Des Objets aux Acteurs
1982-1989: 7 Ans de Réflexion
Habilitation à Diriger des Recherches
Mémoire de Synthèse

Jean-Pierre BRIOT

Laboratoire d'Informatique Théorique et de Programmation
Université Pierre et Marie Curie

10 Juillet 1989

LITP Research Report, No 89–68, Paris, France, September 1989.

1 Introduction

Ce mémoire se propose de rappeler mon activité de recherche entre les années 1982 et 1989. L'ensemble de ces travaux porte sur l'analyse des mécanismes fondamentaux des langages à objets.

Les *langages à objets* sont une famille de langages de programmation restée très active depuis la fin des années 70. Le style de programmation associé, appelé *programmation par objets*, se caractérise par la description du problème à modéliser sous la forme d'un grand nombre d'entités autonomes (appelées *objets*) et de leurs protocoles d'interaction (appelés *messages*). La programmation par objets permet au concepteur d'exprimer les concepts mis en jeu directement dans le langage. Ce n'est pas un hasard si un domaine d'application privilégié est la simulation, où les objets physiques à modéliser seront spécifiés et représentés en tant que tels au niveau du langage [Lieberman 84]. La simulation est d'ailleurs historiquement le domaine d'application qui a conduit au premier langage de programmation de ce type, Simula-67 [Birtwistle et al. 73].

Le deuxième évènement fondamental fut l'invention du système Smalltalk par Alan Kay [Kay 69]. Le concept d'objets communiquant par envois de messages a été mis au service d'une vision révolutionnaire de la programmation, c'est à dire principalement l'interaction entre l'utilisateur et la machine, plutôt que la seule conception des programmes (algorithmique), jusque-là considérée comme l'unique enjeu. La philosophie du système Smalltalk ne peut être raisonnablement réduite à la seule composante langage de programmation.

Cette nouvelle vision de la programmation est maintenant connue du grand public, à défaut d'être toujours prise en compte par les informaticiens. L'exemple le plus frappant en est le Macintosh d'Apple, retombée directe de la machine Smalltalk-72. La programmation par objets permet à la machine de se rapprocher de l'homme, d'une part en exprimant les problèmes de manière antropomorphique (et pour plus être plus précis comme une société d'individus), et d'autre part en la dotant d'une interface homme/machine visuelle et sophistiquée. Les domaines d'application sont multiples et sans limite théorique. Parmi ses avantages reconnus, les principaux sont d'après nous: l'uniformité, la modularité, et le haut niveau d'expression.

Du fait de leur succès considérable, le nombre de langages ou de systèmes se réclamant de la programmation par objets ne cesse d'augmenter. Nombre langages de programmation plus classiques subissent maintenant des "liftings" à l'aide d'*extensions à objets* plus ou moins bien fondées. De plus le terme *objet*, de par sa généralité, est le point de ralliement de beaucoup de concepts et de domaines d'application spécifiques, tels: les types abstraits de données, la représentation de connaissance structurée, les modules, les fonctions génériques, les agents... Les sentiments sont généralement tranchés: soit on revendique les objets, parfois pour dire que tel Monsieur Jourdain, on les utilisait déjà avant de connaître cette terminologie, soit on les rejette comme un phénomène de mode superficiel. D'un groupe de pensée à un autre, la terminologie varie parfois énormément. Plusieurs termes cohabitent, du fait d'une traduction non standardisée du terme américain original *object-oriented*. Les langages orientés objet(s), les langages objet(s). Nous utiliserons le terme *langages à objets*.

C'est dans ce contexte d'explosion et de dispersion des réalisations et des concepts, que j'ai commencé mes premiers pas d'explorateur informatique. La première étape de mon travail a consisté en une analyse expérimentale des différents courants de la programmation par objets, puis en leur synthèse, augmentée de nouveaux résultats. La seconde étape, aujourd'hui en plein développement, vise à maîtriser un des aspects les plus fascinants du monde des objets: leur aptitude à évoluer simultanément. Cette sous-famille spécialisée dans l'étude du parallélisme prend le nom de *langages d'acteurs*. Mon étude a de nouveau suivi la même démarche d'analyse expérimentale en vue d'une synthèse.

2 Plan

La description de mes travaux se décompose en quatre parties.

Le paragraphe 3 introduit la problématique expérimentale à travers le développement du projet d'informatique musicale *Formes* auquel j'ai participé.

Le paragraphe 4 résume ma *thèse de 3ième cycle*, analyse et synthèse des relations entre objets.

Le paragraphe 5 présente les résultats nouveaux, *programmation par métaclasses*, que j'ai apportés au domaine.

Le paragraphe 6 introduit les langages d'acteurs à travers le développement du projet *ABCL/1* auquel j'ai participé.

Enfin le paragraphe 7 en présente une synthèse et une plateforme de modélisation, nommée *Actalk*.

3 Introduction Musicale aux Objets

Le point de départ de mes travaux se trouve dans mon stage de DEA Langages Algorithmes et Programmation de l'Université Pierre et Marie Curie (UPMC) en 1982. Ce stage se déroula au Laboratoire de Reconnaissance des Formes de l'UPMC en binôme avec Bernard Serpette. Le sujet du stage était le portage et le développement d'un langage à objets dédié à la composition et synthèse musicales par ordinateur.

Ce langage, nommé Formes, était en cours de développement à l'Institut de Recherche et de Coordination Acoustique/Musique (IRCAM), sous la direction de Xavier Rodet. Le projet avait pour but de fournir à des compositeurs un système unifié d'aide à la composition musicale et de synthèse en temps réel. Formes était le langage de spécification et d'exécution d'objets musicaux.

3.1 Cadre Général

Formes est un système qui élargit la programmation par objets traditionnelle vers la composante temporelle [Cointe et Rodet 84]. Il ne s'agit pas seulement de décrire statiquement la structure des objets musicaux, mais également de spécifier leur ordonnancement dans le temps. De ce fait on parlera plutôt de *processus* musicaux, dont l'activité temporelle est contrôlée par des *moniteurs*. Les processus musicaux élémentaires recouvrent des notions aussi variées qu'une note de musique, un phonème de la voix chantée, un effet tel un vibrato, ou encore une phase de transition entre deux processus. Les différents types de moniteurs décrivent les différentes hiérarchies de processus, par exemple: atomiques, séquences, multi-voix... Une horloge synchronise les processus actifs à un instant donné. Leurs contributions à la construction musicale sont regroupées en une structure, nommée l'*arbre de calcul*, qui sera évaluée à chaque tic d'horloge, pour produire les données d'entrée du synthétiseur [Rodet et al. 84].

3.2 Développement

Au Printemps 1982, Pierre Cointe venait de concevoir et d'implémenter le premier prototype de Formes en VLisp 10.3, sur le PDP 10 de l'IRCAM. Notre première tâche (à Bernard et moi-même) pendant le stage fut le portage de Formes sur un autre dialecte de Lisp en cours de développement, Le_Lisp.

Au cours du portage, nous avions commencé à expérimenter différentes stratégies d'implémentation et à cerner certains problèmes tels celui de l'ubiquité d'un processus musical (voir le paragraphe 3.4). La synthèse de ce travail fut présentée sous forme d'un rapport de stage [Briot et Serpette 82]. Je participai ensuite au développement du système [Rodet et al. 83] [Rodet et al. 84] et à son utilisation par des compositeurs. J'ai en particulier conçu des modules de transition automatique entre processus musicaux [Briot 84].

3.3 Environnement

En plus de ses capacités de spécification et d'exécution de processus musicaux, Formes propose un environnement interactif d'aide à la composition et de documentation. Il faut en effet souligner que la plupart des utilisateurs de cet outil étaient des compositeurs non informaticiens. Il était donc important de développer un environnement adapté incluant la gestion de l'ensemble des ressources fournies par le système ainsi que celles apportées par l'ensemble des compositeurs sous la forme d'une *bibliothèque*. Je fus responsable de la mise en oeuvre d'un système de gestion et d'auto-documentation de cette bibliothèque au Printemps 1985. Cette bibliothèque est constituée de modèles musicaux que les compositeurs peuvent réutiliser tels quels ou encore, le plus souvent, en dériver de nouveaux. Elle contient des processus musicaux prédéfinis, par exemple un certain nombre de phonèmes de la voix chantée, des moniteurs spécifiques, et des fonctions Lisp de contrôle des paramètres sonores. Le compositeur peut puiser dans cette bibliothèque pour construire étape par étape sa pièce musicale. Un fichier unique contient le programme décrivant l'objet musical Formes ou la fonction Lisp, la documentation associée utilisée par un logiciel de visualisation, et les procédures de test. Ce choix minimise les inconsistences entre ces trois données habituellement réparties sur des fichiers distincts.

3.4 Un Modèle Dual de Création

Les besoins propres au domaine de la composition musicale entrainèrent de manière naturelle une intéressante cohabitation de deux différents modes de création d'objets: la définition à partir d'une *structure abstraite*, et une création par *imitation*.

Chaque processus musical est en effet caractérisé par son *moniteur*, qui représente son type de structure hiérarchique et temporelle. Ainsi par exemple un processus dont le moniteur est la séquence doit spécifier l'ensemble ordonné de ses fils, mais ceci n'a aucun sens si le moniteur est la feuille (atomique). Un processus apparaît donc comme un exemplaire, ou encore *instance*, d'une *classe* de structure et de contrôle.

Parallèlement à la définition d'un processus comme instance d'une classe, un processus peut également être créé à partir d'un autre processus. Les compositeurs travaillent souvent par imitation, c'est à dire en définissant de nouveaux matériaux sonores à partir d'objets sonores déjà existants. Formes permet donc au compositeur de définir un nouveau processus musical comme copie d'un autre, copie qui peut être modulée par le changement de certaines de ses caractéristiques. Ce mode de création s'appelle *copie différenciée* [Cointe et al. 87], puisque seules les différences avec l'original sont spécifiées. Un mode de création par copie coexiste donc avec un mode de création à partir d'une abstraction.

Signalons que la recopie d'un processus peut également être effectuée automatiquement lors de l'exécution par le système lui-même, pour résoudre des problèmes d'ubiquité. En effet un même processus musical peut soutenir deux activités simultanées mais dans des contextes distincts (par exemple lors de l'exécution de plusieurs voix/instruments contrôlées par un moniteur multi-voix). Il doit être dupliqué car les données de gestion de son activité temporelle (en particulier sa durée d'activation) sont propres à un contexte d'activation particulier. Bernard Serpette et moi-même avons décrit ce problème puis présenté une première solution dans [Briot et Serpette 82].

Cette duplicité des modes de création et de relation entre un objet et son modèle m'amena dès le début du projet à analyser les modèles de création des objets et les mécanismes associés. Ce travail de fond, simultanément au développement de Formes, déboucha sur ma thèse de 3ème cycle [Briot 84] sous la direction efficace de Jean-François Perrot.

4 Sémantique des Relations entre Objets

Au cours de ma thèse j'ai entrepris un travail d'analyse puis de synthèse des concepts fondamentaux des langages à objets, et classifié les grandes familles en présence. Je me suis intéressé plus particulièrement aux relations entre les objets, et plus spécialement aux *liens entre un objet et l'objet à partir duquel il est généralement créé* [Briot 83].

Ma thèse, intitulée "Instanciation et Héritage dans Les Langages Objets" [Briot 84], présente des réalisations concrètes d'interprètes permettant d'exhiber les différents modèles et mécanismes en jeu. La méthodologie employée est celle des *micro-interprètes* [Greussay 76] qui permettent de matérialiser l'essentiel des concepts d'un langage de programmation. Je vais maintenant résumer les apports de ma thèse à la compréhension des langages à objets.

4.1 Création

Un objet est rarement créé ex-nihilo, mais plutôt à partir d'un ou plusieurs objets existants. Le mécanisme fondamental de définition est la *décomposition*, c'est-à-dire la définition d'un objet à partir de ses composants. En plus de cela apparaissent trois autres mécanismes de définition:

- la création d'un exemplaire à partir d'un *modèle abstrait*, appelé *classe*,
- la *copie* parfaite ou *différenciée* d'un autre objet,
- et enfin la *spécialisation*, également appelée *dérivation*, à partir d'un objet ou une classe.

On peut regrouper les deux premiers mécanismes en la relation de *création* entre un objet et son modèle créateur, qu'il soit une classe abstraite ou un objet. Le troisième mécanisme correspond à la relation de *dérivation* d'un objet (ou classe) à partir d'un(e) autre dont il étend les fonctionnalités. J'ai montré dans ma thèse en quoi ces deux relations, création et dérivation, sont fondamentales et leurs variations de sémantique correspondent aux différentes familles de langages à objets.

4.2 Les Deux Modèles

Mon étude a fait apparaître deux familles/modèles principaux qui s'opposent et sont rarement mélangés:

- le "modèle abstrait" basé sur l'*instanciation* des classes, et dont la dérivation des classes se fait par *héritage*,
- le "modèle concret" basé sur la *copie* des objets ou/et la dérivation des objets par *délégation*. Dans ce dernier modèle, tout objet est un *prototype* potentiel dont on pourra dériver d'autres objets.

Le premier modèle représente la quasi-majorité des langages ou systèmes à objets, alors que le second est réservé à des systèmes de représentation des connaissances (de type *frame*) et à des réalisations prototypes notamment dans le domaine des *langages d'acteurs*, c'est-à-dire à objets actifs et concurrents.

4.3 Modèle Abstrait: Concept de Classe

Dans ce premier modèle, l'abstraction prime. On définit un modèle abstrait d'objets, la *classe*, avant d'en créer des exemplaires concrets, ses *instances* dont la classe décrit la structure et les comportements. La structure de données est matérialisée par un ensemble de variables, paramètres de la structure, ainsi nommées *variables d'instance*. Les comportements sont caractérisés par un ensemble de *méthodes* communes à toutes les instances d'une classe et ainsi détenues par elle.

Une classe est donc l'analogue d'un moule de fonderie, dont pourront sortir ses instances, des pièces de fonderie. Toutes les pièces auront la même forme mais pourront différer par leurs caractéristiques, matière par exemple.

4.4 Classe = Objet ?

Un des avantages des langages à objets est l'uniformité du mode de représentation. L'existence des classes dans le modèle abstrait amène donc la question suivante: "Une classe est-elle un objet?".

Certains langages, tels C++ [Stroustrup 86], répondent par la négative. Une classe n'est qu'une description mais pas une entité du langage. Dans certaines extensions à objets de langages, tels les Flavors de la Machine Lisp [Moon 86], les classes sont des entités implémentées en Lisp, mais n'ayant pas de statut d'objet du langage.

Si l'on répond par l'affirmative, la conséquence immédiate est la régression infinie. La classe étant un objet, doit être instance d'une classe, et ainsi de suite... Nous verrons au paragraphe 5 comment résoudre ce problème [Briot et Cointe 86b].

4.5 Objet = Classe ?

Si l'on inverse la question, la nouvelle interrogation devient: "Un objet est-il une classe?". Le modèle abstrait impose une réponse négative. En effet il y a des objets qui ne sont pas des classes. Nous les appelons *instances terminales*, puisqu'ils sont des objets non instanciables.

Au contraire, le modèle concret semble répondre par l'affirmative, puisque tout objet devient un générateur potentiel. Mais alors les classes disparaissent, et la sémantique de la création change.

4.6 Modèle Concret: Concept de Prototype

A l'opposé du modèle abstrait, le modèle concret n'introduit pas d'abstraction. En effet on peut définir directement et complètement un objet, sans avoir besoin de définir préalablement sa classe. Pour autant il faut bien introduire un moyen de définir un objet à partir d'un autre. C'est le concept de *prototype*, c'est à dire que tout objet est un générateur potentiel dont on peut en dériver de nouveaux.

Il y a deux alternatives de dérivation. La première est la *copie*. Un nouvel objet peut être créé à partir de n'importe quel autre, en spécifiant éventuellement ses différences avec l'original, comme on l'a vu en Formes [Cointe et al. 87].

Une autre alternative minimise la duplication de connaissances en proposant la *délégation* comme mécanisme de partage des connaissances [Lieberman 87]. Le principe en est simple et intuitif: un objet peut être défini comme une extension d'un objet déjà existant, et lui délèguera (renverra) les messages qu'il ne peut reconnaître. Du fait de sa simplicité ce mécanisme impose une unification des variables avec les méthodes, pour que toute requête, y compris l'accès aux données, se passe par envoi de message pour pouvoir être délégué. Ainsi l'héritage des variables est nécessairement dynamique par opposition au modèle d'héritage des classes. Le modèle perd hélas un peu de sa simplicité intuitive quand il s'agit de l'utiliser pour programmer des transactions concurrentes entre acteurs [Briot et Yonezawa 87].

Les deux alternatives, copie et délégation, peuvent être combinées, comme par exemple dans le langage Act1 [Lieberman 81], mais la délégation, à la différence de la copie [Borning 86], est le concept fondamental permettant de partager des connaissances.

4.7 Prototype ou Acteur: un Point de Terminologie

Le concept de délégation est historiquement apparu dans le langage d'acteurs Act1 [Lieberman 81]. Il s'en est suivi une certaine confusion terminologique entre *prototype* et *acteur* (confusion dont j'ai moi-même été à l'époque en partie responsable dans ma thèse). Il est utile de bien séparer ces deux termes. Le terme de *prototype* définit un générateur potentiel et implicite, par opposition aux classes abstraites et explicites. Le terme d'*acteur* définit un objet actif et autonome communiquant de manière asynchrone (voir le paragraphe 6.1). Ces deux notions sont donc orthonales. Ainsi certains systèmes acteurs, à la différence d'Act1, utilisent les classes comme outil de description (voir le paragraphe 4.10).

4.8 Des Prototypes aux Frames

A titre d'exemple, j'ai montré dans ma thèse comment les langages de frames se réduisent à des langages basés sur le concept de prototype. J'ai choisi le langage de représentation des connaissances Mering [Ferber 83] pour langage cible. Dans Mering, les objets sont définis par un ensemble d'attributs selon le principe de décomposition. Les attributs sont eux-mêmes décomposés en un ensemble d'attributs d'attributs, appelés *facettes*. Certaines de ces facettes sont des données passives, par exemple la valeur de l'attribut et son domaine de valeurs. D'autres sont appelés *attachements procéduraux*, ou encore *réflexes*, et expriment les actions réflexes à la suite d'accès à un attribut. Des mécanismes complexes gèrent la recherche de l'information de l'attribut d'un objet, ce qui entraîne les déclenchements éventuels de réflexes. J'ai montré que de tels mécanismes se réduisent à deux concepts: la *définition récursive*, c'est-à-dire le fait qu'un attribut soit lui-même un objet, et la *délégation*. La combinaison des deux concepts conduit à une délégation récursive pour les attributs, c'est-à-dire que les attributs d'un objet A qui délègue à un objet B vont déléguer aux attributs équivalents de l'objet B. Le micro-interprète Micro-Mering défini dans le micro-interprète Micro-Act1 matérialise cette démonstration. Il est d'ailleurs intéressant de signaler que Jacques Ferber a conçu, simultanément et indépendamment de ces conclusions, une deuxième version, nommée Mering-II [Ferber 84], basée sur ces principes ainsi doublement validés.

4.9 Classe ou Prototype

Deux concepts (et leurs modèles associés) s'opposent donc [Briot 84] [Lieberman 87]: le concept de *classe* et le concept de *prototype*, avec leurs mécanismes de partage de connaissance associés: l'*héritage* de classes et la *délégation*. Ma thèse en présente une analyse comparée que l'on peut résumer ainsi:

philosophique

dans le concept de classe, il y a primauté et préséance de l'abstrait sur le concret. A ce titre le modèle de classe est Platonicien.

A l'opposé avec le concept de prototype c'est l'objet concret qui prime, puisqu'il est toujours susceptible de donner naissance à d'autres objets. L'existence prime donc sur l'essence, et à ce titre le concept de prototype peut être considéré comme existentialiste.

méthodologique

le concept de classe impose le général sur le particulier. Ce modèle est bien adapté aux classes avec beaucoup d'instances, mais beaucoup moins si un unique objet doit être modélisé, auquel cas il nécessite la définition préalable d'une classe spécifique.

Dans le modèle des prototypes, au contraire c'est l'individu qui prime. Les relations entre objets sont plus flexibles et dynamiques. Le modèle est également plus uniforme puisqu'il n'y a plus de distinction entre objets générateurs (classes) et objets concrets. Mais c'est au détriment des capacités de structuration dues à l'absence de l'abstraction et plus encore de l'héritage qui permet de classifier explicitement les modèles d'objets [Briot et Yonezawa 87].

implémentatoire

Comme nous l'avons déjà souligné, le modèle de classe semble le meilleur compromis entre partage de code, flexibilité et efficacité. Cependant le partage de code et de données étant implicite et important, l'adéquation de ce modèle aux multiprocesseurs répartis (sans partage de mémoire) est moins naturelle.

Le modèle des prototypes est plus flexible mais son efficacité est plus difficile à optimiser. Ce n'est donc pas un hasard s'il est utilisé majoritairement pour des systèmes expérimentaux, et plus particulièrement de représentations de connaissances, où la flexibilité prime sur l'efficacité [Lieberman 87].

4.10 Classe et Prototype

Les deux modèles apparaissent relativement antinomiques mais de capacités de description équivalentes. Suivant le type de problème à modéliser l'un ou l'autre des modèles peut apparaître plus adapté. La cohabitation des deux modèles semble superflue et peu souhaitable car elle amoindrit les principes de simplicité et d'uniformité des langages à objets. Cependant, comme nous l'avons vu précédemment, le langage Formes propose pour des raisons spécifiques une cohabitation partielle, deux modes de création distincts étant proposés.

Pour mieux la cerner, j'en ai présenté dans ma thèse une implémentation minimale, nommée Micro-Formes. Un processus Micro-Formes apparaît comme une fusion d'instance (d'un moniteur) et de prototype. Le moniteur décrit la structure ainsi que les procédures de contrôle du processus qui est en plus doté d'un environnement personnel et extensible. Cela traduit la distinction entre les variables système nécessaires à la gestion du calcul, et l'environnement propre au processus, constitué de règles et de paramètres musicaux. Enfin les deux modes de création, instanciation d'un moniteur et copie différenciée cohabitent harmonieusement [Briot 84].

Un projet récent de système réparti à objets [Yokote et al. 89] propose de combiner héritage des classes et délégation entre objets, mais à des niveaux distincts. L'héritage est utilisé pour la spécification statique tandis que la délégation permet la répartition dynamique des comportements à l'exécution. Le système Actalk [Briot 89b], présenté au paragraphe 7, permet également une telle cohabitation.

5 Classes de Première Classe

Nous allons maintenant revenir au problème évoqué précédemment du statut des classes en tant qu'objet.

5.1 Métaclasses

Comme tout objet, une classe est instance d'une classe. Ces classes de classes sont appelées *métaclasses* puisqu'elles décrivent la structure et le comportement des classes [Briot 85b]. Ce problème rebondit alors sur le statut des métaclasses: si ce sont aussi des objets, elles doivent être instances de *métamétaclasses*, et ainsi de suite...

Les premiers langages à objets proposaient des solutions plus ou moins figées. Smalltalk-76 [Ingalls 78] proposait une seule métaclasse primitive au système, nommée `Class`, avec impossibilité d'en définir d'autres. Smalltalk-80 [Goldberg et Robson 83] propose des métaclasses implicites au système. En plus des métaclasses primitives formant le noyau de l'architecture du système, une métaclasse est implicitement associée à chaque classe définie par l'utilisateur. Mais l'utilisateur ne peut définir explicitement les métaclasses, et l'architecture implicite du système conduit parfois à des constructions non naturelles [Briot et Cointe 89]. Le système Loops [Bobrow et Stefik 83] fut un des premiers à considérer explicitement la notion de métaclasses et leurs possibilités.

5.2 Métaclasses Explicites

A la suite du travail de réflexion sur le statut des classes dans les langages à objets [Briot 83], j'ai proposé un modèle d'architecture permettant des métaclasses explicites, tout en gardant une uniformité du modèle. J'en ai ensuite réalisé une première implémentation prototype, nommée

Meta-Smalltalk [Briot et Saint-James 84], dans le cadre du projet Aglae dirigé par Emmanuel Saint-James. Deux métaclasses primitives constituent le noyau de l'architecture: `Metaclass`, générateur des métaclasses, et `Class`, générateur des classes. Chacune des deux classes détient une méthode de création, nommée `new`, la première créant des classes, et la seconde des instances terminales. J'ai également proposé de remplacer les concepts de variable de classe et de méthode de classe introduits par Smalltalk-80 [Goldberg et Robson 83] par des variables d'instance et des méthodes définies au niveau de la métaclasse [Briot et Saint-James 84]. Cette architecture a permis de commencer à expérimenter l'intérêt et l'utilisation des métaclasses [Briot 85b].

5.3 Auto-Description

Dans cette première architecture, les métaclasses `Metaclass` et `Class` sont auto-définies, c'est-à-dire définies à partir d'elles-mêmes. Pour être plus précis, `Metaclass` est définie comme instance de `Metaclass` et sous-classe de `Object`, classe la plus générale, tandis que `Class` est définie comme instance *et* sousclasse de `Metaclass`. L'auto-description de cette architecture permet une compréhension plus aisée, puisque l'architecture du langage est définie dans un langage de haut niveau: lui-même, et une extensibilité ouverte, puisque l'architecture est extensible et reconfigurable à partir d'elle-même. L'implémentation de ces définitions mutuelles est assurée par un mécanisme d'amorçe, ou *bootstrap*, en définissant une première version des objets dans le langage d'implémentation, en l'occurence Lisp, puis en les redéfinissant dans le langage à partir d'eux-mêmes.

Malgré ces volontés, l'auto-description était encore limitée. En effet, la description des métaclasses primitives était encore fixée par l'implémentation, les attributs définissant leur structure (classe dont elle hérite, dictionnaire des méthodes...) n'étant pas des variables d'instance explicites. L'implémentation de l'architecture n'était donc pas encore élevée au niveau de l'architecture elle-même.

Un deuxième inconvénient était la non-uniformité de la création des objets. En effet deux méthodes de création cohabitaient, reflétant les deux types d'objets présents: les classes, et les instances terminales.

Pierre Cointe et moi-même avons apporté la solution complète et minimale à la première limitation dans le cadre du projet ObjVlisp.

5.4 Auto-Description et Minimalité: ObjVlisp

5.4.1 Modèle

ObjVlisp est un modèle d'implémentation d'objets en Lisp. Ce projet [Cointe 84b] a été débuté en 1983 par Pierre Cointe [Cointe 84a], comme souci de décrire de manière concise et uniforme les concepts de la programmation par objets en les intégrant dans l'univers de la programmation Lisp.

L'architecture d'ObjVlisp est bâtie à partir de deux classes primitives: `Class` et `Object`. `Class` est la première métaclasse du système, et c'est à partir d'elle que l'on créera récursivement tous les objets. `Object` est la classe la plus générale du système, et toute classe en est une sous-classe (directe ou indirecte). `Class` et `Object` sont donc respectivement les racines des graphes d'instanciation et d'héritage.

L'architecture d'ObjVlisp peut s'implémenter dans différents dialectes de Lisp. Le type de mode de liaison des variables en Lisp induit différents choix d'implémentation des objets. Ainsi les dialectes de Lisp à liaison dynamique des variables entrainent des problèmes de transparence [Briot et al. 86].

Les premières versions d'ObjVlisp ne considéraient pas les classes comme des objets. Plus tard, les classes devinrent des objets, tous instances de l'unique métaclasse prédéfinie, nommée `Class` selon le modèle de Smalltalk-76.

5.4.2 Uniformité

Mon apport au modèle ObjVlisp, en collaboration avec Pierre Cointe [Briot et Cointe 86b], fut de fusionner les deux types d'objets, c'est-à-dire classes et instances terminales. Cette décision s'écartait des choix de Loops et se rapprochait des choix de Smalltalk-80 où instances terminales et classes sont toutes deux représentées par les valeurs associées aux variables d'instance définies par leurs classes.

La première conséquence fut la fusion des deux métaclasses `Metaclass` et `Class` en une seule: `Class`, ce qui assura la minimalité de l'architecture. Les deux méthodes de création furent également fusionnées en une seule unique, pour refléter l'existence d'un seul type d'objets.

5.4.3 Auto-Description Complète

La deuxième conséquence fut l'auto-description complète de la structure des classes. Ainsi la classe `Class` put décrire ses instances, les classes, à travers un ensemble minimal de variables d'instance explicites:

`class`
: qui décrit la classe de la classe vue en tant qu'instance,

`superClasses`
: qui décrit l'ensemble des classes dont la classe hérite,

`instanceVariableNames`
: qui décrit l'ensemble des variables d'instance que définit la classe,

`methodDictionary`
: qui décrit le dictionnaire des méthodes reconnues par les instances de la classe.

Notons que la variable d'instance `class` est définie dans la classe `Object`, classe la plus générale du modèle. Ceci précise que tout objet référence la classe dont il est instance. Comme `Class` est définie comme sous-classe de `Object`, elle hérite de cette variable d'instance. Les trois autres (`superClasses`, `instanceVariableNames`, `methodDictionary`) sont définies par `Class`.

Remarquons que cette auto-description de la structure est une forme réduite et efficace du concept général d'auto-description, appelé en anglais *reflection*. Le concept de méta-objet, représentation de l'implémentation d'un objet en terme d'objet [Maes 87], en est la base. Dans ObjVlisp, la classe est une factorisation de la description structurelle d'un ensemble d'objets, mais ne décrit pas leur activation dynamique. [Cointe et Graube 88] proposent que les métaclasses qui définissent l'allocation des objets (méthode de création) puissent également totalement contrôler leurs représentations internes. Enfin [Watanabe et Yonezawa 88] proposent un modèle de description incluant également l'activité de l'objet.

5.4.4 Class

La classe `Class` en tant qu'objet doit être instance d'une classe. La solution minimale choisie consista à déclarer `Class` instance d'elle-même. Il y a donc auto-description de `Class`. Voici les valeurs associées à ses variables d'instance:

`class`	`Class`
`superClasses`	`(Object)`
`instanceVariableNames`	(class superClasses instanceVariableNames methodDictionary)
`methodDictionary`	`(new` λnew`)`

5.4.5 Object

La classe `Object` est définie comme instance de `Class`. Pour `Object` les valeurs associées aux variables d'instance sont:

`class`	`Class`
`superClasses`	`()`
`instanceVariableNames`	`(class)`
`methodDictionary`	`()`

5.4.6 Bootstrap

On peut remarquer que `Class` et `Object` sont instances de `Class`, et que `Class` hérite de `Object`. Pour pouvoir implémenter ces références croisées, l'installation de l'architecture procède en quatre étapes:

1. implémentation en Lisp d'une version provisoire de la classe `Class`, nommée `Class0`,
2. création en ObjVlisp de la classe `Object` comme instance de `Class0`,
3. création en ObjVlisp de la classe `Class` comme instance de `Class0` et sous-classe de `Object`,
4. remise à jour des références pour que la classe de `Class` et `Object` soit `Class` et non plus `Class0`.

(Notons que la dernière étape n'est pas nécessaire si la variable d'instance `class` ne référence pas la classe mais son nom.)

5.4.7 Uniformité de la Création

L'architecture d'ObjVlisp est ainsi devenue totalement auto-décrite et minimale. Cependant le mode de création n'était toujours pas complètement uniforme. En effet la création d'une classe demande plus d'effort que la création d'une instance terminale, puisqu'il faut entre autres assurer l'héritage des variables d'instance. De ce fait la méthode de création, nommée `new` et détenue par `Class`, testait le type du créateur: métaclasse ou classe, pour déterminer le type de création: classe ou instance terminale. Pierre Cointe et Nicolas Graube ont doté la dernière version d'ObjVlisp [Cointe 87] d'une méthode de création pleinement uniforme. Elle se décompose en la combinaison de l'allocation de l'instance puis de son initialisation. La méthode d'initialisation, nommée `initialize`, est *générique*, c'est-à-dire qu'il y en a un exemplaire dans la classe `Object` et un autre dans la classe `Class`. La première initialise les variables d'instance de l'instance créée à partir de valeurs spécifiées ou par défaut. La deuxième représente l'initialisation des classes, et après avoir appelé la première méthode assure notamment l'héritage statique des variables d'instance.

5.5 Programmation par Métaclasses

Ce modèle d'architecture a donné lieu à un nouveau style de programmation *par métaclasses* [Briot 85b] puis à une méthodologie associée [Briot et Cointe 86a]. Pour définir de nouvelles métaclasses en ObjVlisp, il suffit de définir une classe (donc une instance d'une métaclasse) qui hérite d'une autre métaclasse [Briot et Cointe 86a]. Cette méthodologie de programmation permet de paramétrer les classes par des variables d'instance définies dans des métaclasses [Briot 84] [Cointe 87], et de contrôler l'implémentation des objets [Cointe et Graube 88].

L'architecture d'ObjVlisp par sa minimalité et son auto-description ainsi que la programmation par métaclasses ont influencé nombre réalisations parmi lesquelles: une architecture de types en Lisp permettant un meilleur contrôle de l'implémentation [Queinnec et Cointe 88], plusieurs implémentations de CLOS [Graube 88] [Attardi et al. 89], une extension à objets de Prolog contrôlant l'exécution [Malenfant et al. 89], l'architecture du noyau d'un langage d'acteurs/agents [Ferber et Briot 88], et l'architecture d'un système réparti à objets [Yokote et al. 89]. ObjVlisp a également été abondamment utilisé par Pierre Cointe et moi-même pour enseigner la programmation par objets en DEA et en DESS à l'UPMC depuis 1987.

5.6 Classtalk

Malgré ses intérêts, ObjVlisp en tant que tel est resté par trop minimal, sans environnement ni bibliothèque d'extensions ou d'applications pour entreprendre une expérimentation et des réalisations en vraie grandeur.

Pour cette raison, Pierre Cointe et moi-même avons tout récemment transposé l'architecture d'ObjVlisp dans Smalltalk-80. Cette réalisation [Briot et Cointe 89] va permettre une expérimentation large de la méthodologie induite par l'architecture. L'intérêt de l'implémentation, nommée Classtalk, réside dans la combinaison avec les classes standard, et dans l'intégration dans l'environnement de programmation standard Smalltalk-80. Ceci permet de combiner les bibliothèques de classes Smalltalk-80 avec les métaclasses définies en Classtalk, et d'utiliser l'environnement de programmation comme outil de développement et de classification. L'environnement standard de Smalltalk-80 a été étendu vers un environnement spécifique prenant en compte les spécificités des classes Classtalk. Classtalk va être utilisé comme support d'enseignement dans le cours de DEA 89-90.

6 Objets et Parallélisme

Lors de mon analyse du langage d'acteurs Act1 au cours de ma thèse, j'ai volontairement laissé de côté les aspects de parallélisme. Ensuite, et fort de mon expérience dans le développement du langage à objets concurrents Formes [Cointe et al. 87], j'ai voulu pousser plus loin mon étude des rapports entre objets et parallélisme. J'ai ainsi séjourné pendant 2 ans, comme chercheur invité, dans l'équipe du Professeur Akinori Yonezawa, pour participer à la conception et à l'expérimentation du langage d'acteurs ABCL/1 [Yonezawa et al. 87].

6.1 Acteurs

Les *langages d'acteurs* sont une sous-famille des langages à objets adaptée au parallélisme. Les objets sont actifs, autonomes et concurrents, et ainsi appelés *acteurs*. (Le terme *concurrence* signifie *parallélisme potentiel*, ou logique, c'est-à-dire susceptible d'être implémenté par du parallélisme réel, ou physique, sur des multi-processeurs [Agha 86].) A la différence des objets classiques qui communiquent par transmissions de messages synchrones, le mode primitif de communication entre acteurs est *asynchrone*, ce qui leur évite d'attendre inutilement la fin et le retour en valeur des requêtes de communication.

Le concept d'acteur prend sa source dans les travaux de Carl Hewitt [Hewitt 77]. Plusieurs générations de langages d'acteurs ont été développées dans son équipe: Plasma [Hewitt 77], Act1 [Lieberman 81], Act2 [Theriault 83], et enfin Acore [Manning 87] défini par le dernier modèle de calcul introduit par Gul Agha [Agha 86].

(Signalons un nouveau point de terminologie: notre *définition* des acteurs donnée plus haut ne doit pas être confondue avec le *modèle de calcul* des acteurs défini par Gul Agha. Ainsi nous considérons le langage Act1 comme un langage d'acteurs, bien que défini avant et ne suivant pas ce modèle de calcul.)

Malgré leur grande influence sur beaucoup de modèles de calcul et langages concurrents, la plupart de ces langages d'acteurs ne sont hélas restés qu'à l'état de prototypes peu ou pas diffusés. De plus leur efficacité était souvent médiocre et leur application réduite, du fait du parti-pris "tout acteur" trop anticipateur pour les machines et les programmeurs contemporains. Le langage ABCL/1 (ce qui signifie Actor Based Concurrent Language) [Yonezawa et al. 87] est un descendant de cette famille de pensée dont l'approche diffère par son pragmatisme, c'est-à-dire la considération de l'efficacité de l'implémentation et l'ergonomie du programmeur. De plus le langage a été appliqué avec succès à un certain nombre de domaines (contrôle de processus, simulation, système d'exploitation, analyse syntaxique ...).

6.2 Acteurs et Pragmatisme: ABCL

ABCL/1 présente une approche pragmatique, c'est-à-dire qu'il combine la programmation par acteurs avec la programmation plus traditionnelle. ABCL/1 n'est pas un langage de programmation autonome, mais se combine avec (et est implémenté dans) un langage de programmation plus classique. La version standard est implémentée en Lisp, mais une implémentation a également été réalisée en C [Doi et al. 88]. Le programmeur définit en général des acteurs de grain moyen dont les comportements peuvent s'exprimer en partie dans le langage de base, en l'occurence Lisp. ABCL/1 offre à l'utilisateur plusieurs protocoles de communication évolués: synchrone, anticipé, interruptions, en plus du seul mode de communication asynchrone primitif entre les acteurs.

6.3 Apports

Pendant mon séjour dans l'équipe j'ai participé à la conception du modèle de calcul [Yonezawa et al. 86b] et du langage de programmation [Yonezawa et al. 86a], ainsi qu'à son implémentation à travers une implémentation prototype sur des machines réparties, nommée Micro-ABCL. Ce travail a d'ailleurs été repris puis développé pendant le stage de DEA d'un étudiant sous ma direction [Briot et DeRatuld 88]. Enfin j'ai participé à l'expérimentation du langage par des algorithmes variés [Yonezawa et al. 86a]. J'ai également étudié des modèles et des implémentations de différents langages d'acteurs, tels Act1 et Acore [Manning 87], à travers des micro-interprètes, ainsi qu'un courant voisin, celui des langages de programmation logique concurrente [Kahn et al. 86], profitant en cela de ma proximité du projet de recherche sur les ordinateurs de 5ème génération.

6.4 Partage des Connaissances et Parallélisme

La présence de mécanismes de partage des connaissances (classe, héritage ...) dans un contexte d'exécution parallèle et répartie pose des problèmes importants d'adéquation de ces mécanismes aux contraintes de répartition physique. Il était important de conduire une telle étude dans le cadre du projet, pour évaluer la faisabilité de l'introduction de moyens de spécialisation et de classification dans le langage. Cette étude donna lieu à une évaluation des techniques habituelles dans ce contexte et à de nouvelles propositions [Briot et Yonezawa 87].

6.5 Des Acteurs aux Agents

De retour en France, j'ai collaboré au projet de Jacques Ferber pour concevoir un langage d'acteurs dédié aux architectures multi-agents. Le concept d'acteur est en effet probablement la fondation la plus appropriée pour décrire et implémenter des systèmes multi-agents [Ferber 87]. Par opposition aux approches plus traditionnelles de l'IA qui prennent généralement comme modèle les connaissances et les capacités de raisonnement d'un être humain, l'approche *multi-agents*, également connue sous le nom *Distributed Artificial Intelligence* [DAI 87], prend comme modèle les organisations sociales. Résoudre une tâche complexe est exprimé comme une coopération d'un ensemble d'agents, véritable communauté d'experts interagissant.

Dans ce cadre j'ai collaboré à la définition du modèle de calcul du langage Mering-IV de Jacques Ferber [Ferber et Briot 88]. Ce langage est bâti sur une architecture auto-décrite, inspirée en partie de l'architecture d'ObjVlisp, et est structuré en couches successives. La première couche correspond à un modèle de calcul primitif, tandis que la couche supérieure intègre des mécanismes de manipulation de connaissances.

7 Une Plateforme de Modélisation des Langages d'Acteurs

L'aboutissement de mon travail d'analyse et de synthèse de divers langages d'acteurs, tels Act1, ABCL/1 et Acore est le modèle Actalk [Briot 89a]. Actalk (qui signifie *act*eurs en Small*talk*-80) est l'analogue du modèle ObjVlisp pour les langages d'acteurs. Actalk offre en plus un environnement complet de classification et d'expérimentation, du fait du choix de Smalltalk-80 comme base plutôt que Lisp.

7.1 Motivations

La conception de cette plateforme répond à des besoins profonds. J'ai en effet constaté que, du point de vue du concepteur et du pédagogue, il est souvent difficile d'analyser et plus encore de comparer différents langages d'acteurs. Les différents choix de syntaxe et d'implémentation ont tendance à masquer leur sémantique profonde.

Du point de vue du concepteur et de l'implémenteur, l'expérience (y compris la mienne) montre que le manque d'outils de conception et de mise au point est flagrant. Trop d'implémentations prototypes sont souvent perdues après nombre essais avortés, à la suite desquels le concepteur repart généralement de zéro, avant d'arriver à des concepts stabilisés. La *réutilisabilité* prônée par le génie logiciel apparaît hélas le plus souvent comme un idéal encore non mis en pratique.

Enfin du point de vue de l'utilisateur, toute expérimentation dans le domaine de la programmation parallèle nécessite, plus encore que pour la programmation séquentielle, des environnements spécialisés de visualisation et de contrôle. De tels environnements sont encore trop rares et à l'état de prototypes, bien qu'un effort de recherche dans ce sens soit maintenant réel.

De tels objectifs de modularité, réutilisabilité, structuration et d'environnements intégrés sont parmi les avantages de la programmation par objets. Cette constatation m'a ainsi conduit à concevoir une plateforme de modélisation et d'expérimentation de langages d'acteurs basée sur la programmation par objets.

7.2 Objectifs et Choix de Conception

Je vais maintenant résumer les objectifs ayant guidé les choix de conception d'Actalk:

- *uniformité et modularité*

  Je voulais pouvoir unifier les différents langages d'acteurs dans un environnement unique et pouvoir les modéliser couche par couche. J'ai donc choisi comme base la méthodologie de la programmation par objets. Les acteurs, ou objets actifs, sont introduits dans le monde des objets sans pour autant changer la nature des objets standard.

- *minimalité et extensibilité*

  Je voulais pouvoir définir un noyau minimal du système pour exprimer la sémantique générale des langages d'acteurs puis l'étendre de manière à simuler les langages existants ainsi qu'en concevoir de nouveaux. J'ai donc retenu une architecture minimale dans l'esprit d'ObjVlisp. L'héritage permet en plus de classifier les différents langages d'acteurs à partir du noyau commun.

- *un environnement intégré*

  Je ne souhaitais pas restreindre la plateforme à la seule conception de langages, mais au contraire proposer un environnement complet d'expérimentation. Pour cette raison Smalltalk-80 s'imposait de par son environnement de programmation sophistiqué sur lequel greffer la plateforme. Du fait de l'intégration du noyau d'acteurs dans le système Smalltalk-80, le concepteur et l'expérimentateur peuvent ainsi réutiliser l'environnement standard de Smalltalk-80.

Le choix de Smalltalk-80 permet également une implémentation aisée et minimale d'Actalk, de par la présence en standard de toutes les entités appropriées pour construire des acteurs: objets, classes et messages, et pour exprimer leurs activités simultanées: processus et sémaphores [Bézivin 88]. Le noyau d'Actalk a été aisément étendu pour simuler les principaux modèles et langages d'acteurs, notamment: le modèle de calcul de Gul Agha [Agha 86] et le langage ABCL/1 [Yonezawa et al. 87]. Ainsi Actalk offre une excellente introduction pédagogique aux langages d'acteurs et a déjà été utilisé avec succès dans plusieurs enseignements de DEA, DESS et autres.

Plusieurs projets extérieurs ont choisi Actalk comme support d'étude pour: l'aide à la conception du langages d'acteurs Mering-IV [Ferber et Briot 88], la modélisation de protocoles de communication entre processeurs [Fron 89] (ce projet vient d'ailleurs de faire l'objet d'une soumission jointe à un programme de recherche ANRT), la modélisation de stratégies pour l'allocation d'acteurs et de tâches, et la compilation de règles de production en des démons parallèles implémentés par des acteurs [Voyer 89].

7.3 Environnement

Les acteurs Actalk sont bien intégrés dans l'environnement Smalltalk-80, par conséquent ils bénéficient automatiquement de son environnement de programmation standard. Cependant le modèle d'interface utilisateur de Smalltalk-80 (le MVC, acronyme de Model View Controller) se trouve parfois inadapté à la représentation des acteurs. Les hypothèses cachées mais fortes du MVC sur le séquencement des calculs peuvent se trouver mises en défaut du fait des activités simultanées pendant une interaction avec l'utilisateur. Un projet d'étudiants a donc conçu et réalisé sous ma direction une extension du MVC adaptée aux acteurs Actalk. Un premier prototype de générateur automatique d'interface spécifique aux acteurs Actalk a même été réalisé [Dadi et al. 89]. Un projet plus ambitieux et actuellement en cours de développement par deux étudiants en stage de DESS sous ma direction consiste à concevoir un environnement complet d'animation et de contrôle interactif de programmes d'acteurs Actalk.

Dans le même ordre d'idées, le debugger standard de Smalltalk-80 se trouve inapproprié en cas de transmissions asynchrones entre acteurs. J'ai donc réalisé un premier prototype de debugger mixte objets/acteurs, comme extension du debugger standard. Il utilise des messages étendus qui contiennent le contexte de l'envoyeur pour pouvoir reconstruire la chaîne appropriée de contextes.

7.4 Parallélisme Physique

Actalk malgré ses avantages souffre d'une limitation structurelle: il ne s'agit que de simulation, et il n'y a pas de réelle exécution parallèle. Plutôt que de chercher à exécuter directement des programmes Actalk/Smalltalk sur des machines parallèles, je propose d'adjoindre à la plateforme Actalk actuelle un environnement d'exécution sur machines parallèles de programmes équivalents. Ainsi plutôt que de faire des compromis, on cherche à prendre le meilleur de deux mondes: un environnement de spécification et d'expérimentation (Actalk) bâti en Smalltalk-80 sur station de travail, et un environnement d'exécution bâti en C++ sur machine parallèle répartie. La plateforme complète sera ainsi constituée de deux composantes autonomes mais complémentaires. La composante d'exécution répartie sera constituée d'une extension du langage C++ vers les acteurs, équivalente au noyau d'Actalk, actuellement en cours de conception par une étudiante en thèse sous ma direction, Sylvie Lemarié [Lemarié 89], et d'un système de répartition d'objets dirigé par Philippe Gautron [Shapiro et al. 89]. Ce travail se fait en collaboration avec l'équipe de recherche de Dennis Kafura qui conçoit, dans des buts légèrement distincts, une extension analogue [Kafura et Lee 89].

Cette extension d'acteurs en C++, équivalente au noyau d'Actalk, aura pour première cible une machine parallèle à base de *transputers*. Une interface incluant un traducteur permettra une exécution transparente sur la machine parallèle des programmes développés dans l'environnement Actalk.

8 Prospective

Mon projet à court terme est le développement de la plateforme Actalk et de son environnement d'exécution répartie associé. Ce projet, dont j'ai la responsabilité et qui est mené avec Philippe Gautron, vient d'être accepté comme programme de recherche Esprit de la CEE sur la programmation parallèle.

Mes projets à moyen terme sont le développement de mon étude des concepts d'auto-description des acteurs à la suite des travaux de [Watanabe et Yonezawa 88], avec application notamment aux problèmes de synchronisation, tels le *dead-lock*, et l'application de la plateforme à de nombreux domaines en relation avec d'autres équipes, tels les systèmes réactifs [Kafura et Lee 89], l'IA [Voyer 89] [Ferber et Briot 88] [DAI 87], la modélisation de machines spécialisées [Fron 89]...

Mes projets à long terme envisagent une investigation et une collaboration active avec les principaux acteurs des modèles alternatifs de calcul parallèle (modèles physiques, chimiques, biologiques, éthologiques...) [Steels 88]. Ces domaines aideront notamment à caractériser la notion de milieu, la topologie explicite et point à point des acteurs étant par trop de bas niveau, et la relation macroscopique/microscopique qui lie le comportement global d'un système parallèle aux activités de chacun de ses éléments.

References

[Agha 86] G. Agha, Actors: a Model of Concurrent Computation in Distributed Systems, *Series in Artificial Intelligence*, MIT Press, Cambridge MA, Etats-Unis, 1986.

[Attardi et al. 89] G. Attardi, C. Bonini, M.-R. Boscotrecase, T. Flagella et M. Gaspari, Metalevel Programming in CLOS, dans [ECOOP 89], pages 244-256.

[Bézivin 88] J. Bézivin, Langages à Objets et Programmation Concurrente: quelques Expérimentations avec Smalltalk-80, *Actes des Journées Afcet-Groplan Langages et Algorithmes*, Bigre+Globule, No 59, pages 176-187, Irisa, Rennes, Avril 1988.

[Birtwistle et al. 73] G. Birtwistle, O.-J. Dahl, B. Myhrhaug et K. Nygaard, Simula Begin, Petrocelli - Charter, New York NY, Etats-Unis, 1973.

[Bobrow et Stefik 83] D. Bobrow et M. Stefik, The LOOPS Manual, Xerox Parc, Palo Alto CA, Etats-Unis, 1983.

[Borning 86] A. Borning, Classes Versus Prototypes in Object-Oriented Languages, *Proceedings of the Fall 86 Joint Computer Conference*, ACM, pages 36-39, Novembre 1986.

[Cointe 84a] P. Cointe, Une Extension de VLisp par les Objets, *Science of Computer Programming*, No 161, pages 291-322, North-Holland, 1984.

[Cointe 84b] P. Cointe, Implémentation et Interprétation des Langages Objets, Application aux Langages Formes, ObjVlisp et Smalltalk, *Thèse d'Etat, Université Pierre et Marie Curie, 17 Décembre 1984*, Rapport de Recherche LITP 85-55, Octobre 1985.

[Cointe et Rodet 84] P. Cointe et X. Rodet, Formes: an Object & Time Oriented System for Composition and Synthesis, *1984 Symposium on Lisp and Functional Programming (LFP'84)*, Austin, Texas, 5-8 Août 1984.

[Cointe 85] P. Cointe, le Modèle ObjVlisp: une Plate-Forme pour l'Expérimentation des Formalismes Objets, *5ème Congrès Afcet de Reconnaissance des Formes et Intelligence Artificielle (RFIA'85)*, Tome II, pages 737-754, Dunod, Novembre 1985.

[Cointe 87] P. Cointe, Metaclasses are First Class: the ObjVlisp Model, dans [OOPSLA 87], pages 156-167.

[Cointe et Graube 88] P. Cointe et N. Graube, Programming with Metaclasses in CLOS, *First CLOS Users and Implementors Workshop*, pages 23-29, Xerox Parc, Palo Alto CA, Etats-Unis, Octobre 1988.

[Dadi et al. 89] K. Dadi, C.-H. Ky, M. LeBris et L. Lescaudron, Actalk: Extension MVC - Réalisation d'un Générateur Automatique d'Interface Utilisateur Graphique Dédiée aux Acteurs, *Mémoire de Projet de DESS, direction: J.-P. Briot*, Université Pierre et Marie Curie, Paris, France, Avril 1989.

[DAI 87] Distributed Artificial Intelligence, édité par M.N. Huhns, Pitman - Morgan Kaufman, London, Royaume-Uni, 1987.

[Doi et al. 88] N. Doi, Y. Kodama et K. Hirose, An Implementation of an Operating System Kernel using Concurrent Object-Oriented Language ABCL/c+, dans [ECOOP'88], pages 250-266.

[ECOOP 87] *European Conference on Object-Oriented Programming (ECOOP'87)*, Lecture Notes in Computer Science, No 276, Springer-Verlag, Berlin, RFA, 1987.

[ECOOP 88] *European Conference on Object-Oriented Programming (ECOOP'88)*, Lecture Notes in Computer Science, No 322, Springer-Verlag, Berlin, RFA, Août 1988.

[ECOOP 89] *European Conference on Object-Oriented Programming (ECOOP'89), Nottingham, Royaume-Uni, 11-14 Juillet 1989*, BCS Workshop Series, Cambridge University Press, Royaume-Uni, Juillet 1989.

[Ferber 83] J. Ferber, Mering: un Langage d'Acteur pour la Repésentation et la Manipulation des Connaissances, *Thèse de Docteur Ingénieur*, Université Pierre et Marie Curie, Décembre 1983.

[Ferber 84] J. Ferber, Quelques Aspects du Caractère Self-Réflexif du Langage Mering, *Deuxièmes Journées d'Etude du Groupe de Travail Afcet-Informatique sur les Langages Orientés Objet*, Bigre+Globule, No 41, pages 277-290, Novembre 1984.

[Ferber 87] J. Ferber, Des Objets aux Agents: une Architecture Stratifiée, *6ème Congrès Afcet de Reconnaissance des Formes et Intelligence Artificielle (RFIA'87)*, Vol. 1, pages 275-286, Dunod, Novembre 1987.

[Fron 89] A. Fron, Instanciation of OBC in Real World, *Position Statements for the ECOOP'89 Workshop on Object-Based Concurrency*, Centre Universitaire Informatique, Université de Genève, Genève, Suisse, Mai 1989.

[Goldberg et Robson 83] A. Goldberg et D. Robson, Smalltalk-80: the Language and its Implementation, *Series in Computer Science*, Addison Wesley, Reading MA, Etats-Unis, 1983.

[Graube 88] N. Graube, Reflexive Architectures: From ObjVlisp to CLOS, dans [ECOOP'88], pages 110-127.

[Greussay 76] P. Greussay, Descriptions Compactes d'Interprètes Implémentables: une Application au Langage Conniver, *2ième Colloque International sur la Programmation*, éditeur: B. Robinet, pages 218-231, Dunod, Avril 1976.

[Hewitt 77] C.E. Hewitt, Viewing Control Structures as Patterns of Passing Messages, *Journal of Artificial Intelligence*, Vol. 8 No 3, pages 323-364, 1977.

[Ingalls 78] D.H.H. Ingalls, The Smalltalk-76 Programming System Design and Implementation, *5th Annual ACM Symposium on Principles of Programming Languages*, Tucson AZ, Etats-Unis, ACM, pages 9-15, Janvier 1978.

[Kafura et Lee 89] ACT++: Building a Concurrent C++ with Actors, D. Kafura et K.H. Lee, Rapport de Recherche, Dept. of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg VA, Etats-Unis, Mai 1989.

[Kahn et al. 86] K. Kahn, E. Dean Trebble, M.S. Miller et D.G. Bobrow, Objects in Concurrent Logic Programming Languages, dans [OOPSLA 86], pages 242-257.

[Kay 69] A. Kay, The Reactive Engine, Phd Thesis, University of Utah, Salt-Lake City UT, Etats-Unis, 1969.

[Lalonde et al. 86] W.R. Lalonde, D.A. Thomas et J.R. Pugh, Actors in a Smalltalk Multiprocessor: a Case for Limited Parallelism, *Technical Report SCS-TR-91*, School of Computer Science, Carleton University, Ottawa, Canada, Mai 1986.

[Lemarié 89] S. Lemarié, Sur la Trace des Vikings: Beta - Comparaison entre Beta et C++, *Mémoire de DEA LAP*, Université Pierre et Marie Curie, Février 1989.

[Lieberman 81] H. Lieberman, Concurrent Object-Oriented Programming in Act1, dans [OOCP 87], pages 9-36.

[Lieberman 84] H. Lieberman, Object-Oriented Programming Languages, entrée dans l'Encyclopedia of AI, éditeur: S. Shapiro, Willey, New-York NY, Etats-Unis, 1985.

[Lieberman 87] H. Lieberman, Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems, dans [OOPSLA 87], pages 214-223.

[Maes 87] P. Maes, Concepts and Experiments in Computational Reflection, dans [OOPSLA 87], pages 147-155.

[Malenfant et al. 89] J. Malenfant, G. Lapalme et J. Vaucher, ObjVProlog: Metaclasses in Logic, dans [ECOOP 89], pages 257-269.

[Manning 87] C.R. Manning, Acore: The Design of a Core Actor Language and its Compiler, *Revised Master Thesis*, EE and CS dept., MIT, Cambridge MA, Etats-Unis, Mai 1987.

[Moon 86] D. Moon, Object-Oriented programming with Flavors, in [OOPSLA 86], pages 1-8.

[OOCP 87] Object-Oriented Concurrent Programming, edité par A. Yonezawa et M. Tokoro, *Computer Systems Series*, MIT Press, Cambridge MA, Etats-Unis, 1987.

[OOPSLA 86] *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86)*, Special Issue of Sigplan Notices, ACM, Vol. 21, No 11, Novembre 1986.

[OOPSLA 87] *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, Special Issue of Sigplan Notices, ACM, Vol. 22, No 12, Décembre 1987.

[OOPSLA 88] *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, Special Issue of Sigplan Notices, ACM, Vol. 23, No 11, Novembre 1988.

[OOPSLA 89] *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89), New-Orleans LO, Etats-Unis, 2 - 6 Octobre 1989*, à paraître en Special Issue of Sigplan Notices, ACM, Hiver 1989.

[Queinnec et Cointe 88] C. Queinnec et P. Cointe, An Open Ended Data Representation Model for EU_LISP, *1988 Conference on Lisp and Functional Programming (LFP'88)*, ACM, Snowbird UT, Etats-Unis, Juillet 1988.

[Shapiro et al. 89] M. Shapiro, P. Gautron et L. Mossieri, Persistence and Migration for C++ Objects, dans [ECOOP 89], pages 191-204.

[Steels 88] L. Steels, Artificial Intelligence and Complex Dynamics, Research Report 88-1, AI Lab, Vrije Universiteit Brussels, Bruxelles, Belgique, 1988.

[Stroustrup 86] B. Stroustrup, The C++ Programming Language, Addison-Wesley, Reading MA, Etats-Unis, 1986.

[Theriault 83] D. Theriault, Issues in the Design and Implementation of Act2, *Technical Report No 728*, AI Lab, MIT, Cambridge MA, Etats-Unis, Juin 1983.

[Voyer 89] R. Voyer, Implémentation d'Architectures Efficaces pour la Représentation des Connaissances. Application aux Langages Loopsiris et Oks, *Thèse d'Université*, Université Pierre et Marie Curie, Paris, Février 1989.

[Watanabe et Yonezawa 88] T. Watanabe et A. Yonezawa, Reflection in an Object-Oriented Concurrent Language, dans [OOPSLA 88], pages 306-315.

[Yokote et Tokoro 87] Y. Yokote et M. Tokoro, Experience and Evolution of ConcurrentSmalltalk, dans [OOPSLA 87], pages 406-415.

[Yokote et al. 89] Y. Yokote, F. Teraoka et M. Tokoro, A Reflective Architecture for an Object-Oriented Distribution Operating System, dans [ECOOP 89], pages 89-106.

[Yonezawa et al. 87] A. Yonezawa, E. Shibayama, T. Takada et Y. Honda, Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, dans [OOCP 87], pages 55-89.

[Briot et Serpette 82] J.-P. Briot et B. Serpette, Le Langage Formes - Analyse et Implémentation, *Mémoire de DEA LAP*, Université Pierre et Marie Curie, Juillet 1982.

[Briot 83] J.-P. Briot, L'Instanciation dans les Langages Objets, *Journées d'Etude sur les Langages Orientés Objet*, Bigre, No 37, pages 173-209, Décembre 1983.

[Rodet et al. 83] X. Rodet, P. Cointe, J.-B. Barrière, Y. Potard, J.-P. Briot et B. Serpette, Applications and Developments of the Formes Programming Environment, *International Computer Music Conference (ICMC'83)*, Rochester NJ, Etats-Unis, Octobre 1983.

[Rodet et al. 84] X. Rodet, P. Cointe, J.-B. Barrière, Y. Potard, J.-P. Briot et B. Serpette, Démonstration de Formes et son Utilisation pour Contrôler des Synthétiseurs, *International Computer Music Conference (ICMC'84)*, pages 110-111 (241-242 en version Anglaise), Ircam, 19-23 Octobre 1984.

[Briot et Saint-James 84] J.-P. Briot et E. Saint-James, Intégration de Smalltalk dans le Projet Aglae, *Deuxièmes Journées d'Etude du Groupe de Travail Afcet-Informatique sur les Langages Orientés Objet*, Bigre+Globule, No 41, pages 52-72, Novembre 1984.

[Briot 84] J.-P. Briot, Instanciation et Héritage dans les Langages Objets, *Thèse de 3ème cycle, Université Pierre et Marie Curie, 15 Décembre 1984*, Rapport de Recherche LITP 85-21, Mai 1985.

[Briot 85a] J.-P. Briot, Présentation du Groupe "Langages Orientés Objets", *Actes du Séminaire d'Informatique Théorique 1984-1985*, LITP, pages 77-85, Automne 1985.

[Briot 85b] J.-P. Briot, Les Métaclasses dans les Langages Orientés Objets, *5ème Congrès Afcet de Reconnaissance des Formes et Intelligence Artificielle (RFIA'85)*, Tome II, pages 755-764, Dunod, Novembre 1985.

[Briot et al. 86] J.-P. Briot, P. Cointe et E. Saint-James, Reécriture et Récursion dans une Fermeture - Etude dans un Lisp à Liaison Superficielle - Application aux Objets, *Actes des Journées Afcet-Informatique Langages Orientés Objet*, Bigre+Globule, No 48, pages 90-100, Janvier 1986.

[Briot et Cointe 86a] J.-P. Briot et P. Cointe, Construction Réflexive d'un Langage Objet: ObjVLisp, *3ème Colloque Afcet sur le Génie Logiciel (CGL3)*, pages 281-294 Afcet, Mai 1986.

[Briot et Cointe 86b] J.-P. Briot et P. Cointe, Definition of a Uniform, Reflexive and Extensible Object-Oriented Language, *European Conference on Artificial Intelligence (ECAI'86)*, Advances in Artificial Intelligence-II, North-Holland, Amsterdam, Pays-Bas, pages 225-232, 1987.

[Briot et Yonezawa 86] J.-P. Briot et A. Yonezawa, Inheritance Mechanisms in Distributed Object-Oriented Languages, *Conference on Software Science and Engineering (SSE'86)*, RIMS, Kyoto University, Kyoto, Japon, 8-10 Septembre 1986.

[Yonezawa et al. 86a] A. Yonezawa, J.-P. Briot et E. Shibayama, Object-Oriented Concurrent Programming in ABCL/1, dans [OOPSLA 86], pages 258-268.

[Yonezawa et al. 86b] A. Yonezawa, E. Shibayama et J.-P. Briot, Le Modèle ABCM/1, *(en Japonais)*, JCSJ, Vol. 3, No 3, pages 9-23, Tokyo, Japon, 1986.

[Briot et Yonezawa 87] J.-P. Briot et A. Yonezawa, Inheritance Mechanisms in Object-Oriented Concurrent Languages, dans [ECOOP 87], pages 32-40.

[Briot et Cointe 87] J.-P. Briot et P. Cointe, A Uniform Model for Object-Oriented Languages Using the Class Abstraction, *Tenth International Joint Conference on Artificial Intelligence (IJCAI'87)*, Vol. 1, pages 40-43, Milan, Italie, 23-28 Août 1987.

[Cointe et al. 87] P. Cointe, J.-P. Briot et B. Serpette, The Formes Language: A Musical Application of Object-Oriented Concurrent Programming, dans [OOCP 87], pages 221-258.

[Briot 88] J.-P. Briot, From Objects to Actors: Study of a Limited Symbiosis in Smalltalk-80, *Rapport de Recherche RXF-LITP 88-58*, Septembre 1988.
Egalement paru en Extended Abstract au *Workshop on Object-Based Concurrent Programming (OBCP'88)*, Sigplan Notices, Vol. 24, No 4, pages 69-72, ACM, Avril 1989.

[Briot et DeRatuld 88] J.-P. Briot et J. DeRatuld, Design of a Distributed Implementation of ABCL/1, *Workshop on Object-Based Concurrent Programming (OBCP'88)*, Sigplan Notices, Vol. 24, No 4, pages 15-17, ACM, Avril 1989.

[Ferber et Briot 88] J. Ferber et J.-P. Briot, Design of a Concurrent Language for Distributed Artificial Intelligence, *International Conference on Fifth Generation Computer Systems (FGCS'88)*, Vol. 2, pages 755-762, Icot, Tokyo, Japon, Novembre-Décembre 1988.

[Briot 89a] J.-P. Briot, Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment, *Rapport de Recherche RXF-LITP 89-33*, Mai 1989.
Egalement paru dans [ECOOP 89], pages 109-129.

[Briot et Cointe 89] J.-P. Briot et P. Cointe, Programming with ObjVlisp Metaclasses in Smalltalk-80, *Rapport de Recherche RXF-LITP 89-28*, Avril 1989.
Egalement à paraître dans [OOPSLA 89].

[Briot 89b] J.-P. Briot, Actalk: une Plateforme de Modélisation de Langages d'Acteurs en Smalltalk-80, *Rapport de Recherche RXF-LITP 89-34*, Mai 1989.
Egalement paru dans les *Actes des Mardis Objets du CRIN, Nancy, 20 Octobre 1987-31 Mai 1988*, *Rapport de Recherche CRIN 89-R-072*, Printemps 1989,
Egalement à paraître dans les *Actes des Journées Afcet-Groplan sur le Développement de Programmes pour Machines Parallèles, Chamonix, 17-19 Mai 1989*, Bigre-Globule, Eté 1989,
et dans les *Actes du 7ème Congrès Afcet de Reconnaissance des Formes et Intelligence Artificielle (RFIA'89), Paris, 29 Novembre-1er Décembre 1989*, Dunod, Hiver 1989.

[Briot 89c] J.-P. Briot, OOCP = OOP + C, *Position Statements for the ECOOP'89 Workshop on Object-Based Concurrency*, Centre Universitaire Informatique, Université de Genève, Genève, Suisse, Mai 1989.

Contents

1 Introduction **2**

2 Plan **3**

3 Introduction Musicale aux Objets **3**
3.1 Cadre Général 3
3.2 Développement 4
3.3 Environnement 4
3.4 Un Modèle Dual de Création 4

4 Sémantique des Relations entre Objets **5**
4.1 Création 5
4.2 Les Deux Modèles 6
4.3 Modèle Abstrait: Concept de Classe 6
4.4 Classe = Objet ? 6
4.5 Objet = Classe ? 6
4.6 Modèle Concret: Concept de Prototype 7
4.7 Prototype ou Acteur: un Point de Terminologie 7
4.8 Des Prototypes aux Frames 7
4.9 Classe ou Prototype 8
4.10 Classe et Prototype 9

5 Classes de Première Classe **9**
5.1 Métaclasses 9
5.2 Métaclasses Explicites 9
5.3 Auto-Description 10
5.4 Auto-Description et Minimalité: ObjVlisp 10
5.4.1 Modèle 10
5.4.2 Uniformité 11
5.4.3 Auto-Description Complète 11
5.4.4 Class 12
5.4.5 Object 12
5.4.6 Bootstrap 12
5.4.7 Uniformité de la Création 13
5.5 Programmation par Métaclasses 13
5.6 Classtalk 13

6 Objets et Parallélisme **14**
6.1 Acteurs 14
6.2 Acteurs et Pragmatisme: ABCL 14
6.3 Apports 15
6.4 Partage des Connaissances et Parallélisme 15
6.5 Des Acteurs aux Agents 15

7 Une Plateforme de Modélisation des Langages d'Acteurs **16**
7.1 Motivations 16
7.2 Objectifs et Choix de Conception 16
7.3 Environnement 17
7.4 Parallélisme Physique 17

8 Prospective **18**