <Integer> number of block arguments
	startpc <Integer>
	home <Context>
'!


!BlockContext methodsFor: 'initialize-release'!

home: aContextPart startpc: position nargs: anInteger
	"This is the initialization message. The receiver has been
	initialized with the correct size only."

	home ← aContextPart.
	startpc ← position.
	nargs ← anInteger.
	pc ← position.
	stackp ← 0! !

!BlockContext methodsFor: 'accessing'!

fixTemps
	"Fix the values of the temporary variables used in the block that are
	ordinarily shared with the method in which the block is defined."
```

```
        home ← home copy.
        home swapSender: nil!

hasMethodReturn
        "answer true if there is an ↑ in the code of this block"
        | method scanner end |
        method ← self method.
        "Determine end of block from long jump preceding it"
        end ← (method at: startpc-2)\16-4*256 + (method at: startpc-1) + startpc - 1.
        scanner ← InstructionStream new method: method pc: startpc.
        scanner scanFor: [:byte | (byte between: 120 and: 124) or: [scanner pc > end]].
        ↑ scanner pc <= end!

home
        "Answer the context in which the receiver was defined."

        ↑home!

method
        "Answer the compiled method in which the receiver was defined."

        ↑home method!

receiver
        ↑home receiver! !

!BlockContext methodsFor: 'temporaries'!

tempAt: index
        ↑home at: index!

tempAt: index put: value
        ↑home at: index put: value! !

!BlockContext methodsFor: 'evaluating'!

value
        "Evaluate the block represented by the receiver.  Fail if the block expects any
        arguments or if the block is already being executed.  Optional.  No Lookup.
        See
        Object documentation whatIsAPrimitive."

        <primitive: 81>
        ↑self valueWithArguments: #()!

value: arg
        "Evaluate the block represented by the receiver.  Fail if the block expects other
        than one argument or if the block is already being executed.  Optional.  No

        Lookup.  See Object documentation whatIsAPrimitive."

        <primitive: 81>
        ↑self valueWithArguments: (Array with: arg)!

value: arg1 value: arg2
```

```
    "Evaluate the block represented by the receiver.  Fail if the block expects other
    than two arguments or if the block is already being executed.  Optional.  See

    Object documentation whatIsAPrimitive."

    <primitive: 81>
    ↑self valueWithArguments: (Array with: arg1 with: arg2)!

value: arg1 value: arg2 value: arg3
    "Evaluate the block represented by the receiver.  Fail if the block expects other
    than three arguments or if the block is already being executed.  Optional.
    See
    Object documentation whatIsAPrimitive."

    <primitive: 81>
    ↑self valueWithArguments:
    (Array
        with: arg1
        with: arg2
        with: arg3)!

valueWithArguments: anArray
    "Evaluate the block represented by the receiver.  The argument is an Array

    whose elements are the arguments for the block.  Fail if the length of the Array
    is not the same as the the number of arguments that the block was expecting.

    Fail if the block is already being executed.  Essential.  See Object
    documentation whatIsAPrimitive."

    <primitive: 82>
    nargs = anArray size
    ifTrue: [self valueError]
    ifFalse: [self error: 'The block needs more or fewer arguments defined']!
!

!BlockContext methodsFor: 'controlling'!

whileFalse
    "Evaluate the receiver once and then repeatedly as long as the value
    returned by the evaluation is false."

    ↑[self value] whileFalse: []!

whileFalse: aBlock
    "Evaluate the argument, aBlock, as long as the value
    of the receiver is false. Ordinarily compiled in-line.
    But could also be done in Smalltalk as follows"

    ↑self value
    ifFalse:
        [aBlock value.
        self whileFalse: aBlock]!

whileTrue
```

"Evaluate the receiver once and then repeatedly as long as the value returned by the evaluation is true."

↑[self value] whileTrue: []!

whileTrue: aBlock

"Evaluate the argument, aBlock, as long as the value of the receiver is true. Ordinarily compiled in-line. But could also be done in Smalltalk as follows"

```
↑self value
ifTrue:
        [aBlock value.
        self whileTrue: aBlock]! !
```

!BlockContext methodsFor: 'scheduling'!

fork

"Create and schedule a process running the code in the receiver."

self newProcess resume!

forkAt: priority

"Create and schedule a process running the code in the receiver. The priority of the process is the argument, priority."

```
| forkedProcess |
forkedProcess ← self newProcess.
forkedProcess priority: priority.
forkedProcess resume!
```

newProcess

"Answer a new process running the code in the receiver. The process is not scheduled."

```
↑Process
forContext:
        [self value.
        Processor terminateActive]
priority: Processor activePriority!
```

newProcessWith: anArray

"Answer a new process running the code in the receiver. The receiver's block arguments are bound to the contents of the argument, anArray. The process is not scheduled."

```
↑Process
forContext:
        [self valueWithArguments: anArray.
        Processor terminateActive]
priority: Processor activePriority! !
```

!BlockContext methodsFor: 'instruction decoding'!

blockReturnTop

```
        "Simulate the interpreter's action when a ReturnTopOfStack
        bytecode is encountered in the receiver."

        | save dest |
        save ← home. "Needed because return code will nil it"
        dest ← self return: self pop to: self sender.
        home ← save.
        sender ← nil.
        ↑dest!
```

pushArgs: args from: sendr

```
        "Simulates action of the value primitive."

        args size ~= nargs ifTrue: [↑self error: 'incorrect number of args'].
        stackp ← 0.
        args do: [:arg | self push: arg].
        sender ← sendr.
        pc ← startpc! !
```

!BlockContext methodsFor: 'printing'!

printOn: aStream

```
        home == nil ifTrue: [↑aStream nextPutAll: 'a BlockContext with home=nil'].
        aStream nextPutAll: '[] in '.
        super printOn: aStream! !
```

!BlockContext methodsFor: 'private'!

valueError

```
        self error: 'Incompatible number of args, or already active'! !
```

ANNEXE C: SPECIFICATION DE L'INTERFACE DU GESTIONNAIRE D'OBJETS.

Notations:

L'interface offerte par la mémoire des objets utilise des index pour désigner les différents champs de l'état d'un objet. Ceux sont des valeurs entières positives ou nulles. Le champ zéro désigne le premier champ de l'état d'un objet. L'état d'un objet peut donc être assimilé à un tableau de champs indexé à partir de la valeur zéro. Un champ peut contenir soit une valeur immédiate ou soit un pointeur (nom local d'un objet).

Un paramètre en italique est un pointeur vers un objet réel ou vers un objet représentant.

Un paramètre qui n'est pas en italique est nécessairement un pointeur vers un objet réel.

Comme il a été exposé précédemment, un objet représentant est constitué de deux champs: le nom du site distant qui possède l'objet réel et le nom local de cet objet sur ce site.

Internal reference counting.

procédure increaseReferencesTo (*objectPointer*)

Cette procédure incrémente d'une unité le compteur de références locales de l'objet désigné par le pointeur "objectPointer".

procédure decreaseReferencesTo (*objectPointer*)

Cette procédure décrémente d'une unité le compteur de références locales de l'objet désigné par le pointeur "objectPointer".

De la même façon, il a été nécessaire de définir les procédures qui gèrent le compteur de références externes d'un objet.

External reference counting.

procédure increaseExternalReferencesTo (*objectPointer*)

Cette procédure incrémente d'une unité le compteur de références externes de l'objet désigné par le pointeur "objectPointer".

procédure decreaseExternalReferencesTo (*objectPointer*)

Cette procédure décrémente d'une unité le compteur de références externes de l'objet désigné par le pointeur "objectPointer".

Class Pointer access.

fonction fetchClassOf (*objectPointer*) → *classPointer*

Cette fonction retourne le pointeur de la classe dont l'objet "objectPointer" est un exemplaire.

Length access.

fonction fetchWordLengthOf (*objectPointer*) → size

Cette fonction retourne le nombre de champs (valeurs ou pointeurs) composant l'état de l'objet de nom "objectPointer".

fonction fetchByteLengthOf (*objectPointer*) → size

Cette fonction retourne le nombre de champs-octets composant l'état de l'objet de nom "objectPointer".

Object pointer access.

fonction fetchPointer (*objectPointer* , fieldIndex) → *valuePointer*

Cette fonction retourne la valeur du champ-pointeur du champ numéro "fieldIndex" de l'objet de nom "objectPointer".

procédure storePointer (*objectPointer* , fieldIndex , *valuePointer*)

Cette procédure affecte la valeur-pointeur "valuePointer" au champ d'index "fieldIndex" de l'objet de nom "objectPointer".

Word access.

fonction fetchWord (*objectPointer* , fieldIndex) → valueWord

Cette fonction retourne la valeur numérique sur 32-bits du champ numéro "fieldIndex" de l'objet de nom "objectPointer".

procédure storeWord (*objectPointer* , fieldIndex , ValueWord)

Cette procédure affecte la valeur numérique sur 32-bits "valueWord" au champ numéro "fieldIndex" de l'objet de nom "objectPointer".

Byte access.

fonction fetchByte (*objectPointer* , byteIndex) → valueByte

Cette fonction retourne la valeur-octet sur 8-bits du champ numéro "byteIndex" de l'objet de nom "objectPointer".

procédure storeByte (*objectPointer* , byteIndex , valueByte)

Cette procédure affecte la valeur-octet sur 8-bits "valueByte" au champ numéro "fieldIndex" de l'objet de nom "objectPointer".

Integer access.

Les entiers sont directement codés avec le même format que les pointeurs (nom local d'un objet). Chaque entier est représenté sur 32 bits où le bit de poids fort est initialisé à 1 et où la valeur en complément à deux est codé sur les 31 autres bits. Il est a noter que les pointeurs suivent le même format, avec cependant le bit de poids fort initialisé à 0. Les entiers sont les exemplaires de la classe "SmallInteger". Les quatre fonctions suivantes permettent de manipuler ce type particulier d'objets.

fonction integerValueOf (objectPointer) → value

Cette fonction retourne la valeur entière de l'objet-entier de nom "objectPointer".

fonction integerObjectOf (value) → objectPointer

Cette fonction retourne le pointeur d'un objet de type entier dont la valeur entière est "value".

fonction isIntegerObject (objectPointer) → boolean

Cette fonction retourne la valeur booléenne "vrai" si l'objet de nom "objectPointer" est un exemplaire de la classe "SmallInteger", et "faux" sinon.

fonction isIntegerValue (value) → boolean

Cette fonction retourne "vrai" si la valeur entière "value" peut être représentée sous forme d'un exemplaire de la classe "SmallInteger", et "faux" sinon.

Object creation.

fonction instantiateClassWithPointers (*classPointer* , instanceSize) → objectPointer

Cette fonction crée une nouvelle exemplaire de la classe dont le pointeur est "classPointer" avec "instanceSize" champs qui pourront

contenir des pointeurs vers d'autres objets. Cette fonction retourne le pointeur du nouvel objet.

fonction instantiateClassWithWords (*classPointer* , instanceSize) → objectPointer

Cette fonction crée une nouvelle exemplaire de la classe dont le pointeur est "classPointer" avec "instanceSize" champs qui pourront contenir des valeurs numériques sur 32 bits. Cette fonction retourne le pointeur du nouvel objet.

fonction instantiateClassWithBytes (*classPointer* , instanceByteSize) → objectPointer

Cette fonction crée une nouvelle exemplaire de la classe dont le pointeur est "classPointer" avec "instanceSize" champs qui pourront contenir des valeurs numériques sur 8 bits (octets); Cette fonction retourne le pointeur du nouvel objet.

fonction createProxyObject (site , remoteObjectPointer) → localObjectPointer

Cette fonction crée un objet représentant pour un objet distant dont le pointeur est "remoteObjectPointer" sur le site "site". Cette fonction retourne le pointeur (nom local) du nouvel objet représentant.

Pointer swapping.

procédure swapPointersOf (*firstPointer* , *secondPointer*)

Cette procédure permute les états des objets pointés par les pointeurs "firstPointer" et "secondPointer". Cette procédure échange les valeurs des deux objets, et sert entre autre à implémenter les objets dynamiques du système Smalltalk.

Instance enumeration.

Les deux fonctions suivantes énumèrent les exemplaires d'une classe donnée. Si le pointeur "classPointer" est le pointeur d'un objet-classe distant, alors ces fonctions énumèrent les exemplaires locaux d'une classe distante.

fonction initialInstanceOf (*classPointer*) → *firstInstanceObjectPointer*

Cette fonction retourne le premier exemplaire de la classe dont le pointeur est "classPointer" suivant l'ordre défini (l'ordre de classement des exemplaires n'est pas significatif).

fonction instanceAfter (*objectPointer*) → *nextInstanceObjectPointer*

Cette fonction retourne le pointeur de l'objet qui est l'exemplaire suivant de l'objet dont le pointeur est "objectPointer" suivant l'ordre

défini. Cet objet appartient forcément à la même classe que l'objet de pointeur "objectPointer".

Object migration.

procédure migrateObject (objectPointer,toSite)

Cette procédure a pour effet de déplacer l'objet dont le pointeur est "objectPointer" sur le site "site". Si l'objet est un représentant, cette procédure effectue de même la migration de l'objet depuis son site de résidence vers le site "site". Ainsi, un objet peut être déplacé d'un site B vers un site C par l'exécution de cette primitive sur un site A possédant un représentant.

ANNEXE D: REALISATION D'UNE PRIMITIVE

D'ACCES AUX OBJETS.

Cette annexe est en fait une étude des différents cas de figures possibles lors du traitement de la fonction "fetchPointer" par le gestionnaire d'objets.

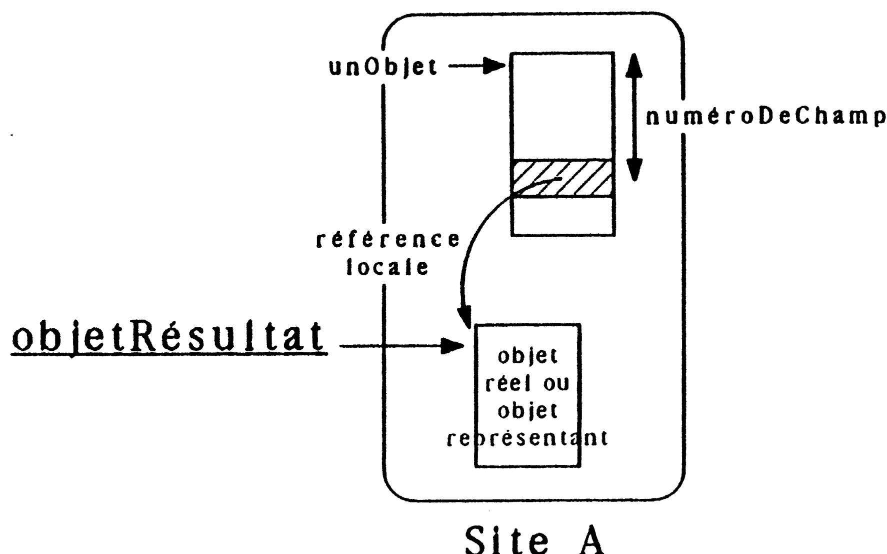
Cet exemple est surtout intéressant car il met en valeur l'utilisation des objets représentants, et il montre comment la spécification initiale de cette primitive, telle qu'elle est explicitée dans le système Smalltalk-80 centralisé, est respectée. La transparence de l'accès à un objet local ou distant est clairement exposée par l'étude des différents cas de figure. Dans chaque cas, le résultat de la primitive doit toujours être un pointeur vers un objet local au site de l'exécution de l'accès.

La fonction "fetchPointer" correspond à une lecture sur un objet. Elle retourne la valeur du "numéroDeChamp" ième champ de l'objet "UnObjet". Cette valeur doit toujours être un pointeur vers un objet local au site où est exécutée la fonction.

Syntaxe: **fonction** *fetchPointer(UnObjet,numéroDeChamp)* → *objetRésultat*

1. L'objet "unObjet" est un objet local au site (objet réel).

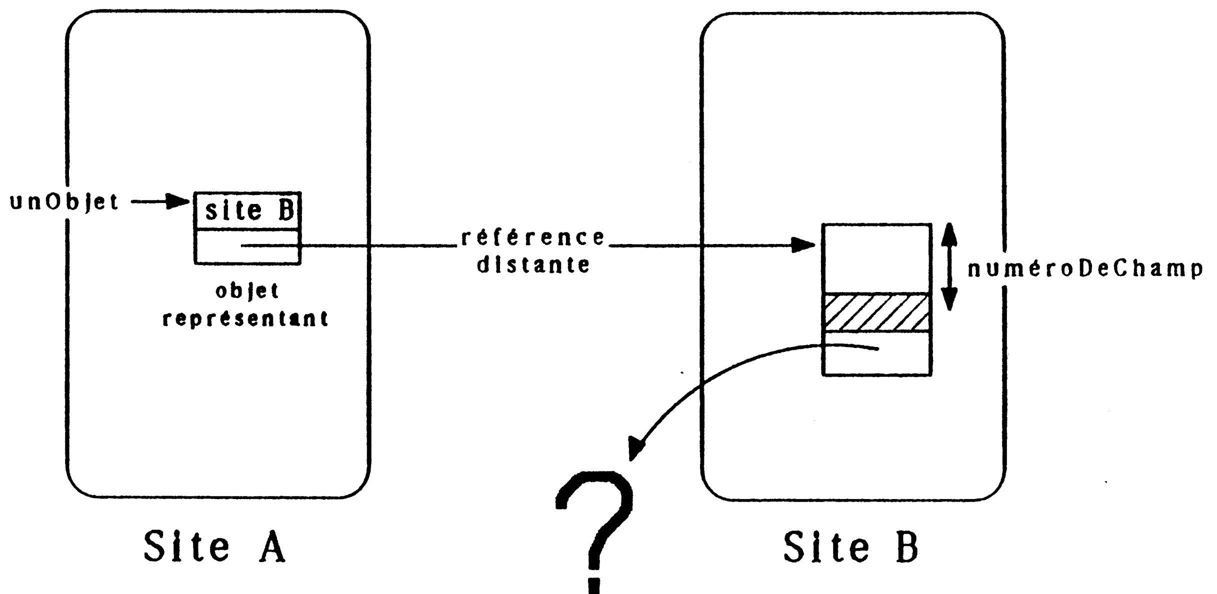
Ceci est le cas général du système Smalltalk-80 centralisé, où tout objet est un objet composé d'autres objets. Cette composition s'arrête avec des références à des objets de bases (entiers, caractères, etc...) qui sont composés de valeurs immédiates. Le résultat de l'appel de la fonction est donc la valeur du champ.



2. L'objet "unObjet" est un objet représentant.

Il se pose alors le problème de représenter correctement le résultat de la fonction suivant la règle de composition des objets de chaque univers Smalltalk. Il faut donc assurer que la fonction délivre toujours un résultat local, même pour un accès distant, ceci pour que le système qui exécute la fonction puisse se servir du résultat pour l'intégrer éventuellement au corps d'un autre objet.

A partir de cela, il se pose alors le problème de connaître la nature du champ "numéroDeChamp" de l'objet distant, pour déterminer et construire le résultat de la fonction.



En effet, trois cas peuvent se présenter. Ceux-ci proviennent de la localisation de l'objet pointé par le champ de l'objet réel. Pour cela, on suppose que l'objet représentant réside sur un site A, et que l'objet réel est localisé sur un site B.

1- L'objet réel référencé par l'objet réel est un objet local au site B.

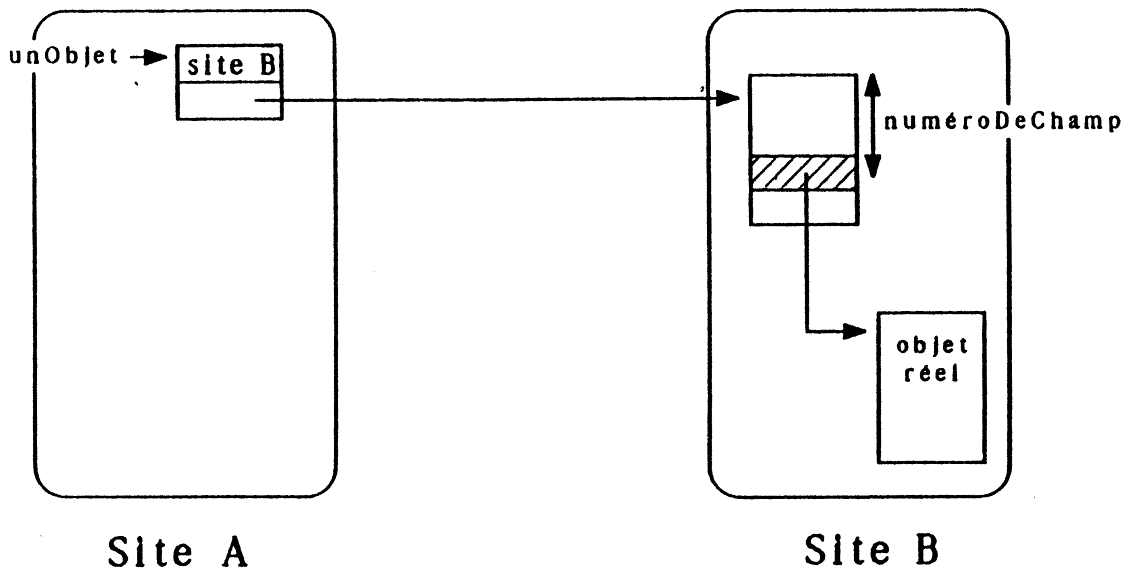
2- L'objet réel référencé est un objet local au site A.

3- L'objet réel référencé réside sur un site (C) autre que A et B.

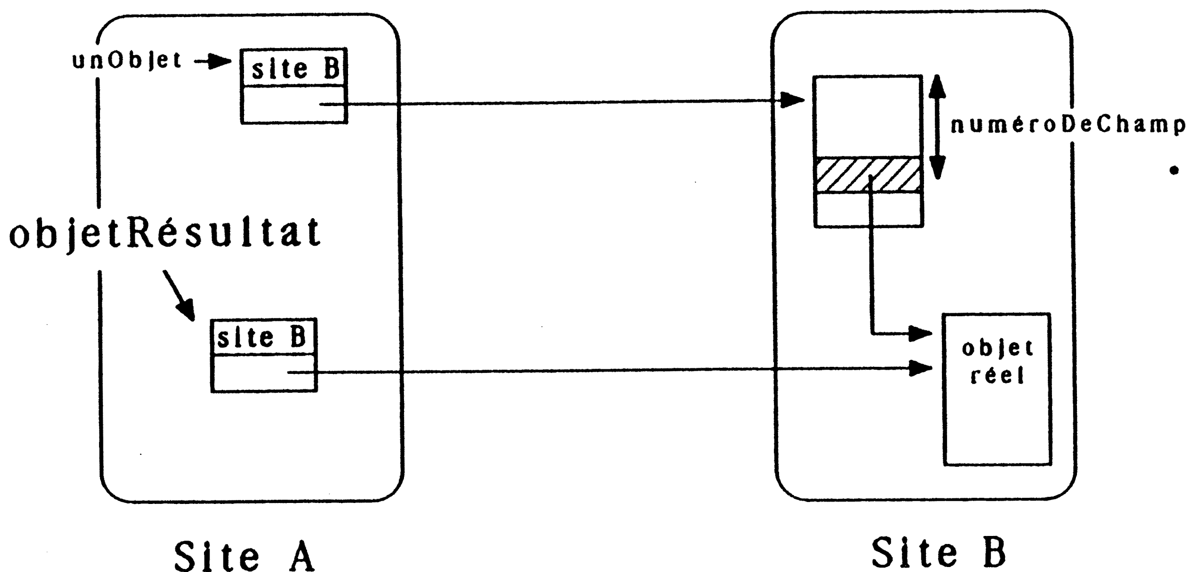
Ces trois cas sont étudiés et résolus ci-après.

2.1. L'objet réel résultat réside sur le site B.

Le champ "numéroDeChamp" de l'objet "unObjet" désigne un objet résident du site B. Il faut donc que localement au site A, la fonction d'accès "fetchPointer" délivre un nom local au site A. Ceci pour qu'ultérieurement ce moyen de désignation et d'accès local puisse être utilisé par d'autres appels de fonctions d'accès. Par conséquent, le nom contenu dans le champ de l'objet "unObjet" du site B ne peut être utilisé sous cette forme sur le site A.

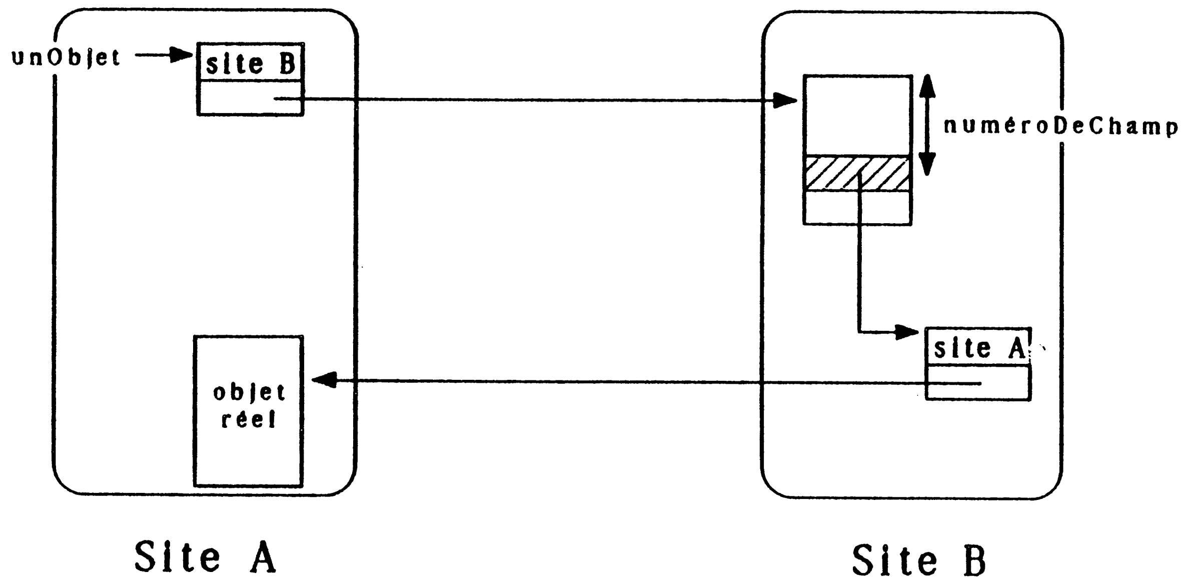


Il suffit de créer, sur le site A, un objet "représentant" qui permet désigner l'objet distant et résultat, par un nom local. La résolution de ce cas, à l'aide des objets "représentants", met bien en évidence la conservation de l'homogénéité de la désignation des objets sur chaque site.

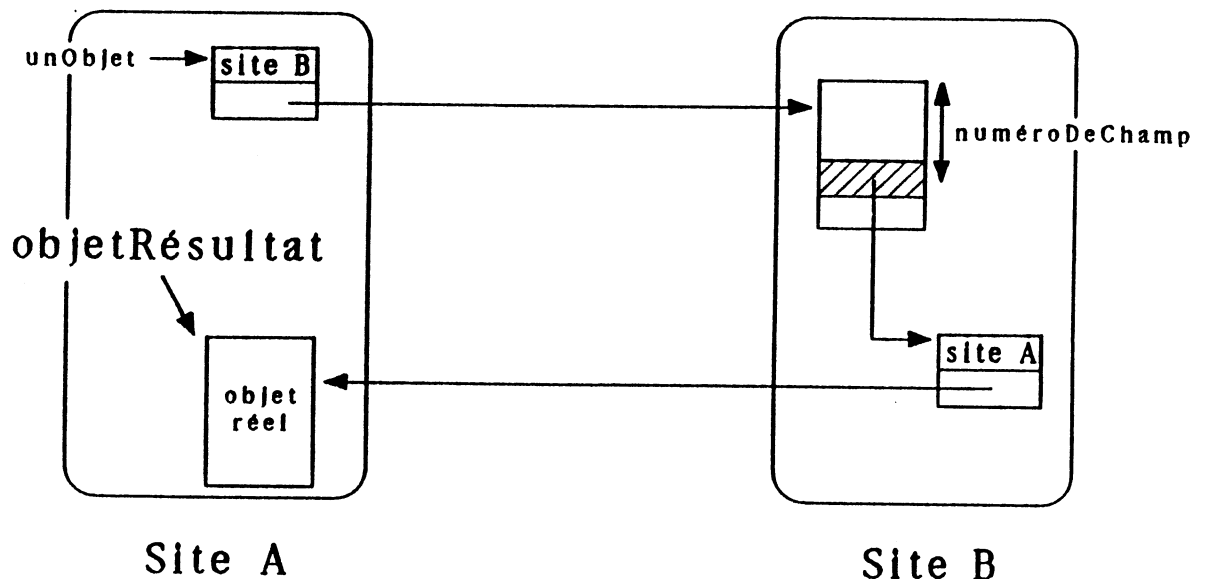


2.2. L'objet réel résultat réside sur le site A.

Le champ "numéroDeChamp" de l'objet "unObjet" désigne un objet "représentant" qui permet d'accéder un objet réel localisé sur le site A.

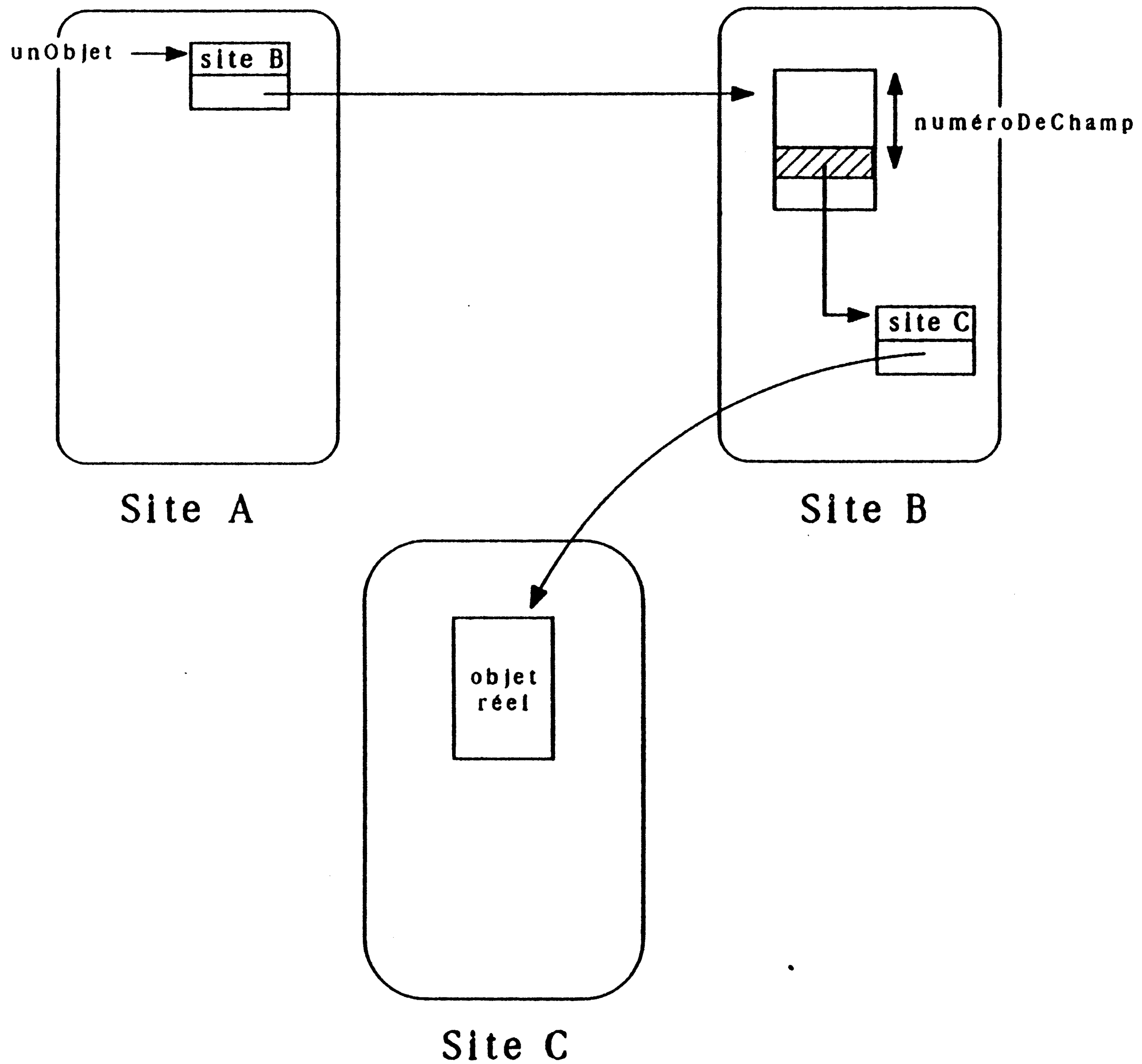


Le résolution de ce cas est simple. Le résultat délivré par la fonction d'accès est le nom local au site A de l'objet réel désigné par l'objet représentant du site B.



2.3. L'objet réel résultat réside sur un site C.

Le champ "numéroDeChamp" de l'objet "unObjet" désigne un objet "représentant" qui permet d'accéder un objet réel localisé sur un troisième site C.



De la même manière que pour le sous_cas précédent, il faut créer un objet "représentant" sur le site A. Ainsi on fournit un nom local qui permet de nommer localement au site A l'objet réel situé sur le site C.

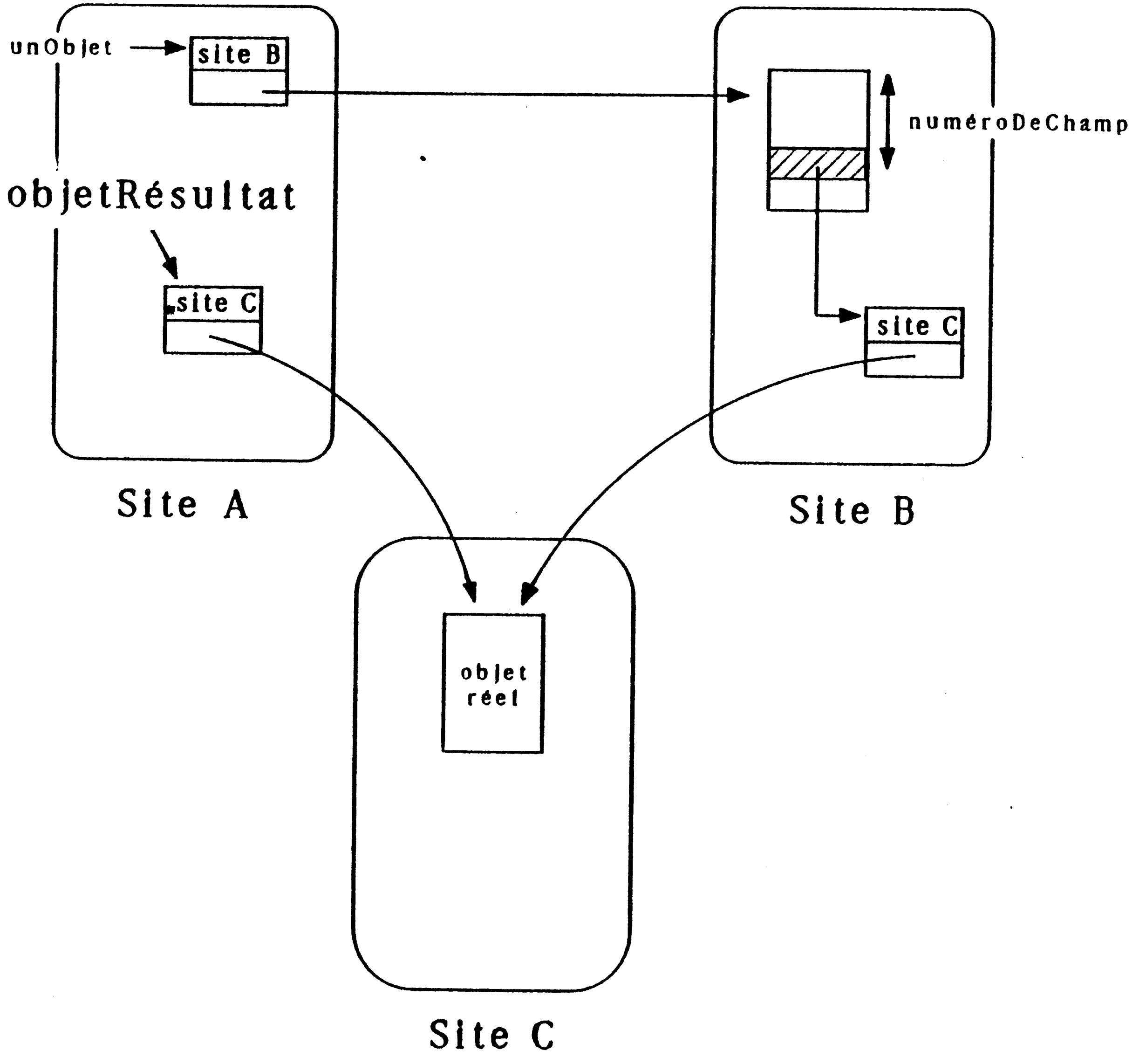

unObjet
site B
numéroDeChamp
objetRésultat
site C
site C
Site A
Site B
objet
réel
Site C

TABLE DES MATIERES

I. INTRODUCTION. 3

1. Motivations. 3

2. Définition des objectifs du travail. 5

3. Travail réalisé. 6

4. Plan de la thèse. 10

II. LE MODELE DES OBJETS ET SA MISE EN OEUVRE. 11

1. Le modèle général de la programmation par objets. 11

2. Le langage Simula67. 14

2.1. Introduction. 14
2.2. La structure de bloc. 14
2.3. Les classes. 15
2.4. Les coroutines. 17
2.5. La structure d'héritage. 17
2.6. Conclusion. 19

3. Le langage et le système Smalltalk-80. 20

3.1. Introduction. 20
3.2. Objets et classes. 21
3.2.1. Les classes et leurs exemplaires. 21
3.2.2. Sous-classe / super-classe. 23
3.2.3. Définition d'une super-classe abstraite. 24
3.2.4. Les métaclasses. 25
3.3. Le modèle d'exécution. 28
3.3.1. Messages et méthodes. 28
3.3.2. Les blocs. 31
3.4. Les processus et le contrôle de l'exécution. 33

4. Conclusion. 34

III. SYSTEMES REPARTIS A OBJETS. 37

1. Systèmes répartis. 37

1.1. Motivations. 37
1.2. Architecture d'ensemble. 38
1.3. Principes de conception. 38
1.4. Problèmes liés à la répartition. 39
1.4.1. Hétérogénéité. 39
1.4.2. Désignation des entités. 39
1.4.3. Allocation de ressources. 39
1.4.4. Protection. 40
1.4.5. Tolérance aux pannes. 40
1.5. Apport de la structuration en objets. 40

2. Quelques expériences. 41

2.1. Eden. 41
2.2. Emerald. 43
2.3. SOS. 45

3. Conclusion et critiques des diverses réalisations. 45

IV. PRINCIPE D'UNE MEMOIRE VIRTUELLE D'OBJETS POUR SMALLTALK-80. 47

1. Introduction. 47

2. Spécification d'une mémoire virtuelle d'objets. 48

2.1. Motivation. 48
2.2. La mémoire des objets et son interface avec l'interprète. 48
2.2.1. Implémentation standard de la mémoire des objets. 48
2.2.2. L'interface interprète / mémoire des objets. 50
2.2.2.1. L'interprète (rappels). 50
2.2.2.2. Les procédures et fonctions d'accès aux objets. 50
2.3. LOOM: Une première mémoire virtuelle pour Smalltalk. 51
Les concepts. 52
Conclusion. 54

3. Principe d'une réalisation centralisée de la mémoire virtuelle d'objets. 54

3.1. Gestion d'une mémoire à deux niveaux. 55
3.1.1. Présentation du problème. 55

3.1.2. Le nouveau cycle de l'interprète. 55
3.1.3. L'accès aux objets. 56
3.1.4. Les fonctions gestionnaire de la mémoire. 58
3.2. Organisation générale du système à mémoire segmentée. 60
3.2.1. Découpage en modules et processus. 60
3.2.2. Le protocole de résolution des défauts d'accès aux objets. 62
3.2.3. Le protocole GMC / GMS. 64
3.2.4. Le protocole de création d'objet. 65

4. Le module de gestion de la mémoire centrale. 66

4.1. Le translateur. 66
4.1.1. Fonction du translateur. 66
4.1.2. Synchronisation sur l'accès au translateur. 68
4.1.3. Composition d'une entrée. 68
4.1.4. Ajout d'informations dans le descripteur d'un objet. 69
4.1.5. Synchronisation sur l'entrée d'un objet. 71
4.2. Protocole d'accès à un objet. 71
4.2.1. Représentation d'un objet. 72
4.2.2. Modifications des procédures d'accès aux objets. 73
4.2.3. Les algorithmes des procédures "début-accès" et "fin-accès". 74
4.3. Le processus gestionnaire de la mémoire. 76
4.3.1. Les fonctions du processus gestionnaire de la mémoire. 76
4.3.2. Structures de données et mécanismes de base. 87
4.3.3. Stratégie de gestion mémoire: "un gestionnaire plus adapté". 94

5. Améliorations possibles. 94

5.1. Fixation d'un objet. 95
5.2. Court-circuit d'accès aux objets par les registres. 97
5.3. Proposition pour une solution plus efficace. 99

6. Conclusion. 100

V. CONCEPTION & REALISATION D'UN SYSTEME SMALLTALK-80 REPARTI. 103

1. Introduction. 103

2. Les problèmes d'un gestionnaire distribué d'objets. 106

2.1. La désignation des objets distribués. 107
2.2. Les accès aux objets. 107

2.3. La protection des objets. 108
2.4. Allocation de mémoire et récupération d'objets. 109

3. Les principes du gestionnaire distribué d'objets. 110

3.1. Partage d'objets. 111
3.2. Les références distantes et les objets "représentant". 111
3.2.1. L'accès à un objet sur le réseau. 115
3.2.2. Unicité des objets représentants sur les sites. 116
3.3. Migration d'un objet. 118
3.4. Gestion de l'espace des objets et récupération d'objets. 120
3.4.1. Récupération d'objets par compteur de références. 121
3.4.2. Récupération d'objets par marquage. 122
3.4.3. Récupération d'objets fondé sur le vieillissement. 127
3.5. Connexions et déconnexions dynamiques. 128
3.6. Protection des accès. 132

4. Composition de la machine virtuelle Smalltalk. 132

4.1. Le noyau de synchronisation. 133
4.2. Organisation du gestionnaire des objets. 134

VI. IMPLEMENTATION DU SYSTEME SMALLTALK-80 REPARTI. 137

1. La réalisation développée. 137

2. Eléments pour une réalisation future. 141

VII. CONCLUSION. 145

BIBLIOGRAPHIE. 149

ANNEXE A: LES OBJETS COMPOSANT L'ENVIRONNEMENT D'EXECUTION DU SYSTEME SMALLTALK-80.

1. Les méthodes.

2. Les classes.

3. Les contextes de méthodes et de blocs.

ANNEXE B: DEFINITION DE LA CLASSE "BLOCKCONTEXT".

ANNEXE C: SPECIFICATION DE L'INTERFACE DU GESTIONNAIRE D'OBJETS.

ANNEXE D: REALISATION D'UNE PRIMITIVE D'ACCES AUX OBJETS.

1. L'objet "unObjet" est un objet local au site (objet réel).

2. L'objet "unObjet" est un objet représentant.

   2.1. L'objet réel résultat réside sur le site B.
   2.2. L'objet réel résultat réside sur le site A.
   2.3. L'objet réel résultat réside sur un site C.

AUTORISATION DE SOUTENANCE

DOCTORAT 3ème CYCLE, DOCTORAT-INGENIEUR, DOCTORAT USTMG

Vu les dispositions de l'Arrêté du **16 avril 1974,**

Vu les dispositions de l'Arrêté du **5 juillet 1984,**

Vu les rapports de Mr. Jean-Pierre Banâtre ....

Mr. Roland Balter ....

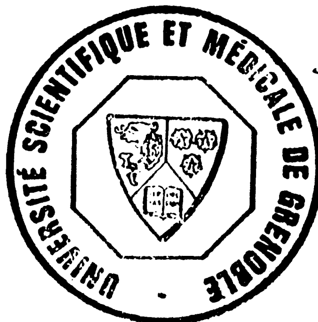
Mr. Dominique Decouchant .... est autorisé

à présenter une thèse en vue de l'obtention du Doctorat de l'USTMG

(spécialité Informatique) ....

Grenoble, le 17 [illegible] 1987 ....

Le Président de l'Université Scientifique Technologique et Médicale



J.J PAYAN

L'objectif de ce travail est d'étudier les problèmes posés par la gestion d'objets dans un système informatique réparti.

Le système Smalltalk-80 a servi de champ d'application; ce système a été conçu initialement sous une forme centralisée, dans laquelle l'ensemble des objets manipulés doit résider en mémoire principale.

Le travail réalisé se décompose en deux phases:
- l'étude et la réalisation d'une mémoire virtuelle d'objets pour un système Smalltalk centralisé,
- l'extension de la mémoire centralisée des objets pour un environnement réparti.

La conception de la mémoire centralisée des objets est analogue à celle d'une mémoire segmentée, dans laquelle les objets jouent le rôle des segments. Les problèmes inhérents à la réalisation d'une mémoire virtuelle ont été étudiés: allocation de la mémoire, résolution des défauts d'accès et récupération d'objets ("ramasse-miettes"). Cependant, cette conception diffère de celle des systèmes classiques par les caractéristiques des segments manipulés (petits segments en grand nombre).

La conception de la mémoire virtuelle des objets a ensuite été étendue au milieu réparti pour offrir la transparence des accès aux objets locaux ou distants, permettre le partage d'objets entre plusieurs sites et autoriser les migrations d'objets. Pour réaliser ces objectifs, une nouvelle entité a été introduite: l'objet "représentant". La répartition rend plus difficiles les problèmes de la désignation des objets, de la protection, de la réalisation des accès aux objets distants et du maintient de la cohérence des objets et des sites. Les nouvelles données de ces problèmes sont précisées et quelques solutions sont présentées. Les problèmes du traitement des erreurs et des pannes n'ont pas été abordés.

Une réalisation du gestionnaire réparti d'objets a été faite, en simulation, au dessus d'Unix 4.2BSD. Elle utilise un noyau de processus (Mux) et des communications par "sockets".

Mots clés:
mémoire virtuelle, gestion distribuée d'objets, représentants, accès transparents, migrations, récupération d'objets.