Programmation par Objets Concurrents, Acteurs et Actalk

(I. Introduction)

II. Des Objets Passifs aux Objets Actifs

III. Concepts

IV. Exemple de Langage de Programmation: ConcurrentSmalltalk (CST)

V. Examples de Programmes en Actalk

VI. Implémentation des Objets Actifs (Actalk)

VII. Le Modèle de Calcul des Acteurs

VIII. Comparaison des Principaux Langages de Programmation Concurrente à Objets

(IX. Applications, Axes de Recherche, et Conclusion)

© 1994 J.-P. Briot

Partie II.

Des Objets Passifs aux Objets Actifs

Objets Passifs

Objets et Concurrence (Processus)

Objets Protégés

Objets Actifs

Evolution des Concepts des Objets vers les Objets Actifs

© 1994 J.-P. Briot

Mono-activité des objets séquentiels

Dans la plupart des langages/systèmes à objets, les objets sont passifs (ils n'ont pas d'activité propre)

L'activité d'un objet repose sur l'allocation du processeur (virtuel) à l'objet

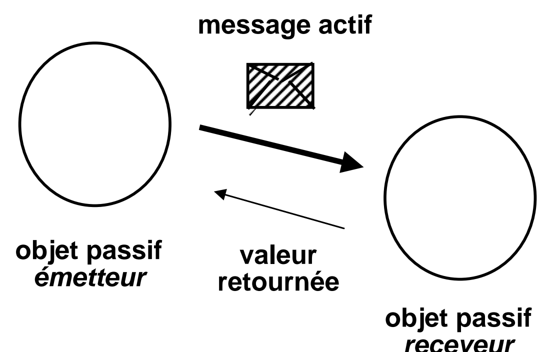
Les objets sont activés par envoi de message qui réalloue le processeur virtuel à l'objet receveur

Un seul objet est actif à la fois

© 1994 J.-P. Briot

Transfert de l'activité

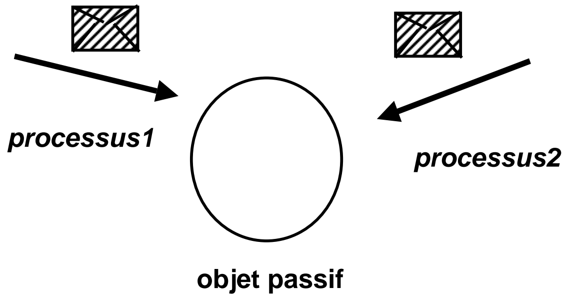
Lors d'une transmission de message:
- l'objet émetteur est suspendu,
- l'activité est transférée à l'objet receveur jusqu'à complétion de son calcul,
- il y a transmission de la réponse au receveur qui reprend alors son activité



© 1994 J.-P. Briot

Concurrence

Pour permettre de multiples activités simultanées, on peut introduire de multiples processeurs virtuels (multi-processus)



Mais les objets demeurent passifs
En conséquence, ils doivent être protégés contre des accès concurrents (potentiellement conflictuels)

Objets et Concurrence

L'introduction de la concurrence impose un certain contrôle sur les accès concurrents à des objets partagés

Il existe des moyens standards (sémaphores, moniteurs...) pour assurer la synchronisation

Mais ceci oblige le programmeur à maitriser deux types distincts (orthogonaux) de programmation:

transmission de message entre objets
et
synchronisation des accès concurrents à des données partagées

Objets Protégés

Une première étape prend en compte la protection des objets contre des accès concurrents en identifiant:

objet
et
grain de données à synchroniser

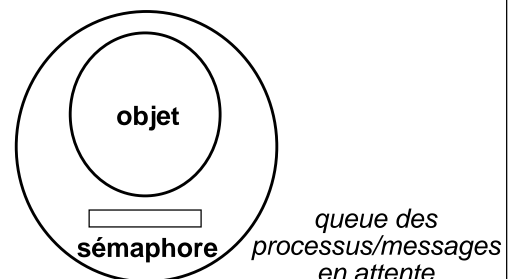
transmission de message
et
synchronisation

Les objets sont encapsulés dans une protection qui assure au minimum l'exclusion mutuelle des envois de message, c'est-à-dire qu'un seul message est traité à la fois

Objets Protégés (1)

Protection Minimale:
exclusion mutuelle

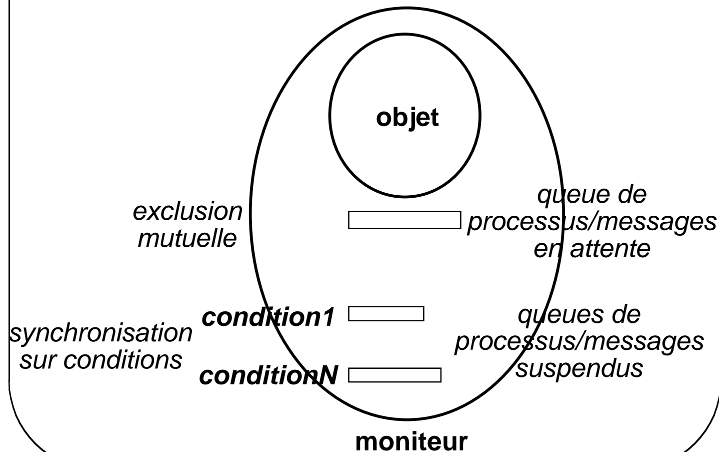
Assurée par un sémaphore qui garantit qu'au plus une activation (envoi de message / processus) est traitée à la fois



Objets Protégés (2)

exclusion mutuelle + synchronisation (suspension sur des conditions)

La synchronisation est assurée par des conditions (variables ou expressions) sur lesquelles les processus seront suspendus tant que les conditions ne sont pas satisfaites



Unification Objet / Processus

L'étape ultime (et naturelle) identifie:

objet
et
activité (processus)

objet
et
grain de données à synchroniser

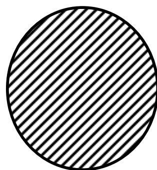
transmission de message
et
synchronisation

Cette intégration permet de conserver la simplicité et la modularité de la programmation par objets

L'unification entre données, méthodes et puissance de calcul fait d'un objet une entité autonome et "d'une seule pièce". Cette atomicité facilite l'allocation et la réallocation (migration) des données et des ressources

Objets Actifs

Un objet actif possède son propre processeur virtuel (processus)



objet actif

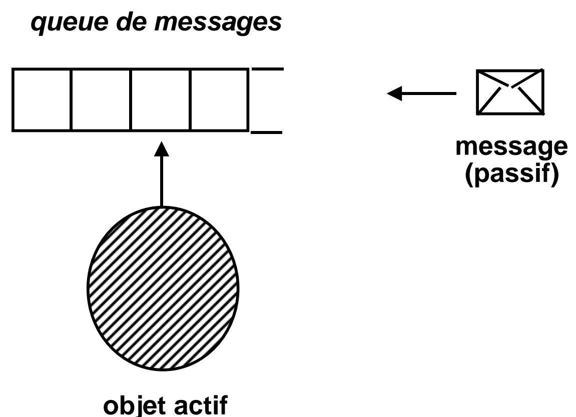
Un objet actif possède sa propre puissance de calcul interne, indépendamment des autres objets

Ceci entraine une concurrence des activités entre les objects (ainsi appelée concurrence <u>inter-objets</u>)

Bufferisation des Messages

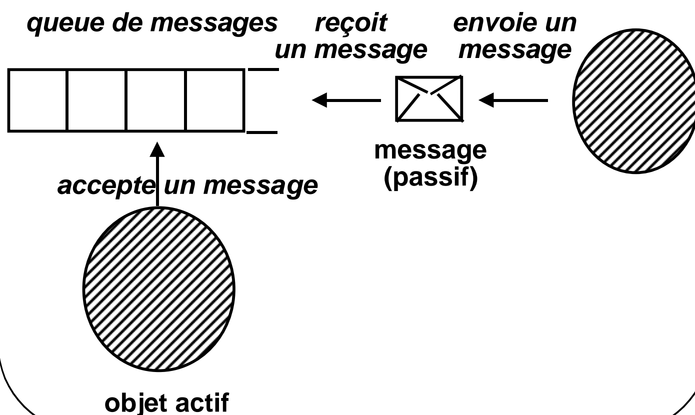
Un objet actif peut être occupé (à traiter un message) pendant qu'il reçoit un message.
Pour cette raison, les messages reçus seront stockés *(bufferisés)* en attente de leur traitement

queue de messages



Bufferisation des Messages

Le concept de bufferisation (queue) de messages permet:
de gérer l'ordre des requêtes,
des vitesses de communication éventuellement différentes entre émetteur et récepteur,
éventuellement de décharger l'émetteur d'une attente superflue, par transmission asynchrone:
il y a alors désynchronisation entre envoi de message, réception de message, et acceptation (puis traitement) de message



Des Objets (Passifs) aux Objets Actifs

Concepts Objet:

- objet
- message
- classe
- héritage

Des Objets (Passifs) aux Objets Actifs: Evolution des Concepts

Concepts Objet -> Concurrence

Concept	Evolution
objet	chaque objet actif possède son propre processus
message	la transmission de message est bufferisée et potentiellement asynchrone
classe héritage	les principes d'implantation doivent être revus pour les architectures réparties autres concepts alternatifs: prototypes, délégation
concurrence et synchronisation interne à un objet	contrôle du traitement de message (concurrence et synchronisation) par un object

Partie III. Concepts

- Envois (de Message) Synchrone et Asynchrone
- Activité des Objets Actifs
- Acteurs
- Evolution des Mécanismes de Structuration (Classe et Héritage)
- Concurrence et Synchronization Interne

Synchronisme vs Asynchronisme

De manière à maximiser la concurrence (en évitant au receveur d'attendre inutilement dans le cas d'une réponse éventuellement non nécessaire),
une grande partie des modèles de langages à objets concurrents considèrent <u>la transmission asynchrone de message</u> (aussi appelé envoi asynchrone) comme le type fondamental de communication

Certains modèles de calcul considèrent au contraire la <u>transmission synchrone</u> comme fondamental

On peut modéliser la transmission synchrone à partir de la transmission asynchrone, et vice-versa

Il s'agit d'un choix entre favoriser concurrence ou synchronisation

La plupart des modèles évolués proposent ces deux types de transmission plus éventuellement des supplémentaires...

© 1994 J.-P. Briot

Uniformité vs Mixité

Certains modèles de langages à objets prônent l'uniformité avec présence unique des objets actifs

D'autres modèles, au contraire, permettent la coexistence d'objets passifs et d'objets actifs
Cela permet pour les extensions concurrentes une compatibilité "vers le haut" et une réutilisation de modules et de méthodologies conventionnelles

Mais il faut alors assurer des règles méthodologiques pour éviter des accès concurrents intempestifs à des objets passifs

Règle:
Les entités partagées et pouvant changer d'état doivent être modélisées par des objets actifs ou au minimum protégés

© 1994 J.-P. Briot

Acteurs

Nous appellerons <u>acteurs</u> des objets actifs <u>réactifs</u> et communiquant par transmission <u>asynchrone</u> de message

Le concept d'acteur a été introduit par Carl Hewitt

Plus récemment, un modèle de calcul plus précis et formalisé a été défini par Gul Agha et Carl Hewitt, et appelé le <u>modèle de calcul des acteurs</u> (the Actor model of computation)

Le terme *acteur* peut être employé dans un sens *générique*, comme défini en haut, ou bien en référence à la sémantique *precise* du modèle de calcul des acteurs *(une de ses caractéristiques principales étant l'absence d'affectation, voir §VII).*

© 1994 J.-P. Briot

Structuration

Dans les langages à objets séquentiels, le concept traditionnel de structuration des objets est le couple <u>classe/héritage</u>

La plupart des langages concurrents à objets conservent les fondements de ce modèle

Les classes permettent l'abstraction et la factorisation de la structure et des comportements des objets concurrents

© 1994 J.-P. Briot

Les Différents Niveaux de Concurrence

Un objet actif possède sa propre puissance de calcul, indépendamment des autres objets actifs

Ceci entraine une concurrence des activités <u>entre</u> les objects (en conséquence appelée concurrence <u>inter-objets</u>)

Un objet actif peut aussi avoir la capacité de traiter plusieurs messages simultanément

Ceci entraine une concurrence supplémentaire des activités <u>à l'intérieur</u> des objects (en conséquence appelée concurrence <u>intra-objet</u>)

Activité(s) Interne(s) à un Objet

L'activité interne d'un objet peut être (P. Wegner):

 # *séquentielle*
l'objet a au plus une activité, les messages sont donc traités séquentiellement

 # *quasi-concurrente*
plusieurs messages peuvent être en cours de traitement, mais un seul est en cours d'activation (les autres étant suspendus: en attente)

 # *concurrente*
l'objet peut posséder plusieurs activités simultanées, et donc traiter plusieurs messages simultanément

Concurrence Intra-Objet

concurrence

concurrence libre (unserialized)

concurrent
concurrence avec contrôle, ex: remplacement de comportement du modèle de calcul des acteurs

quasi-concurrent
objets actifs "monitorés", ex: ConcurrentSmalltalk

séquentiel
séquence pure (serialized)
///////ex: acteurs sérialisés de Actalk

synchronisation

Contrôle de la Concurrence Intra-Objet

Un objet actif peut contrôler l'acceptation et le traitement des messages par:

 # suspension *(quasi-concurrence)*

<u>attente d'un modèle de message</u> (*wait-for* ABCL/1)

<u>suspension du traitement</u> (*relinquish* CST)

<u>interruption du traitement</u> (*priorités* Orient84/K, *mode express* ABCL/1)

 # acceptation *(concurrence)*

<u>spécifier quand le message suivant est accepté</u> (concept de *comportement de remplacement* du modèle de calcul des acteurs)

<u>rendre activables un ou plusieurs modèles de messages</u> (Orient84/K, Rosette)

Comparaison

La suspension pendant le traitement (de type moniteur) est plus "naturelle" pour le programmeur, mais elle est difficile à implémenter efficacement (car elle implique la gestion de suspension/reprise de contextes)

l'activation/désactivation de messages donne un contrôle plus simple et général sur l'acceptation des messages, et permet la concurrence

Les langages fondés sur le principe d' activité autonome permettent un contrôle explicite sur l'acceptation de messages

La réflection *(reflection)* est une nouvelle méthodologie basée sur le concept d'auto-représentation. Elle permet de décrire et contrôler l'exécution d'un objet (actif) par le biais d'un (ou plusieurs) autre objet, appelé son méta-objet

Concurrency and Synchronization

Synchronization is needed to manage activities, that is acceptance and computation of requests, within an object.

Two types of synchronization are involved:

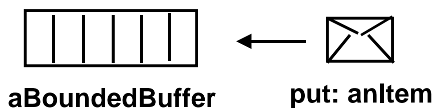
activation synchronization
synchronization between competing activities, that is concurrency control over pending requests
(e.g., strongest control is mutual exclusion)

service synchronization
synchronization onto the availability of services (methods may be disabled depending on the state of the object)

Service Synchronization

Availability of services (methods) may depend on the state of the object.

Let's take the (seminal) example of a bounded buffer (a buffer whose size is bounded)



Requests for putting items onto the buffer should be delayed if/while the buffer is full

(Conversely requests for getting items from the buffer should be delayed/suspended while the buffer is empty)

Synchronization of Method Invocations

Specification of activation synchronization and service synchronization may be clearly separated ([Thomas, PARLE'92]) or mixed within the same specification (currently, most of the case).

Most of the OOCP systems offer quasi-concurrency (for tractability and implementation reasons), that is *implicit mutual exclusion.*
However systems may be concurrent. In the case of the bounded buffer, put and get requests may be handled concurrently (because accessing different data zones). Other seminal example: multiple readers/writers.

Most of the OOCP systems offer synchronization over acceptance of messages (before computing them) as opposed to suspending them.

Control of Acceptance of Messages

Control over acceptance of messages may be:

unconditional and implicit
in reactive models, e.g., ConcurrentSmalltalk

explicit
in autonomous activity models, e.g., *ACCEPT* in POOL, *serve* in Eiffel //
by waiting for some pattern of message, e.g., *wait-for* in ABCL/1

conditional (guarded)
in guarded methods, predicates control acceptance depending on object state and possibly message contents,
e.g., *activation conditions* based on *synchronization counters* in Guide

"mutable"
abstract states define sets of enabled methods (acceptance sets). An object may change its abstract state
e.g., *behavior abstraction* in ACT++, *enabled sets* in Rosette

Synchronization of Method Invocations (2)

The two main schemes for expressing synchronization over message acceptance are:

guarded methods: *a boolean guard predicate is assigned to each method*

acceptance sets: *object specifies the next set of messages to be accepted, that is, methods to be enabled*

guarded methods

class BoundedBuffer
n: number of elements; size: (max) size of the buffer

put when n < size

get when n > 0

We suppose that mutual exlusion is implicit, otherwise we need to explicit it into both guards:

put when n < size and not put and not get

Synchronization of Method Invocations (3)

Notice that such guards are often implemented (as in Guide) with synchronization counters (Verjus & Banatre). They record number of started, completed, pending, ongoing and completed executions of a method.

acceptance sets

class BoundedBuffer

acceptance sets: empty = #{put}; partial = #{put, get}; full = #{get}

initialize become empty

put when (n = size) become full else become partial

get when (n = 0) become empty else become partial

Specifications express transitions between acceptance sets *after* completing execution of a method

Synchronization of Method Invocations (4)

These synchronization specifications control acceptance of method invocations.

One may also control computation of methods after their acceptance (during their activation):

by waiting for some pattern of message, e.g., *wait-for* in ABCL/1

suspend computation onto conditions, e.g., monitor-like *relinquish* in ConcurrentSmalltalk (see Part V)

interrupt computations, e.g., priorities in Orient84/K, express mode message passing in ABCL/1

But such solutions have some weaknesses: mixing programs and synchronization, suspended contexts are expensive, nested monitor calls...

Synchronization of Method Invocations (5)

To specify synchronization conditions on method invocations, important issues are:

control may be centralized (e.g., in the *body* in POOL, or in path expressions) or decentralized in every method

control should be abstract enough and independent of method implementation

what should be the default? mutual exclusion or full internal concurrency?

These are *very important* issues for reuse of synchronization specifications (inheritance and composition of synchronization conditions, see Part IX)

Priorités

Un mécanisme de priorités peut être également introduit pour donner plus de contrôle sur le déroulement et les interruptions des traitements, mais cela réduit la lisibilité et la modularité des programmes

Les priorités peuvent être associées:

aux messages
"traiter d'abord les messages urgents!"
mode express de ABCL/1

aux objects (émetteurs)
"traiter d'abord les messages du chef!"

aux méthodes
"traiter d'abord le message dont la méthode est la plus prioritaire"
16 niveaux de priorité en Orient84/K

Terminologie

Object-Oriented Concurrent Programming

Concurrent Object-Oriented Programming

tentative de traduction:

Programmation Concurrente par Objets

Langage Concurrent à Objets

Il y a une grande famille de langages sous ce terme général:

objets et processus sont conservés distincts
objets et processus sont unifiés (objets actifs)
objets actifs et concurrents (activités internes multiples), ex: le modèle des acteurs

Mais dans la plupart des langages, les objets restent séquentiels, la concurrence n'est qu'inter-objets, pour cette raison la terminologie suivante:

Programmation par Objets Concurrents
et
Langage à Objets Concurrents

est soit trop restrictive, soit ambigue

Partie IV.
Exemple de Langage de Programmation Concurrente par Objets: ConcurrentSmalltalk (CST)

Les Objets Actifs de ConcurrentSmalltalk

Transmission Synchrone

Transmission Asynchrone
Concept de Client *(Reply Destination* ou *Customer)*
Concept de Continuation

Concept de Réponse Anticipée *(Future)*

Post-Exécution de Messages

A Quick Introduction to Smalltalk

Smalltalk semantics is purely object-oriented:

only objects
classes are objects
and message passing
messages are objects

Smalltalk syntax is intuitive but not standard to programmers *(initially designed for non programmers)*

Defining a class, e.g. class Counter

is the quotation character

'this is a string'

```
Object subclass: #Counter
    instanceVariableNames: 'contents '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Basic-Examples'
```

creating a new class, subclass of class Object, whose name is #Counter

one instance variable is defined as contents

shared variables are not used here

the class is organized in a library for the class Browser

syntax for variables:

variables which are shared start with an uppercase character, e.g., class names which are global variables (e.g., Counter)

variables which are owned by a single object or context start with a lowercase character, e.g., temporary variables and instance variables (e.g., contents)

Creating a new instance of a class (Counter)

```
| aCounter |
aCounter := Counter new
```

declaring a temporary variable

:= stands for assignment
new is the instance creation message

syntax for message passing:
receiver message

where message:
selector
selector argument
sel: arg1 ector: arg2 ...

```
Counter new      --->  a Counter
1 + 2            --->  3
#(a b c) at: 2   --->  b
```

this last case is specific to Smalltalk and is called a keyword message (split into keywords). An example with several keywords is the message for class definition (subclass: #Counter instanceVariableNames:...)

Note about the fileOut syntax (format)

Classes and methods are usually defined in the class browser. This definitions are saved onto files by the fileOut menu. This automatically adds bangs (!) to the definitions in order to separate definitions.
Here we observe this program print-out convention

Defining methods (of class Counter)

```
!Counter methodsFor: 'operations'!

contents
  ^contents!

incr
  contents := contents + 1!

reset
  contents := 0! !
```

the caret (^) returns the following object as the result value of the method invocation

these two methods change the state of the counter (by assigning its instance variable)

Note about returning values

A method invocation always returns a value. If no return value is specified, the default value returned is the object itself (pseudo-variable self)

The object currently performing the method is self referenced by pseudo-variable self. This allows anonymous recursive message invocation

Initializing the values of instance variables

Because of encapsulation, the only way is by defining initialization method(s):

```
!Counter methodsFor: 'initialization'!

contents: v
  contents := v! !
```

Example of session:

```
| aCounter |

aCounter := Counter new.

aCounter contents: 100.

aCounter incr;
         incr;
         incr.

aCounter contents      --->

Counter new contents: 10
```

creation of a counter

instructions in a sequence are separated by periods (.)

initialization of its contents

a cascade is a sequence of messages sent to the same receiver; messages are separated by semi-colons (;)

example of composition of messages (creation and initialization)

Defining class methods

Class methods define messages which are sent to the class and not to its instances

They are used primarily for:
* defining or redefining creation protocols
* defining example methods

```
!Counter class methodsFor: 'instance creation'!

new
  ^super new reset! !
```

the instance creation method new is redefined in class Counter to automatically initialize the contents to 0

When there is such a recursive redefinition, the pseudo variable super is used in place of self to call the previous definition (in the superclass)

```
!Counter class methodsFor: 'example'!

example
  "Counter example"

  Counter new contents: 200;
    incr; incr;
    contents! !
```

"this is a comment"

example methods provide samples of use, entry points and demos of the class. This is part of the library documentation

ConcurrentSmalltalk

Extension de Smalltalk-80 vers la concurrence réalisée par Yasuhiko Yokote et Mario Tokoro

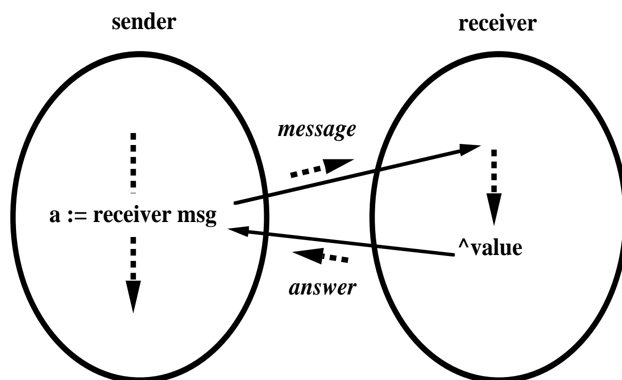
ConcurrentSmalltalk (CST)
- est compatible "vers le haut" avec Smalltalk-80
- introduit les objets actifs, ils sont quasi-concurrents (de type monitor)
- introduit deux nouveaux types de transmission de message: asynchrone et anticipée (future)

Les classes d'objets actifs sont définies à l'aide du mot clé atomicSubclass:

```
Object atomicSubclass: #Counter
    instanceVariableNames: 'contents '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'CST-Examples'!
```


Transmission Synchrone

requête en attente de la réponse

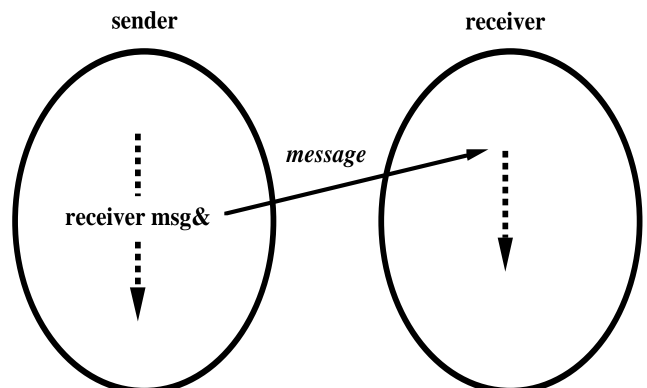


L'exécution de sender est suspendue en attendant l'acceptation du message, puis la fin du traitement par receiver,
la réponse (answer) est alors retournée à sender qui peut reprendre son exécution

Note:
Cette transmission est compatible avec la transmission de message standard de Smalltalk-80. Cependant la sémantique est légèrement différente:
Le receveur n'est pas nécessairement immédiatement disponible pour accepter le message (il peut être occupé à traiter d'autres messages antérieurs)

Transmission Asynchrone

requête sans attente ni réponse



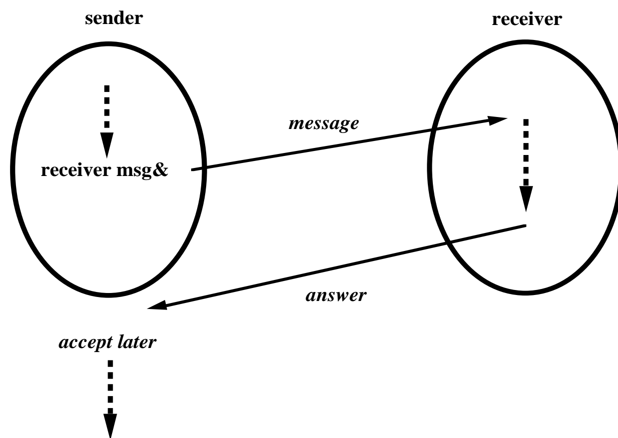
L'exécution de sender continue (reprend) aussitôt après envoi du message à receiver
le traitement du message par receiver commence alors simultanément à la suite de l'exécution de sender

Du Synchrone Bidirectionnel à l'Asynchrone Unidirectionnel

La transmission asynchrone accroit la concurrence des programmes (elle déclenche une nouvelle activité et évite une attente)

Question: Quand et comment peut-on remplacer une transmission synchrone par une transmission asynchrone?

Réponse: sender envoie sa requête de manière asynchrone à receiver tout en lui indiquant de renvoyer la réponse (également de manière asynchrone) directement à lui-même (sender)



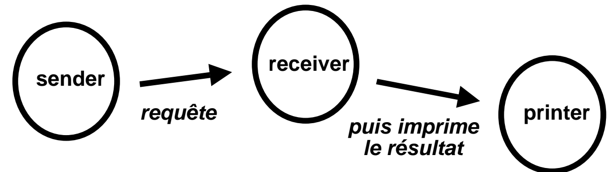
Concept de Client

(Reply Destination ou Customer)

L'objet à qui renvoyer la réponse est appelé le client (customer ou encore reply destination) et est le destinataire de la réponse

Cette technique permet de transformer:
une transmission synchrone bidirectionnelle *en* deux transmissions asynchrones unidirectionnelles

Le client peut être l'émetteur lui-même ou bien un autre objet quelconque, par exemple un scribe *(printer)*



Concept de Continuation

Le client peut être:
- # préexistant, par ex: printer,
- # créé dynamiquement

Le principe est le suivant:
plutôt que mener la transaction à lui tout seul, l'émetteur *(sender)* délègue le reste du calcul (ce qui reste à faire après réception de la réponse) à un autre objet spécialement créé pour la circonstance

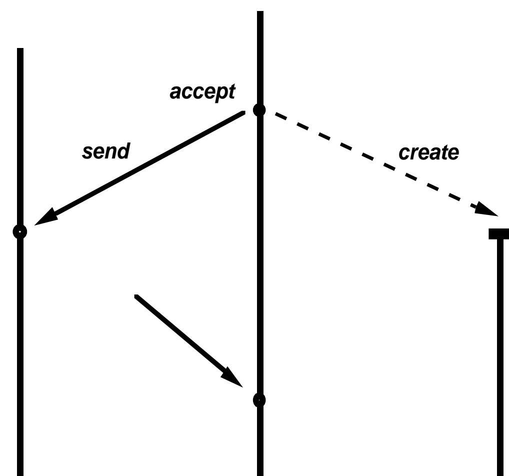
Cet objet représente la continuation (suite) du calcul, et en conséquence est appelé la continuation

Les continuations entrainent la fluidité des calculs, c'est-à-dire, une disponibilité constante des objets, chaque requête n'induisant qu'une étape de calcul sans attente

Diagramme d'Evénements

(R. Feynmann, G. Agha)

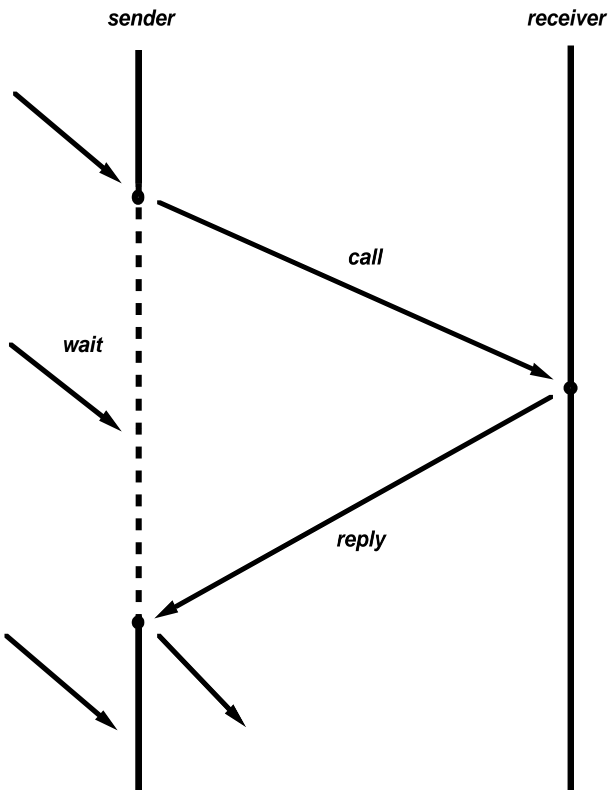
événement = acceptation d'un message



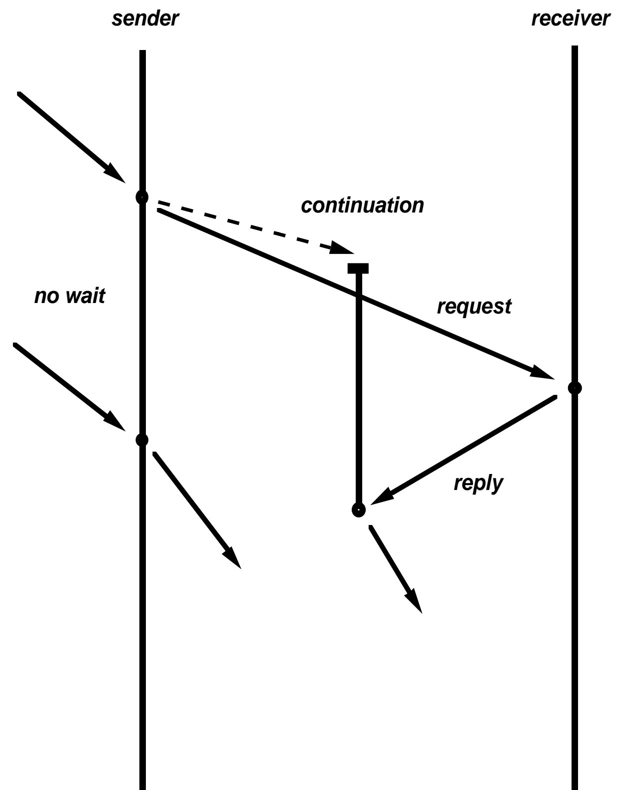
actions:
- # envoyer un message
- # créer un objet

Transmission Synchrone

(avec attente)



Transmission Asynchrone avec (Passage à la) Continuation *(sans attente)*



Nécessité d'Attente

Dans certains cas l'émetteur a besoin de la réponse pour savoir (comment) répondre aux messages suivants

Dans ce cas, l'attente de la réponse est nécessaire; il faut alors utiliser une transmission synchrone

Cependant!
L'émetteur n'a pas pour autant toujours besoin de la réponse immédiatement, c'est-à-dire qu'il pourrait conduire un autre sous-calcul tout en attendant la réponse

Une réponse anticipée, ou promesse de réponse, (future) sera alors immédiatement retournée à la place de la vraie valeur en cours de calcul

Ainsi la restriction de la transmission synchrone au minimum de cas permet une concurrence optimale

Le Concept de *Future*

Calcul anticipé

Promesse de réponse

La métaphore de l'oeuf et du poulet:

Un fermier vous vend un oeuf à la place (comme promesse) d'un (futur) poulet

Le contrat garantit que l'oeuf deviendra tôt ou tard un poulet

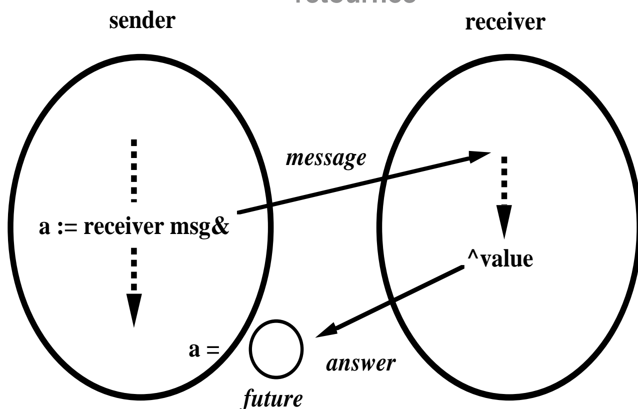
L'oeuf peut être transmis (revendu), par envoi de message, à un autre propriétaire; le contrat reste toujours valide

Le propriétaire espère qu'avant qu'il en aie besoin (pour un repas), l'oeuf sera déjà devenu un poulet

Dans le cas contraire, le propriétaire affamé doit attendre

Transmission Anticipée (*Future*)

une réponse anticipée est immédiatement retournée



- L'exécution de sender continue aussitôt après envoi du message à receiver, un future (valeur anticipée) est immédiatement retourné comme valeur de la transmission
- le traitement du message par receiver est simultané à l'exécution de sender
- receiver retourne sa valeur directement au future *(qui est le destinataire de la réponse)*
- dans l'intervalle, si sender veut utiliser (déterminer) la réponse, il verra son exécution suspendue jusqu'à sa détermination

Un *Future* est-il "Gluant"?

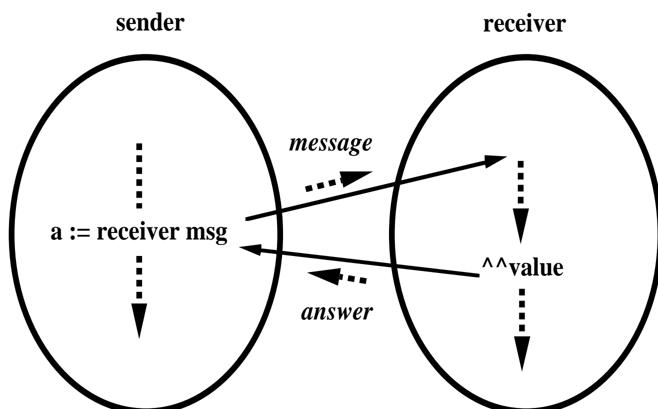
Question:

Y a-t'il un moyen de tester si la valeur d'un *future* est déterminée (test de complétion du calcul), ou bien reste-t'on "englué" dès qu'on l'a touché (veut accéder à sa valeur)?

deux points de vue:

- Dans le modèle des acteurs, on n'offre pas au programmeur de test de complétion. La synchronization doit rester implicite et transparente pour éviter tout risque d'erreur
- En ABCL/1, le programmeur peut explicitement tester la complétion d'un future (par la fonction :ready?) pour lui donner une chance de mener un autre calcul ou action avant une nouvelle tentative d'accès

Transmission Synchrone avec Post-Exécution

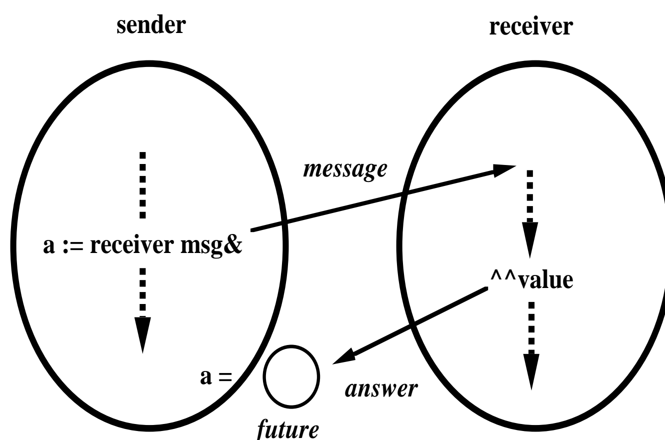


- Comme pour une transmission synchrone mais receiver continue son exécution après avoir répondu

- ^^ retourne une réponse, mais avec post-exécution à la différence de ^ qui termine l'exécution

- Cela permet notamment une réinitialisation en fin de traitement sans retarder le renvoi de la réponse

Transmission Anticipée avec Post-Exécution



pas de commentaire

(laissé en exercice!)

Partie V.
Exemples de Programmes

Exemples de Programmes dans le Langage ConcurrentSmalltalk (CST) pour illustrer les Concepts et la Méthodologie de la Programmation Concurrente par Objets

Concept	*Exemple*
Client et Serializer	Printer
Continuation	Factorial
Divide and Conquer et Join Continuation	MultiplyInRange
Pipeline	Prime Numbers
Synchronisation	Bounded Buffer

© 1994 J.-P. Briot

Printer

L'impression des résultats intervient en général à la fin d'un calcul, c'est-à-dire, comme client *(reply destination)* final d'un calcul

En Smalltalk-80 la fenêtre Transcript peut être utilisée pour un tel usage

Mais le Transcript est un objet passif, donc non protégé contre d'éventuelles requêtes simultanées d'impression

Un objet actif ConcurrentSmalltalk traite les messages séquentiellement

Il peut donc être utilisé pour "sérialiser" un objet passif, c'est-à-dire imposer la séquentialité des messages

A première vue, cela peut sembler paradoxal: utiliser le concept d'objet de la programmation concurrente pour sérialiser (restreindre la concurrence) d'un objet passif!

© 1994 J.-P. Briot

classe Printer

Mais un objet passif n'est pas protégé contre les accès concurrents;
un objet actif l'est

La classe Printer définit le comportement d'un tel scribe qui encapsule le Transcript.
Elle crée une instance prédéfinie, référencée par la variable Print

Les messages retournant les réponses aux continuations auront par convention toujours pour sélecteur le mot-clé reply:

```
ActiveObject subclass: #Printer
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Tutorial-Examples'!

!Printer methodsFor: 'script'!

reply: v
  Transcript show: '> ' , v printString; cr! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --"!

!Printer class methodsFor: 'initialization'!

initialize
  Smalltalk at: #Print put: self new! !
```

© 1994 J.-P. Briot

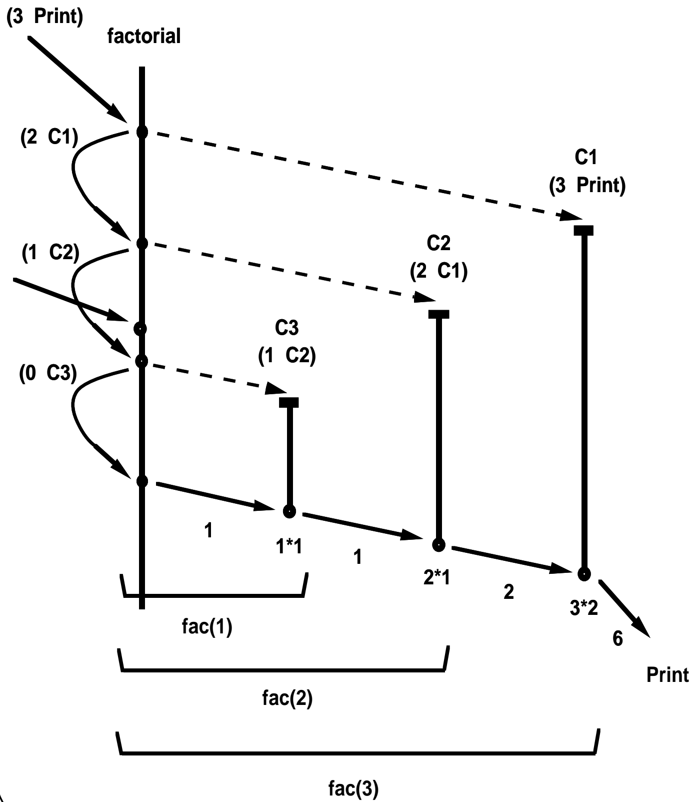
Calcul de la Factorielle avec Continuations

Le calcul récursif de factorial(n) est séparé entre:
* l'appel récursif de factorial(n-1),
* la multiplication de cette valeur par n et le retour en valeur (au client courant)

Cette seconde tâche est déléguée à une continuation qui encapsule:
le reste du calcul
(multiplier factorial(n-1) par n, et retourner le résultat au client courant),
le contexte du calcul courant nécessaire à sa complétion ensuite
(n et le client courant)

© 1994 J.-P. Briot

Factorielle



Factorial in Scheme (lexical scope Lisp)

```
; factorial with recursion

(define (factorial-r n)
  (if (= n 0)
    1
    (* (factorial-r (- n 1)) n)))
```

```
? (factorial-r 10)
= 3628800
```

```
; factorial with continuation

(define (factorial-c n c)
  (if (= n 0)
    (c 1)
    (factorial-c (- n 1)
                 (lambda (v) (c (* v n))))))
```

```
? (factorial-c 10 (lambda (v) v))
= 3628800
```


classe Factorial

```
ActiveObject: #Factorial
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!Factorial methodsFor: 'script'!

n: n replyTo: r
  n = 0
    ifTrue: [r reply: 1]
    ifFalse: [aself n: n-1 replyTo:
      (FactorialContinuation new n: n r: r) active]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --"!

!Factorial class methodsFor: 'example'!

example
  "Factorial example"
  | aFac |
  aFac := Factorial new active.
  aFac n: 10 replyTo: Print.
  aFac n: 5 replyTo: Print! !
```

Print is a predefined active object which represents the display (Transcript window)

classe FactorialContinuation

```
ActiveObject subclass: #FactorialContinuation
  instanceVariableNames: 'n r '
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!FactorialContinuation methodsFor: 'initialization'!

n: anInteger r: aContinuation
  n := anInteger.
  r := aContinuation! !

!FactorialContinuation methodsFor: 'script'!

reply: v
  r reply: n * v! !
```

Avantages

<u>transmission asynchrone + continuation</u>
vs
appel fonctionnel synchrone

divergence
ex: factorial(-1)
l'objet reste disponible,
mais tôt ou tard les ressources seront épuisées (plus de mémoire!)

deadlock (étreinte fatale)
une récursion n'entraine plus systématiquement un deadlock

Concurrence

Mais le calcul d'une factorielle reste en définitive séquentiel (et plus lent qu'un algorithme séquentiel)!
La concurrence n'intervient qu'en cas de calculs simultanés

Nous allons donc voir un deuxième algorithme qui décompose le calcul en sous-calculs concurrents

Le principe est celui de la décomposition d'un calcul en sous-calculs exécutés simultanément avec ensuite recomposition des différents résultats partiels

Cette stratégie s'appelle:
<u>divide and conquer</u>

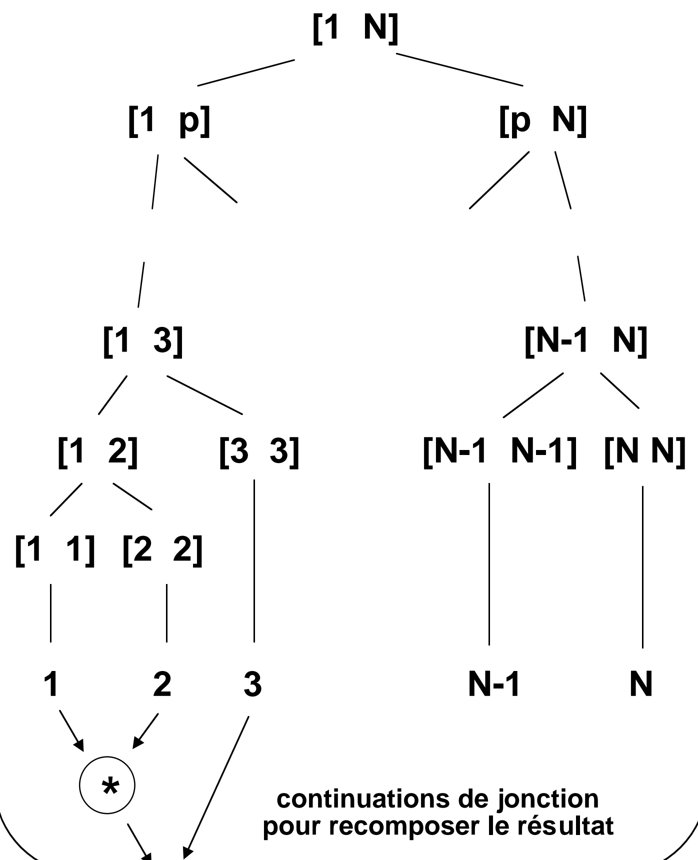
Factorielle par Divide and Conquer

Le calcul de N! se réduit à la multiplication des nombres appartenant à l'intervalle [1 N]

Ces multiplications peuvent intervenir simultanément. L'idée est donc de diviser l'intervalle en deux parties:
[1 pivot] et [pivot+1 N]
et d'opérer les deux sous calculs de manière concurrente

Le calcul est donc récursif et crée donc deux nouveaux objets actifs chargés de traiter les deux sous calculs avec comme destinataire commun une continuation (dite de jonction ou de jointure, *join continuation*) qui multipliera les deux résultats partiels

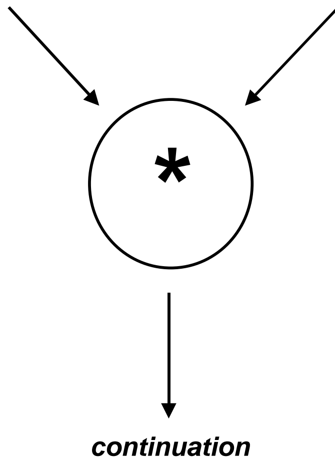
Factorielle par Divide and Conquer



Divide and Conquer et Continuations de Jonction

Les continuations de jonction (*join continuations*) resynchronisent les sous-calculs et combinent leurs sous-résultats

Une continuation de jonction est équivalente à un opérateur de data-flow, qui "attend" deux (ou plus) valeurs à recombiner, et envoie le résultat à la continuation enveloppante



continuation

classe MultiplyInRange

```
ActiveObject subclass: #MultiplyInRange
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: ' Tutorial-Examples'!


!MultiplyInRange methodsFor: 'script'!

from: i to: n replyTo: r
   | jc mid |                          "[i  n]"
   i = n ifTrue: [r reply: i]
      ifFalse:
      [jc := (MultiplyJoinContinuation new r: r) active.
       mid := (i+n)//2.
       MultiplyInRange new        "[i  mid]"
          from: i to: mid replyTo: jc.
       MultiplyInRange new        "[mid+1  n]"
          from: mid+1 to: n replyTo: jc]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

MultiplyInRange class
   instanceVariableNames: ''!


!MultiplyInRange class methodsFor: 'example'!

example
   "MultiplyInRange example"
   MultiplyInRange new active
      from: 1 to: 10 replyTo: Print! !
```


classe MultiplyJoinContinuation

```
ActiveObject subclass: #MultiplyJoinContinuation
   instanceVariableNames: 'v1 c '
   classVariableNames: ''
   poolDictionaries: ''
   category: ' Tutorial-Examples'!


!MultiplyJoinContinuation methodsFor: 'initialization'!

c: aContinuation
   c := aContinuation! !

!MultiplyJoinContinuation methodsFor: 'script'!

reply: v
   v1 isNil              "if no value received yet"
      ifTrue: [v1 := v] "memorize first value"
      ifFalse: [c reply: v1*v]! ! "otherwise, compute"
```

The behavior of the join continuation changes serially:

1) memorize value

2) combine (multiply) it with the first value and return the result to the continuation

A test is necessary to check if a first value has already been accepted. This problem will be solved gracefully in the actor model of computation (see Part VII).

Algorithmique/Résolution de Problèmes Concurrents

On peut diviser la majorité des algorithmes concurrents de résolution de problèmes en trois grandes classes (G. Agha):

<u>divide and conquer</u>
le problème est décomposé en sous problèmes, les sous-solutions sont ensuite recombinées

<u>pipeline</u>
le problème est décomposé en un enchaînement de calculs à travers lequel les données circulent

<u>coopération</u>
le problème est décomposé en un ensemble d'entités (agents) qui vont elles-mêmes décider de la répartition et la coordination de leurs tâches
(voir multi-agents, §IX)

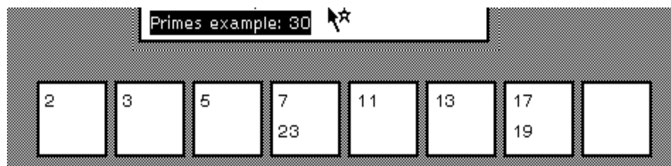
Pipeline: Calcul des Nombres Premiers

On détermine les nombres premiers à l'aide d'une chaîne ordonnée de filtres

Chaque filtre correspond à un nombre premier déjà "trouvé"

Les entiers successifs sont envoyés dans le "pipeline" des filtres, chaque filtre teste s'il est un diviseur de l'entier à tester

Si un entier atteint le dernier filtre avec succès, un nouveau nombre premier a été déterminé, le filtre associé sera alors ajouté en fin de chaîne



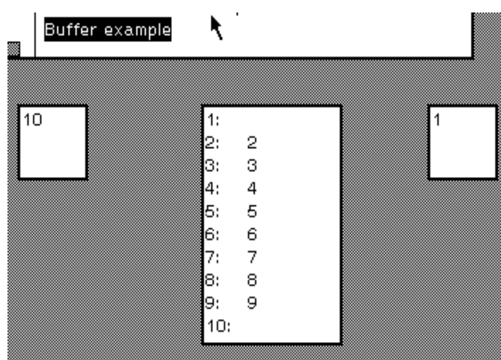
classe PrimeFilter

```
ActiveObject subclass: #PrimeFilter
  instanceVariableNames: 'n next '
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!PrimeFilter methodsFor: 'initializing'!

n: aPrimeNumber
  n := aPrimeNumber! !

!PrimeFilter methodsFor: 'script'!

filter: i
  i \\ n = 0                    "if i is not divided by n"
    ifFalse: [next isNil        "if end of the chain"
        "a new prime number is added to the chain"
      ifTrue: [next := (PrimeFilter new n: i) active]
        "otherwise pass the test to next in the chain"
      ifFalse: [next filter: i]]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --"!

!PrimeFilter class methodsFor: 'example'!

checkUntil: max
  "PrimeFilter checkUntil: 50"
  | two |
  two := (PrimeFilter new n: 2) active.
  2 to: max do: [:i | two filter: i]! !
```

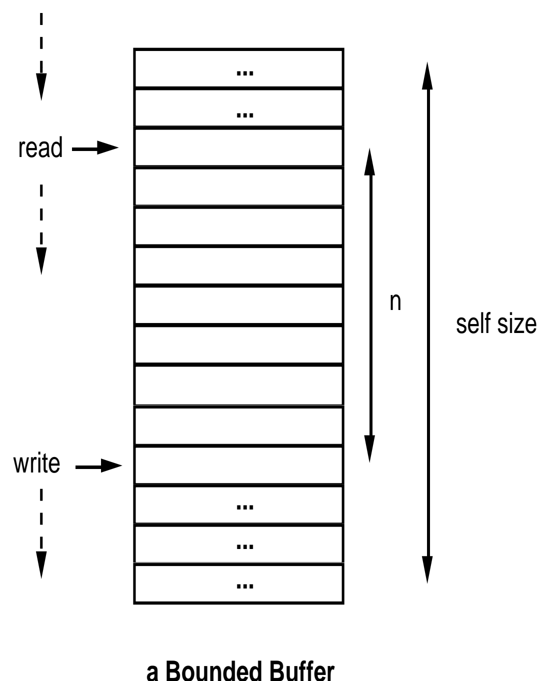

Synchronisation: Producteur/Consommateur avec un Buffer Borné

Un producteur et un consommateur échangent des données à travers un buffer (tampon) borné

Le problème est de synchroniser la production et la consommation à la disponibilité du buffer (par ex: suspendre les requêtes d'ajout tant que le buffer est plein)



Implémentation du Buffer



classe BoundedBufferActivity

(modèle de synchronisation Abstract States)

```
AbstractStatesActivity subclass: #ASBBActivity
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!ASBBActivity methodsFor: 'abstract states'!

empty
  ^#(put:)!

full
  ^#(get)!

initialAbstractState
  ^#empty!

partial
  ^(self empty) + (self full)! !

!ASBBActivity methodsFor: 'state transition'!

nextAbstractStateAfter: selector
  ^oself is empty
      ifTrue: [#empty]
      ifFalse: [oself isFull
                  ifTrue: [#full]
                  ifFalse: [#partial]]! !
```


classe Producer

```
ActiveObject subclass: #Producer
  instanceVariableNames: 'buffer delay '
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!Producer methodsFor: 'initializing'!

buffer: aBoundedBuffer delay: seconds
  buffer := aBoundedBuffer.
  delay: seconds! !

!Producer methodsFor: 'script'!

run: max
  1 to: max do: [:i |
    buffer put: i.
    (Delay forSeconds: delay) wait]! !
```


classe Consumer

```
ActiveObject subclass: #Consumer
  instanceVariableNames: 'buffer delay '
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!Consumer methodsFor: 'initializing'!

buffer: aBoundedBuffer delay: seconds
  buffer := aBoundedBuffer.
  delay: seconds! !

!Consumer methodsFor: 'script'!

run: max
  max timesRepeat:
    [buffer get.
     (Delay forSeconds: delay) wait]! !
```


Partie VI.

Actalk: Implémentation des Objets Actifs en Smalltalk-80

- Buts et Architecture de la Plateforme Actalk
- Implémentation des Objets Actifs Sérializés
- Implémentation de la Transmission de Message Asynchrone

La Plateforme Actalk

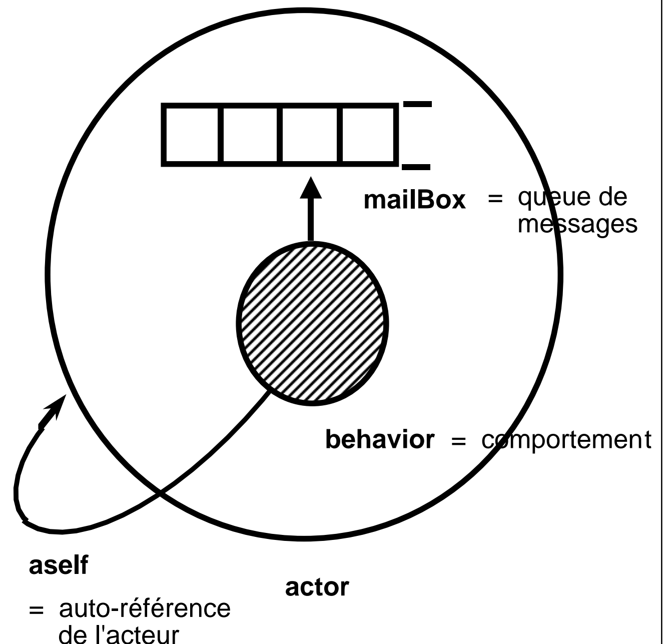
Une plateforme de modélisation, classification, et expérimentation avec les langages concurrents à objets dans un environnement de programmation unifié

Le noyau de la plateforme représente une implémentation minimale des objets actifs sérializés communiquant par transmissions asynchrones (i.e., de type acteur) en Smalltalk-80

Actalk signifie ainsi Acteurs en Smalltalk (J.-P. Briot)

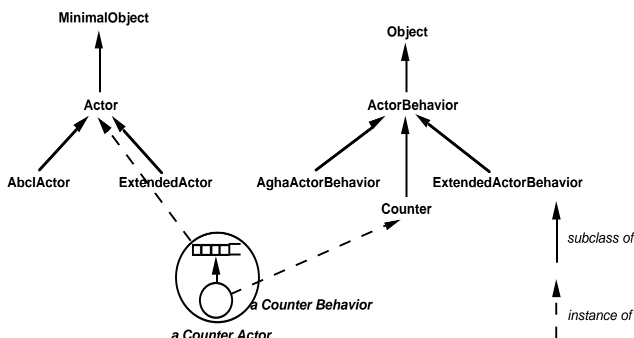
Le noyau d'Actalk est ensuite étendu pour simuler de nombreux langages et modèles de calcul concurrents à objets (Modèle des acteurs, ABCL/1, POOL...)

Représentation d'un Acteur en Actalk



Architecture d'Actalk

classe Actor structure des acteurs (queue de messages et comportement) et sémantique de la transmission asynchrone
classe ActorBehavior activité (processus) et sémantique du traitement des messages (séquentiel)



Le programmeur définit des classes de comportements d'acteurs comme sousclasses de ActorBehavior,
par exemple la classe Counter

Le concepteur (de langages) simule des classes de langages par des sousclasses des classes du noyau: Actor et ActorBehavior,
par exemple les classes AghaActorBehavior, AbclActor

Implémentation

<u>classe Actor</u>
* <u>structure des acteurs</u> variables d'instance mailBox et behavior
* <u>sémantique de la transmission asynchrone</u> méthode asynchronousSend:

<u>classe ActorBehavior</u>
* <u>activité</u> méthode setProcess installe un processus infini et en tâche de fond pour accepter séquentiellement les messages
* <u>acceptation des messages</u> méthode acceptNextMessage
* <u>création d'un acteur</u> méthode actor
* <u>auto-référence</u> variable d'instance aself permet la récursion

Des transmissions récursives asynchrones (envoyées à aself) peuvent être combinées avec des transmissions récursives standard "internes" (envoyées à self)

classe Actor

```
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'!

new
  ^super new initialize! !
```


classe ActorBehavior

```
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 perform: aMessage selector
       withArguments: aMessage arguments! !

!ActorBehavior methodsFor: 'actor creation'!

actor
  ^Actor new initializeBehavior: self! !
```


Synchronization de l'Acceptation de Messages

La classe SharedQueue assure implicitement:

l'exclusion mutuelle des acteurs émetteurs sur la queue de messages (méthode d'ajout de messages nextPut:)

la synchronisation (suspension) du processus du comportement de l'acteur qui ne peut accepter un nouveau message tant que la boîte aux lettres est vide (méthode de retrait de messages next)

Implémentation Transparente de la Transmission Asynchrone

Cette implémentation utilise
les capacités réflexives de Smalltalk:
si le message n'est pas reconnu par l'acteur (classe Actor), ceci déclenche l'envoi du message d'erreur doesNotUnderstand: avec le message réifié pour argument

La méthode doesNotUnderstand: a été redéfinie dans la classe Actor pour renvoyer le message de manière asynchrone:

```
!Actor methodsFor: 'error handling'!
doesNotUnderstand: aMessage self asynchronousSend:
  aMessage! !
```

La classe MinimalObject (surclasse de Actor) minimise les conflits de nom éventuels entre messages envoyés aux acteurs et méthodes de la classe Object. MinimalObject implémente l'ensemble mimimal de méthodes système de la classe Object

Le message sera ultimement accepté par le comportement de l'acteur, et perform: va réinstaller et reprendre l'exécution du message, mais après que le receveur et le type de transmission aient été modifiés

Partie VII.

Le Modèle de Calcul des Acteurs

Buts et spécificités

Le concept de remplacement de comportement exemple: fibonacci et continuations de jonction

Réinterprétation des concepts de sérialization et de future

Insensitivité exemple: compte bancaire avec sauvegarde

Architectures acteur

Le Modèle de Calcul des Acteurs

Concepts:
- inventé par Carl Hewitt,
- modèle de calcul précisé par Gul Agha

Buts:
- modéliser les systèmes parallèles répartis et ouverts

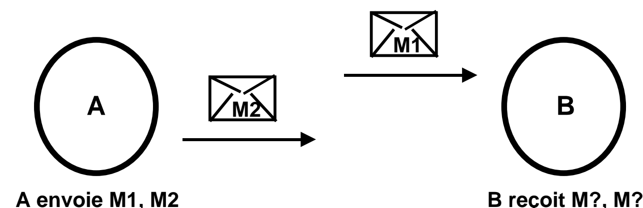
Spécificités de conception:
- concurrence maximale (et par défaut) (inter-objets, intra-object, intra-méthode)
- indépendance par rapport à un type d'architecture spécifique
- non déterminisme équitable
- asynchronisme
- localité (pas de données globales)
- ouverture: compositionnalité, extensibilité et reconfigurabilité dynamiques (configurations, acteurs externes, réceptionnistes)
- objets partagés avec changements d'état sans affectation (remplacement de comportement)

Non Déterminisme

De manière à être indépendant d'un type d'architecture de machine parallèle spécifique, le modèle de calcul des acteurs est non déterministe

Non déterminisme de:
- séquencement des activités
- ordre d'arrivée des messages

En particulier, aucune hypothèse n'est faite sur la conservation de l'ordre des messages lors de leur transmission:
par exemple, si un acteur A envoie successivement les messages M1 et M2 à un acteur B, on ne peut prédire si cet ordre va être conservé à la réception

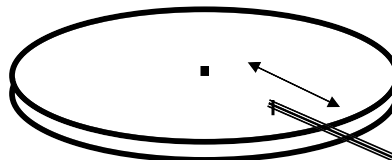


A envoie M1, M2 B reçoit M?, M?

Certains réseaux informatiques ne garantissent pas la conservation de l'ordre des messages, ainsi Arpanet

Non Déterminisisme et Equité

Métaphore (W. Clinger):
un algorithme contrôle les mouvements du bras d'un disque en fonction des demandes d'accès aux pistes



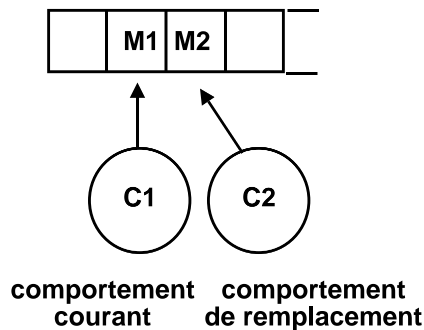
Un algorithme naif:
pour minimiser les mouvements, accéder en premier lieu à la plus proche des pistes demandées

Mais cet algorithme n'est pas <u>équitable</u>:
une requête peut être indéfiniment repoussée, par exemple en cas de séquence:
pistes n, n-1, n, n-1...,
la piste 0 ne sera jamais atteinte

L'arbitre qui sérialise les messages arrivant dans la boîte aux lettres d'un acteur doit être équitable.
Ainsi il y a <u>garantie de délivrance</u> des messages dans un temps fini

Le Concept de Remplacement de Comportement

Pendant l'exécution d'un message, le comportement courant de l'acteur spécifie le <u>comportement de remplacement</u>, c'est-à-dire le comportement qui acceptera le message suivant, autrement dit *comment* traiter le message suivant



un comportement ne traite qu'un seul message

Puissance d'Expression

Le concept de comportement de remplacement unifie:

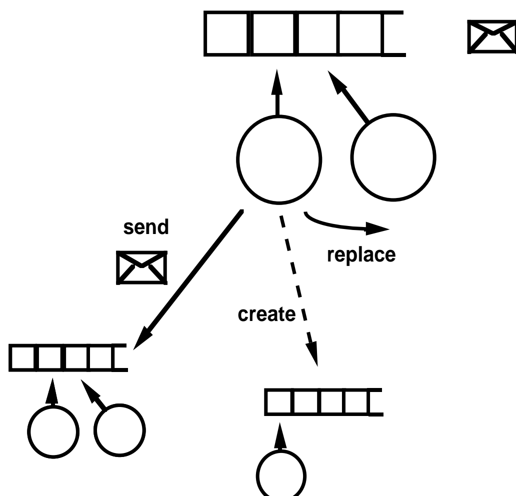
changement d'état
le comportement de remplacement peut être une copie du comportement courant, manifester un changement d'état, ou encore un changement quelconque

concurrence intra-objet
dès que le comportement de remplacement est spécifié, le traitement du message suivant *peut débuter*

synchronisation
tant que le comportement de remplacement n'est pas spécifié, le message suivant *ne peut être accepté*

Les 3 Actions d'un Comportement

send envoyer un message
create créer un acteur
replace spécifier le comportement de remplacement



Un acteur peut prendre une décision de choix entre plusieurs actions
Cette décision ne peut être basée que sur sa connaissance locale

Simulation du Concept de Remplacement de Comportement en Actalk: classe AghaActorBehavior

La classe AghaActorBehavior
redéfinit la méthode setProcess pour accepter un message et un seul
définit la méthode replace: pour spécifier le comportement de remplacement

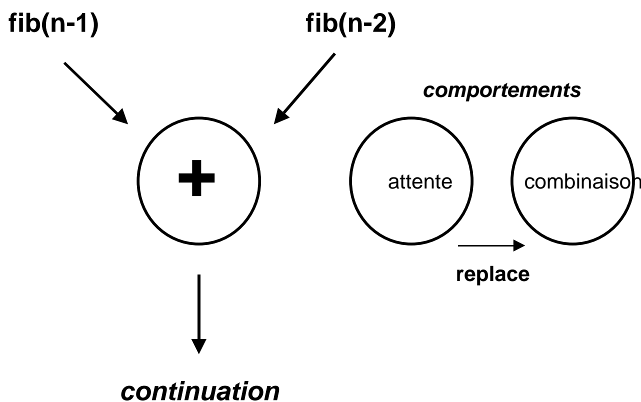
Ainsi on passe d'un objet actif sérialisé à un objet actif concurrent

```
ActorBehavior subclass: #AghaActorBehavior
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Actalk-Extension-Actor'!

!AghaActorBehavior methodsFor: 'initialization'!

setProcess
  [self acceptNextMessage] fork! !

!ActorBehavior methodsFor: 'behavior replacement'!

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

Exemple de Changement Complet de Comportement: Retour aux Continuations de Jonction

La succession des deux comportements, de mémorisation et de combinaison, va pouvoir s'exprimer naturellement, sans avoir recours à des tests (voir les continuations de jonction pour le calcul de la factorielle, §V).

Exemple: fibonacci



Exemple: classe Fibonacci

```
ActiveObject subclass: #Fibonacci
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!Fibonacci methodsFor: 'script'!

n: n replyTo: c
    | join |
  n < 2
    ifTrue: [c reply: 1&]
    ifFalse: [join := (BinaryAdder new c: c).
              self n: n-1 replyTo: join&.
              self n: n-2 replyTo: join&]! !
```


Les Deux Comportements de la Continuation Additionneur

```
AghaActorBehavior subclass: #BinaryAdder
  instanceVariableNames: 'c '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Join-Continuations'!

!BinaryAdder methodsFor: 'initialization'!

c: aContinuation
  c := aContinuation! !

!BinaryAdder methodsFor: 'script'!

reply: v
  self replace: (UnaryAdder new v1: v c: c)! !
```

```
AghaActorBehavior subclass: #UnaryAdder
  instanceVariableNames: 'v1 c '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Join-Continuations'!

!BinaryAdder methodsFor: 'initialization'!

v1: anInteger c: aContinuation
  v1 := anInteger. c := aContinuation! !

!UnaryAdder methodsFor: 'script'!

reply: v2
  c reply: v1 + v2&! !
```


Serialized/Unserialized

Un acteur dont le comportement est constant est appelé un acteur non sérializé (unserialized)

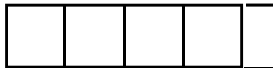
Il peut traiter ses requêtes simultanément sans aucune restriction

Il peut être recopié autant que désiré pour pouvoir exploiter cette concurrence sur des multi-processeurs (il est immutable)

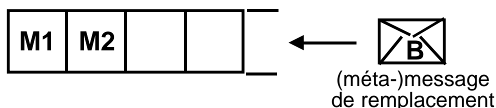
Un acteur non sérializé est équivalent à une fonction

Calcul Externe du Comportement de Remplacement: Futures

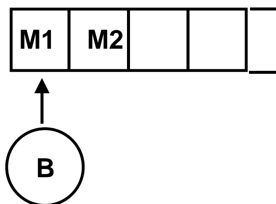
Un acteur <u>future</u> est un nouvel acteur qui n'a pas encore de comportement, il sera calculé de manière <u>externe</u>



Quand la valeur (le comportement) de ce "futur" acteur est déterminée, le comportement résultant est envoyé à l'acteur par un <u>(méta-)message de remplacement</u>



L'acteur (ex-future) peut alors commencer à accepter les messages (éventuellement déjà en attente)



Autres Applications du Calcul Externe de Comportement:

Acteur Externe

Le programmeur peut <u>composer</u> des modules (appelés des <u>*configurations*</u>) d'acteurs <u>dynamiquement</u>

Cela implique que les représensants (interface) d'autres configurations ne sont pas encore connus *(système ouvert)*

Ces représentants sont appelés <u>acteurs externes</u> et constituent l'interface de sortie (les acteurs qui constituent l'interface d'entrée sont appelés <u>réceptionnistes</u>)

Les acteurs externes sont implémentés par des acteurs futures dont les comportements seront spécifiés au moment de la composition quand les autres configurations qu'ils représentent seront connues; en attendant ils vont stocker les messages

Insensitivité

Prenons le cas d'un acteur qui a besoin d'une information non locale pour pouvoir spécifier son comportement de remplacement

mais d'un autre côté, il ne peut accepter cette information avant d'avoir spécifié son comportement de remplacement

cercle vicieux!

Concept d'Insensitivité

Ce cercle vicieux est résolu par le concept d'<u>insensitivité</u>, application directe du principe de calcul externe du comportement de remplacement

L'acteur ne spécifie pas de comportement (en conséquence il ne peut accepter le prochain message),

le comportement de remplacement sera plus tard calculé de manière externe, envoyé (sous forme d'un méta-message de remplacement), et accepté par l'acteur insensitif,

le remplacement de comportement alors effectué, l'acteur redeviendra sensitif, c'est-à-dire pourra à nouveau accepter le message suivant

Exemple: Compte Bancaire avec Protection

Un compte bancaire possède un compte de protection,

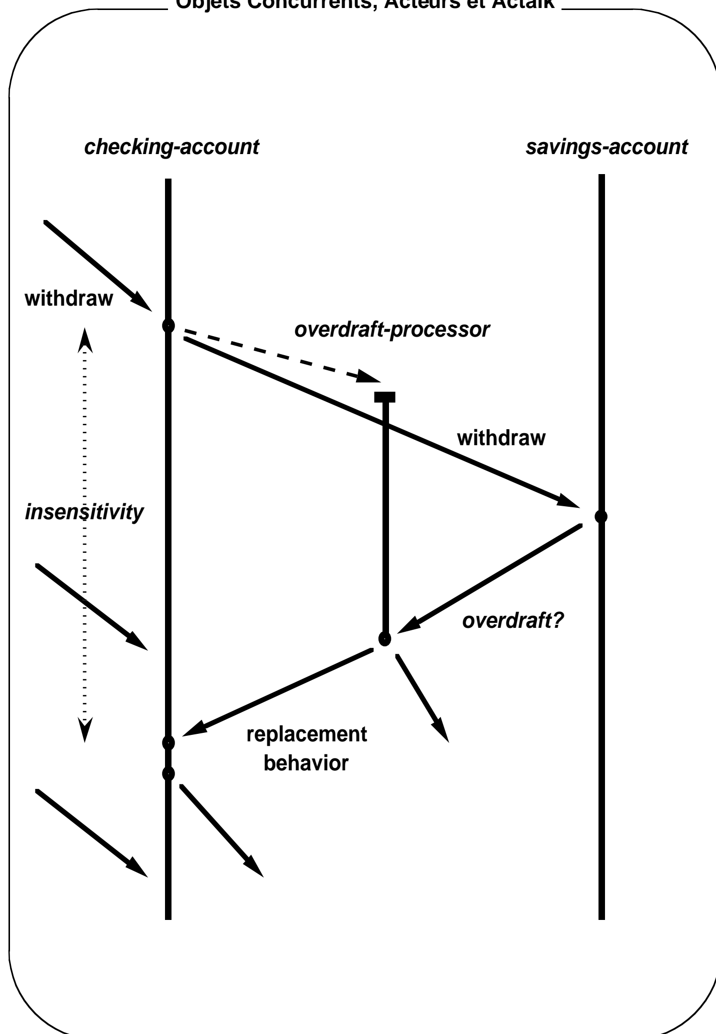
si un retrait entraine un découvert, le compte principal va vérifier si le compte de protection peut couvrir le montant du découvert,

l'état résultant du compte principal (c'est-à-dire son comportement de remplacement) dépend de la réponse du compte de protection (couverture du découvert ou non?)

En conséquence, le compte principal devient insensitif,

il crée également une continuation de traitement du découvert,

cette continuation aura pour tâche de calculer le comportement de remplacement à partir de la réponse du compte de protection, et de l'envoyer au compte principal qui pourra alors redevenir sensitif



Identification de Messages

Supposons que par erreur de programmation, un autre méta-message de remplacement de comportement atteigne un acteur insensitif

Le problème est de distinguer parmi de multiples messages de même sélecteur et de même cible

solutions:

utiliser une clé (numérique ou symbolique)

créer un nouvel acteur et l'utiliser comme un nouvel identificateur garanti unique

Dans l'exemple, la continuation (overdraft-processor) est utilisée comme clé par l'acteur insensitif (checking-account) pour authentifier le (méta-)message de remplacement

Architectures Acteur

Les Langages de programmation acteur (par ex: Pract/Acore de C. Manning, MIT) ont en général (au moins) deux niveaux:

continuations explicites (niveau de base)
transmission de message asynchrone et continuations explicites

continuations implicites (haut niveau)
transmission de message bidirectionnelle (réponse implicite) comme en programmation par objets classique
le compilateur génère des continuations, et de l'insensitivité si une communication est nécessaire pour spécifier le comportement de remplacement

Le compilateur a aussi pour tâche d'implémenter de manière efficace la concurrence à l'aide de séquence, pseudo-parallélisme, et parallélisme

Dans des implémentations monoprocesseur, la concurrence sera réduite (par exemple, le compilateur peut optimiser le remplacement de comportement en sérialisant arbitrairement pour pouvoir générer des affectations sur le même comportement ainsi réutilisé)

Partie VIII.

Comparaison des Principaux Langages Concurrents à Objets

ConcurrentSmalltalk Y.Yokote et M. Tokoro

Modèle de calcul des Acteurs G. Agha et C. Hewitt et al.

ABCL/1 A. Yonezawa et al.

POOL P. America

Concurrent Eiffel D. Caromel

Concurrent Prolog E. Shapiro

Filtre / Perspectives

uniforme / hybride

transmission asynchrone
synchrone
anticipée (future)

concurrence intra-objet?

synchronisation intra-objet?

réactif / activité autonome

acceptation des messages implicite / explicite

priorités de messages?

conservation de l'ordre des messages?

post-actions?

ConcurrentSmalltalk

uniforme / **hybride**
objets Smalltalk-80 standard (passifs)
transmission **asynchrone**
synchrone
anticipée (future)

concurrence intra-objet? **non**
pseudo-concurrence seulement
synchronisation intra-objet? **oui**
sur conditions (mySecretary relinquish: <Block>)
réactif / activité autonome

acceptation des messages **implicite** / explicite

priorités de messages? **non**

conservation de l'ordre des messages? **oui**

post-actions? **oui**
en utilisant l'acknowledgement: ^^<expression>

Modèle (de calcul) des Acteurs

uniforme / hybride

transmission **asynchrone**
synchrone *à plus haut niveau*
anticipée (future) *à plus haut niveau*
compilées en continuations et insensitivité
concurrence intra-objet? **oui**
remplacement de comportement
synchronisation intra-objet? **oui**
seulement à l'acceptation par le remplacement de comportement
réactif / activité autonome

acceptation des messages **implicite** / explicite

priorités de messages? **non**

conservation de l'ordre des messages? **non**
non déterminisme équitable
post-actions? **oui**
implicites, car l'envoi d'une réponse est explicite

Modèle (de calcul) des Acteurs

sémantique pure (pas d'affectation)

indépendant d'une architecture spécifique

très grande concurrence et à grain très fin

grand non déterminisme

mais

bas-niveau (*assembleur pour giga-processeurs*)

le programmer a besoin de constructions de plus haut niveau,
cela amène à une architecture multi-couches,
mais par voie de conséquence, il est alors non trivial de conserver une bonne corrélation entre spécification et exécution, notamment pour la mise au point

ABCL/1

uniforme / **hybride**
structures de données et routines internes (Lisp)
transmission **asynchrone**
synchrone
anticipée (future)

concurrence intra-objet? **non**
quasi-concurrence seulement
synchronisation intra-objet? **oui**
mode d'attente de message (wait-for)
réactif / activité autonome

acceptation des messages **implicite** / **explicite**
attente de message explicite (wait-for)
priorités de messages? **oui**
mode de transmission express (interruption)
conservation de l'ordre des messages? **oui**

post-actions? **oui**
implicites, car l'envoi d'une réponse est explicite

© 1994 J.-P. Briot

ABCL/1

calcul hybride
(programmation concurrente par objets et plus traditionnelle)

granularité moyenne

communication de haut niveau
(types, contraintes...)

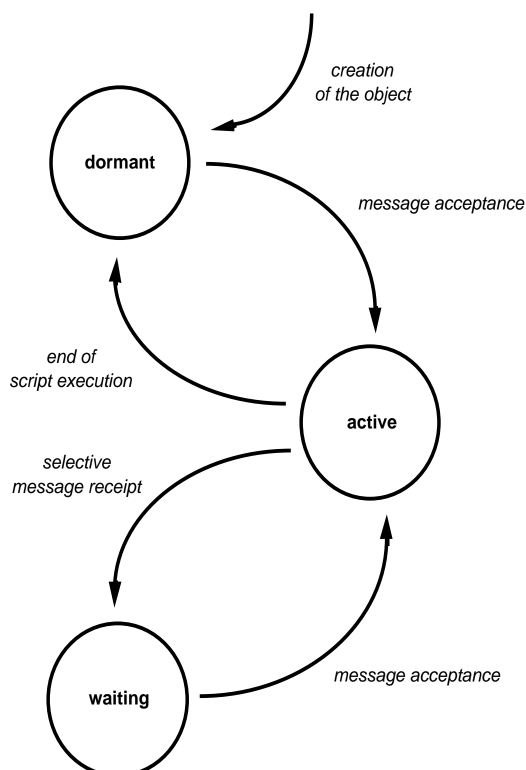
mode de transmission de messages en priorité

concurrence et non déterminisme borné

Ces choix facilitent la modélisation conceptuelle et la réutilisation de modules logiciels plus conventionnels.
Ils facilitent également des implémentations efficaces sur les architectures courantes.
Mais ces choix complexifient l'implémentation et rendent la sémantique du langage difficile à formaliser

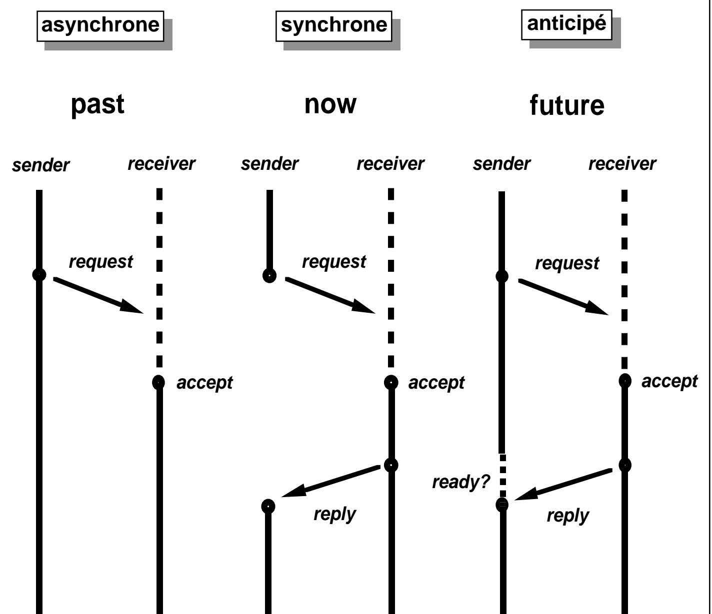
© 1994 J.-P. Briot

Les 3 Modes d'Activité d'un Objet



© 1994 J.-P. Briot

Les 3 Types de Transmission



Les types de transmission now et future peuvent être décomposés en types past avec mode waiting

© 1994 J.-P. Briot

POOL

uniforme / hybride
mais objets primitifs sans activité et sans transmission de message
transmission asynchrone
synchrone
anticipée (future)

concurrence intra-objet? **non**

synchronisation intra-objet? **oui**
seulement par acceptation de message
réactif / **activité autonome**
concept de *"body"*
acceptation des messages implicite / **explicite**
commande ANSWER
priorités de messages? **non**

conservation de l'ordre des messages? **oui**

post-actions? **oui**
après le mot-clé POST

© 1994 J.-P. Briot

POOL: Le Concept de "Body"

En POOL la concurrence nait de l'activité propre et indépendante de chaque objet, plutôt qu'initiée par des transmissions asynchrones

En conséquence:

 # transmission synchrone

 # activité autonome (*body*) et acceptation explicite des messages (ANSWER)

Une implémentation sur une multi-processeur à mémoire répartie spécifique (DOOM) a été réalisée

© 1994 J.-P. Briot

Eiffel //

uniforme / **hybride**
seules les sousclasses de la classe PROCESS génèrent des objets actifs
transmission **asynchrone**
synchrone
anticipée (future)

concurrence intra-objet? **non**

synchronisation intra-objet? **oui**
seulement par acceptation de message
réactif / **activité autonome**
routine Live
acceptation des messages implicite / **explicite**
routine serve
priorités de messages? **non**

conservation de l'ordre des messages? **oui**

post-actions? **non**

© 1994 J.-P. Briot

Programmation Concurrente par Objets en Programmation Concurrente Logique

objet
= processus perpétuel se réincarnant

identité de l'objet
= canal de communication
= variable logique partagée
= [FirstMessage | QueueOfMessages]

plus proche de la programmation par acteurs que de la programmation logique
(le retour arrière automatique et l'unification complète sont perdus)

© 1994 J.-P. Briot

Exemple: un Compteur

terminaison

```
Counter([],Contents).

Counter([reset | QueueMessages], Contents)
    :- Counter(QueueMessages?, 0).

Counter([incr | QueueMessages], Contents)
    :- plus(Contents, 1, NewContents),
       Counter(QueueMessages?, NewContents).
```

message incomplet

```
Counter([contents(Contents) | QueueMessages],
    Contents)
       :- Counter(QueueMessages?, NewContents).
```

exemple:

```
Counter(Channel, 100).
Channel = [incr, incr, contents(X)].
```

solution:

X = 102

© 1994 J.-P. Briot