CHAPITRE 6 Programmation des Processus

Définir le corps, le contrôle, la synchronisation, l'activité des processus

6.1 Programmation du contrôle

6.1.1 Analyse d'exemples

a) Ada

```
task body buffer is
begin
    loop
        select
            when notFull =>
                accept put (x) do
                    ...
                end;
            or
            when notEmpty=>
                accept get (x) do
                    ...
                end;
            or
                terminate;
        end select;
    end loop;
end buffer ;
```

Caractéristiques :

---> Mainly Centralized (but code of entry point)
---> Mainly Explicit (but non-determinism)

b) Guide

Inspiré des Compteurs de Synchronisation

```
CLASS buffer is

METHOD put (x)
    BEGIN
        ...
    END;

METHOD get (x) do
    BEGIN
        ...
    END;

CONTROL
    put:
    (completed(put) - completed(get)<size)
    and current(put) = 0;
    get:
    (completed(put) < completed(get))
    and current(get) = 0;
end buffer ;
```

Caractéristiques :

---> Centralized (fully declarative)
---> Implicit (no thread at all)

c) Hybrid

Univ. of Geneva

```
type buffer of ( ... )
    ...
private
{
    ...
     put: ( x ) -> ;
            uses notFull;
    {
        ...
        » signal notEmpty »
    }

     get: -> x ;
            uses notEmpty;
    {
        ...
        » signal notFull »
    }
}
```

Caractéristiques :

---> Not centralized
---> Implicit (not an explicit thread)
---> Very much a monitor like solution

6.1.2 Choix: Contrôle explicite

Default:

---> Centralized
---> Explicit control

The Process class:

```
// An asynchronous class based on Process
class Process  {
  public:
    Process (...) {  // Process creation
      ...
    }
    virtual void Live () {   // Process body
      ... // Default FIFO behavior
    }
    ...
 };
```

Why ? :

Centralized:

- --> • Readability: separation between Actions / Synchronization
- --> • Reusability: add the synchronization

Explicit:

- --> • Lower level: imperative programming
- --> • An implicit control can be programmed

6.1.3 Exemple: Buffer

An example: Bounded Buffer in C++//

```
// An asynchronous class based on Process
class Buffer: public Process, public List  {
      virtual void Live () {    // Process body
          while (not ‘‘termination") {
             if (!full)
                ‘‘serve the oldest on put "
             if (!empty)
                ‘‘serve the oldest on get"
          }
      }  };
```

On retrouve:

Centralized:

--> • Readability:
no Action / Synchronization mixed up

--> • Reusability: of List

Explicit:

--> • Lower level: imperative programming
classical control structure

New question: how to achieve:

“serve the oldest on put” ?

plus: termination?, non-active wait ?

6.1.4 Service des routines: exemples

a) Ada

```
task body buffer is
begin
    loop
        select
            when notFull =>
                accept put (x) do
                    ...
                end;
            ...
    when startwrite'COUNT = 0 =>
                accept startread do
                    ...
```

Caractéristiques :

---> Serve the oldest request on put
(No other choice)

---> Selection upon the state of the request line

b) Concurrent C (N. Gehani)

accept lock (id) suchthat libre (id)

...

Caractéristiques :

---> Selection upon the parameters value of a request

Conclusions :

---> Expressivenes is limited

---> Always new extensions that have to be added to the language (cf. Add 95)

---> Very low modularity:
how to program new services, and make them available to other users?

6.1.5 Choix: bibliothèque de services

What is needed to achieve it ?

Service:

Some Reflection and Reification

---> Routines as first class objects
(ROUTINE class, operator &)

```
ROUTINE rout;
rout = & put;
```

---> Requests as first class objects
(REQUEST class)

```
REQUEST req;
req.serve();
```

---> Access to the list of pending requests

```
from pendReq.start til pendReq.end
do
    req = pendReq.current;
    if .... then req.serve ();
```

Library of service routines:

Non-blocking services

- serve_oldest(); // Serve the oldest request of all
- serve_oldest(f); // The oldest request on f

Blocking services

- bl_serve_oldest(); // Serve the oldest request of all

  bl_serve_oldest(f); // The oldest request on f

Timed services

- bl_serve_oldest(Time limit); // Blocking, with time out
- etc.

Waiting primitives

- wait_a_request(); // Wait until a request arrives
- wait_a_request(f); // A request to serve on f
- wait_a_request_to_serve(); // Wait until a request to serve

==> non active Wait

Extensible by the end user:

- serve_oldest_of2 (f1, f2); // The oldest request on f1 and f2
- etc.

(Typical program.: critical section, select, service, ...)

Member functions representations:

- mid(put)
- mid(A::put)
- mid(A::put, A::get);
- mid(A::put(int, P *));

6.1.6 Exemple: Buffer

An example: Bounded Buffer in C++//

```
// An asynchronous class based on Process
class Buffer: public Process, public List  {
      virtual void Live () {   // Process body
         while (!stop) {
           if (!full)
              serve_oldest ( mid (put) );
           if (!empty)
              serve_oldest ( mid(get) );
           wait_a_request_to_serve();
         }
    }  };
```

Caractéristiques :

---> si ni vide ni plein, ne respecte pas l'ordre d'arrivé, mais alterne put et get.

Other policy:

```
// Respecting arrival ordering of the requests
        if (full)
          serve_oldest ( mid (get) );
        else if (empty)
          serve_oldest ( mid(put) );
        else
          serve_oldest ( mid(put), mid(get));
```

---> si ni vide ni plein, respecte l'ordre d'arrivé

6.1.7 Exemple: Reader-Writer (ProActive)

An example: Reader-Writer in ProActive

```
// An asynchronous class based on Process
public class ReaderWriter implements Active {

// Even Policy
public void live(Body myBody){
   // Loops over lifetime...
   while(!done){
      // if there is no writer and no write requests
      // are before the first Read rq.
      if(writeCount==0 &&
         isOlder(myBody,"startRead","startWrite"))
               myBody.serveOldest("startRead");
      // If ther is no activity and the first request
      // is startWrite
      if(readCount==0 && writeCount==0 &&
         isOlder(myBody,"startWrite","startRead"))
               myBody.serveOldest("startWrite");
      // Allow endXX requests
      myBody.serveOldestWithoutBlocking("endRead");
      myBody.serveOldestWithoutBlocking("endWrite");
    myBody.waitForRequest(); // Passive wait.
}}
```

```
/** * isOlder
* @return True if methodA is before B, False otherwise */
private boolean isOlder(Body myBody,
                        String methodA,String methodB){
// Buferize new incoming requests
myBody.bufferizeNewRequests();
// Iterate through the pending requests
MethodCall mc=null;
String method=null;
RequestLineIterator rq=myBody.getRequestLineIterator();
while(rq.hasMoreMethodCalls()){
  mc=rq.nextMethodCall();
  method=mc.getName();
  // if methodA is present before MethodB
  if(method.equals(methodA)){
  // Accept new requests and put the bufferred into the queue
    myBody.acceptNewRequests();
    return true;
  }else{
    if(method.equals(methodB)){
    // methodb is present Before methodA
      myBody.acceptNewRequests();
      return false;
}  }  }
// There is no request matching methodA
   myBody.acceptNewRequests();
   return false;
  }
```

Conclusion:

Explicit control and Services:

powerful,

sometimes needed,

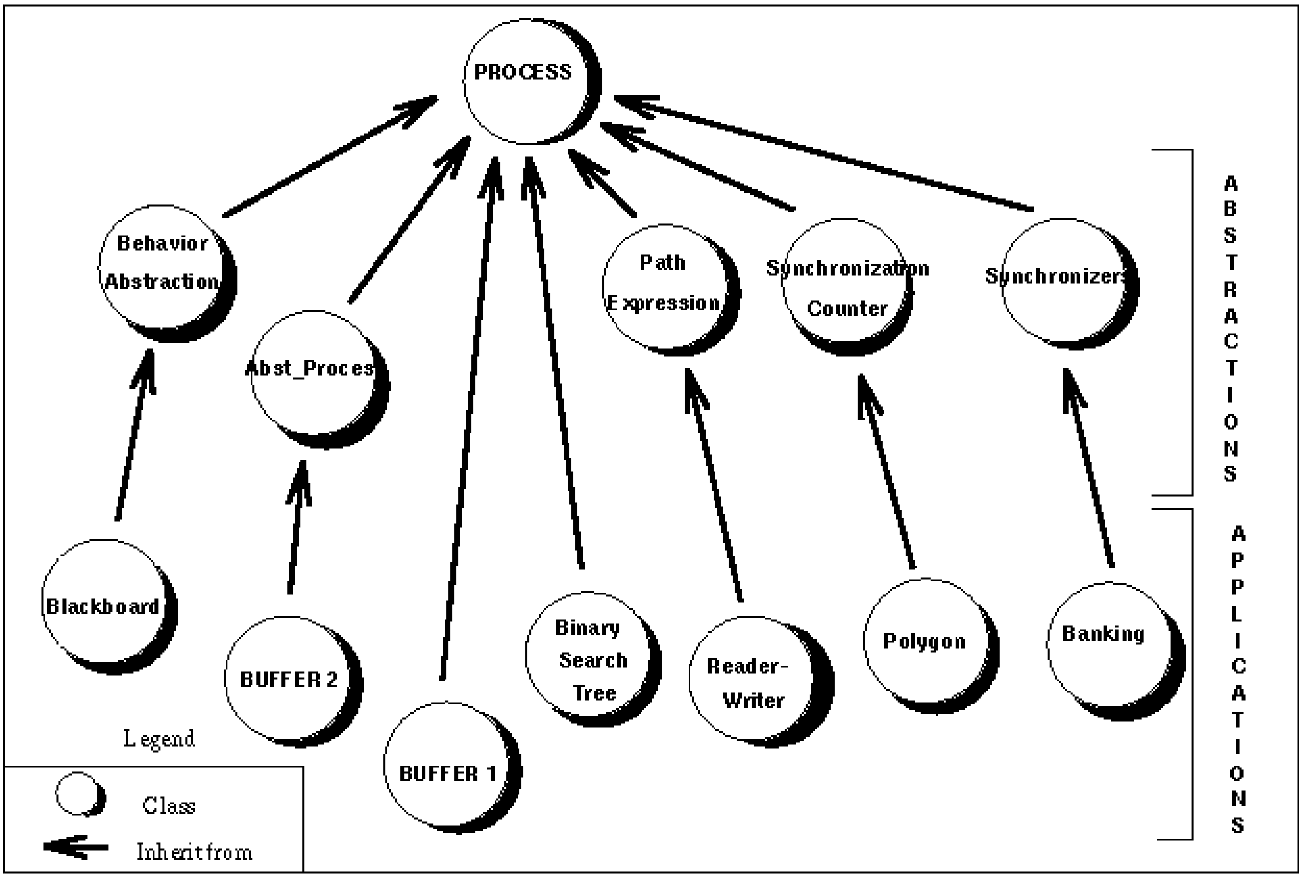
sometimes, we want a more abstract expression:

for instance with non-determinism

--> • Abstraction for concurrency control

6.1.8 Contrôle implicite: Abtraction

Abstractions for concurrency control:



- Enabled set: **Rosette**, C. Tomlinson, V. Singh
- Behavior Abstraction: **ACT++**, D. G. Kafura, K. H. Lee
- Activation Conditions: **Guide**, D. Decouchant, S. Krakowiak
- Path Expression: R. Campbell, A. Haberman
- Synchronization Counters: P. Robert, J.-P. Verjus

6.1.9 Example: Buffer déclaratif

```
// An asynchronous class based on Process
class Buffer: public Abst_Process, public List  {
      virtual void Synchronization () {
          associate (mid (put), mid (full) );
          associate (mid (get), mid (empty) );
      }
};
```

Note:

Select vs. abstraction:

non determinism,

terminaison

6.1.10 Réutilisation de synchronisation

Programmation incrémentale de synchronisations (impossible en explicite)

Nouveau buffer:

Deux nouvelles routines:

nb_element:

retourne le nb. d'éléments dans le buffer

get_last

retourne le dernier élément ajouté

```
// An reuse of synchonization
class Buffer2: public Buffer{
    virtual void Synchronization () {
        Buffer::Synchronization();
        associate (mid (get_last), mid (empty) );
    }
};
```

6.1.11 Programmation d'une abstraction

```
// The Abst_Process
class Abst_Process: public Process{
      virtual void Live () {   // Process body
          synchronization();
          while (!stop) {
              from <beg. of request line> until <end>
              loop
              <Get request, Test blocking routine>
              if <false> then // Request can be served
                 req.serve();
                 <remove req from the list>
              end;
              <next request>
              end loop;
              wait_a_request_to_serve();
          }
        }

        associate (rout:ROUTINE,sync:ROUTINE) {
            ....< Associate a blocking condition>
        }

        synchronization () {
          // Default synchronization: empty= FIFO
        } // A redefinir par l'utilisateur

};
```

6.2 Une méthode de parallélisation

A programming method

6.2.1 Conception et programmation séquentielle

a) Sequential design and programming:

6.2.2 Parallélisation -- Distribution

a) Process identification:

(1) Initial activities:
Parallelism, Distribution, Concurrence

(2) Shared objects:

- Default copy semantics (check semantics)
- Reference semn. needed: Active object
---> back to --> a.2

b) Process programming:

(1) Define the type of the control

(2) Define the process class

(3) Define the interface

(4) Define the process activity -- synchronization

(5) Use the process: polymorphism

c) Adaptation to constraints:

(1) Synchronization refinement

(2) Routine redefinition

(3) Topology refinement

(4) New process definition (--> a.2)

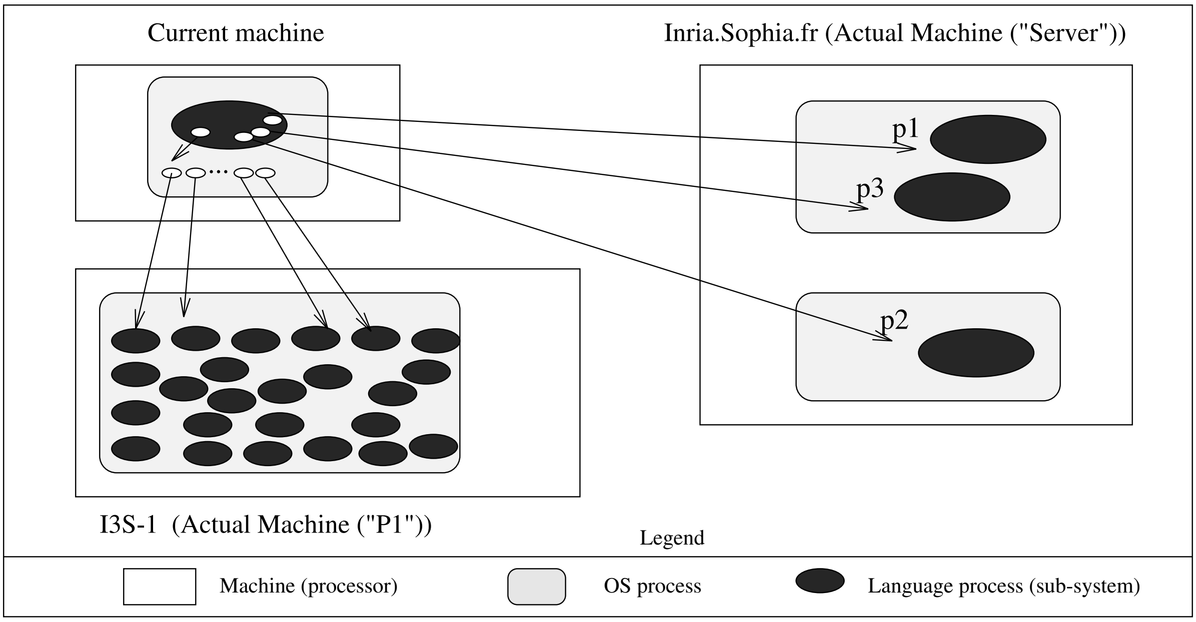
6.3 Placement

Mapping:

Assigns each active object to an operating system process on an actual machine or processor.

6.3.1 Modèle de placement

Mapping model:



Virtual node names: File .c++ll-mapping

// Virtual name	Actual name
Mach_A	Nice
Mach_B	bolas.isi.edu GLOBUS
Proc1	Inria.Sophia.fr
Server	pitcairn.mcs.anl.gov GLOBUS

6.3.2 Programmer le placement

Controlling the mapping:

```
// Controlling active object mapping
class Mappings {
 public:
     virtual void on_machine(const String& m);
                            // Machine virtual name
     virtual void with_process(Process* p);
                            // Same as active object p
     virtual void set_light(); // light-weight process
     virtual void set_heavy(); // heavy-weight (OS) };
```

Combination of the Mapping Criteria:

Process nature \ Machine	Virtual machine M	Machine where the language process p is running
Light	On machine actual_name (M) On an arbitrary existing OS process (a new one if none)	Same machine and OS process as p
Heavy	On machine actual_name (M) On a new OS process	Same machine as p On a new OS process

"existing OS process"

Difficult to ensure:

none when checking, then one, so 2 !

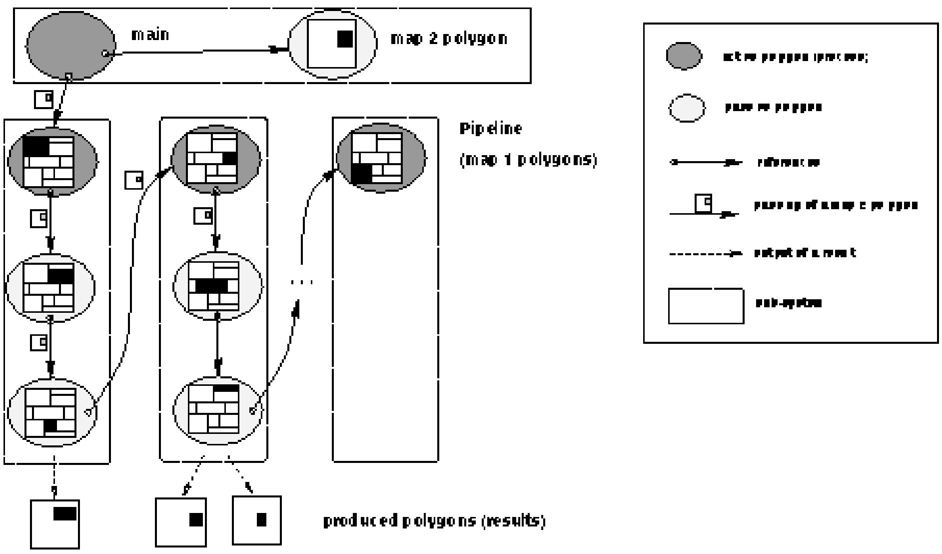
6.3.3 Exemple: Polygon Overlay

Programming the mapping:

```
P_A *p1, *p2; // Active objects
Mapping *map1, *map2; // Mapping objects

map1->set_heavy(); map1->on_machine("Server");
p1 = new (map1)  P_A(...);
map2->set_light(); map2->with_process(p1);
p2 = new (map2)  P_A(...);
```

Polygon Overlay mapping:



6.4 Résumé du modèle

Based on the Eiffel// model:

---> Heterogeneous model
Passive and active objects (processes, actors

---> Sequential processes

---> Polymorphism between objects and processes

---> No shared passive objects
Call-by-value between processes

---> Unified syntax between message passing and inter-process communication

---> Systematic asynchronous communications towards active objects

---> Wait-by-necessity
Automatic and transparent futures

---> Automatic Continuation (CA)
No un-needed waits
A transparent delegation mechanism

---> Centralized and explicit control
Libraries of Abstractions

Extensions / generalizations:

---> Allocation-based active object

---> Migration
With forwarding (or centralized) + optimizations

---> Light-weight active objects
Threads

---> Control of mapping

6.5 Exemples

6.5.1 Contrôleur de vitesse

Cruise control

6.5.2 Un système de blackboard

Reconnaissance de parole

6.6 Réactivité

If the system can react "immediately " at the frequency imposed by external stimuli: reactive

(not transformational)

Reactive to its environment

Examples are: any event-driven system,

CSCW (Computer Supported Cooperative Work) or Groupware

Real-time systems

etc.

Usual parallel object-oriented systems are not really reactives (even if said so) because a call to another object function can take an arbitrary (long) time, and during that treatment, an object won't be able to react to other events, for instance another call, that might be of a higher priority.

Some languages use the notion of "Express Message" (cf. ABCL language, Yonezawa et al.), but it is only a two-level partition, and done statically on the member functions.

Also, does not allow reuse.

6.6.1 Principes

Within our systems, reactivity is based on the construction of a specific abstraction (being reactive)

It also relies on reflective capabilities of the system (see next chapter).

The reactive abstraction provides

reactive routines

Reactive routines can have an interruption semantics with the following promitives:

abort:

permits aborting the current request service activity

suspend:

suspends the current request service

resume:

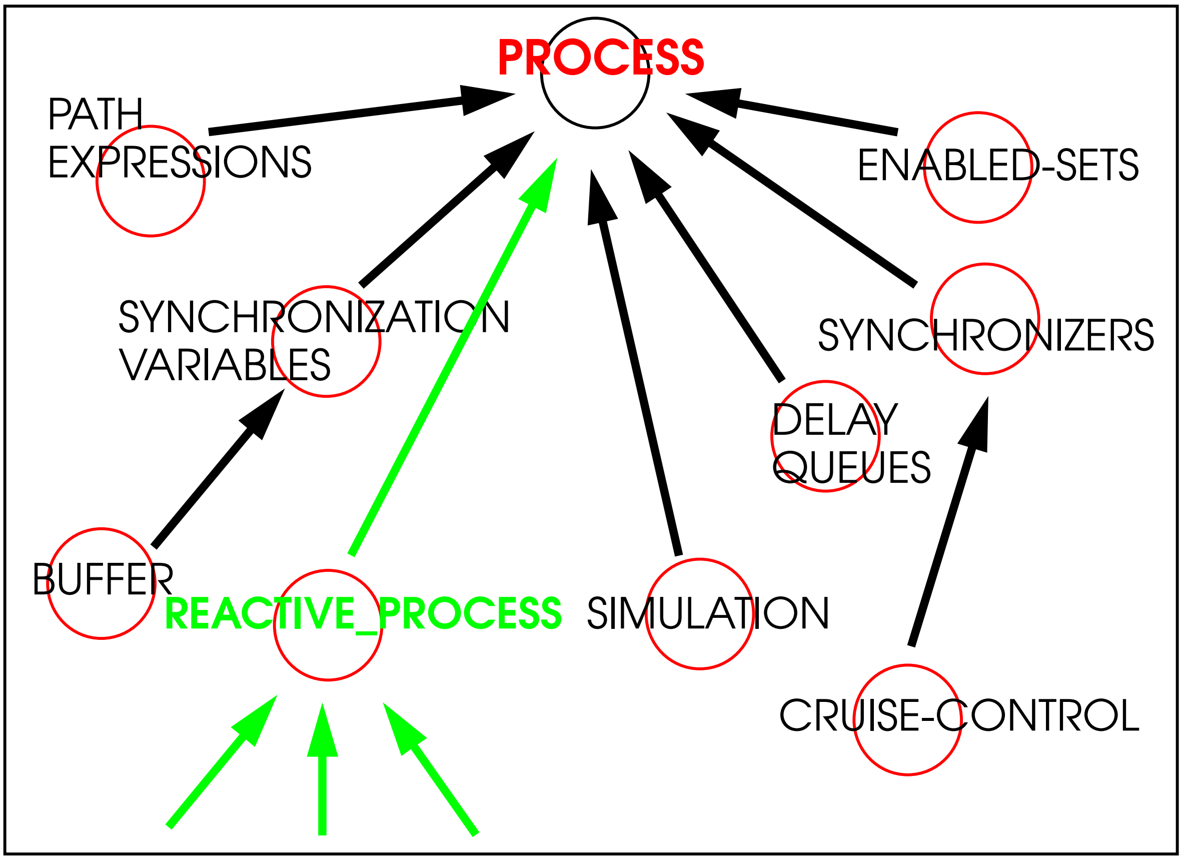
resumes a previously suspended execution

A reactive process (active object) is also able to answer normal requests

An important feature is the capability to take a standard object (both passive or active) and to turn it into a reactive object (reuse):

inheritance of reactive process

6.6.2 Abstractions reactives



The reactive process class :

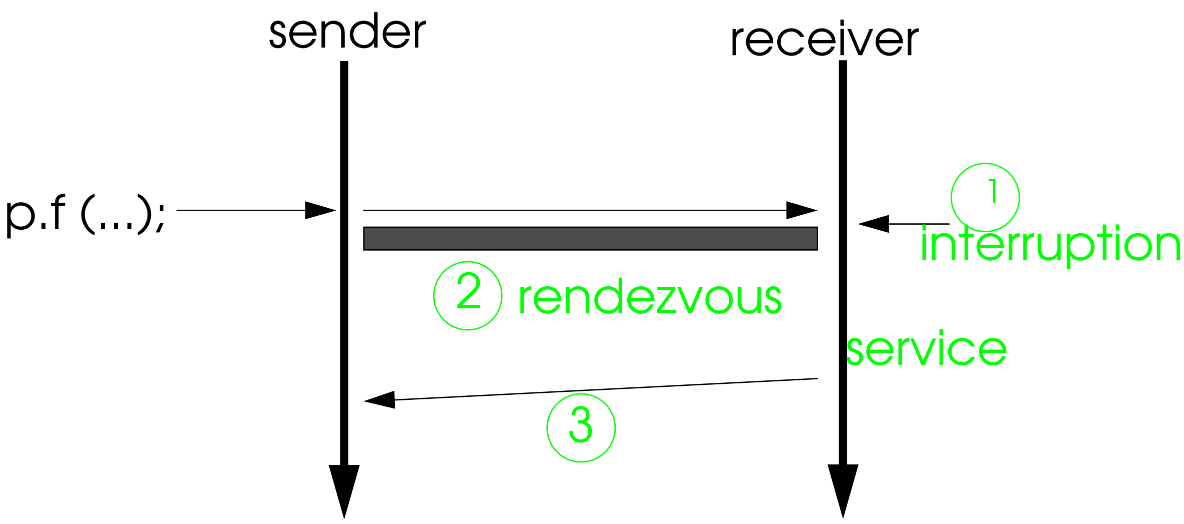
```
class REACTIVE_PROCESS
export
        suspend, resume, abort
inherit
            PROCESS redefine Live
feature
Live is
   do
        < FIFO >
                ...
Receive_request is
   do
        < programming of reactive behavior>
                ...
```

6.6.3 Programmation d'abstractions reactives

Based on the communication semantics:

---> 1. Interruption - 2. Rendezvous

---> (3. Asynchronous service)



The communication implies a rendevous, and as such an interruption of the called object.

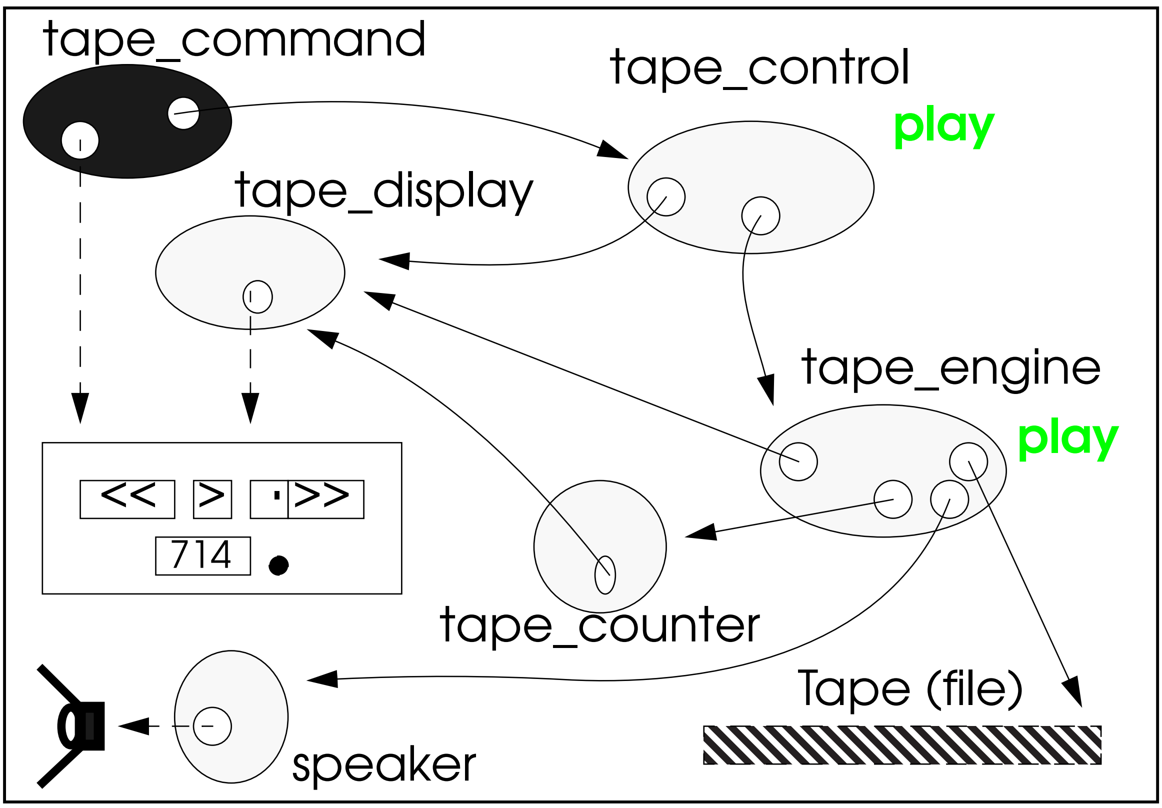
During that RDV, a special behavior can be defined (e.g. suspension of the current service).

6.6.4 Exemple: Tape recorder

a) Principle :

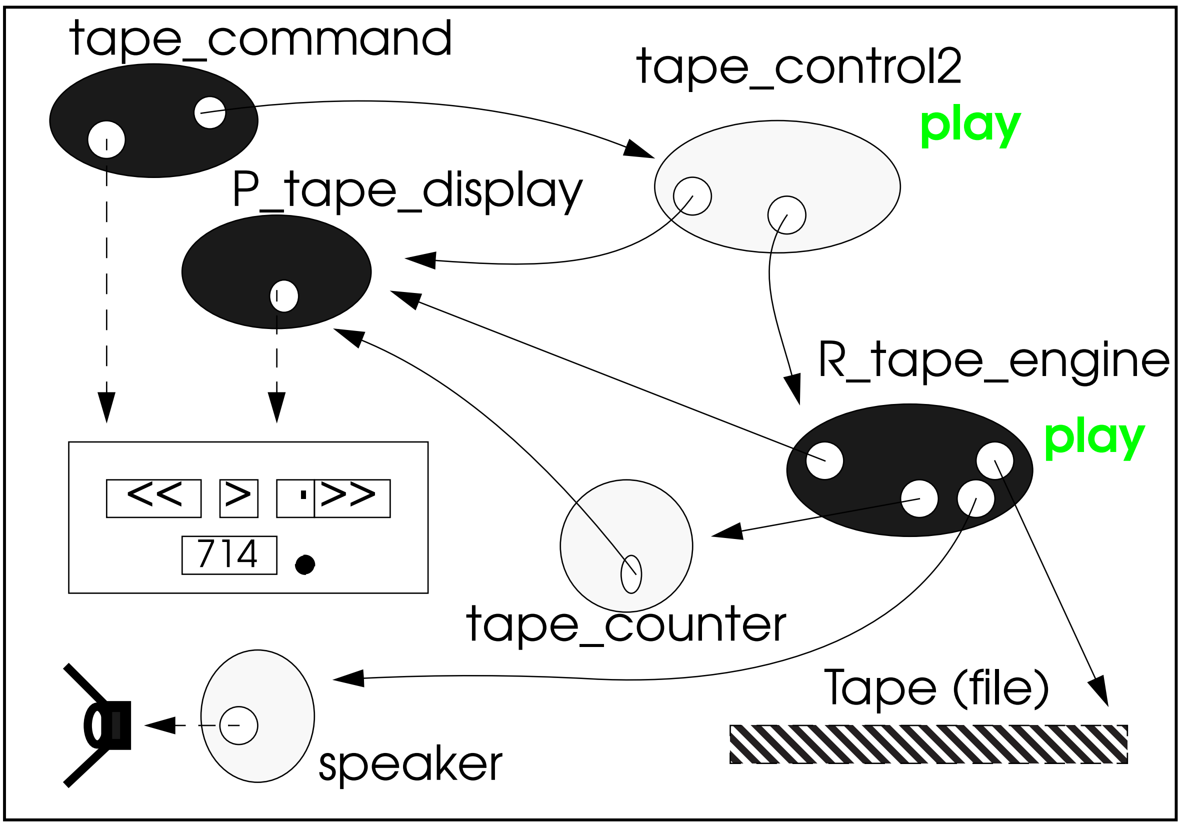
- play, stop, forward, backward,
- engine (reading the file), counter,
- display and command

b) Sequential design:

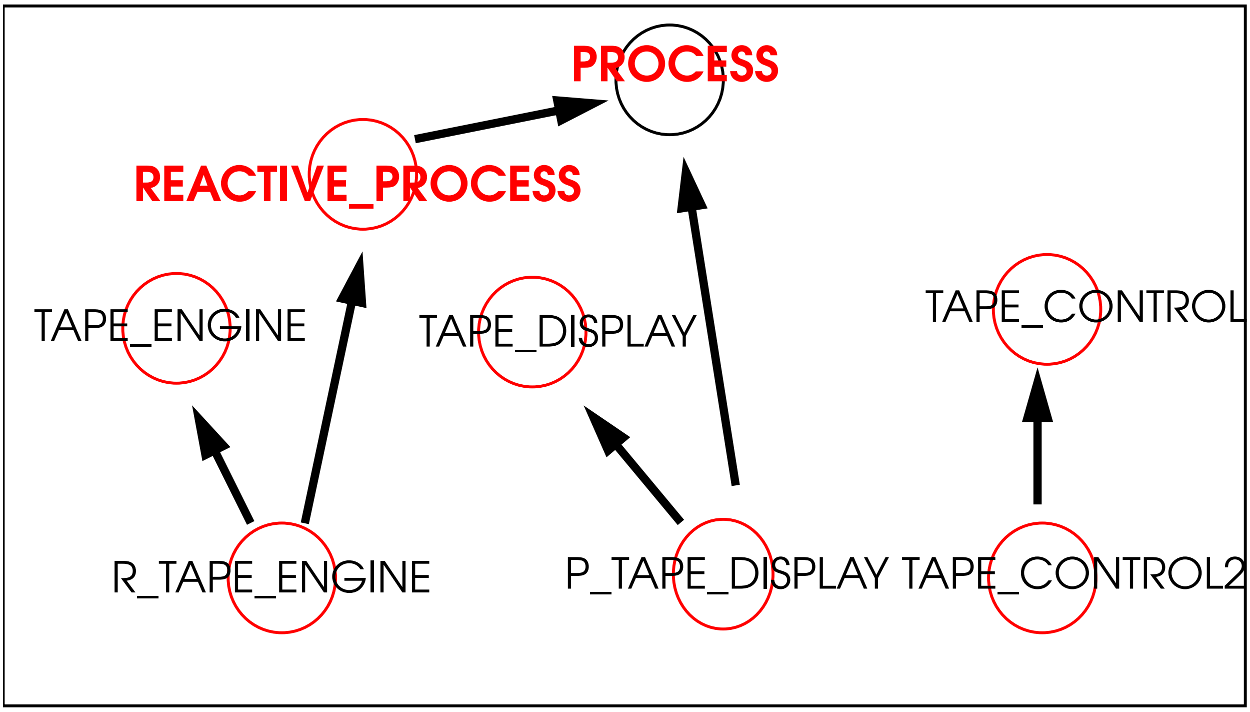


tape_control	tape_engine
play **is**	play **is**
do	**do**
if loaded **then**	**until** end_of_tape
display.set_play;	**loop**
engine.play;	physical_play;
else	**end**;
...	display.set_stop;
	end; -- play

c) Parallel design:



Process programming:



Programming the classes

```
class R_TAPE_ENGINE
...
inherit
        REACTIVE_PROCESS
        TAPE_ENGINE;
feature

end
```

```
class  TAPE_CONTROL2
...
inherit TAPE_CONTROL
      redefine play, ...;
...
play is
  do
    engine.abort;
    old_play;
  end;
...
end
```

Conclusions:

- Sequential to reactive programming:

---> Possible

---> Probably better

- More to do:

---> Specific abstractions

---> Extract specific rules for the design and the programming

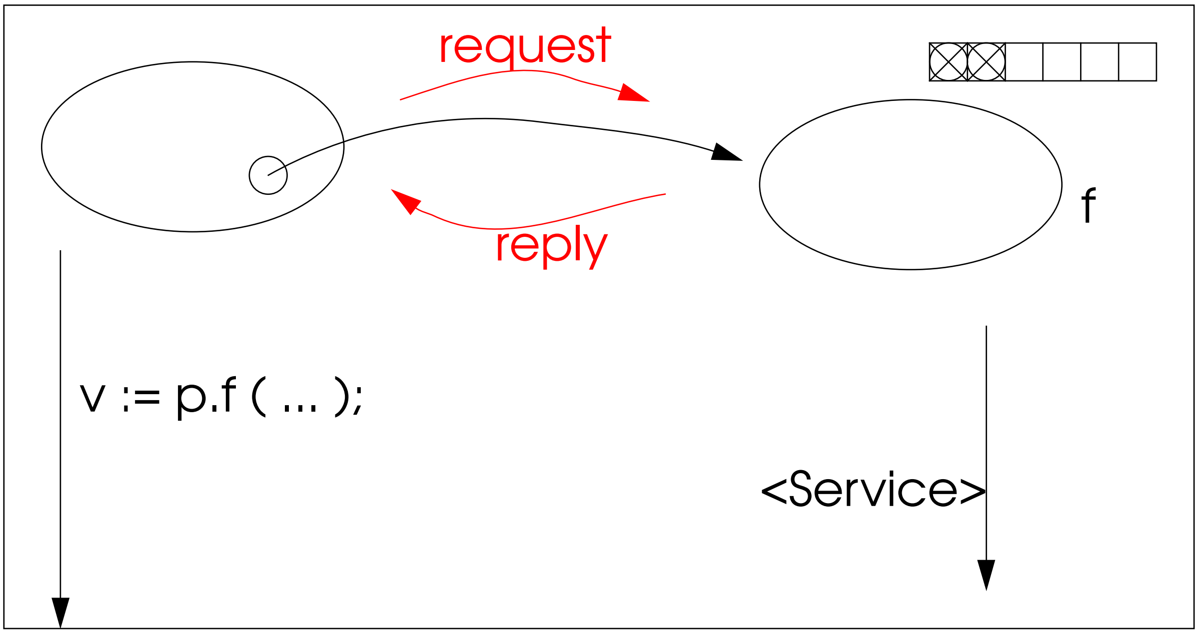
---> Verifications

---> Topology is very important

CHAPITRE 7 Réflexion

7.1 Introduction

Inter-Process Communication:



2 Reifications: objects:

Request:

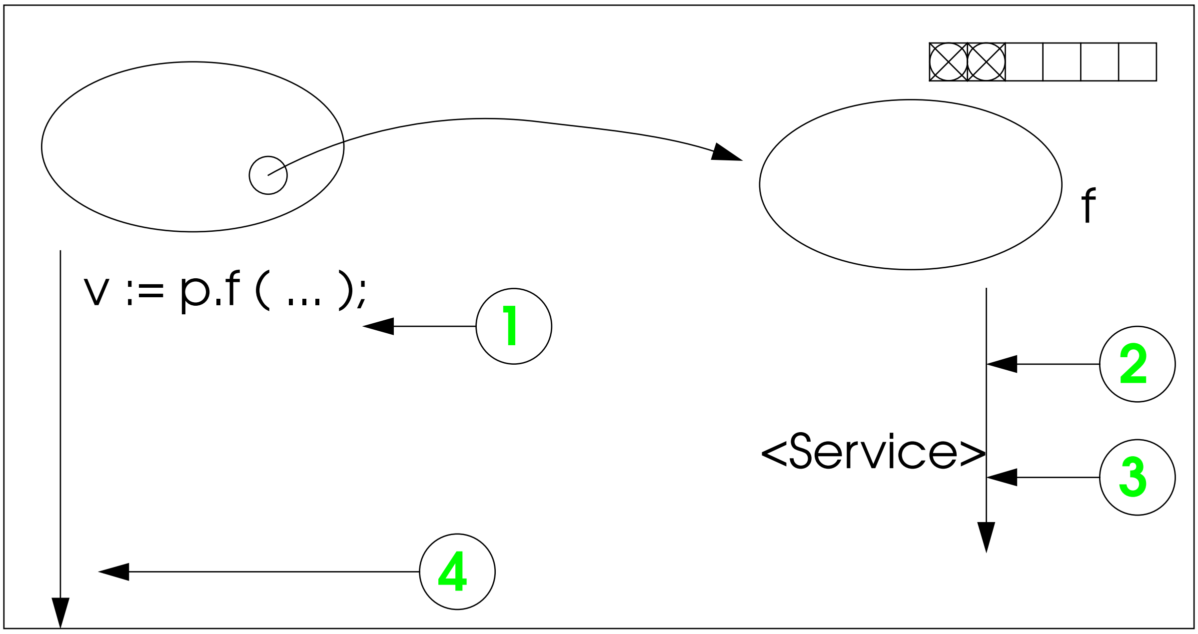
* ID
* Sender
* Receiver
* Target routine
* Effective Parameters
* ...

Reply:

* ID
* Sender
* Receiver
* Result
* ...

7.2 Reification points

Inter-Process Communication:



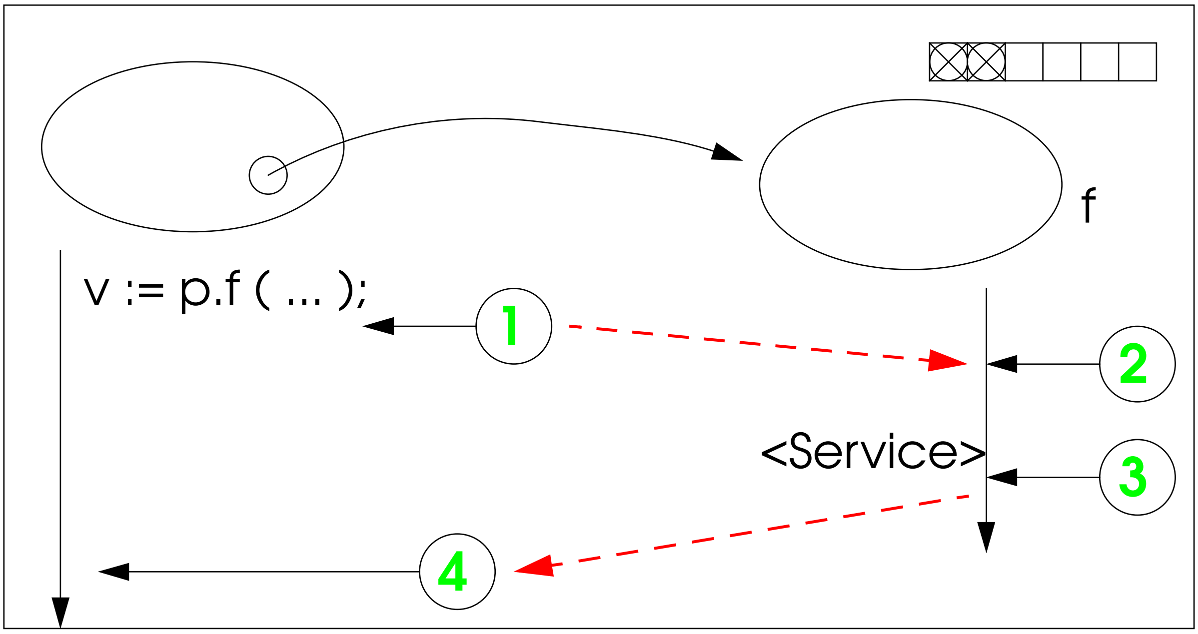
4 Reification points:

1. Send Request
2. Receive Request
3. Send Reply
4. Receive Reply

Each reification point:

---> Specific routine
---> Default behavior
---> Possible redefinition

Reification points



Default behaviors:

Send Request

Establish communication

Send the request object

Receive Request

Put it in the request line

Send Reply

Establish communication

Send the reply object

Receive Reply

Assign the reply object into the awaited entity

7.3 Adaptation: utilisation de la réification

Modification of the default behavior:

---> Specific Abstractions

Reactivity:

Interruption of the current service,
Management of priority (express messages),
Co-routines, or monitor-like programming

Unreliable communications:

Delay the sending of a reply

Fault-tolerance:

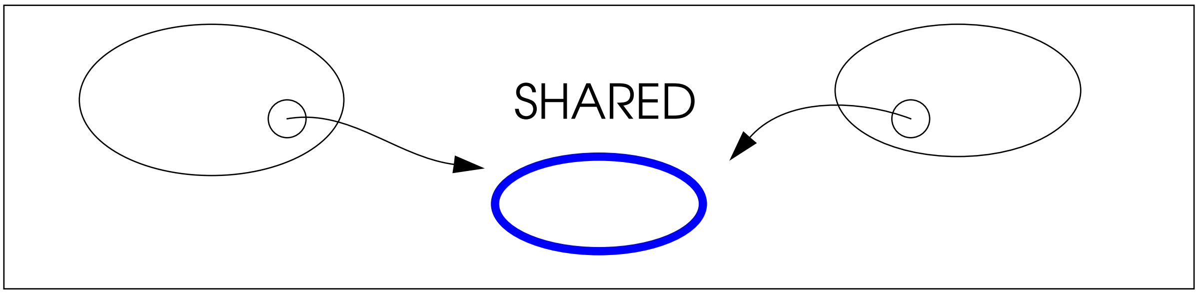
Management of the request line in persistent storage (with RDV communication semantics)

Replication

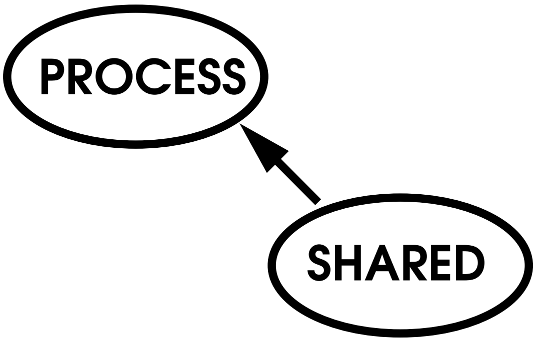
For availability, and maybe resilience ...

7.3.1 Replication

From shared:



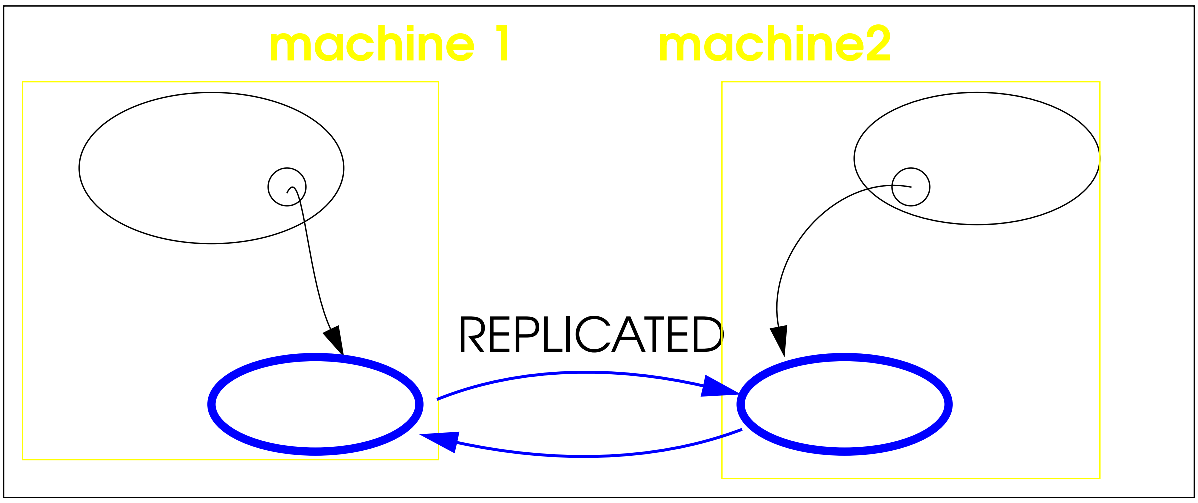
A specific abstraction:



A process without activity, and with synchronous communication

---> same interface and mechanisms

To replicated:



Replication (2):

The user controls the distribution:

```
p = new P_A;
p -> Create ( mapping );
```

Manual :

Virtual name (Appli.) --> Actual name (OS)
New OS process, or within existing one

Automatic:

Default strategy,
load balancing (hopefully),
optimization through locality, etc.

We manage:

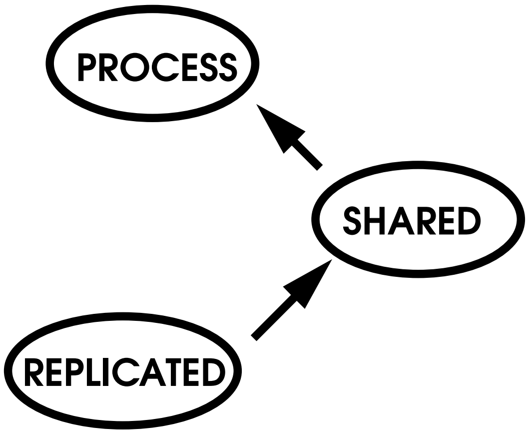
- Compilation problems:
  ---> binary generation, polymorphism
- Objects and processes addressing
  ---> machine ID, OS process ID, object ID

You should define:

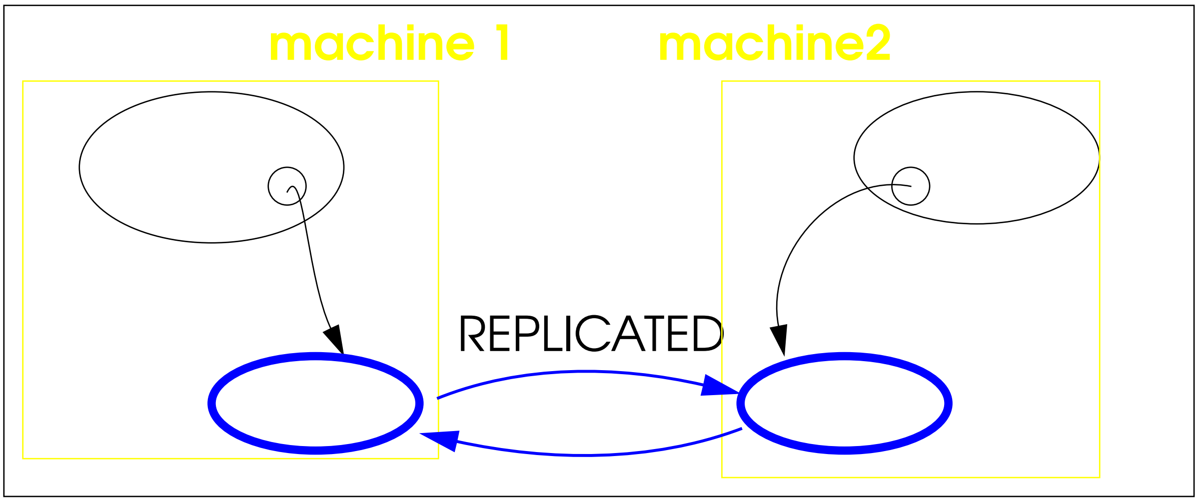
- Replication strategy,
- Replication protocol
- End-user control and interface

Programming Replication:

A specific class (Abstraction):



At run-time:



4 Reification points:

1. Send Request
2. Receive Request
3. Send Reply
4. Receive Reply

Probably needed:

---> Access to the proxy: find the closest copy

7.3.2 Atomic Objects:

---> User-defined atomic data types

Declarative language to express conflict relationship

----> ABSTRACTIONS

From Wu et al.

```
class Account
inherit
  ATOMIC
feature

conflict is
    do
      associate ( & credit, OK, & check, OK);
      associate ( & debit, OK, & check, OK);
      associate ( & debit, OK, & debit, OK);
      associate ( & credit, OK, & debit,FAILED);
    end;

end --
```

This is *implicit* atomic objects (system responsible for synchronization and recovery code):

Other approaches:

---> Explicit (programmer is responsible)
---> Hybrid

7.3.3 Other Features (2):

Atomic and Replicated Objects

From Kaiser et al.

Using replication protocol to implement consistency

Other strategies are probably needed

Transactions

How to define, implement *transactions* ?
How explicit should it be for the end-user?

Atomicity

How to build Global Atomicity from the basic features?

etc.

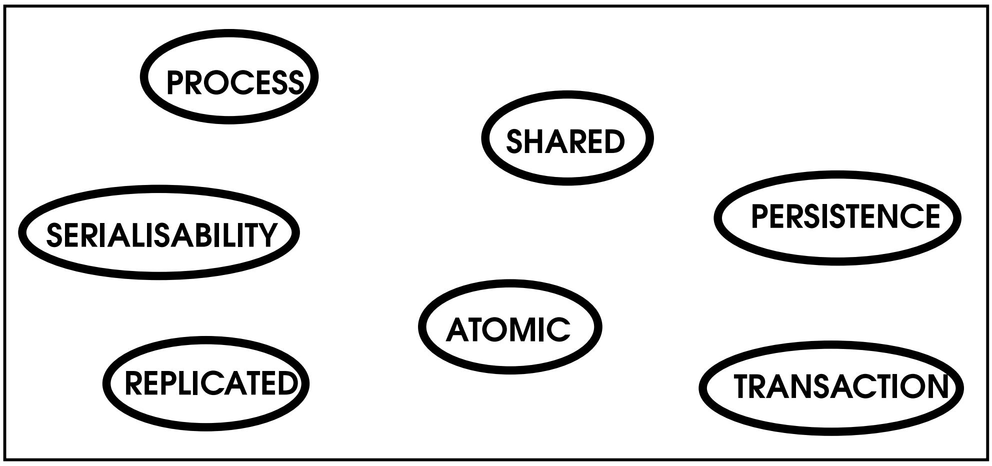
7.3.4 Conclusion on REIFICATION

Is it adapted?

Is it enough ? What else do you need?

What are the limits?

How do we organize all the dimensions? (user and implementation)



How to map onto various hardwares for efficiency?

How does everything relate to persistence?

Our goal:

- A communication mechanism which is flexible and powerful
- The programming of concepts usually embedded into the language definition
- A platform to experiment with ideas, protocols, etc.

CHAPITRE 8 Synthèse, Analyse et Bibliographie

8.1 Dimensions de la programmations à Objets Parallèle

Autres langages,

Voir these + article Briot + article Papathomas

8.2 10 features to achieve reuse

8.3 ACT: Abstract Control Type

8.4

8.5