Actalk: une Plateforme de Modélisation de Langages d'Acteurs en Smalltalk-80
Actalk: a Platform to Model Actor Languages in Smalltalk-80

Jean-Pierre BRIOT

Equipe Mixte LITP - Rank Xerox France,
Université Pierre et Marie Curie,
4 place Jussieu, 75005 Paris, France
briot@litp.ibp.fr

RESUME - Cet article décrit une plateforme de modélisation, classification et expérimentation de langages d'acteurs, premiers pas vers les systèmes multi-agents. Ce système, nommé *Actalk*, qui signifie *act*eurs en Small*talk*-80[1] étend les objets passifs et séquentiels activés par envois de messages synchrones vers des acteurs actifs et parallèles communiquant par messages asynchrones. Le noyau d'Actalk est aisément étendu pour simuler les principaux modèles de calcul et langages d'acteurs, notamment le modèle de calcul de Agha, et le langage Abcl/1 de Yonezawa, ainsi implicitement classifiés par héritage. L'environnement standard de Smalltalk-80 est spécialisé pour visualiser et contrôler dynamiquement l'exécution des acteurs.

Mots clés : objet, Smalltalk-80, parallélisme, acteur, agent, plateforme, spécification, expérimentation, modularité, combinaison, Act*, Abcl/1, classification, environnement.

ABSTRACT - This paper describes a platform to model, classify and experiment with *actor* languages, foundations for multi-agents systems. This system, named *Actalk*, which stands for *act*ors in Small*talk*-80, extends passive and serial objects activated through synchronous message passing towards active and concurrent actors communicating through asynchronous messages. The kernel of Actalk is easily extended to simulate the main actor computation models and languages, Agha's actor computation model, and Yonezawa's Abcl/1 language, thus implicitly classified with inheritance. Smalltalk-80 standard programming environment is specialized in order to dynamically visualize and control execution of actors.

Keywords : object, Smalltalk-80, parallelism, actor, agent, platform, specification, experiment, modularity, combination, Act*, Abcl/1, classification, environment.

[1]Smalltalk-80 est une marque déposée de ParcPlace Systems.
Cette recherche a bénéficié du soutien du GRECO Programmation.

1 Introduction

Cet article décrit la modélisation de langages d'*acteurs*. Les langages d'acteurs sont basés sur la notion d'objets actifs et autonomes qui interagissent selon des protocoles de communication fondés sur l'envoi de message. Le concept d'*acteur* est sans doute 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 la coopération d'un ensemble d'agents, véritable communauté d'experts interagissants.

La théorie des acteurs a été inventée par Carl Hewitt [Hewitt 77] comme un formalisme unificateur pour représenter connaissances et interactions. Plasma fut le premier langage conçu selon ces idées, et a été suivi par Act1, Act2, puis le représentant de la nouvelle génération: Pract/Acore, basé sur le plus récent modèle de calcul défini par Gul Agha [Agha 86]. En plus de cette famille de langages conçus autour de Carl Hewitt au laboratoire d'IA du MIT, le modèle des acteurs a donné lieu à nombre propositions. La constante de tous ces langages d'acteurs est la notion d'objets actifs et coopérants [OOCP 87]. Un exemple est le langage Abcl/1 proposé par Akinori Yonezawa [Yonezawa et al. 86].

Nous avons 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. De plus certains de ces prototypes sont par trop liés à des machines spécifiques (par exemple une machine Lisp) ou sont trop peu diffusés.

Du point de vue du concepteur et de l'implémenteur, l'expérience (y compris la nôtre) montre que le manque d'outils de conception et d'expérimentation est flagrant. Trop d'implémentations prototypes sont souvent sacrifiées avant d'arriver à des concepts stabilisés. Comme une implémentation modulaire n'est éventuellement atteinte qu'après stabilisation du langage, le concepteur reprend généralement son implémentation de zéro après chaque essai avorté. La réutilisabilité prônée par le génie logiciel n'est alors pas mise 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 par 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 nous a récemment conduit à concevoir une plateforme de modélisation et d'expérimentation de langages d'acteurs basée sur la programmation par objets. Nous allons résumer les étapes de sa conception.

2 Buts et Choix de Conception

Détaillons maintenant les buts et les choix qui nous ont conduit à concevoir notre plateforme:

1. *uniformité et modularité*
   Nous voulions unifier les différents langages d'acteurs dans un environnement unique et pouvoir les modéliser couche par couche. Dans ce but nous avons choisi la méthodologie

de la programmation par objets. Nous introduisons les acteurs dans le monde des objets en définissant un sous-monde d'objets actifs, sans pour autant changer la nature des objets standard.

2. *minimalité et extensibilité*
   Nous voulions définir un noyau minimal du système pour exprimer la sémantique générale des langages d'acteurs puis l'étendre de manière à simuler les langages d'acteurs actuels ainsi qu'en concevoir de nouveaux. Pour cette raison nous avons retenu une architecture minimale dans l'esprit d'ObjVlisp [Briot et Cointe 87, Cointe 87], c'est-à-dire dotée d'un noyau minimal étendu de manière uniforme. L'héritage nous permet de classifier les différents langages d'acteurs à partir du noyau commun.

3. *un environnement intégré*
   Nous ne souhaitions pas restreindre notre plateforme à un seul outil de conception de langages, mais au contraire proposer un environnement complet d'expérimentation. Pour cette raison nous avons choisi Smalltalk-80 comme environnement sur lequel greffer notre plateforme. Smalltalk-80 est actuellement le système de programmation par objets le plus complet et doté d'un environnement de programmation sophistiqué. Du fait de l'intégration du noyau d'acteurs dans le système Smalltalk-80, le concepteur et l'expérimentateur peuvent réutiliser l'environnement standard de Smalltalk-80. Un projet plus ambitieux et actuellement en cours de développement étend l'environnement standard vers un environnement spécifique aux acteurs pour prendre en compte le parallélisme. Comme il n'est ici question que de spécification et d'expérimentation du parallélisme, nous emploierons le terme *concurrence*, pour signifier tout parallélisme potentiel [Agha 86].

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. Nous nommons le système résultant *Actalk* (ce qui veut dire *act*eurs en Small*talk*-80), en tant que fils spirituel d'*ObjVlisp*. Actalk est une plateforme de classification, conception et d'expérimentation de langages d'acteurs dans un environnement unifié. Plutôt que de changer le système Smalltalk-80 sur lequel il se greffe, Actalk y est parfaitement intégré. De ce fait, Actalk offre également une excellente plateforme d'étude et de comparaison entre objets et acteurs. Nout utilisons maintenant couramment Actalk pour introduire et enseigner la programmation par acteurs.

3 Des Objets aux Acteurs

Nous allons maintenant brièvement introduire le modèle des acteurs, et de quelle manière nous l'incluons dans Smalltalk-80. Nous nous intéresserons particulièrement ici au *pourquoi* et au *comment* étendre un système d'objets séquentiels tel que Smalltalk-80 vers la concurrence.

En Smalltalk-80 (et de manière générale en programmation par objets) les objets sont activés par envois de messages. Les objets sont *passifs* parce qu'ils subissent la requête d'activation de l'envoyeur du message. Ils n'ont pas d'activité propre. Envoyer un message représente le transfert d'activité d'un objet à un autre, avant le retour à l'appelant dont l'activité a été temporairement interrompue.

Pour simuler le parallélisme, Smalltalk-80 introduit de multiples activités, appelées *processus*. Ainsi plusieurs messages peuvent activer simultanément plusieurs objets. Mais si un même objet reçoit simultanément deux messages, les deux activations peuvent se disputer le contrôle de l'objet. Si les deux méthodes changent simultanément la valeur d'une variable de l'objet, l'état résultant peut être inconsistant. Le problème est qu'un objet reste passif, et par conséquent doit être protégé contre des activations simultanées.

La solution la plus simple et la plus pragmatique est d'empêcher plus d'une activation à la fois pour un même objet, elle est appelée *exclusion mutuelle*. Le problème est que l'utilisateur est conduit à décrire simultanément la modélisation en terme d'objets et la synchronisation des accès aux données contenues par ces objets. Ces deux niveaux de programmation sont orthogonaux et correspondent à la différence de nature entre objet et processus.

De manière à libérer le programmeur de cette tâche supplémentaire, une synchronisation automatique est proposée en Smalltalk-80 dans [Pascoe 86]. Cette solution consiste à associer à un objet (en l'*encapsulant*) un sémaphore assurant systématiquement l'exclusion mutuelle.

Une autre solution plus définitive garantit cette hypothèse de mono-activité en associant une activité propre à l'objet. Cela fait évoluer le modèle des objets passifs vers des objets actifs. Un objet va maintenant détenir sa propre activité interne, et ne dépendra plus d'une activation externe par envoi de message. Un objet devient *actif* et *autonome*. Il peut maintenant décider par lui même *quand* et *comment* interpréter un message et dispose de la puissance de calcul nécessaire à son traitement.

Puisque le receveur d'un message possède sa propre activité pour interpréter un message, il n'est plus indispensable pour l'envoyeur de suspendre son activité pour la transférer au receveur. Si aucune réponse n'est attendue, l'envoyeur peut immédiatement continuer son activité aussitôt après l'envoi du message. La transmission de message devient unidirectionnelle et asynchrone. Donc, envoyer un message se réduit à sa délivrance (dans un temps fini) au receveur. Le receveur peut interpréter un message à tout moment après sa livraison, ce qui impose aux messages d'être stockés dans une *boîte aux lettres* associée au receveur, avant d'être interprétés.

Le modèle initial des objets activés par des envois de messages synchrones a donc été étendu vers un nouveau modèle d'objets actifs et concurrents qui communiquent de manière asynchrone. Nous appellerons ce nouveau type d'objets des *acteurs*. Un acteur est composé d'une boîte aux lettres, où les messages qui lui sont destinés sont stockés selon leur ordre d'arrivée, ainsi que de l'objet actif qui les interprètera, que nous appelerons son *comportement*.

4 Implémentation du Noyau d'Actalk

4.1 Définition du Noyau Minimal d'Actalk

Nous allons maintenant définir le noyau d'Actalk qui soit minimal et le plus général. Deux classes le composent: la classe `Actor` définit la sémantique des acteurs, et la classe `ActorBehavior` définit la sémantique des comportements d'acteurs.

La classe `Actor` représente un acteur à travers ses deux composants: la boîte aux lettres ou file d'attente qui stockera les messages, et le comportement qui les interprètera. L'implémentation de la transmission de message asynchrone nous a conduit à introduire une troisième classe, présentée au paragraphe 4.4, de manière à clairement séparer la sémantique du noyau d'Actalk des techniques employées pour son implémentation.

Figure 1: Implémentation d'un acteur en Actalk.

Définir la classe `ActorBehavior` est moins neutre. Nous voulons en effet exprimer le plus généralement possible la stratégie d'interprétation des messages par un comportement. Le modèle de calcul des acteurs, défini par Gul Agha, déclare qu'un comportement interprète un seul message et doit spécifier quel autre comportement traitera le message suivant. Le concept de *remplacement de comportement* généralise le changement d'état (affectation des variables d'instance) traditionnel. Ce modèle de calcul est minimal et assez général pour exprimer tout autre type de modèle de calcul [Agha 86]. Cependant nous ne l'avons pas choisi comme candidat pour notre noyau car nous voulons une implémentation minimale et une intégration maximale avec Smalltalk-80. Il nous faut donc conserver l'affectation des variables, pour que le programmeur puisse exprimer le comportement des acteurs tout comme il définit le comportement d'objets standard. Parce qu'il peut y avoir des changements d'état durant l'interprétation des messages, leur traitement doit être *sérialisé*, c'est-à-dire interprétés l'un après l'autre.

Nous sommes convaincus que cette définition de l'activité d'un comportement d'acteur est minimale *et* suffisamment générale. Elle sera affinée ou redéfinie en étendant le noyau d'Actalk. Ainsi le modèle de calcul des acteurs de Gul Agha est ensuite immédiatement défini comme une légère extension du noyau (une méthode est redéfinie et une autre est rajoutée) au paragraphe 7.

La figure 1 illustre l'implémentation d'un acteur.

4.2 La classe Actor

La classe `Actor` implémente les acteurs comme des *encapsulateurs* construits autour d'objets Smalltalk-80 standard. Elle est sous-classe de la classe `MinimalObject` qui est utilisée pour l'implémentation de la transmission de message asynchrone, et sera décrite au paragraphe 4.4. La classe `Actor` spécifie deux variables d'instance:

`mailbox` référence la file d'attente (queue) de messages, une instance de la classe standard `SharedQueue`,

`behavior` référence le comportement qui interprète les messages, une instance d'une sous-classe de la classe `ActorBehavior` décrite au prochain paragraphe.

```
MinimalObject subclass: #Actor
   instanceVariableNames: 'mailBox behavior '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Kernel'!
```

```
!Actor methodsFor: 'initialization'!

initialize
   mailBox := SharedQueue new!

initializeBehavior: aBehavior
   behavior := aBehavior.
   behavior initializeAself: self! !

!Actor methodsFor: 'access to instance variables'!

mailBox
   ^mailBox! !

!Actor methodsFor: 'message passing'!

asynchronousSend: aMessage
   mailBox nextPut: aMessage! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

!Actor class methodsFor: 'instance creation and initialization'!

behavior: aBehavior
   ^self new initializeBehavior: aBehavior!

new
   ^super new initialize! !
```

La méthode de classe `behavior:` crée un acteur et initialise son comportement (la méthode d'instance `initializeBehavior:`). La méthode de classe `new` est redéfinie pour initialiser la boîte aux lettres de l'acteur (la méthode d'instance `initialize`). Nous avons choisi de décomposer l'initialisation d'un acteur en ces deux protocoles distincts pour une meilleure modularité et réutilisabilité. Cela s'avère utile quand nous étendons cette implémentation minimale par exemple pour l'optimiser ou la protéger (mais cela ne sera pas illustré dans cet article). Enfin la méthode d'instance `asynchronousSend:` implémente l'envoi de message asynchrone à un acteur en rajoutant le message en queue de sa boîte aux lettres.

4.3 La classe ActorBehavior

La classe `ActorBehavior` implémente le comportement général d'un acteur. Les utilisateurs décriront des comportements d'acteurs en définissant des classes de comportements, sous-classes de `ActorBehavior`. La classe `ActorBehavior` définit une variable d'instance: `aself`, qui référence l'acteur dont le comportement est en cours d'activité. `aself` permet à un acteur de s'envoyer des messages récursifs (et asynchrones) à lui-même. La différence avec la pseudo-variable standard de Smalltalk-80, `self`, sera abordée au paragraphe 4.4.

Un processus sera créé en même temps que l'acteur pour implémenter l'activité et l'autonomie de son comportement. Ce processus est indépendant (*background*) et infini. En accord avec la définition du noyau d'Actalk, le comportement aura pour activité immuable d'extraire le prochain message de la boîte aux lettres (c'est-à-dire le premier dans la file d'attente), de l'interpréter, puis de recommencer.

```
Object subclass: #ActorBehavior
   instanceVariableNames: 'aself '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Kernel'!

!ActorBehavior methodsFor: 'initialization'!

initializeAself: anActor
   aself := anActor.
   self setProcess!

setProcess
   [[true] whileTrue: [self acceptNextMessage]] fork! !

!ActorBehavior methodsFor: 'message acceptance'!

acceptNextMessage
   self acceptMessage: aself mailBox next!

acceptMessage: aMessage
   self performMessage: aMessage! !

!ActorBehavior methodsFor: 'actor creation'!

actor
   ^Actor behavior: self! !
```

La méthode d'instance `initializeAself:` initialise l'auto-référence (`aself`) de l'acteur et lance l'activité de son comportement (la méthode `setProcess`). Les méthodes `acceptNextMessage` et `acceptMessage:` acceptent et interprètent le prochain message dans la boîte aux lettres. Tant que la boîte aux lettres est vide, le processus est suspendu par le sémaphore qui synchronise la lecture de la queue [Goldberg et Robson 83, page 262]. La méthode `actor` crée un acteur à partir d'un objet passif choisi comme comportement. C'est la seule méthode du noyau d'Actalk que l'utilisateur doit connaître, c'est-à-dire l'interface utilisateur, comme illustré dans le paragraphe 5.1.

Notons que le noyau d'Actalk se réduit à 2 classes et 11 simples méthodes. Le nombre de méthodes pourrait être encore plus réduit, mais résulte d'une décomposition modulaire. Dans le même but, nous définissons la méthode `performMessage:` dans la classe `Object`:

```
!Object methodsFor: 'message passing'!

performMessage: aMessage
   ^self perform: aMessage selector withArguments: aMessage arguments! !
```

4.4 Transparence de la Transmission de Message Asynchrone

4.4.1 Localité

Nous souhaitons maintenant rendre ce nouveau type de transmission de message entre acteurs (la méthode `asynchronousSend:`) compatible avec la syntaxe standard de Smalltalk-80. Nous utilisons une technique déja introduite en Smalltalk-80 pour changer la syntaxe (interpréter des sélecteurs composés pour l'héritage multiple) ou la sémantique (encapsuler des objects [Pascoe 86]) de la transmission de message en redéfinissant le traitement d'erreur. Un message est envoyé à un objet qui, volontairement, ne reconnait pas ce message. Cette erreur est trappée par redéfinition de la méthode standard d'erreur "message inconnu" (`doesNotUnderstand:`) qui suit alors une stratégie spécifique. La redéfinition locale de la méthode `doesNotUnderstand:` garantit la localité des modifications. Envoyer un message à un acteur entrainera une transmission asynchrone alors qu'envoyer le même message à un objet Smalltalk-80 standard restera une activation synchrone. Notre implémentation redéfinit donc cette méthode dans la classe `Actor`:

```
!Actor methodsFor: 'message passing'!

doesNotUnderstand: aMessage
   self asynchronousSend: aMessage! !
```

Pour spécifier le comportement d'un acteur qui s'envoie un message à lui-même, le programmeur peut choisir entre deux types de récursion, matérialisées par les deux *pseudo-variables* `aself` et `self`. Un envoi à `aself` entraine une transmission Actalk asynchrone. Le message est déposé dans la boîte aux lettres de l'acteur, et son comportement l'interprètera dès que possible. Un envoi à `self` entraine une activation synchrone Smalltalk-80 standard. Le message sera interprété immédiatement de manière interne par le comportement.

4.4.2 Implémentation

Notre utilisation de cette technique de redéfinition d'erreur fait l'hypothèse qu'un acteur ne reconnait pas le sélecteur du message qu'il reçoit parce que nous *voulons* déclencher une erreur, pour pouvoir changer la sémantique de l'envoi de message. Or les méthodes standard définies dans la classe `Object` sont héritées par toute classe, et donc en particulier par la classe `Actor`. Par conséquent l'envoi d'un message standard ne provoquera pas l'erreur attendue, et sera interprété directement (comme le sont les messages "système" définis dans la classe `Actor`).

Différentes stratégies d'implémentation ont été proposées, notamment par [Pascoe 86], pour garantir cette hypothèse de non reconnaissance de sélecteur de message. La solution simplifiée que nous proposons consiste à définir une nouvelle racine de l'héritage autre que la classe `Object`, pour la court-circuiter. Le dictionnaire de méthodes de cette nouvelle classe ne doit pas être vide car un ensemble minimal de méthodes système est nécessaire pour traiter les erreurs, imprimer, comparer, et inspecter ses instances. Sinon l'interprète et l'environnement de Smalltalk-80 ne serait plus capable de manipuler correctement de tels objets.

Malheureusement l'environnement Smalltalk-80 standard ne permet pas de définir une classe sans sur-classe. L'astuce d'implémentation consiste en une première définition de cette nouvelle classe, nommée `MinimalObject`, comme sous-classe de `Object`. Puis elle est automatiquement réajustée (activation de la méthode `initialize`) en deux temps: son lien d'héritage est détruit (`superclass := nil`), puis un ensemble minimal de méthodes système appartenant à `Object` est recopié (`recompile:from:`) dans son dictionnaire de méthodes.

```
Object subclass: #MinimalObject
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Kernel-Encapsulator'!

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

!MinimalObject class methodsFor: 'initialization'!

initialize
   superclass := nil.
   #(doesNotUnderstand: error:   ~~ isNil   = ==   printString printOn: class
     inspect basicInspect basicAt: basicSize instVarAt: instVarAt:put:)
         do: [:selector | self recompile: selector from: Object]! !

MinimalObject initialize!
```

5 L'Exemple du Compteur

Présentons maintenant un des exemples les plus simples et représentatifs de la programmation par objets: le compteur. Nous allons ainsi montrer la parfaite intégration des acteurs Actalk dans le langage Smalltalk-80. La classe `Compteur` décrit le comportement d'un acteur compteur, c'est-à-dire un acteur qui se comporte comme un compteur.

5.1 Définition des Compteurs

La classe `Compteur` définit deux méthodes d'instance pour remettre à zéro (`raz`), et incrémenter (`incr`) la variable d'instance `contenu` d'un compteur. Selon la terminologie des acteurs, ces deux méthodes constituent le *script* d'un acteur.

```
ActorBehavior subclass: #Compteur
   instanceVariableNames: 'contenu '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Examples'!

!Compteur methodsFor: 'script'!

incr
   contenu := contenu + 1!

raz
   contenu := 0! !
```

Nous avons défini cette classe de comportements d'acteurs compteurs exactement comme d'habitude, excepté que la classe `Compteur` doit être définie comme sous-classe de la classe `ActorBehavior`. Pour créer un acteur compteur, il suffit de créer comme d'habitude une instance de `Compteur` et de postfixer cet objet par le sélecteur `actor`. L'on peut alors envoyer des messages à cet acteur nouvellement créé. L'envoi de message sera implicitement asynchrone.

```
| unCompteur |
unCompteur := Compteur new actor.
unCompteur raz; incr; incr
```

Remarquons que l'envoi du message `actor` est la seule marque spécifique de la création d'acteurs. La définition et l'envoi de messages sont transparents pour le langage Smalltalk-80 dans lequel les acteurs Actalk sont intégrés. Nous verrons au prochain paragraphe que l'abandon du retour implicite en valeur du message entrainera par contre un nouveau style de programmation.

5.2 Le Concept de Destinataire: L'Exemple du Scribe

Supposons que nous souhaitions consulter le contenu du compteur (par exemple pour l'afficher). Du fait de la nature maintenant asynchrone de la transmission de message, la valeur de l'expression d'envoi ne correspond plus à la valeur retournée comme en Smalltalk-80 standard. La valeur d'une transmission asynchrone n'est pas significative. En pratique, elle est égale au receveur du message, du fait de la convention de Smalltalk-80 sur la valeur retournée par défaut.

Pour récupérer la valeur du message, nous allons simuler la transmission standard bidirectionnelle à l'aide d'une seconde transmission unidirectionnelle pour la réponse, en spécifiant lors de l'envoi à quel acteur doit être retourné le résultat. Nous appellerons un tel acteur un *destinataire* (ou *reply destination* [Yonezawa et al. 86]). Les destinataires sont utilisés en particulier pour implémenter les *continuations*, qui sont un concept de base de la programmation par acteurs [Hewitt 77]. (Ce concept ne sera pas explicité dans cet article.)

Ains, pour consulter notre compteur, nous devons lui spécifier un destinataire en envoyant la requête. Nous définissons donc une nouvelle méthode de consultation, dotée d'un paramètre supplémentaire:

```
!Compteur methodsFor: 'script'!

consulteEtRepondA: destinataire
   destinataire reply: contenu! !
```

Nous faisons l'hypothèse que tout acteur spécifié comme destinataire reconnait le sélecteur `reply:`, qui est notre convention pour retourner une valeur. Il existe un tel acteur destinataire prédéfini en Actalk, référencé par la variable globale `Print`, qui affiche les valeurs dans la fenêtre spécialisée de Smalltalk-80 (appelée le `System Transcript`). Le comportement de ce scribe est défini par la classe `Printer`:

```
ActorBehavior subclass: #Printer
   instanceVariableNames: ''
   classVariableNames: ''
```

```
    poolDictionaries: ''
    category: 'Actalk-Examples'!

!Printer methodsFor: 'script'!

reply: value
    Transcript show: '> ' , value printString; cr! !
```

Voici un exemple d'utilisation:

```
Compteur new actor
    raz; incr; incr; consulteEtRepondA: Print
```

qui affiche:

```
> 2
```

Pour éviter au programmeur de devoir expliciter les destinataires et continuations nous développons une couche de plus haut niveau, de niveau d'expression analogue au Smalltalk standard. Un compilateur, analogue au compilateur Acore [Manning 87], traduira cette couche en Actalk avec génération automatique des continuations appropriées. Un premier prototype de ce compilateur a déjà été réalisé.

6 Intégration d'Actalk dans Smalltalk-80

Historiquement, les classes et les objets furent proposés dans les langages Simula et Smalltalk pour décrire des concepts abstraits ou concrets. Les acteurs furent introduits dans le langage Plasma pour décrire des structures de contrôle puis le parallélisme. Actalk propose de combiner ces deux approches, c'est-à-dire d'étendre les classes et les objets vers le parallélisme, et/ou de fournir un environnement de description et d'expérimentation d'acteurs. Parce que le sous-monde des acteurs Actalk est totalement intégré dans le langage Smalltalk-80, les acteurs peuvent envoyer des messages aux objets et vice-versa. Ainsi les deux styles de programmation peuvent être combinés.

6.1 Consistence

Une des motivations de l'introduction des acteurs dans le monde des objets Smalltalk-80 est de résoudre automatiquement les inconsistences entre objets et processus (discutées au paragraphe 3). Pourtant un mélange quelconque entre objets et acteurs risque de voir resurgir ce problème. Si plusieurs acteurs référencent un même objet passif, la situation sera équivalente à plusieurs processus partageant un même objet.

Une solution définitive à ce problème consiste en le remplacement de tout objet standard Smalltalk-80 (passif) par un acteur équivalent. Nous n'avons pas choisi cette voie pour ne changer Smalltalk-80 en aucune manière.

La solution que nous proposons est un ensemble de règles de combinaison, certaines étant vérifiées par l'implémentation d'Actalk ou par son interface utilisateur. La règle de non partage d'un objet passif par plusieurs acteurs est adoucie dans la pratique. Ainsi les classes Smalltalk-80 peuvent être approximées à des objets constants, et par conséquent subir des activations simultanées. (Selon la terminologie des acteurs, un objet Smalltalk-80 constant est équivalent à un acteur *non sérialisé*.)

6.2 Etendre l'Environnement de Smalltalk-80 vers les Acteurs

Les acteurs Actalk sont bien intégrés dans Smalltalk-80, par conséquent ils bénéficient automatiquement de l'environnement de programmation standard de Smalltalk-80. Ce dernier est d'une grande aide pour concevoir des langages d'acteurs et des applications. Cependant il ne s'avère pas totalement adapté aux spécificités des acteurs Actalk.

Ainsi le modèle standard d'interface graphique de Smalltalk-80, appelé MVC (acronyme pour Model View Controller), ne fonctionne plus correctement si l'on modifie (par exemple en déplacant) la représentation d'un acteur en train d'interpréter des messages.

Pour ces raisons, nous étendons l'environnement standard pour l'adapter aux spécificités des acteurs Actalk. Une extension du MVC standard a été réalisée dans le cadre d'un projet d'étudiants de DESS. Elle permet de visualiser et contrôler (par exemple entrer en mode pas à pas) un acteur au cours de son activité. Nous avons notamment introduit un contrôle générique des évènements d'acteurs (recevoir un message, l'interpréter...) en associant à chaque évènement une méthode générique. Ces méthodes n'ont par défaut aucune action, mais sont redéfinissables pour des classes d'application pour provoquer des actions telles que réaffichage, séquencement... Cela donne un nouvel exemple de l'utilisation des concepts de la programmation par objets pour la plateforme Actalk. Signalons enfin qu'un générateur automatique d'interface est disponible pour permettre une utilisation transparente par le programmeur.

Composante indispensable d'un environnement de programmation évolué, le debugger standard a également été étendu pour visualiser les transmissions synchrones et asynchrones. Il utilise des messages étendus qui contiennent le contexte de l'envoyeur pour pouvoir reconstruire la chaîne appropriée de contextes.

La figure 2 montre l'environnement Actalk au cours du calcul des n premiers nombres premiers par la méthode des filtres.

6.3 Simulation du Parallélisme

Le séquenceur de Smalltalk-80 n'assure pas de séquencement automatique entre les processus. La solution minimale consiste en l'utilisation du contrôle générique des évènements pour modifier de façon modulaire la sémantique du séquenceur de processus Smalltalk-80 standard (par des appels de séquencement explicite: `Processor yield`).

Un séquenceur à horloge a également été réalisé pour permettre un contrôle plus systématique et plus fin. Ce séquenceur est hiérarchique pour contrôler de manière distincte la concurrence *inter-acteurs*, c'est-à-dire entre les acteurs, et la concurrence éventuelle *intra-acteur*, c'est-à-dire les activités multiples internes à un même acteur tel défini au paragraphe suivant.

7 Etendre le Noyau pour Simuler Différents Langages d'Acteurs

Nous allons maintenant esquisser quelques extensions du noyau d'Actalk pour simuler certains des langages d'acteurs actuels les plus représentatifs. Une telle simulation n'a pas pour but une réimplémentation complète d'un langage et de son environnement de programmation, mais veut exprimer ses caractéristiques essentielles et les plus spécifiques. Ces extensions utilisent

Figure 2: Calcul des nombres premiers dans l'environnement Actalk.

Figure 3: Architecture d'Actalk (noyau et quelques extensions).

l'héritage pour dériver leur sémantique à partir de celle exprimée par le noyau d'Actalk. Nous détaillerons la description du modèle de calcul des acteurs défini par Gul Agha. Une autre extension standard décrit les différents protocoles de communication entre acteurs définis dans le langage Abcl/1 de Akinori Yonezawa, mais ne sera pas présentée dans ce papier. Ces extensions illustrent l'intérêt de la définition modulaire du noyau d'Actalk. Du fait de la relation d'héritage entre le noyau et ses extensions, la comparaison entre les différents modèles de langages d'acteurs est grandement facilitée. L'héritage ne sert pas seulement à classifier ces différents modèles (outil pédagogique), mais également à clairement connecter et réutiliser leurs implémentations respectives (outil de conception et d'implémentation). Du fait de la séparation entre les classes `Actor` et `ActorBehavior`, l'on peut même tenter des hybridations éventuelles, ainsi un acteur doué des protocoles de communication d'Abcl/1 (la classe `AbclActor`) et dont le comportement suit le modèle de Gul Agha (la classe `AghaActorBehavior`).

La figure 3 montre la hiérarchie des classes du noyau d'Actalk, plus quelques unes de ses extensions actuelles. Remarquons que la classe `ActorBehavior` et toutes ses extensions sont des classes *abstraites*, c'est-à-dire ne génèrent pas d'instances. Ces classes donnent la sémantique du traitement des messages, mais pas la sémantique des messages eux-mêmes. Seules les classes d'application, telles la classe `Compteur`, génèrent des instances: les comportements, et définissent la sémantique des messages qu'ils peuvent interpréter. Les acteurs sont des instances de la classe `Actor` ou de ses sous-classes.

7.1 Le Modèle de Calcul de Gul Agha

Le modèle de calcul des acteurs défini par Gul Agha dans [Agha 86] remplace les changements d'état (affectation de variables) des acteurs sérialisés par un concept de plus haut niveau: le *remplacement de comportement*. Au cours de l'interprétation d'un message, le comportement courant d'un acteur spécifie son *comportement de remplacement*, c'est-à-dire comment sera interprété le prochain message. A la différence du noyau Actalk, un comportement n'acceptera plus qu'un seul message. Le comportement de remplacement spécifiera à son tour son propre comportement de remplacement au cours de l'interprétation du message suivant. Cela conduit à une chaîne de comportements ordonnés par causalité, et isomorphe à la queue de messages. Les deux points importants à souligner sont l'*absence d'affectation* et la *séparation entre les comportements successifs*. Par conséquent les différents comportements d'un acteur

Figure 4: Le concept de remplacement de comportement.

sont susceptibles d'être exécutés simultanément, en *pipe-line*.

7.2 Implémentation du Modèle

Pour représenter cette sémantique légèrement différente de celle de la classe `ActorBehavior`, nous en introduisons une sous-classe, nommée `AghaActorBehavior`:

```
ActorBehavior subclass: #AghaActorBehavior
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Extensions-Agha'!

!AghaActorBehavior methodsFor: 'initialization'!

setProcess
   [self acceptNextMessage] fork! !

!AghaActorBehavior methodsFor: 'behavior replacement'!

replace: replacementBehavior
   aself initializeBehavior: replacementBehavior! !
```

Deux méthodes suffisent à définir cette extension. La méthode `setProcess` est redéfinie, et accepte maintenant un message et un seul. Nous introduisons une nouvelle méthode, nommée `replace:`, pour spécifier le comportement de remplacement (et l'initialiser).

Remarquons que cette redéfinition change légèrement la signification de la variable d'instance nommée `behavior` et définie dans la classe `Actor`. Elle représente maintenant le *plus récent comportement courant* d'un acteur. Au cours d'un remplacement de comportement, cette variable est réaffectée (par la méthode `initializeBehavior:` de `Actor`) au nouveau comportement. Le comportement courant n'est pas perturbé, et finira normalement son activité courante. Il pourra alors être récupéré par la gestion de la mémoire (*garbage collector*), n'étant plus référencé par l'acteur et son processus associé étant terminé. La figure 4 illustre ce modèle d'acteurs.

Pour l'exemple du compteur, les précédentes affectations de variable seront remplacées par la spécification d'un comportement de remplacement. (Nous supposons l'existence d'une méthode de classe `contenu:` pour créer et initialiser un nouveau compteur.) Remarquons que la méthode `consulteEtRepondA:`, bien que ne changeant pas l'état du compteur, doit également spécifier un comportement de remplacement (égal au comportement courant) pour que le compteur puisse interpréter le prochain message.

```
consulteEtRepondA: destinataire
   self replace: self.
   destinataire reply: contenu!

incr
   self replace: (AghaCompteur contenu: contenu + 1)!
```

8 Développements en Cours et Futurs

La plateforme Actalk est actuellement en cours de développement et d'expérimentation dans divers domaines d'application.

Nous avons l'intention de simuler d'autres modèles de langages d'acteurs par des extensions du noyau d'Actalk. Un prototype de système multi-agents a été réalisé en Actalk dans le cadre d'un stage de DESS [Abramovici 89]. Il modélise chaque agent par un acteur Actalk et des KS (modules de connaissance), eux-mêmes implémentés par des acteurs Actalk pour un fonctionnement opportuniste et concurrent du système. Actalk a de plus été aisément intégré dans le système OKS de Robert Voyer pour permettre une activation concurrente de démons, résultats de la compilation d'un ensemble de règles de production [Voyer 89].

Actalk a également été choisi comme support d'études pour la modélisation de protocoles de communication entre processeurs dédiés au traitement du signal [Fron 89], et la modélisation de stratégies de migration d'acteurs dans le cadre du projet Mering-IV spécialisé pour les systèmes multi-agents [Ferber et Briot 88].

Cependant, malgré ses avantages, la plateforme Actalk n'offre qu'une simulation du parallélisme. Pour y remédier, nous débutons actuellement la conception d'un environnement d'exécution d'acteurs sur machines parallèles. Cet environnement sera une extension du langage C++, équivalente au noyau d'Actalk, et aura pour première cible un réseau de Transputers. Une interface incluant un traducteur permettra une exécution transparente sur la machine parallèle des programmes développés dans l'environnement Actalk.

Des projets similaires à Actalk sont une description plus générale des mécanismes de concurrence et synchronisation en Smalltalk-80 [Bézivin 88], et des extensions de Smalltalk-80 vers le parallélisme, notamment ConcurrentSmalltalk [Yokote et Tokoro 87]. A l'inverse notre but n'est pas de proposer un langage spécifique, extension efficace de Smalltalk-80 vers le parallélisme, mais plutôt d'offrir une plateforme ouverte de spécification et d'expérimentation des différentes familles de langages d'acteurs en réutilisant le maximum de l'environnement de programmation Smalltalk-80 standard.

9 Conclusion

Dans cet article nous avons présenté la conception du système Actalk, basé sur Smalltalk-80 et fournissant un environnement unifié pour comparer et concevoir différents langages d'acteurs et leurs stratégies d'implémentation. Les avantages du choix de la programmation par objets comme fondation. sont les suivants: les classes nous permettent de décrire les modèles de comportements d'acteurs, l'héritage des classes permet de classifier les différents modèles d'acteurs, et la généricité permet un contrôle modulaire des évènements. Le noyau d'Actalk a été étendu avec succès pour décrire divers modèles de calcul et d'implémentation. L'extension Abcl/1 est décrite dans [Briot 89], et [Briot 88] détaille plusieurs types d'extensions et exemples.

Nous remercions Pierre Abramovici, François Goret, Nicolas Graube, Chork-Huat Ky, Marc LeBris, Sylvie Lemarié, Loic Lescaudron, et Gilles Moussion, pour leur participation active au projet. Nous remercions également Jean Bézivin, Pierre Cointe, Jacques Ferber, Annick Fron, Les Gasser, Philippe Gautron, Jean-François Perrot, et Robert Voyer, pour leur collaboration et commentaires constructifs.

References

[Abramovici 89] P. Abramovici, Agentalk: Multi-Agents en Actalk, rapport de stage de DESS GLA, direction: J.-P. Briot, Université Pierre et Marie Curie, Septembre 1989.

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

[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.

[Briot 88] J.-P. Briot, From Objects to Actors: Study of a Limited Symbiosis in Smalltalk-80, *Rapport de Recherche LITP 88-58 RXF*, Université Pierre et Marie Curie, Paris, Septembre 1988.

[Briot 89] J.-P. Briot, Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment, *ECOOP'89*, Cambridge University Press, pages 109-129, Juillet 1989.

[Briot et Cointe 87] J.-P. Briot et P. Cointe, Definition of a Uniform, Reflexive and Extensible Object-Oriented Language, *ECAI'86*, Advances in Artificial Intelligence-II, North-Holland, pages 225-232, 1987.

[Cointe 87] P. Cointe, Metaclasses are First Class: the ObjVlisp Model, *OOPSLA'87*, Special Issue of SIGPLAN Notices, Vol. 22, No 12, pages 156-167, Décembre 1987.

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

[Ferber 87] J. Ferber, Des Objets aux Agents: une Architecture Stratifiée, *RFIA'87*, Vol. 1, pages 275-286, Novembre 1987.

[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, Novembre-Décembre 1988.

[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, Mai 1989.

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

[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.

[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.

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

[Pascoe 86] G.A. Pascoe, Encapsulators: A New Software Paradigm in Smalltalk-80, *OOPSLA'86*, Special Issue of SIGPLAN Notices, Vol. 21, No 11, pages 341-346, Novembre 1986.

[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.

[Yokote et Tokoro 87] Y. Yokote et M. Tokoro, Experience and Evolution of ConcurrentSmalltalk, *OOPSLA'87*, Special Issue of SIGPLAN Notices, Vol. 22, No 12, pages 406-415, Décembre 1987.

[Yonezawa et al. 86] A. Yonezawa, J.-P. Briot et E. Shibayama, Object-Oriented Concurrent Programming in ABCL/1, *OOPSLA'86*, Special Issue of SIGPLAN Notices, Vol. 21, No 11, pages 258-268, Novembre 1986.