Modélisation et classification de langages de programmation concurrente à objets : l'expérience Actalk *

Jean-Pierre BRIOT

LITP
Institut Blaise Pascal
4 place Jussieu
75252 Paris Cedex 05, France
fax : +33 (1) 44-27-62-86
e-mail : `Jean-Pierre.Briot@litp.ibp.fr`

Dept. of Information Science
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku
Tokyo 113, Japon
+81 (3) 5689-4365
`briot@is.s.u-tokyo.ac.jp`

Résumé : Cet article présente la conception d'une plateforme de *modélisation*, *classification* et *expérimentation* de *programmation concurrente par objets*. Notre motivation est d'offrir un outil qui peut aider à décrire, analyser, comparer, combiner, expérimenter, et enfin utiliser différents formalismes (modèles de calcul, langages de programmation, stratégies de synchronisation...) de programmation concurrente par objets. Cette plateforme, nommée *Actalk*, est implantée et *intégrée* à l'environnement de programmation *Smalltalk-80*.

Actalk se fonde sur l'existence d'un *noyau* qui modélise la sémantique de base (c'est-à-dire des *objets actifs* et séquentiels communicant par *envoi de messages asynchrone* et unidirectionnel). Ce noyau se compose d'un ensemble de classes ainsi que de *méthodes paramètres* associées destinées à être redéfinies. Différentes *extensions*, définies comme sous-classes des classes du noyau, simulent divers formalismes de programmation concurrente à objets. Elles expriment différents modèles d'*activité* (séquentiel, concurrent...), de *communication* (synchrone, à réponse anticipée...), et de stratégies de *synchronisation* d'activation des requêtes.

Dans cet article nous décrivons les motivations et les choix de conception de la plateforme Actalk ainsi que l'architecture de sa dernière version. Les choix de représentation, décomposition et de *paramétrisation* de l'architecture sont plus particulièrement discutés dans cet article. Nous résumons ensuite les différents formalismes actuellement modélisés comme sous-classes/extensions des classes composant le noyau. Nous présentons enfin un exemple développé pour illustrer l'expressivité de cette plateforme à travers la *combinaison* de deux formalismes de synchronisation dédiés à la programmation concurrente par objets. Enfin, avant de conclure, nous discuterons brièvement les intérêts de même que les limites de la plateforme, puis résumerons ses différents développements ainsi que les travaux similaires.

Mots-clés : modélisation, classification, programmation concurrente à objets, plateforme, Smalltalk-80.

*La version préalable de ce rapport est parue dans les actes du colloque "Langages et Modèles à Objets", INRIA/IMAG/PRC-IA, Grenoble, 13-14 Octobre 1994.

Modeling and Classifying Various OOCP Languages : the Actalk Experience

Jean-Pierre BRIOT

Abstract : This paper describes the design of a testbed for *modeling*, *classifying* and *experimenting* with *object-oriented concurrent programming* (OOCP in short). We aim at providing a tool to help describing, analyzing, comparing, combining, experimenting, and using various OOCP formalisms (computation models, programming languages, synchronization strategies…). This testbed, named *Actalk*, is implemented and *integrated* within the *Smalltalk-80* programming environment.

Actalk is based on some *kernel* which models basic OOCP semantic (i.e., serial *active objects* communicating through unidirectional *asynchronous message send*). This kernel is composed of a set of classes and their associated *parameter methods* intended to be further redefined. Various *extensions*, defined as sub-classes of the kernel classes, simulate various OOCP formalisms. They specify different models of *activity* (serial, concurrent…), *communication* (synchronous, with eager reply…), and *synchronization* strategies (for controlling activation of requests).

In this paper we first discuss motivations and design decisions for the Actalk testbed. We then describe the architecture of the new version of Actalk. Representation choices as well as decomposition and *parameterization* of the architecture are specially discussed in this paper. We summarize various formalisms which have been actually modeled/simulated as sub-classes/extensions of the kernel classes. We finally introduce and detail one example *combining* two synchronization formalisms in order to show the expressivity of the platform. Before concluding we briefly discuss the merits and limitations of the platform and we summarize its various developments as well as related work.

Keywords : modeling, classifying, object-oriented concurrent programming, testbed/platform, Smalltalk-80.

Introduction

Cet article aborde le domaine de la *modélisation* et la *classification* de langages de *programmation concurrente par objets*. Nous avons conçu et développé il y a plusieurs années une plateforme dans ce but, nommée *Actalk*, et implantée et intégrée dans l'environnement Smalltalk-80 [BRIO89]. Cette plateforme a permis de simuler plusieurs types de langages de programmation concurrente par objets, définis comme sous-classes d'un noyau commun. La plateforme Actalk a été utilisée par plusieurs équipes non seulement en enseignement, mais également comme fondation pour expérimenter diverses stratégies de programmation, ainsi que pour développer divers systèmes, en particulier multi-agents [1]. A la suite de ces expériences, nous avons tout récemment rajouté et intégré à la plateforme Actalk un grand nombre de nouveaux formalismes et mécanismes [2]. Ceci nous a amenés à affiner l'architecture initiale de la plateforme de manière à augmenter encore ses modularité et expressivité initiales.

Ce papier a donc pour ambition de présenter notre expérience et nos résultats actuels dans le domaine de la *modélisation* et la *classification* de *langages de programmation concurrente à objets*. Un point crucial est celui des compromis optimaux à trouver entre *expressivité*, *minimalité*, et *praticalité* de notre plateforme. Par rapport à un système de représentation de connaissances général, le domaine de modélisation d'Actalk est fortement restreint au domaine de la programmation concurrente à objets. Mais Actalk doit permettre non seulement une *modélisation* mais également une *programmation* et une *expérimentation* effectives.

1 Contexte et motivations

Nous allons tout d'abord introduire très brièvement le domaine que nous souhaitons modéliser, c'est-à dire celui des langages de programmation concurrente par objets.

1.1 Programmation concurrente par objets

La *programmation concurrente par objets* (en anglais, *object-oriented concurrent programming* [OOCP87]) est un type de programmation issu de la *programmation par objets* et prenant en compte la présence de *multiples activités et communications simultanées*.

Elle répond particulièrement bien à deux des besoins et enjeux clés de la programmation actuelle et future : (1) le besoin de *méthodologies* pour pouvoir gérer la nécessaire *décomposition* de problèmes et programmes dont la complexité va croissant, ainsi que pour gérer leur *évolution* ; et (2) l'exploitation des perspectives d'*efficacité* accrue offertes par les nouvelles architectures de machines *parallèles*.

La programmation concurrente par objets *généralise* les principes et les possibilités de la programmation par objets traditionnelle, c'est-à-dire séquentielle. Elle repose sur l'*intégration* de la notion d'*activité* avec celle d'*objet*. Il est en effet naturel d'identifier les objets comme unité d'activité et d'associer la synchronisation entre ces activités au niveau de la transmission de message entre les objets. Cette unification préserve la plupart des principes et avantages de la programmation par objets traditionnelle tout en renforçant encore l'auto-suffisance et l'autonomie des objets.

[1] Ces différents projets sont évoqués brièvement au §5.2.

[2] Certaines modélisations ont été développées en Actalk à l'origine par d'autres équipes, voir au §5.2.

De tels objets doués d'activité autonome sont appelés *objets actifs*. La forme la plus élémentaire et fondamentale de communication entre ces objets actifs est l'envoi de message *asynchrone* et *unidirectionnel*, c'est-à-dire sans attente d'une réponse par l'émetteur du message.

1.2 Variations et enjeux

Du fait du succès de la programmation concurrente par objets, différents *formalismes* (nous incluons : différents modèles de calcul, langages de programmation, constructions associées, stratégies de synchronisation et de coordination...) ont été et sont encore actuellement proposés. Ces différentes propositions, tout en partageant les principes énoncés plus haut, proposent des stratégies et solutions différentes relatives aux aspects principaux suivants :

- **communication** : un objet communique-t'il par envoi de message asynchrone, synchrone, ou à réponse anticipée ?, implicite ou explicite ?, simple ou multiple ?, avec envoi prioritaire ?... ;
- **activité** : un objet accepte-t'il implicitement ou explicitement les messages ?, avec possibilité éventuelle de traiter plusieurs messages simultanément ?... ;
- **synchronisation et coordination** : comment un objet contrôle-t'il l'activation des messages (quand une requête peut-elle être traitée) ?, en fonction de quelles données et ressources ? (son état, les invocations en cours, les services demandés...), et selon quelle *stratégie* (états abstraits, compteurs de synchronisation...) ? [3]

1.3 Motivations pour une plateforme unificatrice

La variété des modèles de langages de programmation concurrente à objets proposés et en général la difficulté de les comparer aisément (du fait des différences éventuelles de terminologie, de syntaxe, et d'implantation) nous ont amené au cours de l'Eté 1988 à concevoir et développer une plateforme unificatrice de modélisation, classification, et expérimentation. Cette plateforme, développée et intégrée dans l'environnement de programmation Smalltalk-80, a été nommée *Actalk*, dont l'origine du nom signifie : objets *act*ifs (et *act*eurs) en Small*talk*-80.

Ses motivations sont à la fois :

- **pédagogiques** pour permettre d'*analyser* et de *classifier* différents formalismes (modèles, langages...) dont les différents choix de syntaxe et d'implantation ont souvent tendance à masquer leur sémantique profonde ;
- pour faciliter la **conception** de nouveaux formalismes, par *dérivation* et *combinaison* de formalismes déjà existants (un exemple en sera donné au §4.3) ;

[3] Ce point est en fait crucial et fait encore actuellement l'objet d'une importante activité de la part de la communauté scientifique spécialisée. Le problème clé est de pouvoir utiliser les outils méthodologiques classiques, et donc en premier lieu l'héritage, pour pouvoir spécialiser et réutiliser les spécifications de synchronisation. Ce problème s'avère être non trivial, et a même été de ce fait nommé *"inheritance anomaly"* [MATS&93]. L'objectif est donc d'élaborer des formalismes de synchronisation suffisamment abstraits et modulaires pour être aisément *réutilisés*. Notre plateforme inclut déjà la plus grande partie des différents formalismes actuellement proposés. Un exemple de mise en oeuvre sera décrit au §4.

- pour permettre une **expérimentation** active à la méthodologie de programmation concurrente par objets à l'aide d'un *environnement de programmation* incluant des outils de visualisation et de contrôle adaptés.

2 Conception de la plateforme Actalk

2.1 Objectifs et choix de conception

La genèse des objectifs ainsi que des choix de conception de la plateforme Actalk a déjà été discutée dans [BRIO89]. Nous nous bornerons donc ici à les résumer brièvement :

- **uniformité** et **modularité**. Différents types de modèles, langages et stratégies de programmation concurrente par objets sont rassemblés dans un environnement *unique* ;
- **minimalité**. Actalk est composé d'un *noyau minimal* qui modélise la sémantique fondamentale de la programmation concurrente à objets ;
- **extensibilité** et **expressivité**. Ce noyau est ensuite *étendu* pour représenter et *classifier* différents types de langages de programmation concurrente à objets sous forme de *sous-classes* des classes du noyau ;
- **simplicité**. Nous nous intéressons à la représentation des caractéristiques fondamentales des différents langages ainsi simulés, en laissant de côté les considérations d'optimisation, de preuve, et de sécurité quand elles complexifieraient la modélisation ;
- **intégration**. Actalk est intégré à l'environnement de programmation Smalltalk-80 dans lequel il est implanté. Ainsi le programmeur peut combiner différents types de programmation concurrente par objets avec la programmation par objets traditionnelle offerte par Smalltalk-80 (et donc en particulier réutiliser les programmes séquentiels) pour exprimer des *granularités* (taille, concurrence interne, connaissances...) variées ;
- **expérimentation**. Actalk inclut un nombre important de formalismes variés et exemples de programmes, tous regroupés dans un environnement unifié. Il est ainsi aisé de les étudier, comparer, étendre, et enfin dans une certaine mesure de les combiner. Enfin les outils d'environnement proposés par Smalltalk-80 sont disponibles pour le développement de formalismes et de programmes [4].

2.2 Modèles de représentation

Ces différents objectifs affichés peuvent être en partie contradictoires entre eux (ainsi par exemple minimalité et expressivité). Il s'agit donc de tenter de trouver les meilleurs compromis entre simplicité, expressivité *et* praticalité.

Comme fondation de la plateforme il nous fallait choisir un environnement axé soit sur la représentation de connaissances, soit sur la programmation. Les objectifs énoncés plus

[4]En fait certains des outils standard de Smalltalk-80, tels le modèle d'interface homme-machine MVC ainsi que le dévermineur nécessitent quelques ajustements à la nature des objets actifs. Un tel projet a déjà été réalisé et a également débouché sur de nouveaux outils spécialisés (tels un séquenceur générique de processus ainsi que ses outils de visualisation associés) [LESC92]. Ayant été développés dans une version précédente de Smalltalk-80, ils n'ont pas encore été portés dans leur intégralité par manque de temps/moyens.

haut montrent qu'Actalk n'a pas pour vocation d'être une plateforme de représentation générale, mais qu'elle est dédiée à la représentation de divers formalismes de programmation concurrente par objets. Dans le but de pouvoir offrir un environnement de programmation effective suffisamment élaboré, nous avions choisi comme fondation de la plateforme un environnement de programmation élaboré, en l'occurence Smalltalk-80.

Nous avons choisi Smalltalk-80 comme fondation pour ses qualités suivantes : simplicité (des concepts), haut-niveau, modularité, flexibilité, richesse de l'environnement, et enfin diffusion large. Smalltalk-80 est assez modulaire et flexible pour permettre une excellente intégration de la plateforme Actalk dans son environnement. Smalltalk-80 permet également une implémentation minimale de par la présence en standard de toutes les entités appropriées pour construire des objets actifs : objets, classes ; pour gérer leurs activités : processus, traitement de message (`perform:`) ; leur communication : messages, et queues (de messages) ; et leur synchronisation : sémaphores, queues partagées (de messages). La représentation des objets actifs d'Actalk à l'aide d'objets standard de Smalltalk-80 offre ainsi une méta-représentation de haut niveau et presque équivalente aux objets d'Actalk [5].

3 Décomposition de l'architecture

Nous allons maintenant détailler les principes de l'architecture d'Actalk en précisant plus particulièrement ici la manière dont elle est *décomposée* et *paramétrée* pour permettre une grande expressivité des aspects énoncés au §1.2. Les considérations de syntaxe seront donc secondaires et unifiées dans la syntaxe standard de Smalltalk-80.

Actalk peut être décomposé en son *noyau* qui définit la sémantique de base des objets actifs, ainsi que ses diverses *extensions*. Ces extensions sont différents modules de classes (appelés *catégories* dans la terminologie Smalltalk-80) représentant divers modèles et constructions de langages de programmation concurrente par objets, ainsi que nombreux exemples.

3.1 Noyau

Le noyau d'Actalk est décomposé en plusieurs classes de base et leurs méthodes qui représentent les différents aspects énoncés au §1.2. Nous avons choisi une architecture avec un ensemble bien délimité de classes et de méthodes définissant ces fonctionnalités à vocation d'être redéfinies dans les sous-classes extensions. Ces fonctionnalités, nommées *méthodes paramètres*, paramétrisent les différents aspects clés (réception, sélection, activation d'un message...) d'un type de langage donné. Notons que l'identification et le nombre de ces classes et méthodes paramètres du noyau représente un équilibre à trouver entre *modularité* souhaitée et *compacité* nécessaire pour que la consistence entre les différents composants reste facile à gérer [6].

[5] Une alternative serait de représenter une portion plus ou moins importante des caractéristiques des objets actifs d'Actalk (structure, activité...) en tant qu'objets actifs Actalk. Une telle solution réflexive a été développée à partir d'Actalk et nommée ReActalk par Sylvain Giroux [GIRO&91]. ReActalk augmente encore le pouvoir d'expression d'Actalk, mais au détriment d'une légère perte de compacité et d'efficacité. Ce problème général de fixer la (meilleure) *granularité de représentation/décomposition* sera abordé de manière plus pratique au §3.

[6] La possibilité de combinaison arbitraire de différents composants (*comportement*, *activité* et *adresse*) provenant de formalismes différents (ex : créer un objet actif suivant le modèle de *remplacement de com-*

3.2 Décomposition du noyau en classes

Le noyau de la version actuelle d'Actalk est décomposé en trois classes de base :

- la classe `ActiveObject` représente le *comportement interne* d'un objet actif, c'est-à-dire celui qui traite les messages et ainsi donne sa sémantique à l'objet actif. Le programmeur définira donc ses classes de comportements d'objets actifs, c'est-à-dire ses programmes concurrents, comme sous-classes de `ActiveObject` et non pas de `Object` comme en Smalltalk-80 standard. Notons que les constructions spécifiques éventuelles d'un langage de programmation donné seront également exprimées par des sous-classes abstraites de `ActiveObject`, ainsi par exemple la sous-classe `Abcl1Object` qui définit une construction d'attente explicite d'un certain type de message ("wait-for" de ABCL/1).

- la classe `Activity` représente l'*activité interne* d'un objet actif. Cette classe apporte l'autonomie (via l'allocation d'un processus) à chaque objet actif. Elle décrit également la manière dont les messages sont sélectionnés, séquencés, et activés. Ceci inclut donc en particulier différentes statégies possibles de synchronisation de l'activation (également appelée *acceptation*) des messages.

- la classe `Address` représente représente l'*adresse* d'un objet actif, c'est-à-dire l'identificateur et la boîte aux lettres associée où vont être envoyés les messages. Elle définit la manière dont la réception des messages sera interprétée, c'est-à-dire différents possibles types de communication [7].

3.2.1 Composantes d'un objet actif

Un objet actif en Actalk est donc formé de trois composants. Le programmeur crée un comportement comme instance d'une classe de comportements sous classe de la classe `ActiveObject` puis lui envoie le message `active` pour l'activer. Ceci entraine la création transparente des deux autres composants, activité et adresse, et le démarrage du processus associé à l'activité.

La fonctionnalité minimale d'un objet actif peut être résumée en suivant les étapes principales du cheminement d'une requête, depuis sa réception jusqu'à son traitement :

- l'adresse *reçoit* un message, ce qui déclenche la méthode paramètre `receiveMessage:` de `Address` qui rajoute le message dans la queue de messages en attente ;

- *indépendamment*, l'activité *sélectionne* le message suivant (méthode paramètre `nextMessage` de `Activity`),

- qu'elle accepte et *traite* (méthode paramètre `performMessage:` de `Activity`) en délégant son traitement effectif au comportement.

portement des acteurs [OOCP87, pages 37–53] *et* le modèle de communication d'ABCL/1 [OOCP87, pages 55–89]) pose le problème de leur inconsistence éventuelle. Pour protéger l'utilisateur contre de telles incompatibilités, la plateforme Actalk inclut un mécanisme simple de spécification de contraintes de compatibilité entre les classes composantes (via des méthodes virtuelles) et de leur vérification.

[7] Dans Actalk, c'est le receveur du message qui détermine la sémantique de l'envoi de message. Ceci s'explique par la manière dont sont implantés de nouveaux envois de messages en Smalltalk-80 (voir [BRIO89]). Ceci permet modularité et encapsulation puisque la sémantique de l'envoi reste locale et liée au receveur.

3.2.2 Avantages

De décomposer ainsi un objet actif en trois composantes permet d'avoir une plus grande modularité de modélisation, en particulier :

- de *séparer ("découpler")* la sémantique de la *transmission* des messages d'avec la sémantique de leur *traitement* ;
- de *séparer* un programme *utilisateur* d'un modèle donné d'*activité* [8] (ex : avec ou sans concurrence interne) ;
- de *séparer* un programme *utilisateur* de la spécification de sa *synchronisation* interne (contrôle de l'activation des messages) ;
- de permettre d'associer éventuellement plusieurs comportements à une même adresse. Ceci permet en particulier de modéliser immédiatement et de manière naturelle le *modèle de calcul des acteurs* [OOCP87, pages 37–53] qui permet l'activité simultanée de plusieurs comportements distincts pour traiter simultanément plusieurs messages.

3.2.3 Décomposition en méthodes paramètres : exemple de la classe Activity

sélecteur	*paramètre*	*valeur par défaut*	*exemples de redéfinitions*
`privateInitialize`	*initialisation*	rien	initialise des compteurs de synchronisation
`start`	*démarrage de l'activité*	démarre le processus créé par `createProcess`	installe un niveau d'activité supplémentaire (pour des messages express)
`createProcess`	*création du processus d'activité*	crée un processus tel que défini par `body`	crée une "poignée" (référence) sur le processus (utile pour le contrôle)
`body`	*nature de l'activité*	accepte un à un chaque message	accepte un seul message
`nextMessage`	*sélection du message suivant*	retourne le premier message de la queue de messages	retourne le premier message parmi les méthodes autorisées (transition d'états abstraits, voir au §4.1)
`acceptMessage:`	*acceptation/ activation d'un message*	`performMessage:`	crée un sous-processus pour traiter le message (plusieurs traités simultanément)
`performMessage:`	*traitement d'un message*	délègue le traitement au comportement	retourne la valeur au destinataire implicite de la réponse
`addressClass`	*classe d'adresse*	`Address`	`ImplicitReplyAddress` (pour retourner une réponse implicite)
`invocationClassFor:`	*classe d'invocation*	`Message`	invocation incluant l'émetteur du message

[8] Cette séparation n'existait pas dans la version précédente d'Actalk où les deux classes `ActiveObject` et `Activity` ne faisaient alors qu'une : la classe `ActorBehavior`.

3.3 Décomposition en niveaux

En plus de la décomposition du noyau en différentes classes, il existe également une décomposition du noyau en *niveaux*[9] de manière à bien séparer les *caractéristiques essentielles*, des *facilités supplémentaires* offertes, et enfin du *point d'entrée (public)* pour l'utilisateur.

Le premier niveau définit une version *minimale* des trois classes du noyau décrites au §3.2 et incluant les méthodes de base et les méthodes paramètres. Le deuxième niveau rajoute des facilités supplémentaires, en particulier le concept de *méthode d'événement générique*, décrit au §3.4. Enfin, le troisième niveau regroupe les trois classes du noyau, points d'entrée pour l'utilisateur destinées à être spécifiées ou sous-classées. Elles comprennent un certain nombre de fonctionnalités utilisateur (génération de méthodes de trace, vérification des contraintes de compatibilité...).

3.4 Méthodes d'événements génériques

Les *méthodes d'événements génériques* sont des méthodes virtuelles associées à des événements : *réception* d'un message, *acceptation*, et enfin *complétion* de son traitement. Ces méthodes génériques peuvent être utilisées par l'utilisateur, par exemple pour fournir des informations de type *trace*. Mais elles sont également utiles pour la paramétrisation de l'architecture puisqu'elles permettent de spécifier en particulier des pré et post-traitements. Enfin elles peuvent également servir à exprimer la gestion de certaines stratégies de synchronisation, par exemple pour mettre à jour des *registres/compteurs* des activations en cours (cet exemple est détaillé au §4.2). De manière à ne pas aboutir à des conflits entre ces différents niveaux d'utilisation des événements génériques, nous avons choisi de définir des méthodes propres à chacune de ces trois utilisations.

3.5 Modularité de la paramétrisation

De manière à paramétrer encore plus finement la plateforme, tout en n'augmentant pas le nombre de classes et de méthodes paramètres au niveau du noyau, nous utilisons une *décomposition modulaire et hiérarchique*. Ceci veut dire décomposer plus avant certaines des méthodes paramètres (et donc en introduire de nouvelles) au niveau de certaines sous-classes abstraites dans la hiérarchie.

Un premier exemple est la classe abstraite d'adresse `GenericSendAddress` qui décompose et ainsi paramétrise l'aiguillage de la réception d'un message selon son type et son mode. Un autre exemple est la classe abstraite d'activité `SelectiveAcceptActivity` qui paramétrise plus finement la sélection du prochain message (méthode paramètre `nextMessage`). Une nouvelle méthode paramètre est définie (`isAcceptableMessage:`) pour paramétrer les critères de sélection, par exemple pour évaluer des *gardes* de méthodes (voir au §4.2).

3.6 Exemples de modélisations

La plateforme Actalk comporte actuellement une trentaine de catégories comprenant plus de 100 classes. Outre le noyau et ses classes de librairies d'objets utilitaires (ainsi des classes prédéfinies de continuations), les librairies incluent :

[9]Cette décomposition a été introduite à l'origine par Loïc Lescaudron [LESC92].

- **langages** : modélisation et implantation des caractéristiques essentielles des langages de programmation suivants [10] : modèle de calcul des acteurs (remplacement de comportement), langages ABCL/1 (3 types et 2 modes d'envois de messages, acceptation explicite...), POOL (acceptation explicite, post actions...), ConcurrentEiffel, ConcurrentSmalltalk (suspension sur condition...)... [11] ;
- **exemples** : fonctions numériques, algorithmes non numériques (tris...), simulation... ;
- **synchronisation** : une attention particulière est apportée aux diverses stratégies de synchronisation, c'est-à-dire de contrôle de l'activation des messages, ainsi : par transition d'états, gardes, suspension/attente, compteurs de synchronisation... Différents exemples classiques de synchronisation (buffer borné, philosophes, lecteurs/écrivains, serveur d'imprimante...) sont dérivés (et ainsi comparés) dans ces différentes stratégies. De plus différentes variations/spécialisations (préservation de l'ordre, priorités, garantie de non famine...) sur ces exemples permettent d'illustrer et de comparer la mise en oeuvre de la spécialisation des spécifications de synchronisation à l'aide de l'héritage.

4 Exemple développé

Pour illustrer plus en pratique les capacités de modélisation de la plateforme, nous présentons ici brièvement un exemple de combinaison de stratégies de synchronisation.

4.1 États abstraits

Une première approche du contrôle de la sélection des messages candidats à l'activation repose sur des états abstraits qui décrivent un ensemble de méthodes *autorisées* (ou *candidates*). Cette stratégie s'applique parfaitement à la synchronisation liée à l'*état* de l'objet. Ainsi par exemple un buffer de taille bornée dans l'état abstrait `full` aura son ensemble de méthodes autorisées réduit à {`get`}, la méthode `put:` étant momentanément non disponible.

Dans la définition ci-dessous de la classe `AbstractStatesActivity`, les méthodes font référence aux deux méthodes virtuelles de spécification de l'état abstrait initial (méthode `initialAbstractState`) et de la transition d'états (méthode `nextAbstractStateAfter:`). Ces deux méthodes virtuelles ne sont pas définies au niveau de cette classe abstraite, mais le seront au niveau d'une sous-classe spécifique (exemple du buffer borné au §4.4).

```
ImplicitReplyActivity subclass: #AbstractStatesActivity
   instanceVariableNames: 'enabledSelectors '

nextMessage
   "retourne le premier message autorise."
   ^self mailBox firstMessageWithCondition:
        [:message | enabledSelectors includes: message selector]
```

[10] Voir [OOCP87] pour une description de ces différents langages. Signalons également qu'un projet de DESS de l'Université de Nantes, encadré par Jean Bézivin et Olivier Roux, a également implanté les langages CSP et OCCAM à partir d'une version précédente de Actalk. Leur implémentation initiale de POOL-T et des compteurs de synchronisation a également servi de base pour leur intégration dans la version actuelle.

[11] Signalons également une extension (classes `ImplicitReplyActivity/Address`) implantant une transmission de message avec valeur retournée et synchronisation implicites (de type *future*). Cette extension permet ainsi une bonne réutilisation du code standard Smalltalk-80.

```
privateInitialize
   super privateInitialize.
   "initialisation de l'ensemble initial de methodes autorisees."
   enabledSelectors := self perform: self initialAbstractState

"methode d'evenement generique associee a l'evenement de completion d'un message."
kernelEventComplete: aMessage
   super kernelEventComplete: aMessage.
   "transition d'etats : calcul du nouvel ensemble de methodes autorisees."
   enabledSelectors := self perform: (self nextAbstractStateAfter: aMessage selector)
```

4.2 Gardes et compteurs de synchronisation

Cette stratégie quoique puissante ne peut toutefois rendre compte des exclusions éventuelles entre les activations de méthodes si l'on permet maintenant le traitement simultané de plusieurs messages. Pour assurer la consistence de l'état du buffer précédent, il est nécessaire d'empêcher l'exécution simultanée de plusieurs `put:`, de même pour `get`. Par contre, on peut permettre une exécution simultanée éventuelle d'un `put:` *et* d'un `get`, du fait qu'ils accèdent à des zones mémoires distinctes. La stratégie des *gardes* faisant référence à des *compteurs de synchronisation* (mémorisant le nombre d'invocations reçues, acceptées, et complétées pour un message/méthode donné) permet de gérer de manière précise de telles *conditions d'activation*. L'implantation (classe `CountersActivity`) principale est décrite ci-dessous.

```
SynchroConcurrentActivity subclass: #CountersActivity
   instanceVariableNames: 'receivedDictionary acceptedDictionary completedDictionary '

privateInitialize
   super privateInitialize.
   "creation des trois dictionnaires gerant les compteurs pour chaque methode."
   self makeSynchroCounterDictionariesOnSelectors: oself class allScriptSelectors

"test de l'acceptabilite d'un message : evaluation de la garde."
isSynchroAcceptableMessage: aMessage
   ^self satisfyGuardSelector: aMessage selector arguments: aMessage arguments

satisfyGuardSelector: selector arguments: argumentsArray
   ^self perform: (self findGuardSelector: selector) withArguments: argumentsArray

"la garde associee a une methode est aussi une methode, prefixee par le symbole guardOF."
findGuardSelector: selector
   ^('guardOF' , selector) asSymbol

"les evenements incrementent les compteurs de synchronization associes."
synchroEventAccept: aMessage
   super synchroEventAccept: aMessage.
   self incrAccepted: aMessage selector

synchroEventComplete: aMessage
   super synchroEventComplete: aMessage.
   self incrCompleted: aMessage selector

synchroEventReceive: aMessage
   super synchroEventReceive: aMessage.
   self incrReceived: aMessage selector
```

4.3 Modèle mixte

Les deux stratégies précédentes sont en fait complémentaires. Nous pouvons donc les combiner, les états abstraits spécifiant la *synchronisation d'états*, et les gardes/compteurs de synchronisation spécifiant la *synchronisation d'activation*. Actalk permet de l'exprimer aisément en combinant [12] (de manière atomique) les deux tests d'acceptation de message. Nous nommons la classe résultante `ASCActivity`, pour "*abstract states* and *synchronization counters*".

```
CountersActivity subclass: #ASCActivity
   instanceVariableNames: 'enabledSelectors '

isSynchroAcceptableMessage: aMessage
   "un message doit satisfaire les deux conditions testees (atomicite garantie par l'appel)."
   ^(enabledSelectors includes: aMessage selector)
      and: [super isSynchroAcceptableMessage: aMessage]

privateInitialize
   super privateInitialize.
   enabledSelectors := self perform: self initialAbstractState

synchroEventComplete: aMessage
   super synchroEventComplete: aMessage.
   enabledSelectors := self perform: (self nextAbstractStateAfter: aMessage selector)
```

Cette stratégie mixte de synchronisation, très modulaire, et à l'origine proposée par [THOM92], a ainsi été aisément modélisée et implantée en Actalk. Plus encore, en suivant le même principe de *combinaison* de stratégies, nous avons également pu aboutir de manière naturelle à une *nouvelle* stratégie encore plus expressive (détaillée dans un papier à paraître).

4.4 L'exemple du buffer borné : classe d'activité/synchronization

```
ASCActivity subclass: #ASCBoundedBufferActivity
   instanceVariableNames: ''

"etats abstraits: synchronisation selon l'etat."
empty                  full                  partial
   ^#(put:)               ^#(get)               ^(self empty) + (self full)

"etat abstrait initial a la creation de l'objet."
initialAbstractState
   ^#empty

"transition d'etat (oself = le comportement de l'objet actif et pas l'activite elle-meme)."
nextAbstractStateAfter: selector
   ^oself isEmpty ifTrue: [#empty]
                  ifFalse: [oself isFull ifTrue: [#full]
                                         ifFalse: [#partial]]

"gardes: conditions d'activation."
guardOFget                                   guardOFput: item
   ^(self current: #get) = 0                    ^(self current: #put:) = 0
```

[12]Comme Smalltalk-80 n'offre pas d'héritage multiple, il nous faut définir la classe résultante comme sous-classe *unique* d'une des deux classes précédentes (en l'occurence on choisit la classe `CountersActivity`, car légèrement plus complexe). Il nous faut donc redéfinir la variable d'instance `enabledSelectors`, et combiner *"à la main"* les méthodes `privateInitialize` et `synchroEventComplete:`.

5 Evaluation et comparaison

5.1 Evaluation et critique

Nous espérons avoir pu montrer à travers cet exemple les capacités de modélisation de la plateforme Actalk. Nous avons pu dégager un noyau minimal et paramétré, puis affiner son expressivité à la suite de l'expérience de nombreuses modélisations. Cependant on ne peut prétendre dégager *à priori* tous les paramètres nécessaires à une modélisation de n'importe quel système. Mais la méthodologie de décomposition modulaire, plus bas dans la hiérarchie (présentée au §3.5), présente l'avantage de ne pas avoir alors à redéfinir le noyau. Signalons enfin qu'Actalk n'est pas une plateforme de spécification formelle, mais privilégie au contraire la modélisation et l'expérimentation prototypes.

5.2 Autres travaux

Actalk a d'ores et déjà fait l'objet d'utilisations pédagogiques (comme support de cours et tutoriaux, et également en modélisation *active* en particulier par le DESS de l'Université de Nantes qui a développé de nombreuses extensions) ; d'expérimentation (en utilisant Actalk comme fondation pour étudier et développer des stratégies de : activation de démons, réflexivité, traitement d'exceptions, résolution de contraintes...) ; et enfin d'applications. Actalk a notamment déjà été utilisé comme fondation pour développer plusieurs plateformes multi-agents de granularités variées (agents *cognitifs* [BOUR&91] ou *réactifs* [DROG&92]) dans l'équipe de Jacques Ferber au LAFORIA. D'autres projets en cours dans d'autres équipes (langage naturel, séquencement de gènes) utilisent également Actalk comme outil.

5.3 Projets connexes et influences

Plusieurs projets ont également choisi Smalltalk-80 comme fondation d'une plateforme de modélisation et de classification. Un des précurseurs est la plateforme Simtalk, dédiée à la modélisation et la classification de modèles de simulation [BEZI87]. Comparé à Simtalk, Actalk restreint son domaine aux langages de programmation *intégrant* les objets et les activités, et paramétrise son noyau minimal pour assurer expressivité et modularité. Un séquenceur générique développé par Loïc Lescaudron dans le cadre du projet Actalk [LESC92] utilise également Smalltalk comme fondation pour paramétrer et classifier différentes stratégies de séquencement. La plateforme Prototalk [DONY&92] est quand à elle dédiée à la modélisation et la classification de langages de prototypes. Enfin le langage de programmation Concurrent-Smalltalk [OOCP87, pages 129–158], et surtout son *évolution*, ont également été une référence ainsi qu'une influence.

Conclusion

Dans cet article nous avons présenté la conception et l'architecture de la plateforme Actalk qui permet de représenter, comparer et explorer divers types de langage de programmation concurrente par objets. Nous avons montré à travers un exemple développé de combinaison de stratégies de synchronisation à la fois la concision et l'expressivité de cette plateforme. Nous souhaitons qu'à terme cette plateforme devienne une forme de *bibliothèque partagée* de divers formalismes de programmation concurrente à objets.

La dernière version (documentée) de la plateforme Actalk est disponible via ftp anonyme : "ftp camille.is.s.u-tokyo.ac.jp; cd pub/actalk" *ou* "ftp ftp.ibp.fr; cd /ibp/softs/litp/actalk" ; et via WWW/Mosaic : "http://web.yl.is.s.u-tokyo.ac.jp/members/briot/actalk/actalk.html".

Remerciements

La place nous manque ici pour détailler l'ensemble des personnes ayant contribué au développement d'Actalk. Nous tenons tout de même à remercier ici particulièrement en premier lieu Loïc Lescaudron, et également Jean Bézivin, Malik Bouabsa, Alexis Drogoul, Jacques Ferber, Sylvain Giroux, Jean-François Perrot, Olivier Roux, et Akinori Yonezawa pour leurs contributions, commentaires, et soutiens respectifs.

References

[BEZI87] J. Bézivin, Some Experiments in Object-Oriented Simulation, *OOPSLA'87*, Special Issue of SIGPLAN Notices, Vol. 22, No 12, pages 394–405, Décembre 1987.

[BOUR&91] T. Bouron, J. Ferber et F. Samuel, MAGES: a Multi-Agent Testbed for Heterogeneous Agents, dans *Decentralized Artificial Intelligence*, Vol. II, édité par Y. Demazeau et J.-P. Muller, North-Holland, 1991.

[BRIO89] J.-P. Briot, Actalk: une Plateforme de Modélisation de Langages d'Acteurs en Smalltalk-80, *7ème Congrès Afcet de Reconnaissance des Formes et Intelligence Artificielle (RFIA'89)*, Tome I, pages 147–161, Paris, Novembre-Décembre 1989.

[DONY&92] C. Dony, J. Malenfant, et P. Cointe, Prototype-Based Languages: From a Taxonomy to Constructive Proposals and their Validation, *OOPSLA'92*, Special Issue of SIGPLAN Notices, Vol. 22, No 10, pages 201–217, Octobre 1992.

[DROG&92] A. Drogoul, J. Ferber, B. Corbara et D. Fresneau, A Behavioral Simulation Model for the Study of Emergent Social Structures, dans *Towards a Practice of Autonomous Systems*, édité par P. Bourgine et F. Varela, MIT Press, pages 161–170, 1992.

[GIRO&91] S. Giroux et A. Senteni, ReActalk, a Reflective Version of Actalk, *OOPSLA'91 Workshop on Reflection and Metalevel Architectures*, Xerox PARC Technical Report, Octobre 1991.

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

[LESC92] L. Lescaudron, Prototypage d'Environnements de Programmation pour les Langages à Objets Concurrents : une Réalisation en Smalltalk-80 pour Actalk, *Thèse d'Université*, LITP, Université Paris VI - CNRS, Mai 1992.

[MATS&93] S. Matsuoka et A. Yonezawa, Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, dans *Research Directions in Concurrent Object-Oriented Programming*, édité par G. Agha, P. Wegner et A. Yonezawa, Mit Press, pages 107–150, 1993.

[OOCP87] Object-Oriented Concurrent Programming, edité par A. Yonezawa et M. Tokoro, *Computer Systems Series*, MIT Press, 1987.

[THOM92] L. Thomas, Extensibility and Reuse of Object-Oriented Synchronization Components, *PARLE92*, LNCS, Vol. 605, Springer-Verlag, pages 261–275, Juin 1992.

Contents