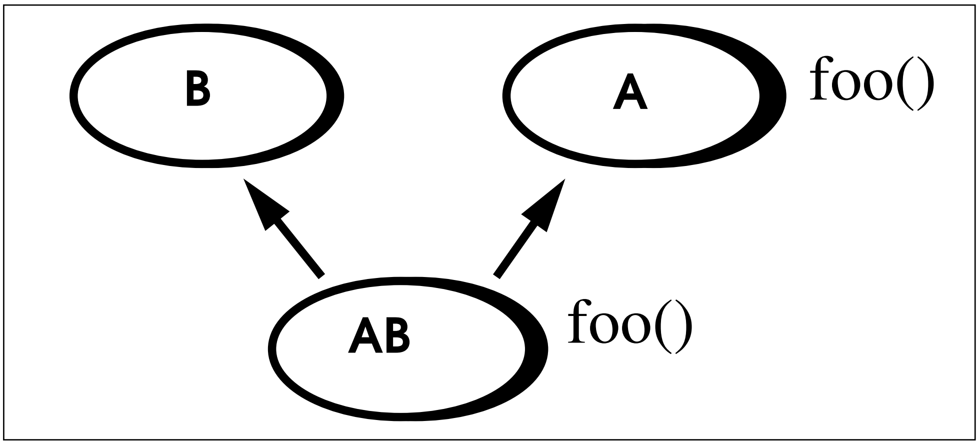
CHAPITRE 5 Programmation à Objets: Modèles et Principes de base

5.1 Programmation à objet séquentielle

Langage de classes, typé

5.1.1 Héritage



Simple, multiple, répété

Interface (Java), Délégation

Redéfinition (overriding ou overwriting), surcharge (overloading), Renommage

Co-variance, No-variance, contra-variance

5.1.2 Polymorphisme

Type statique, Type Dynamique (entité, variable)

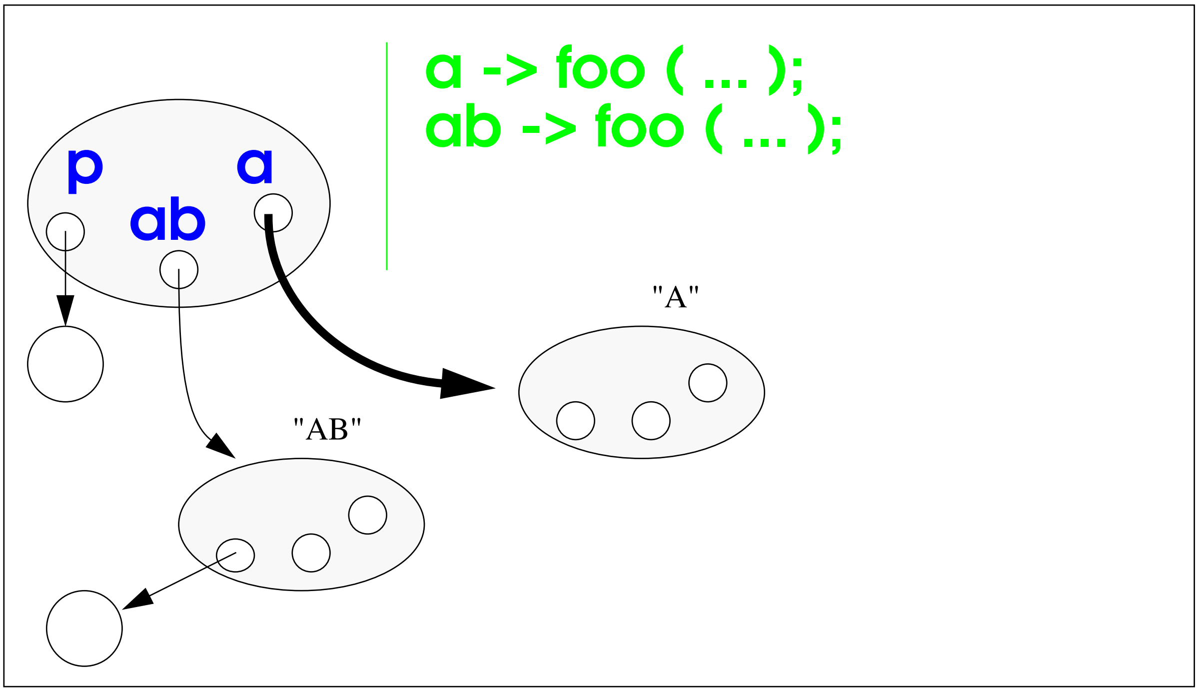
Type des objets (ni statique ni dynamique)

Affectation polymorphe

Affectation polymorphe inverse (?=) ou cast dynamique

5.1.3 Liaison dynamique

Le corps de la routine appelée est décidé dynamiquement, en fonction du type dynamique de la variable.

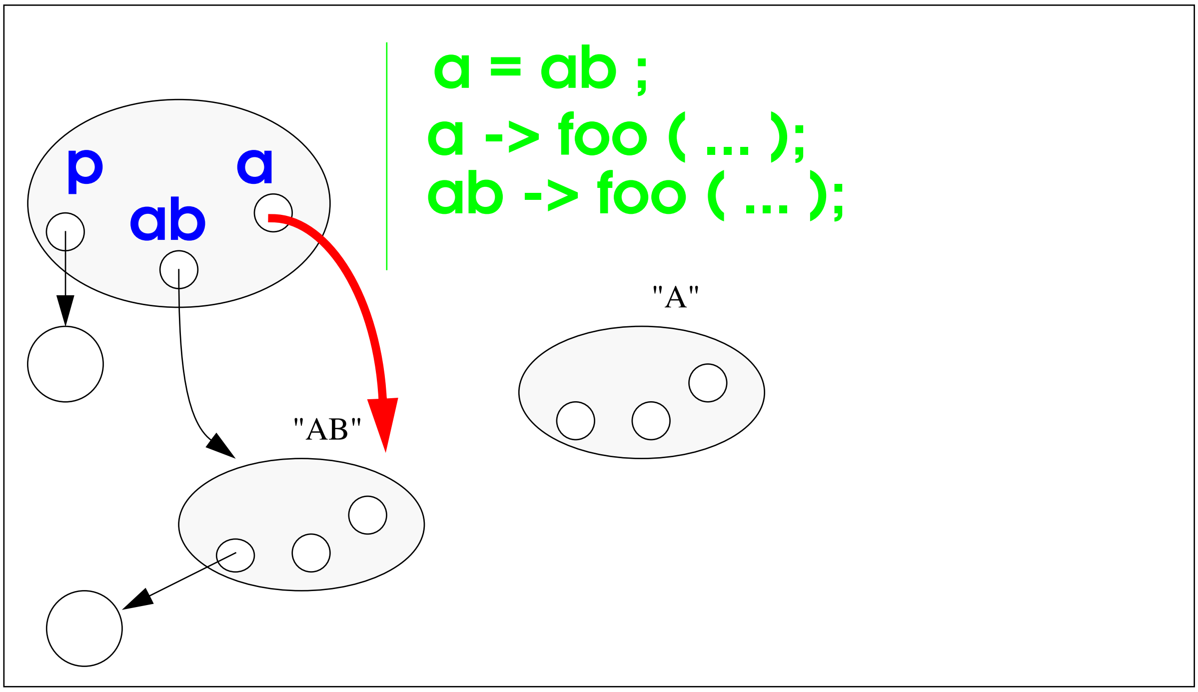


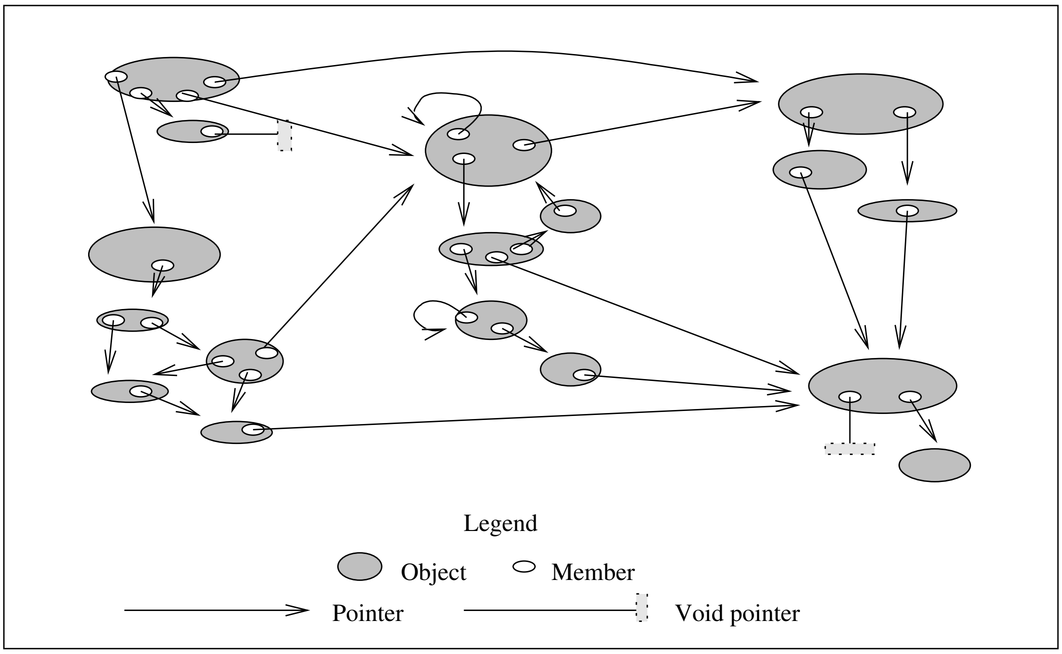
Après affectation polymorphe:

5.1.4 Système séquentiel

Une seule activité (“processus”, “thread”):

l’objet racine du système

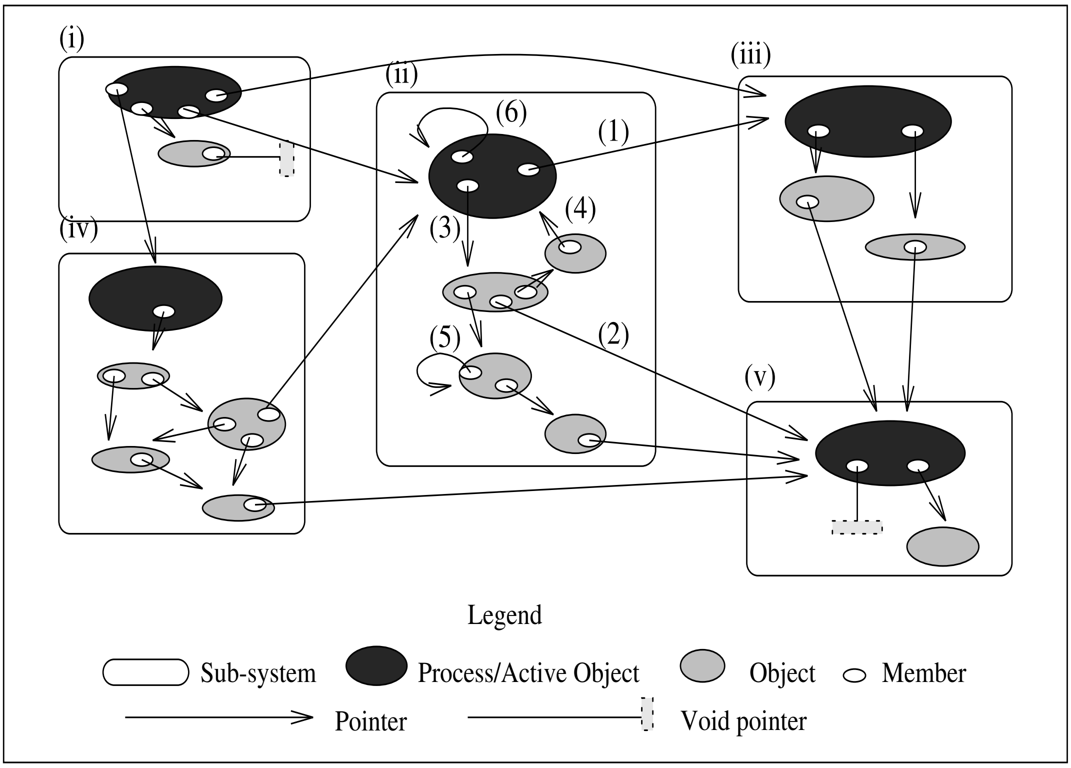




Appel de routine: l'activité "passe" d'un objet à l'autre (pile d'exécution)

On parle quelquefois “d’envoi de messages” pour les langages à objets séquentiels (e.g. Smalltalk), mais c’est un abus de langage car la sémantique demeure celle d’un appel de routine.

Question: comment passer à plusieurs activités ??



5.2 Objets actifs et Parallélisme

Une solution qui nous semble très peu “à objets”:

5.2.1 Parallélisme orthogonal aux objets

Les activités parallèles sont dissociées des objets (elles sont orthogonales aux objets).

Ces solutions sont en général fortement inspirées de la notion de moniteur + Threads

C’est par exemple le cas de Java, ou encore Hybrid, ConcurrentSmalltalk

5.2.2 Parallélisme par objet actif

Chaque activité est associée à un objet actif
Une activité *est* un objet (actif)

Solutions plus novatrices, vrai paradigme de “parallélisme à objet”. Ce que nous allons étudier

Deux grandes catégories:

Modèles Uniformes, ou Homogènes:

Tous les objets sont actifs

e.g. Acteurs

Modèle non-uniforme

Seuls certains sont actifs

Attention:
la première solution est intéressante dans certains cas (parallélisme de plus bas niveau, Système d'Exploitation).
Cf. Java.

A partir de maintenant:
étude d'un modèle particulier:

Eiffel//, C++//, ProActive

Réutilisation (Seq. --> Parallèle, Répartie)

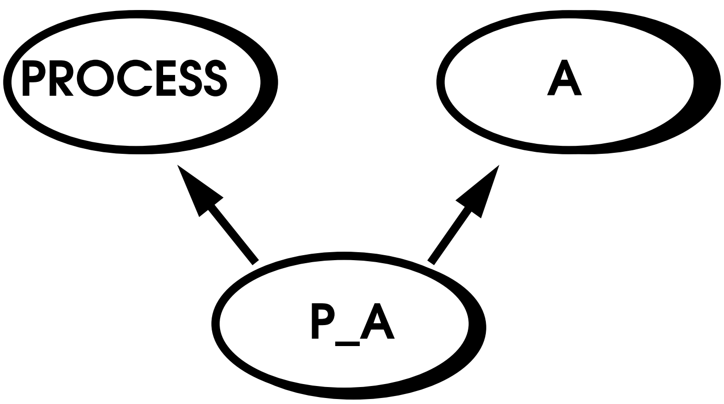
5.3 Dimensions fondamentales

Exécutions //, Communication, Synchronisation, partage, etc.

5.3.1 Exécutions parallèles

Processes are objects

Every object is not a process

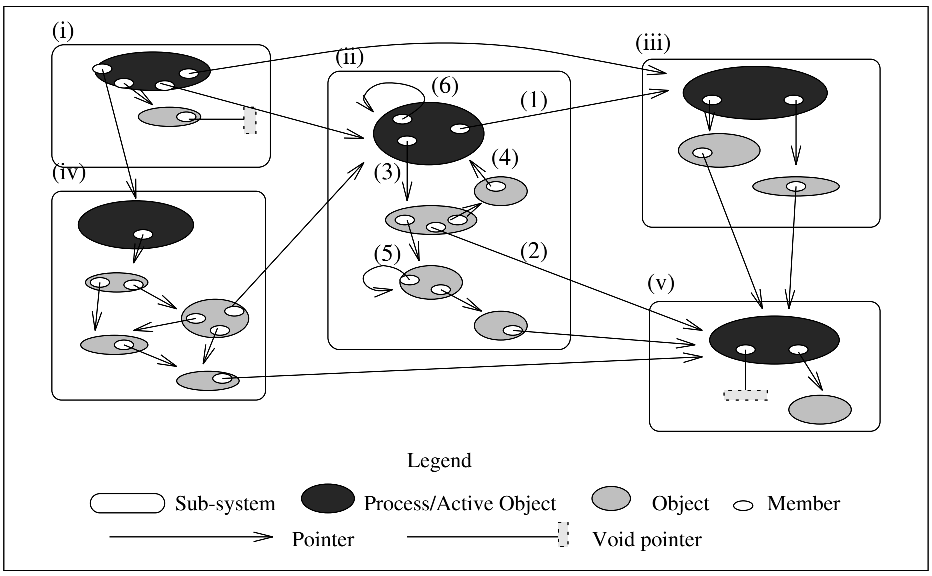



```
A* a;
P_A* p_a;

a -> g ( ... );
p_a -> g (... );
```

Use of inheritance for parallel activities
Directly or indirectly (class or Interface)

Active and passive objects:



Active objects:

Class-based:

Heir of a PROCESS class:

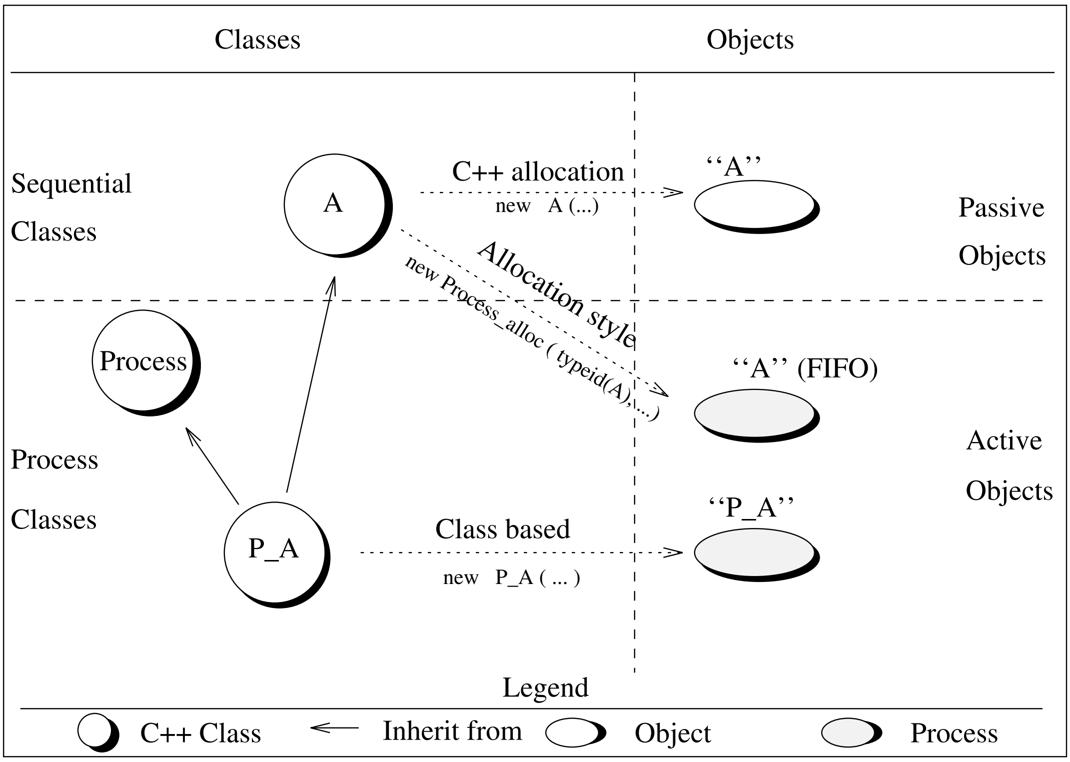
new P_A (...);

Allocation-based:

Instantiation of a seq. class:

new (typeid(A)) Process_alloc(...)

Classes, objects, active objects:



Sequential processes

---> one thread per object

Reuse:

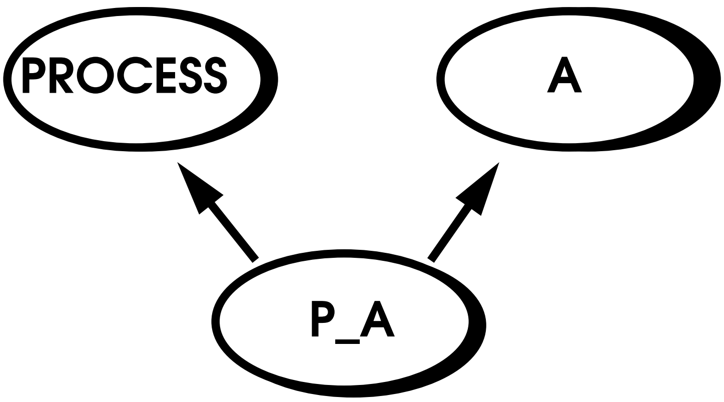
---> A a; ... P_A pa;

---> Polymorphism between objects and processes:
a = pa;

5.3.2 Communication

Communication semantics:

---> Method as IPC (entry point)
---> Systematically asynchronous towards active objects
---> FIFO order is respected




```
A* a;
P_A* p_a;

a -> g ( ... );
p_a -> g (... );
```

Decision at the class level, not the routine one

Structuring of the parallelism is based on the object structuring:

Asynchronous sub-systems

Reuse:

```
---> A a; ...                    P_A pa;
---> a.foo ();
---> ....
---> pa.foo(); // a remote call, a parallel call
               //   an asynchronous call
```

Default behavior:

A FIFO service of requests

5.3.3 Objets partagés

No shared passive objects

---> Call-by-value between processes,

Deep copy for both Calls and Returns

Passive objects only,
Active Objects keep a reference semantics

Initially : no shared object, then if none shared at step N, none at step N+1.

Sub-system structuring:

all asynchronous references are towards the root objects

--> Picture

Asynchronous reference from whithin a sub-system is OK.

5.3.4 Synchronisation

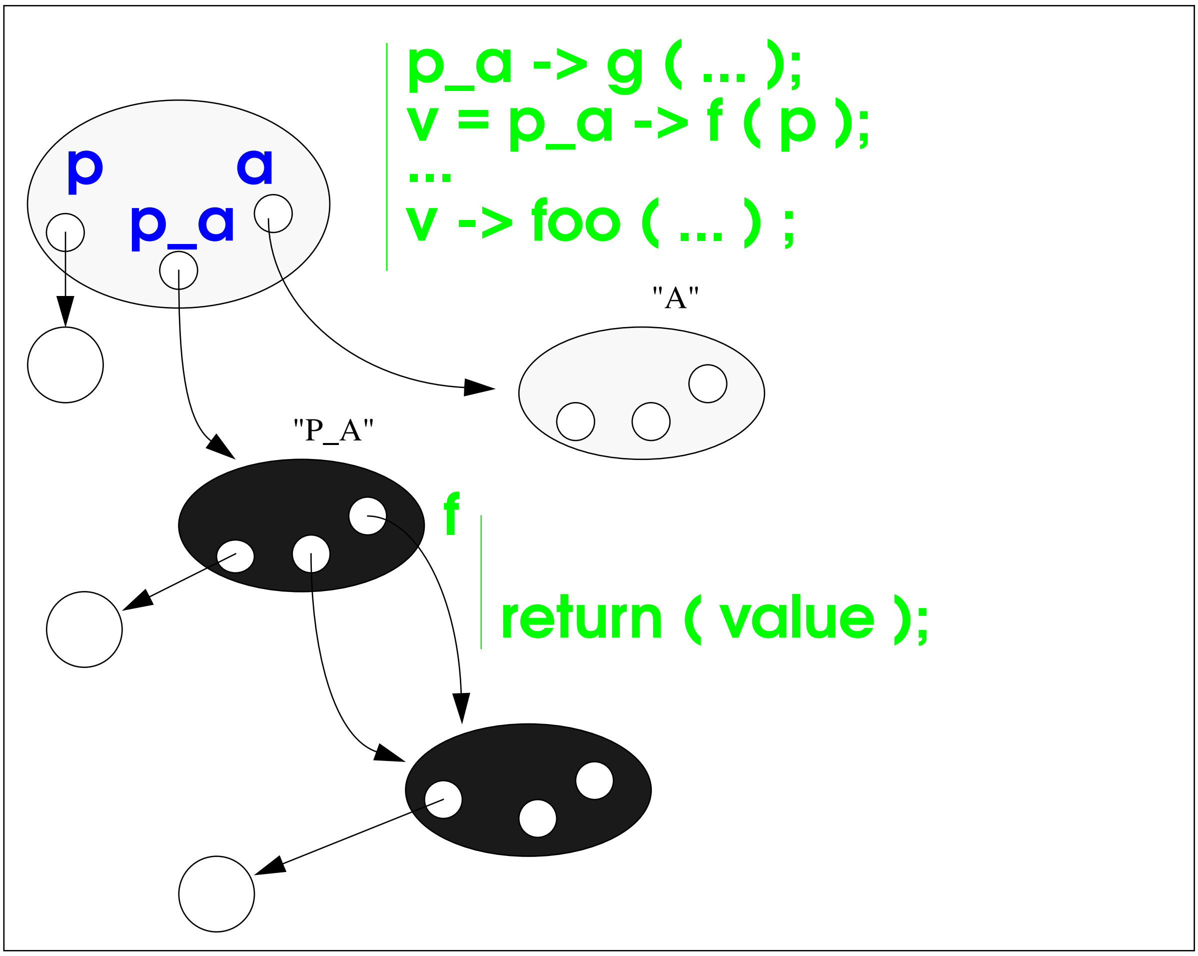
Wait-by-necessity:

---> Systematic, implicit and transparent futures

A wait occurs (automatically):

- upon a strict operation on a future (unknown value)

- upon passing a future as a parameter in an asynchronous call



2 Predefined operations:

Wait, Awaited

Ease the programming of synchronizations

Reuse of routines

To understand futures, find an implementation technique for them.

2 main solutions:

--> • 1 bit per object + check in every method, upon each call

--> • Wrapper object + check in wrapper, upon each call as well

... Pictures ...

5.3.5 Continuations Automatiques

Automatic Continuation:

Goal: Avoid needless wait --- Reuse

Principal: transparent carry of result

If a result is not available when it has to be send back to the caller (upon completion of a request), then the current object is not blocked, and the result will be send back latter on and transparently, when available.

See Binary Tree example (second examples).

5.4 Exemples

5.4.1 Polymorphisme objet - processus

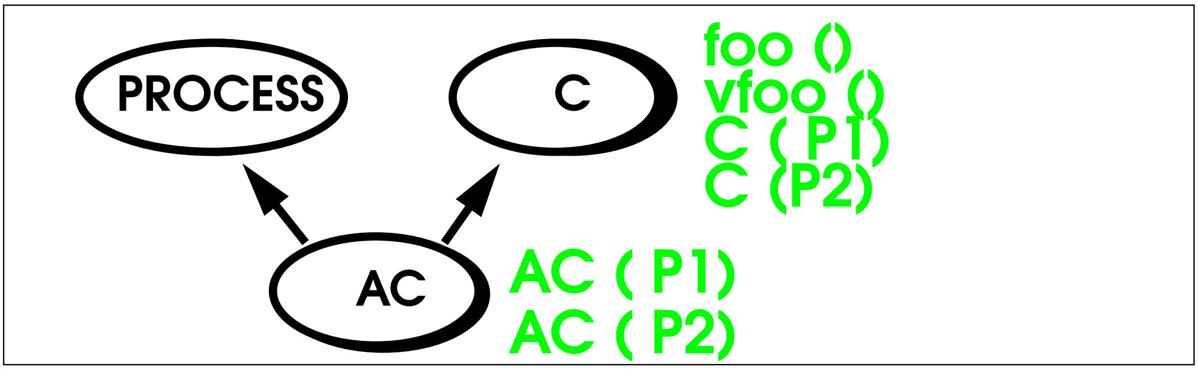
C++//

From a normal (sequential) classes:

```
// Any user class
class C {
  public:
    Res* foo ( param );
    virtual Res* vfoo ( param );
    C(P1 p1);
    C(P2 p2);
};
```

Parallelization with C++//:

```
// An asynchronous class based on Process
class AC: public C, public Process {
  public:
     AC(P1 p1): C(p1);
     AC(P2 p2): C(p2);
 };
```



Usage :

```
// Explicit use
AC* ac = new AC(p1);
Res* r1 = ac->vfoo ( param ); // Asynchronous calls
Res* r2 = ac->foo ( param ); // Async. call

r1->vbar();   // Automatic Wait if awaited
      // r1->bar(); // Cannot be used: not virtual
// ... Picture ...
```

```
// Implicit use (reuse of sequential code)
C* c = ac;    // Polymorphic assigment
...
// Reusability of sequential code:
Res* v = c->vfoo ( param ); // Asynchronous call,
                            // virtuals only
v->vbar();   // Automatic synchronization
// Reuse limit due to Dynamic Binding limitation
// +/- idem in other dist. C++: e.g. CORBA, orbix
```

```
// Explicit use of future through the proxy:
Future_P* f1=ac->get_future (EC_Any(r1));
                 // Use of primitives, get a Future proxy
f1->Wait();    // Wait the value
if (!f1->Awaited()) ...;    // Test available
Res*r=(Res*)f1->Value();//Get the value,
```

5.4.2 Arbre Binaire de recherche

Binary tree, Eiffel//

a) Version séquentielle

Code Séquentielle:

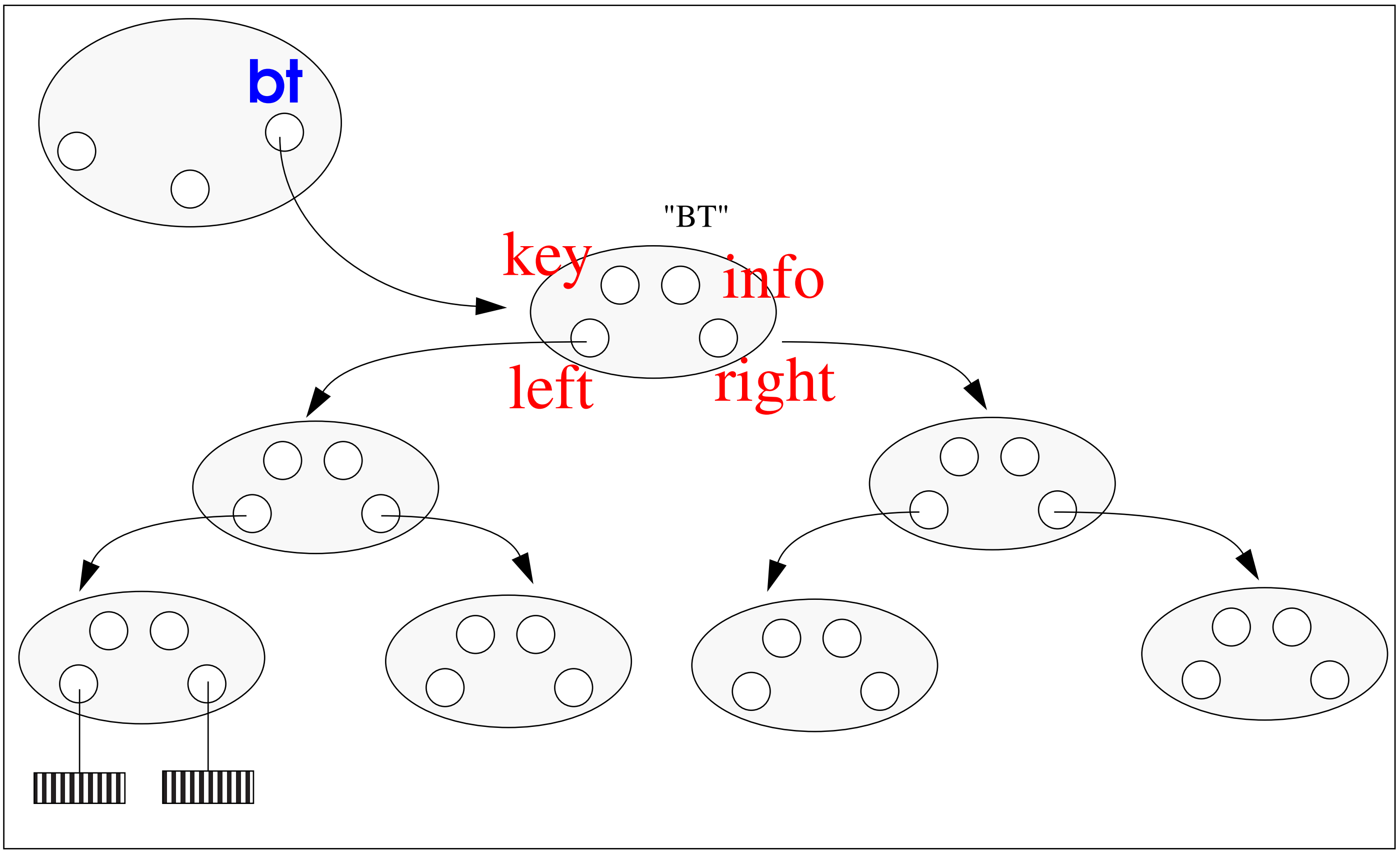
```
class BT [ T -> COMPARABLE, U]
    -- T is the key, U the information
export
        insert, -- Insert k in the BT
        search, -- Value with key k ?
        left, right
feature
    key: T; info: U;
    left, right: like current;
...
```

Routines *insert* et *search*:

```
insert (k: T; i: U) is
    do
        if key.Void then
            key := k; info := i;
            left.Create; right.Create;
        elsif k.equal (key) then
            info := i;
        elsif k.lt (key) then
            left.insert (k, i);
        else
            right.insert (k, i);
        end;
    end; -- insert

search (k: T) : U is
    do
        if key.Void then
            Result := <Not found>
        elsif k.equal (key) then
            Result := info;
        elsif k.lt (key) then
            Result :=left.search (k);
        else
            Result :=right.search (k);
        end;
    end; -- search
```

Topologie à l'exécution:



Exemple d'utilisation:

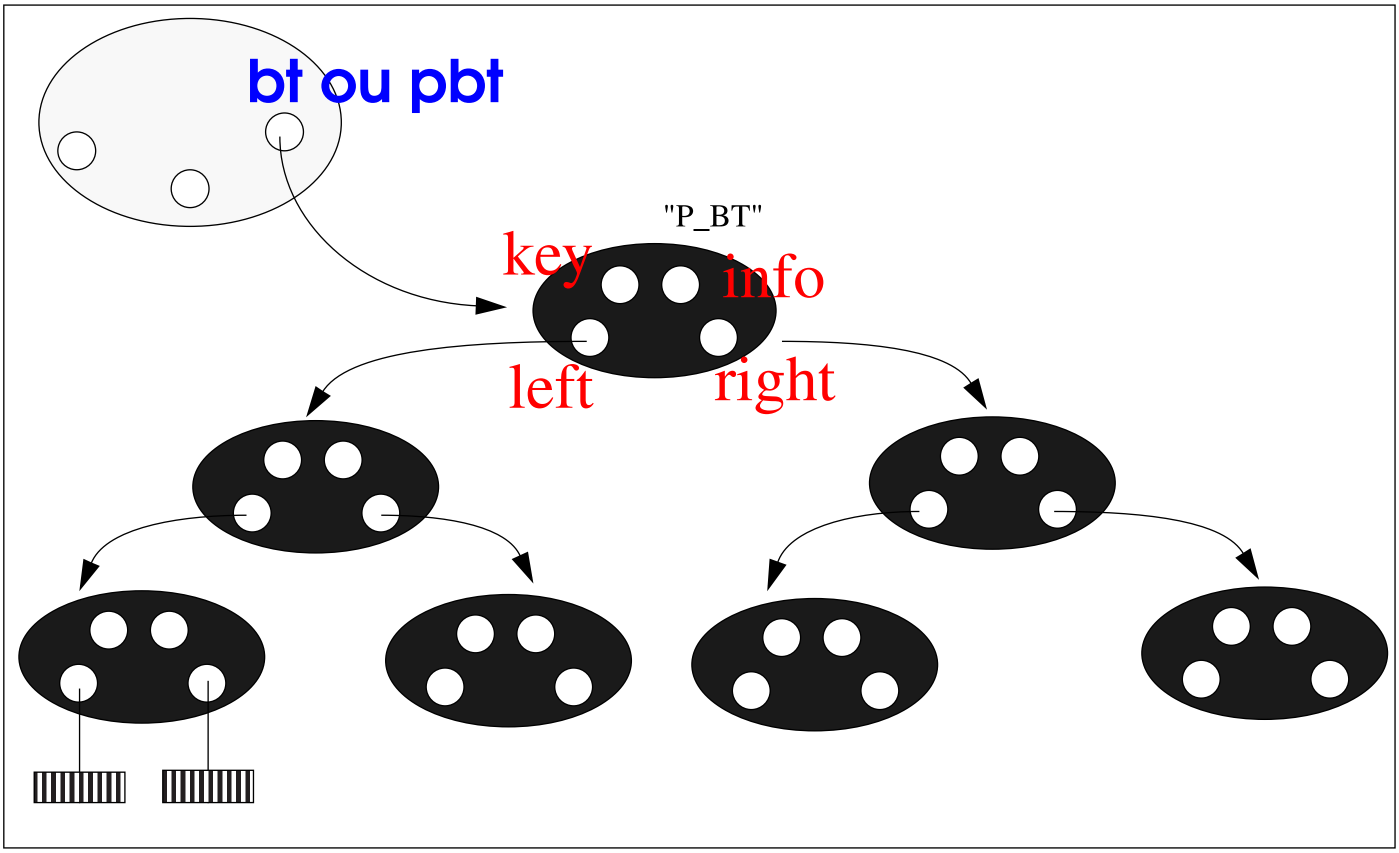
```
bt: BT; bt.Create;
...
bt.insert (k1, v1);
bt.insert (k2, v2);
bt.insert (k3, v3);
...
s2 := bt.search (k2);
s1 := bt.search (k1);
bt.insert (k1, vx);
s3 := bt.search (k1);
...
```

b) Version parallèle

Trouver les objets actifs:

chaque noeud de l'arbre binaire

Topologie à l'exécution:



Note:

La construction "*Like Current*" est importante.

Elle correspond à une technique de la programmation à objets classique (séquentielle) qui permet de définir le type d'une entité comme étant dépendante de l'objet courant (*MyType* dans d'autres langages) -- important pour la réutilisation.
Ici cela permet de passer de séquentiel à parallèle.

Code Parallèle:

```
class PBT [ T -> COMPARABLE, U]
    -- T is the key, U the information
export
        repeat BT
inherit
    BT;
    PROCESS;
feature

end -- Class PBT
```

Exemple d'utilisation:

```
bt: BT;
... pbt: PBT; pbt.Create; bt := pbt;
bt.insert (k1, v1);
bt.insert (k2, v2);
bt.insert (k3, v3);
...
s2 := bt.search (k2);
s1 := bt.search (k1);
bt.insert (k1, vx);
bt.insert (k1, vy);
s3 := bt.search (k1);
s3.print (); // must print vy
```

Affectation polymorphe ou appel de routine

--> • Réutilisation du code

Propriétés :

- Au plus, le niveau de parallélisme est égal au nombre de noeud de l'arbre.

- Le service FIFO est fondamental.

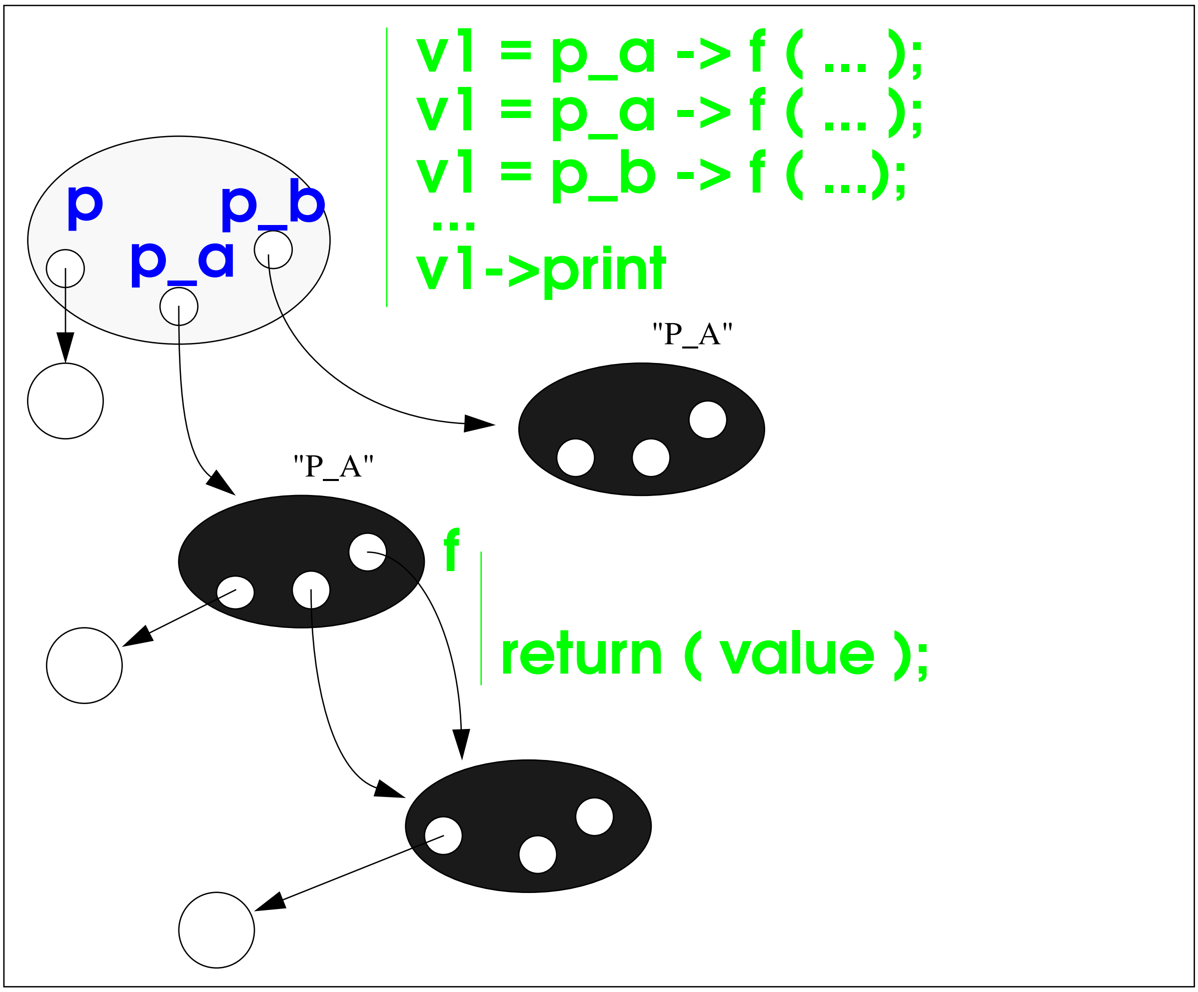
- Le fait que les communications point à point respectent l'ordre FIFO est également fondamental.

- Si l'on cherche à prouver que la version séquentielle est équivalent à la version parallèle (exercice non trivial), il faudra démontrer que:
    - pour chaque noeud dans le cas parallèle, la liste des requêtes reçues respecte l'ordre du cas séquentiel.

5.5 Précisions sur la sémantique

5.5.1 Attente-par-nécessité et affectation

Soit le code suivant:



Quelle sont les sémantiques possibles ?

Quelle est la meilleure?

Comment programmer l'une avec l'autre?

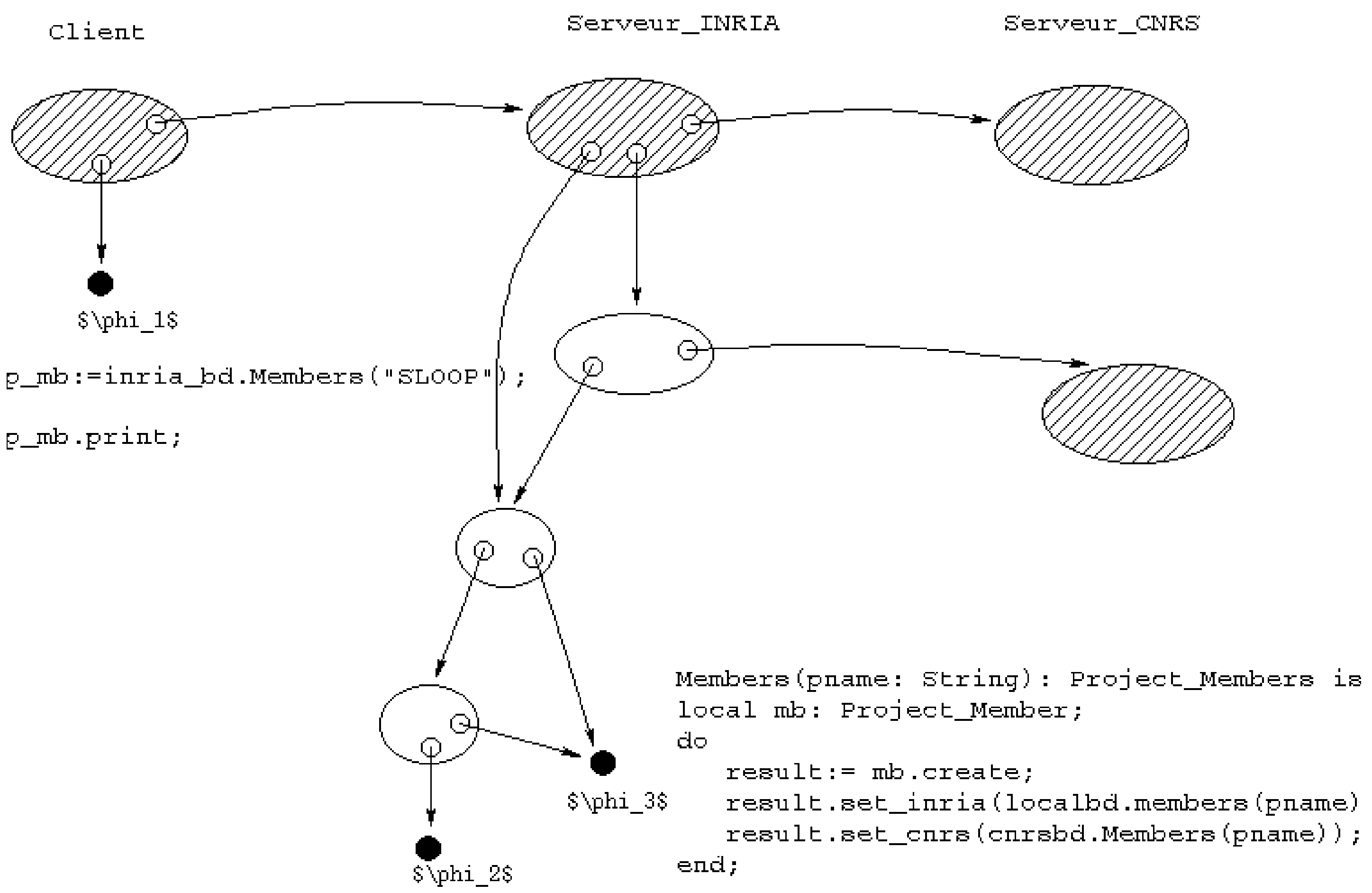
5.5.2 Stratégies de Continuation Automatiques

Dealing with partial results.

General case is more complex than Binary Tree.

Goal: maximize parallelism, avoid dead-locks and livelocks

For instance:



At least two solutions:

1.: Synchronous: a result is returned only when it contains no more futures.

Requires to build a copy of values to be returned when they are known (at the end of the routine) or upon reception. Deep clone.

This avoids potential modifications from the current active object, in order to maintain the sequential semantics.

2.: Asynchronous: partial results are returned as soon as they are available.

Requires to build more futures to be sent between active objects (sub-systems) in place of unknown values.

There is a risk of endless looping when cyclic dependencies occur between asynchronous calls. But:

- Correspond to deadlock with Sol. 1

- Are dynamically detectable

Solution 2 leads to more parallelism, and avoid some deadlock that should not occur (a "lazy solution" in a functionnal sense, where non-strict values that are not correct do not preclude from reaching a correct solution).
At the end: unknown futures remain

Note: these strategies could also be applied upon asynchronous calls, when passing a future as parameter.

5.5.3 Caractéristiques du modèle

Parallel model:

MIMD model: Active objects,

Wait-by-necessity:
Data-flow object-oriented parallelism

Object-flow

Computations are structured in sub-systems