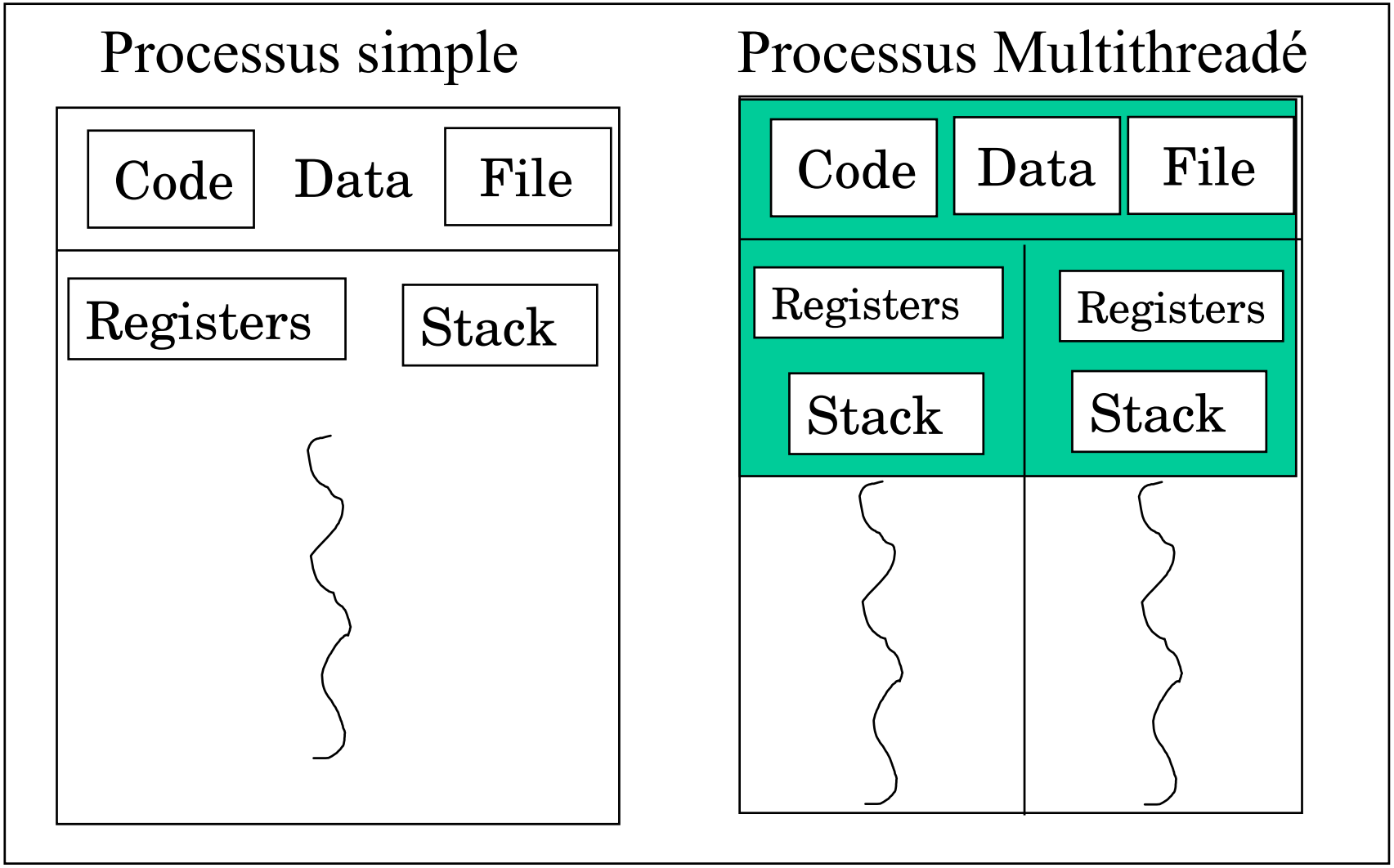
NFP121

janvier de l'an 8

MultiThreading

Thread versus **Processus**



Processus

- possède un espace de mémoire adressable contenant son état, ses variables locales, etc…

- accès protégés aux autres processus, fichiers et ressources E/S

Thread

- Possède un état d'exécution (prêt, bloqué…)
- Possède sa pile et un espace privé pour ses variables locales
- A accès à l'espace adressable et aux ressources du processus :
  - lorsqu'un thread modifie une variable non locale au thread, tous les autres threads voient la modification
  - un fichier ouvert par un thread est accessible aux autres threads (du même processus)

La commutation entre threads est moins dispendieuse que la commutation entre processus

- Un processus possède mémoire et ressources.
- Passer/Changer d'un processus à un autre implique de sauvegarder et rétablir l'état initial
- Passer/ Changer d'un thread à un autre dans un même processus est plus simple

La création et la terminaison de threads dans un processus sont également moins coûteuses

Pourquoi les threads

- Reactivité: un processus peut être subdivisé en plusieurs threads,
  - Par exemple l'un dédié à l'interaction avec les usagers, l'autre dédié à traiter des données
  - l'un peut s'exécuter tandis que l'autre est bloqué
- Partage de ressources: mémoire, fichiers
- Utilisation de multiprocesseurs: les threads peuvent exécuter en parallèle sur des UC différentes

Programmation multithread Java

- Toute méthode d'un objet java s'exécute dans au moins un thread
- Un programme « simple » exécute les threads suivants :
  - Thread principal
  - Garbage collector
  - La file d'événements graphiques

java.lang.Thread

```
class C1 extends Thread
{
  public C1() {
     this.start();
  }

  public void run() {
     // le corps du thread
  }
}
```

java.lang.Runnable

```
interface Runnable {
        public void run() ;
}

class C2 implements Runnable {
        public C2() {
                Thread t = new Thread(this);
                t.start();
        }
        public void run() {
                // le corps du thread
        }
}
```

Exemple : un compteur

```
class Compteur extends Thread {
  protected int count,  inc,  delay
  public Compteur(int init, int inc, int delay){
    this.count=init; this.inc=inc; this.delay=delay;
  }
  public void run(){
    try{
      for(;;){
        System.out.println(count + " "); count +=inc;
        sleep(delay);
      }catch(InterruptedException e){}
  }
  public static void main(String [] args){
    new Compteur(0,1,33).start();
    new Compteur(0,-1,100).start();
  }}
```

Exemple : Un générateur de valeurs aléatoires

```
class Cotation implements Runnable {
  protected int value;
  public cotation(int init){this.value=init;  }

  public void run(){
    try{
      for(;;){
        System.out.println(value);
        value+=(Math.random() -0.4) * (10.0 * Math.random());
        Thread.sleep(100);
      }catch(InterruptedException e){}
  }
  public static void main(String [] args){
    new Thread(new Cotation(100)).start();
  }}
```

Threads et sockets serveur

```
try {
       ServerSocket serveur = new
    ServerSocket(port, 8);
}catch (IOException e) { System.exit(1); }

try {
  while (true) {
    Socket s = serveur.accept();
    Ecoute ec = new Ecoute(s);
    ec.start();
  }
}catch (IOException e) {System.err.println(e);}

serveur.close();
```

Threads et sockets

```
public class Ecoute extends Thread {
  Socket s;
  public Ecoute(Socket s) {
    this.s = s;
  }

  public void run() {
    try {
      OutputStreamWriter sor =
                new OutputStreamWriter(s.getOutputStream());
      sor.write(...);
    }catch (IOException ie) {}
    //…
  }
}
```

Les verrous

- Tout objet Java dispose d'un verrou
- Utilisé avec le mot-clé **`synchronized`** associé à une méthode :

```
synchronized void metAjour() { … }
```

- Alors un seul *thread* à la fois peut appeler la méthode, sinon verrouillage

synchronized

```
class ComptePartageable {
    synchronized void ajouter(int montant) {...}
    synchronized void retirer(int montant) {...}
}

class Client implements Runnable {
     ComptePartageable c;
    public Client(ComptePartageable c) {
            this.c = c;
            Thread t = new Thread(this); t.start();
    }
    public void run() { ... c.ajouter(100); ... c.retirer(10); ...}
}

ComptePartageable c = new ComptePartageable ();
Client c1 = new Client (c); Client c2 = new Client(c);
```

java.lang.Object : wait, notify, …

- tout objet possède les méthodes *(héritées d'Object)*
  - **`wait()`**, **`notify()`** et **`notifyAll()`**

- Dans un bloc **`synchronized{…}`** d'un objet O :
  - **`wait()`** relache le verrou et se met en attente.
  - **`notify()`** reveille un *thread* en attente
  - **`notifyAll()`** reveille tous les *threads* en attente

Producer Consumer Coordination

```
public class Share {
  private int s;
  private boolean empty = true;

  public synchronized int get() {
    while (empty == true) {
      try {
        wait(); // nothing to get, wait
      } catch (InterruptedException e) {}
    }
    empty = true;
    notifyAll(); // wakeup waiting Consumers/Producers
    return s;
  }
  public synchronized void put(int s) {
    while (empty == false) {
      try {
        wait(); // no room
      } catch (InterruptedException e) {}
    }
    this.s = s;
    empty = false;
    notifyAll(); // wakeup waiting Consumers/Producers
  }
}
```

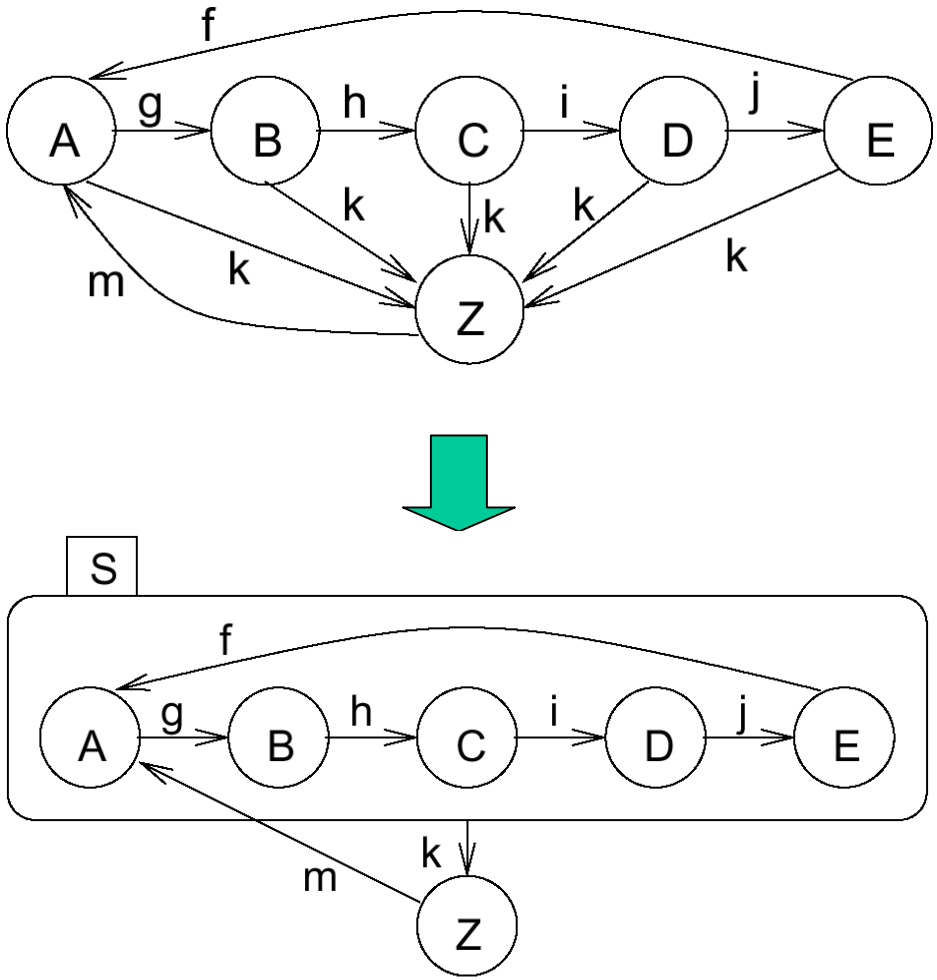
priorités

- Scheduling et priorité :
  - Le scheduling est en partie dépendant des implémentations
  - setPriority() permet de fixer la priorité d'un *thread*
  - Pour 2 threads de même priorité, par défaut : *round robin (à chacun son temps)*
  - T1 cède la place aà T2 quand sleep(), wait(), bloque sur un synchronized, yield(), stop()

Introduction aux statecharts

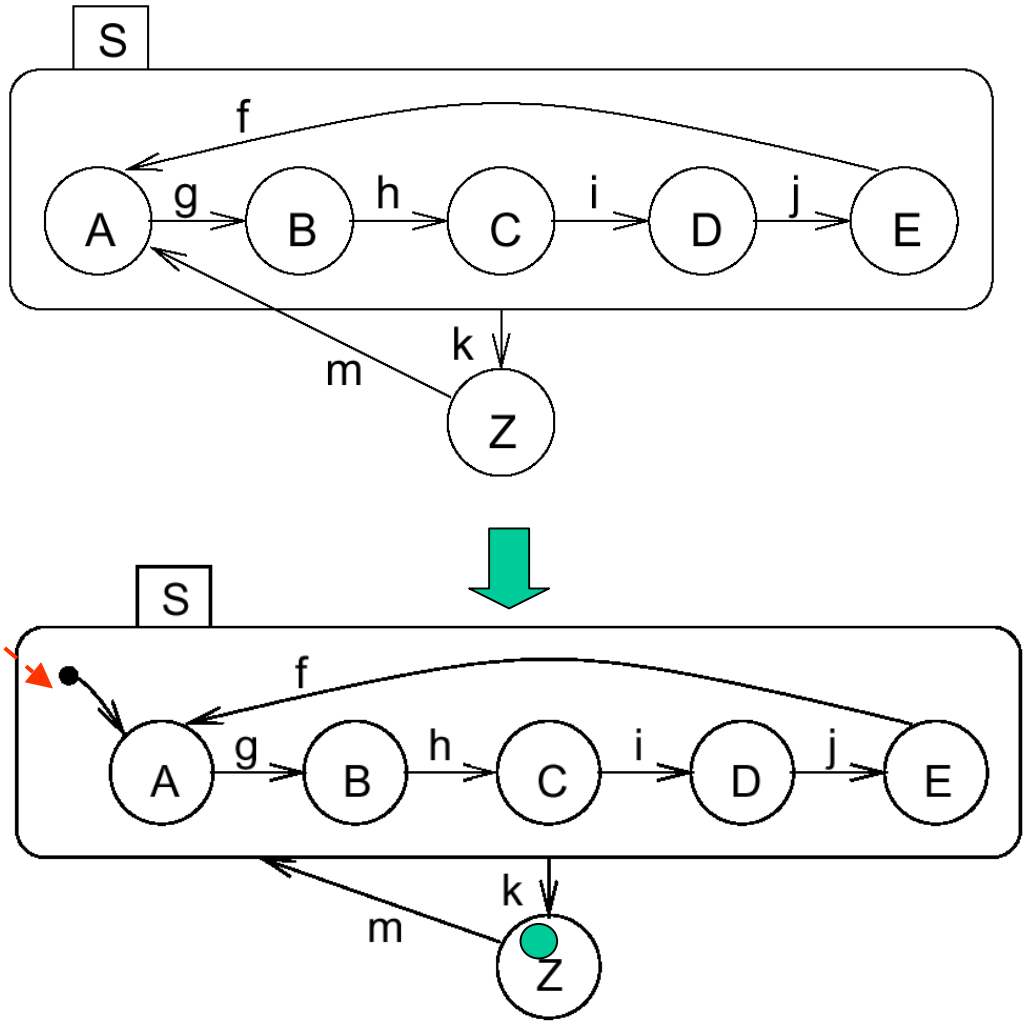
Diagrammes d'états transitions d'UML

Des diagrammes Etats/Transitions aux statecharts hiérachisés



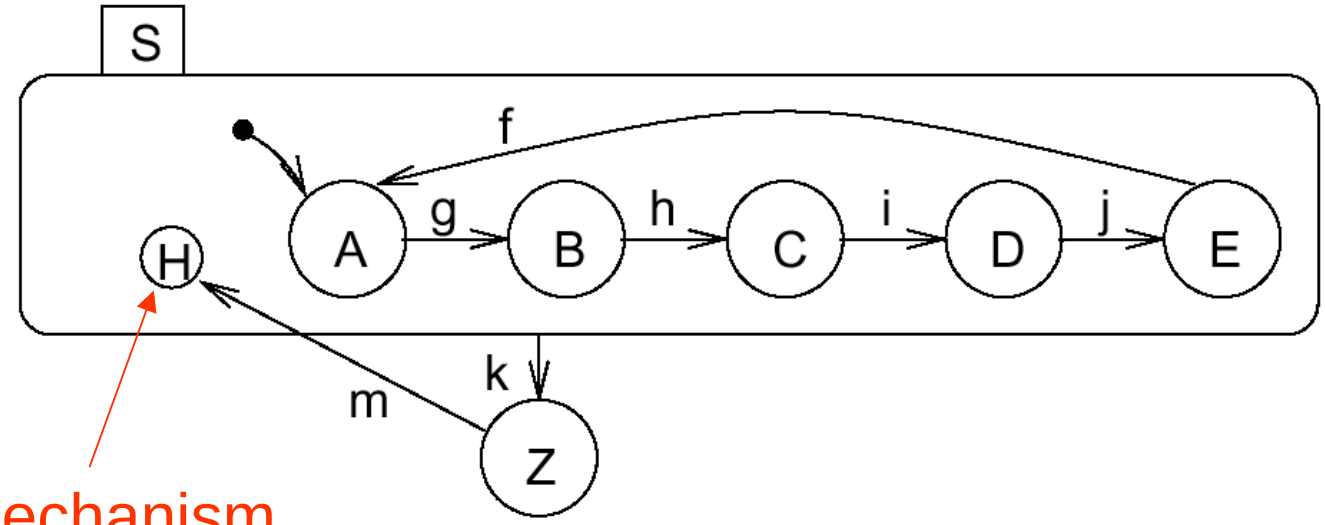
Dans exactement un des sous-états de S si S est **actif**
(soit dans A soit dans B ..)

Etat interne initial



- Désigne l'état initial quand on entre dans S
- Ce n'est pas un état en soit!

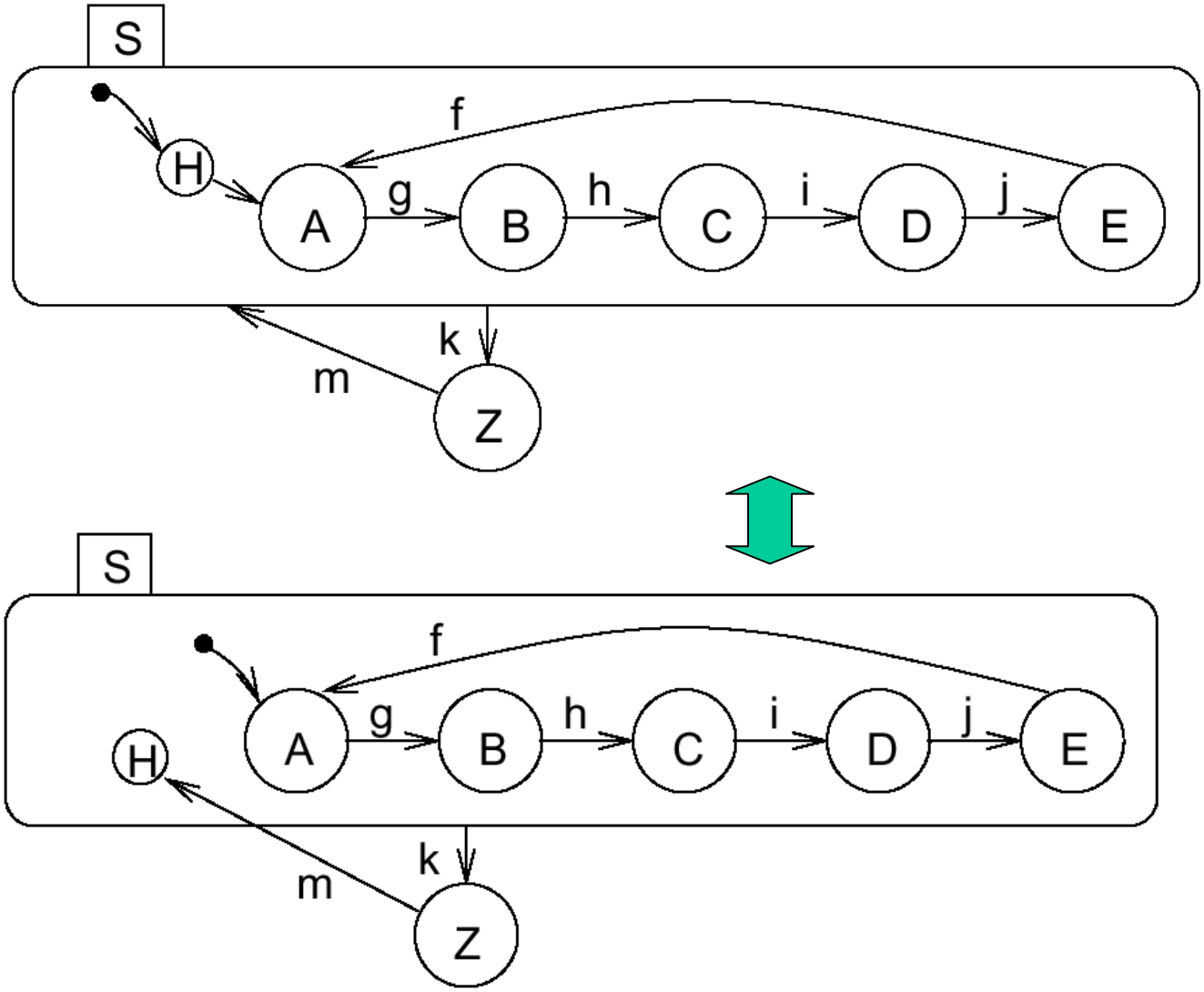
History



History mechanism

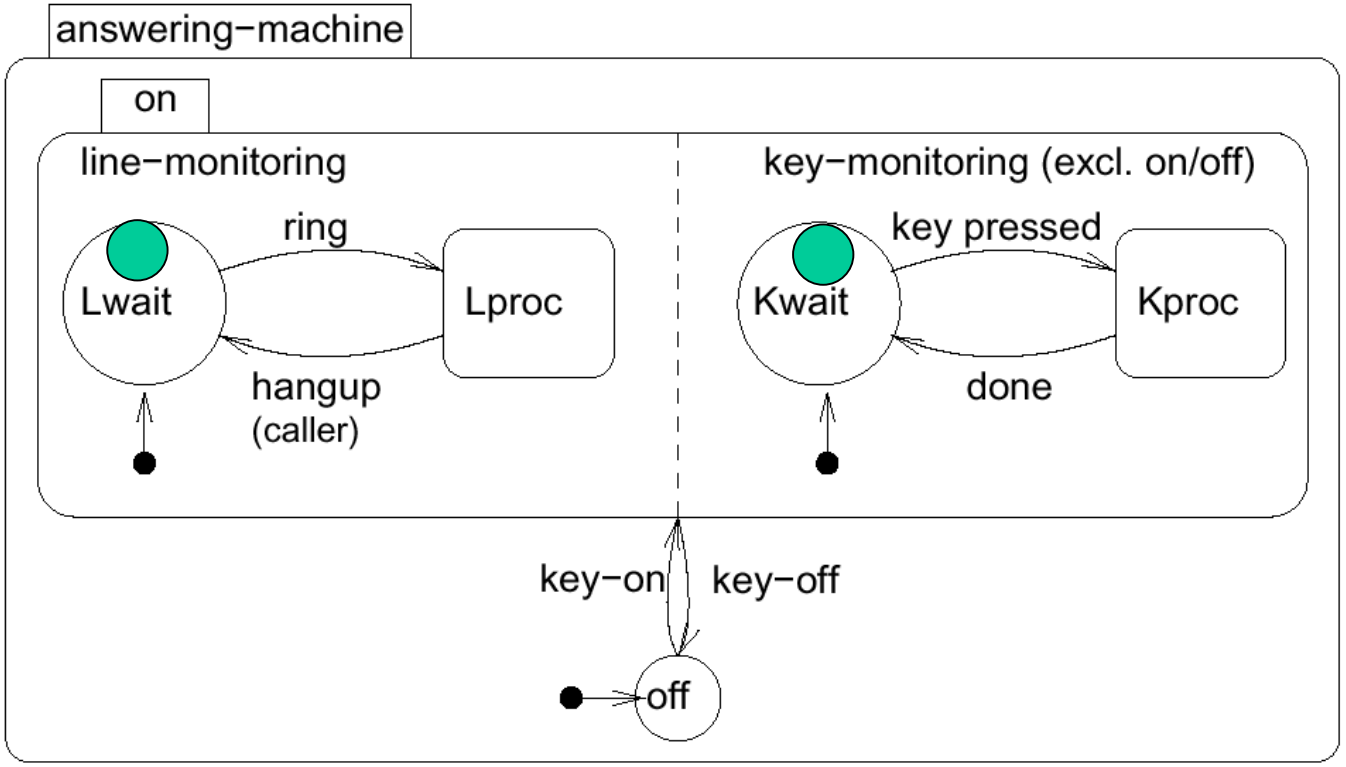
- m provoque l'entrée dans S dans l'état qui était celui du système avant que S ne soit quitté (cela peut-être A, B, C, D, or E).
- Si on accède à S pour la première fois c'est dans l'état A

Notation combinée



Concurrence

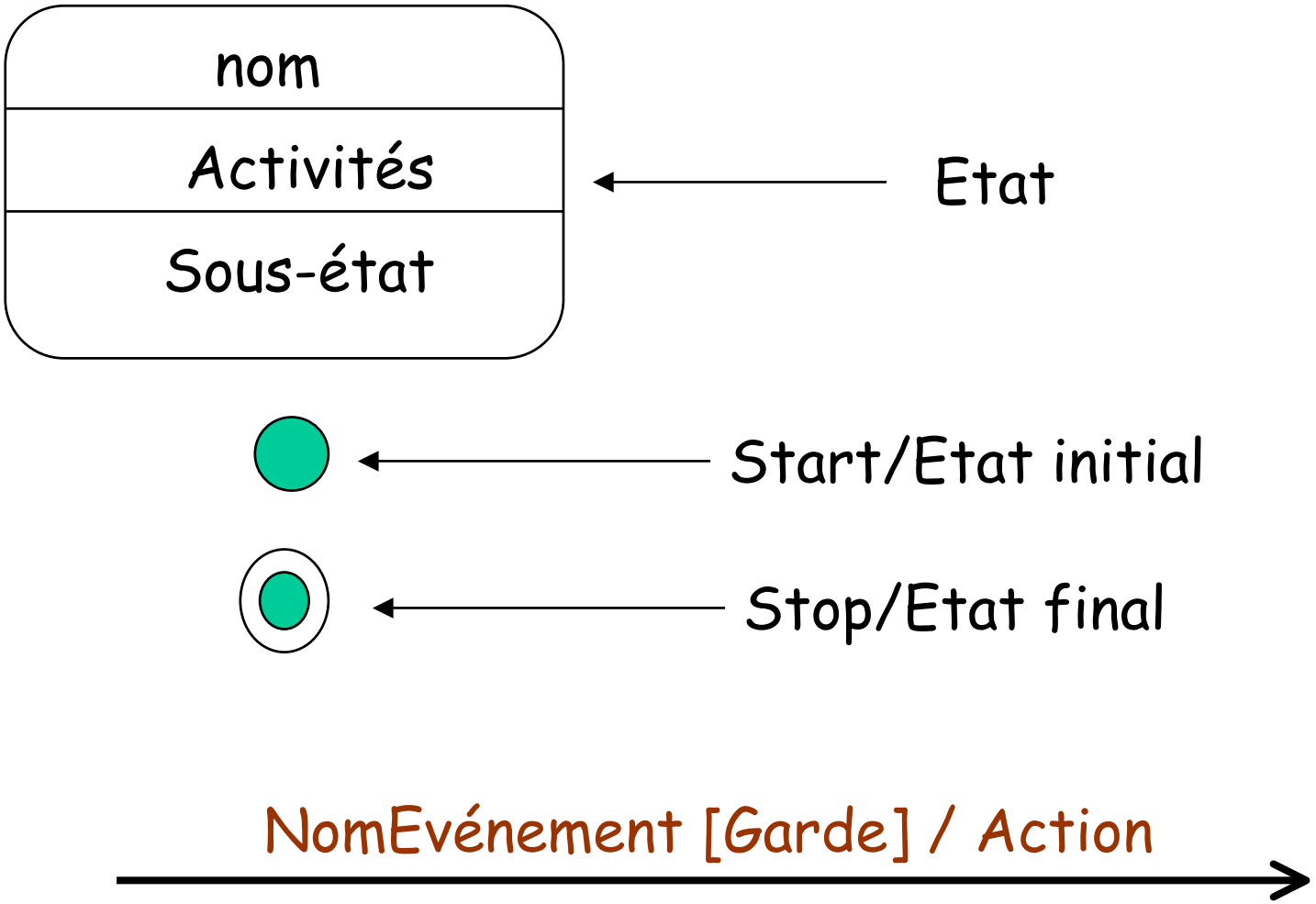
- **AND-super-states**:



Récapitulatif des Concepts

- Etats (simples ou composites)
- Transitions
- Evénements
- Actions
- Activités
  - entry, exit, do,include

Etat et transitions



Evénements et gardes

- nomEvénement (param1:type, param2:type,…)
- Externes (utilisateur, environnement,…) ou internes
- [Garde] : expression Booléenne Ex: [ageCapitaine>50]
- Evénement est observé par le système, la garde est vérifiée lors de la transition (c-à-d pas observée à tout moment)

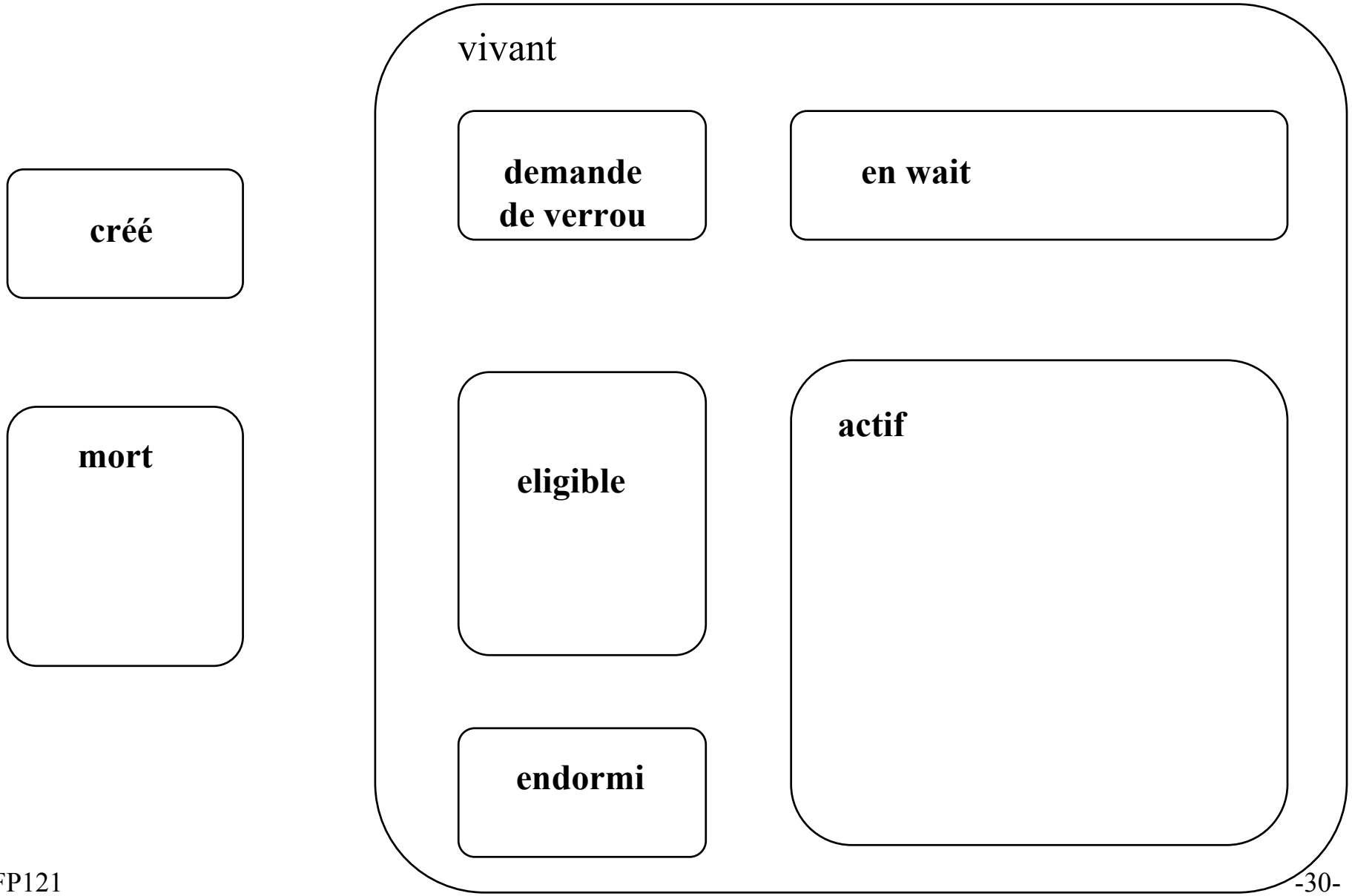
Actions

- Action
  - instruction
    - pseudo-code
- ex. solde = solde – débit
  - appel d'un événement :
    - ^cible.événement(param)
    - Ex : ^Compte.paiement(solde)
- Action interne à un état
  - ActionLabel/Action
    - Action Label :
      - entry, do, exit, on event
    - Ex. : entry/calculateAverage(x:array)

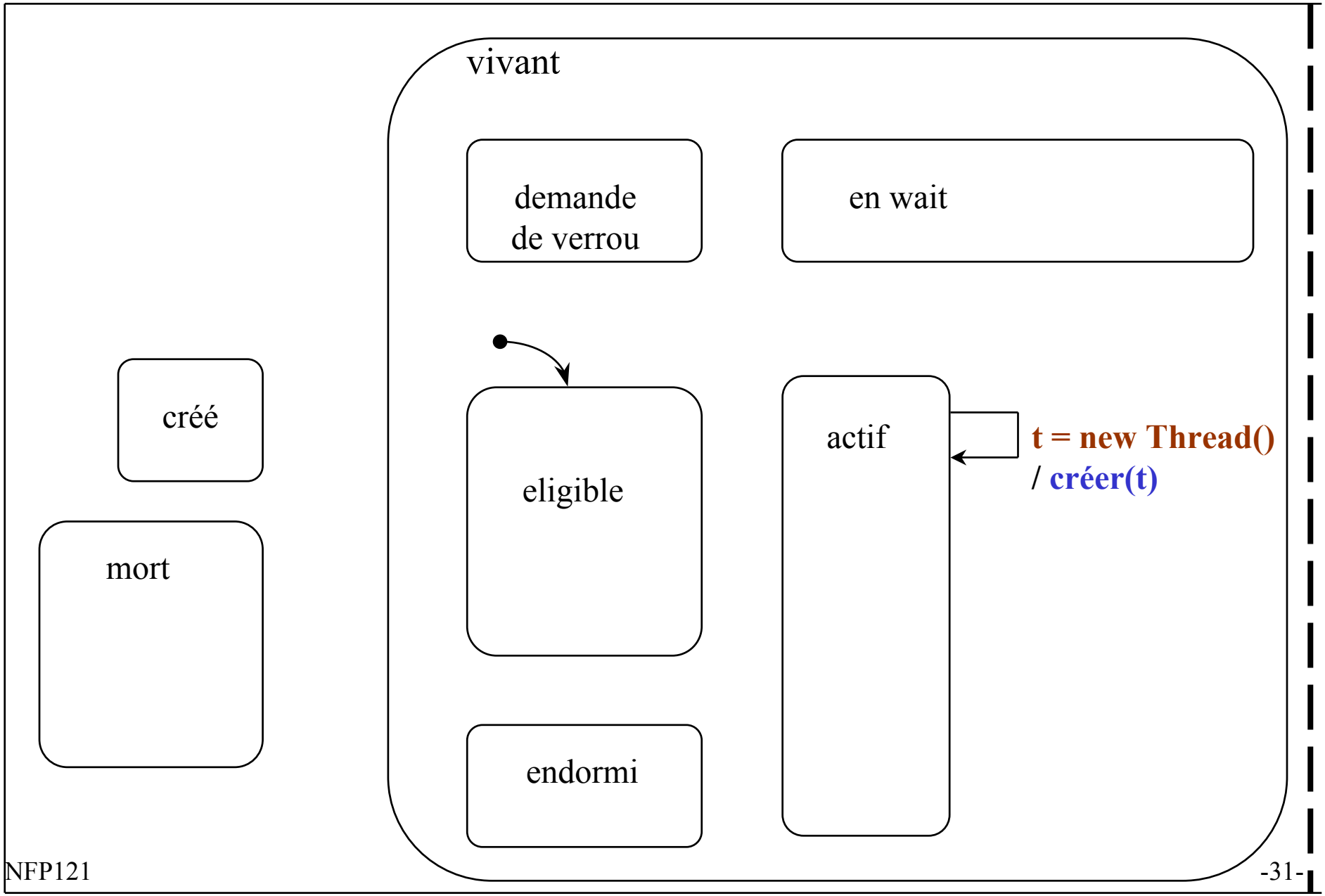
Le thread dans tous ses états !!!

Diagrammes de harel (statecharts)

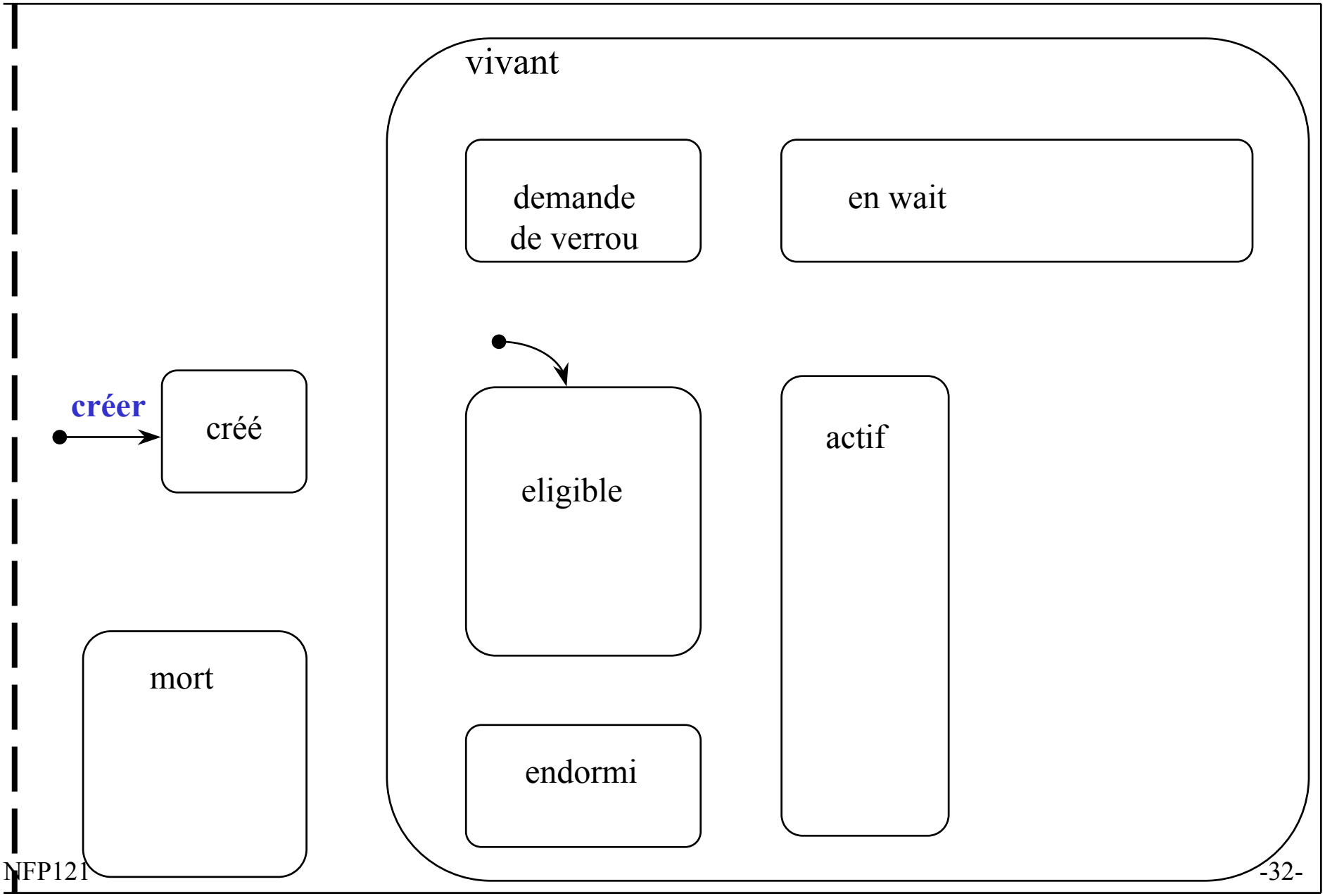
Etats d'un thread



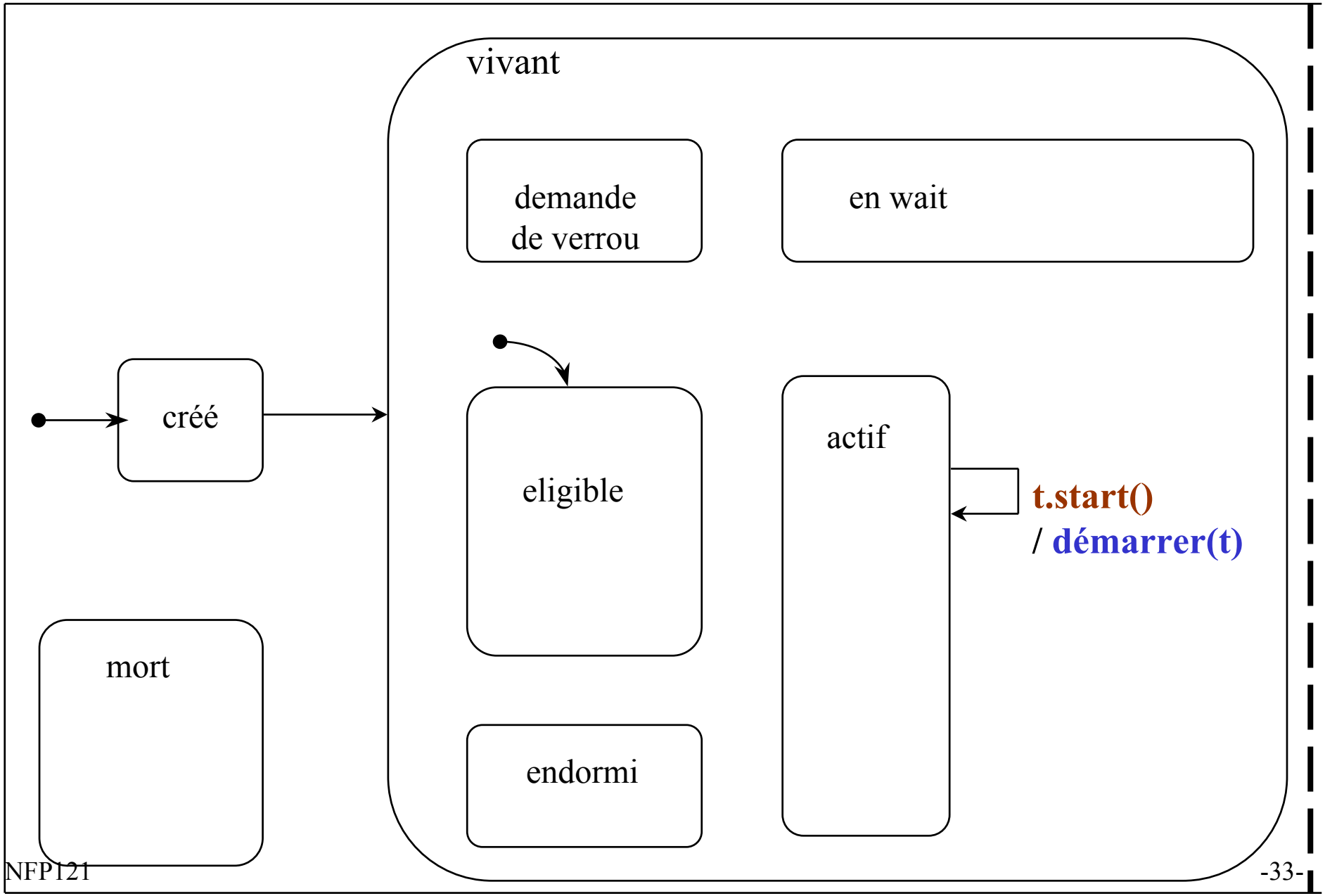
Création d'un thread t dans un thread



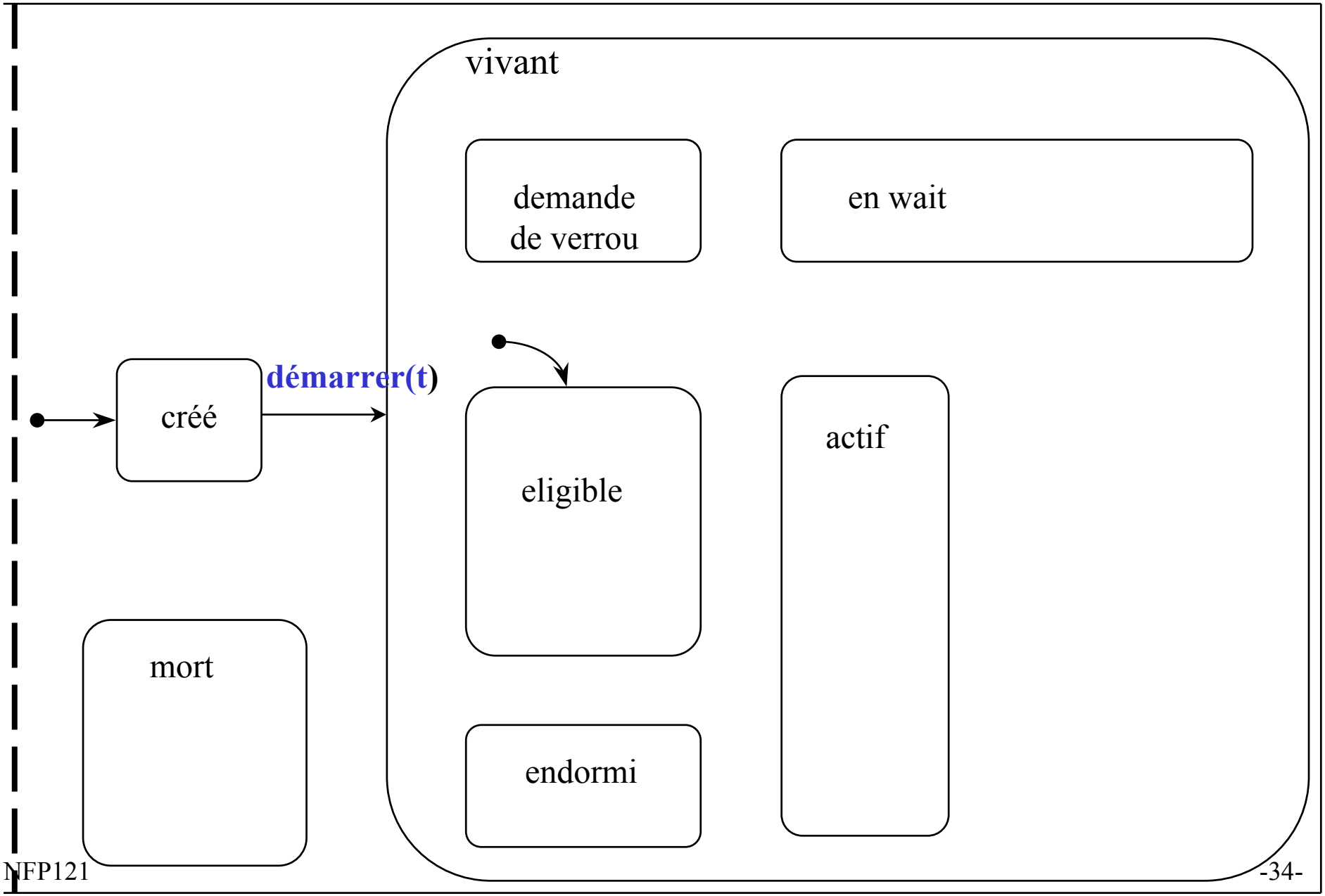
Thread t (créé en //)



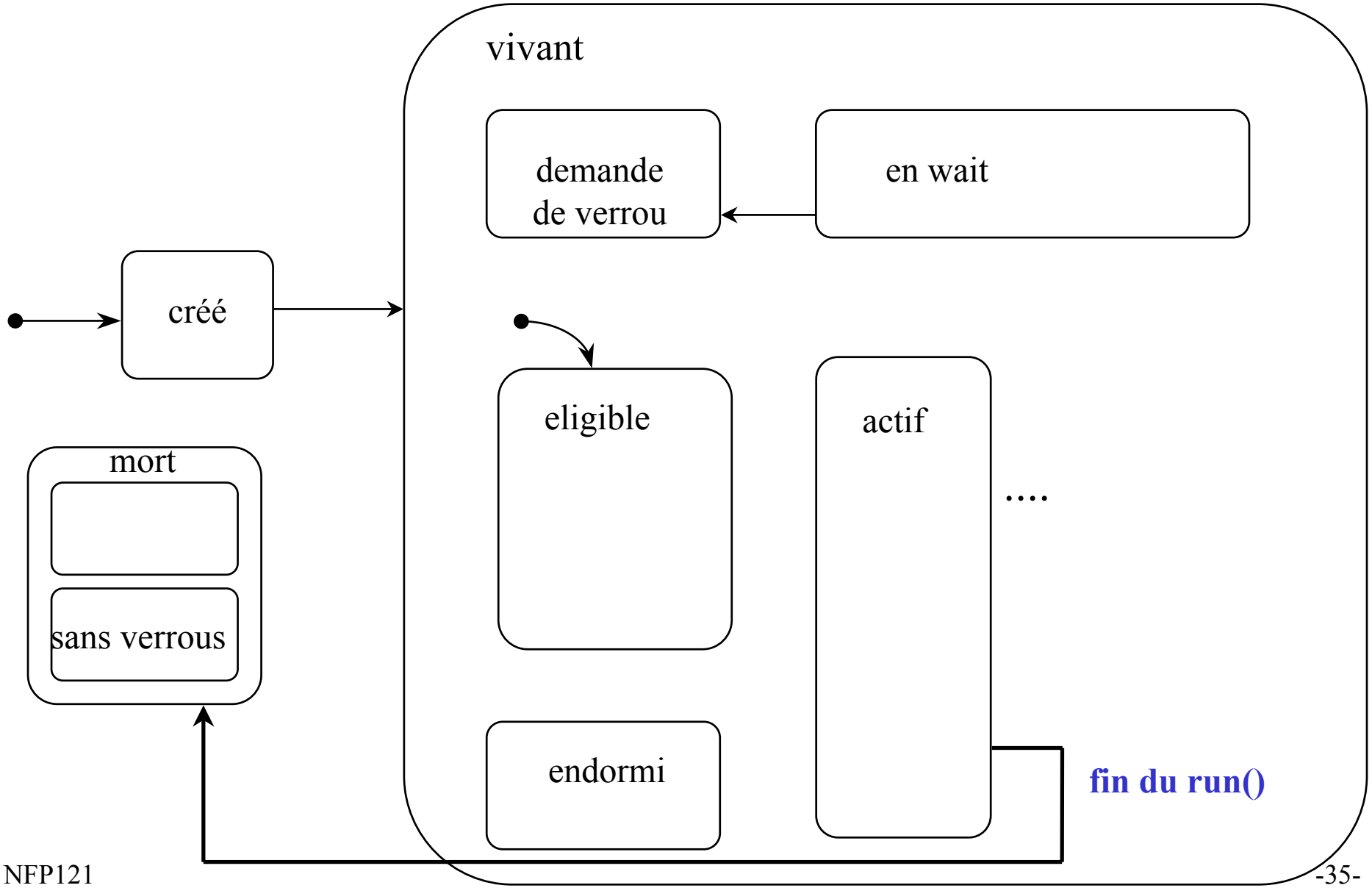
Lancement d'un thread t par un thread



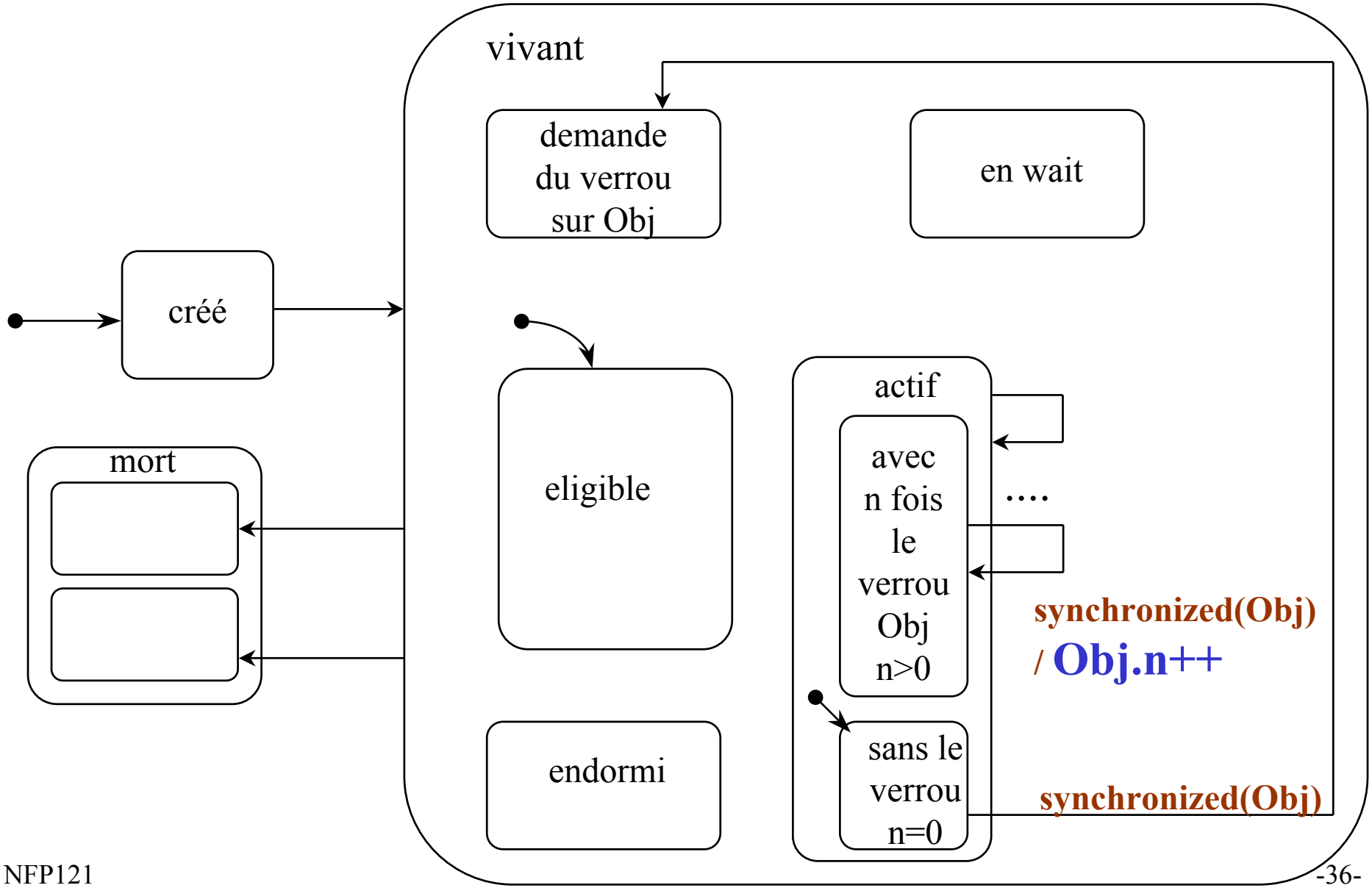
lancement de t



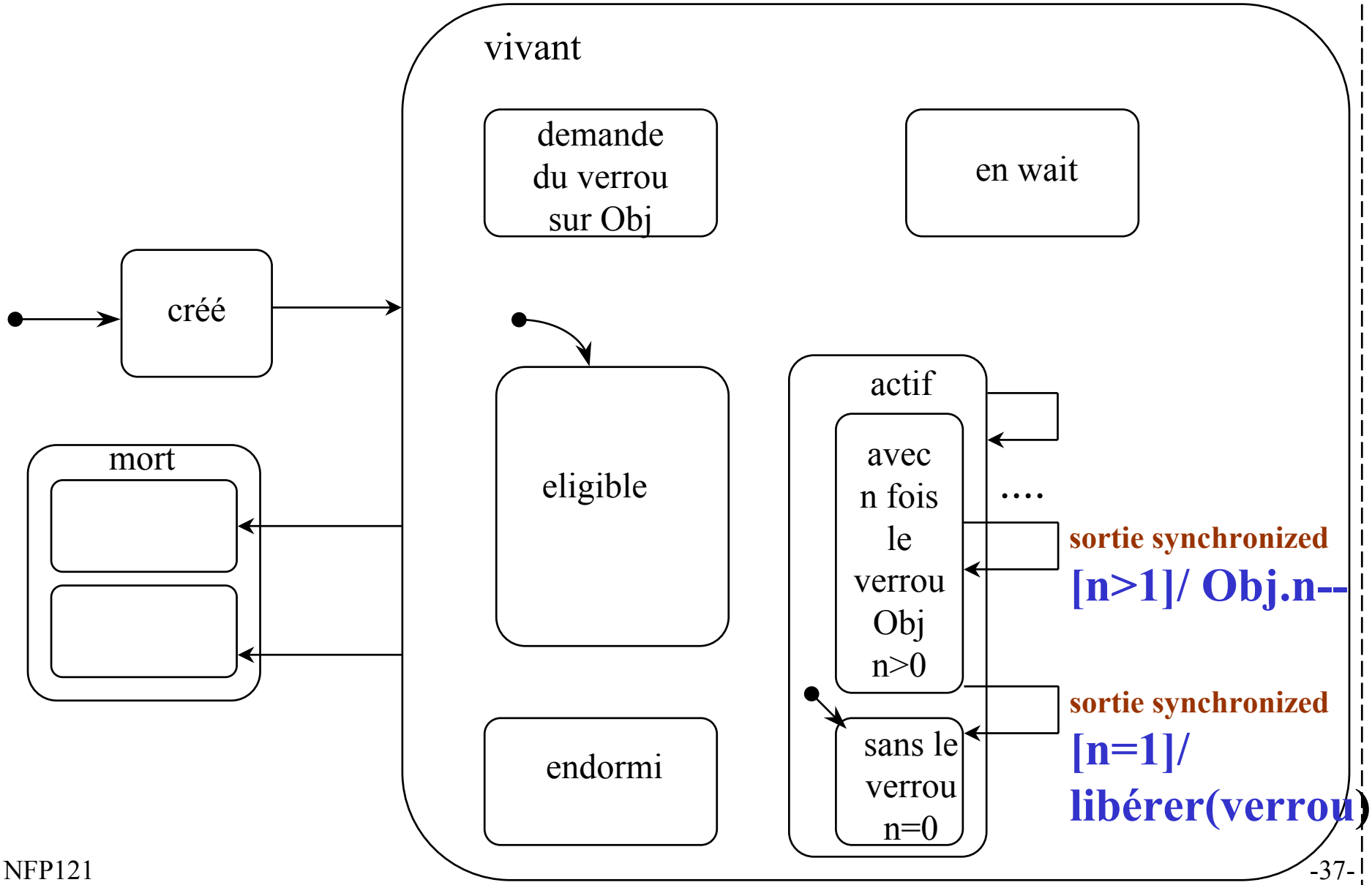
fin tragique d'un Thread (fin du run…)



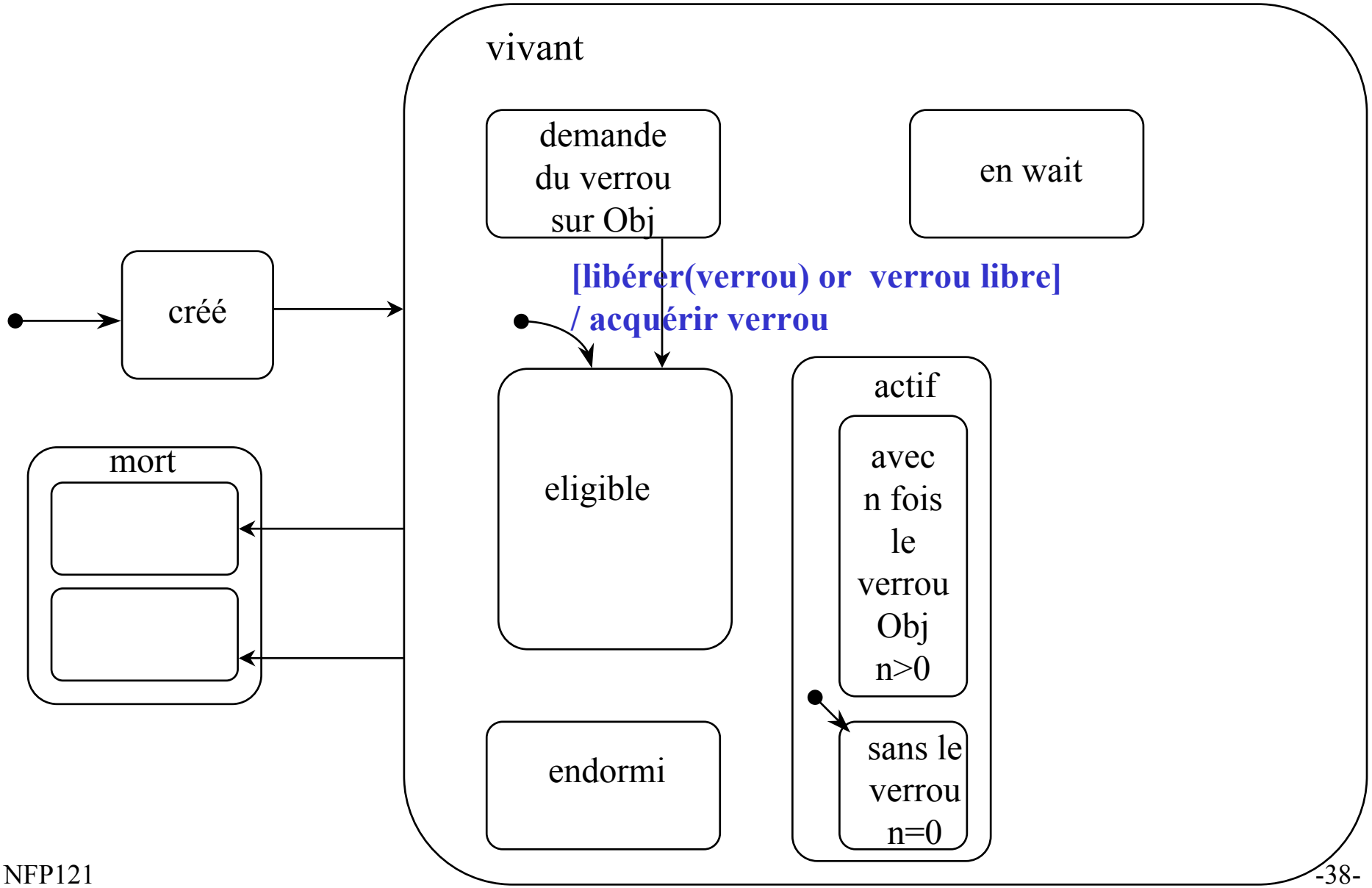
synchronized



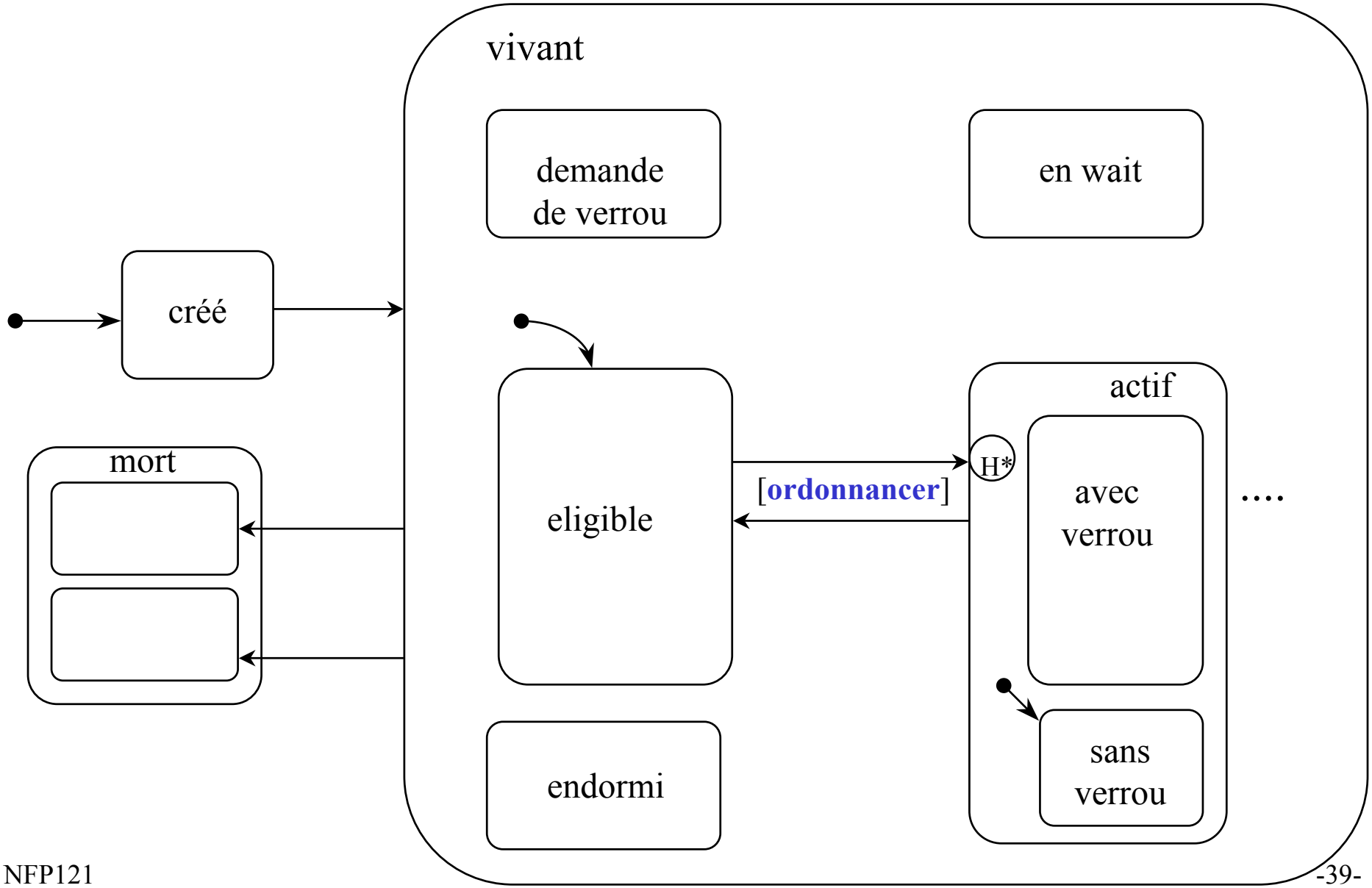
Sortie d'un bloc synchronized



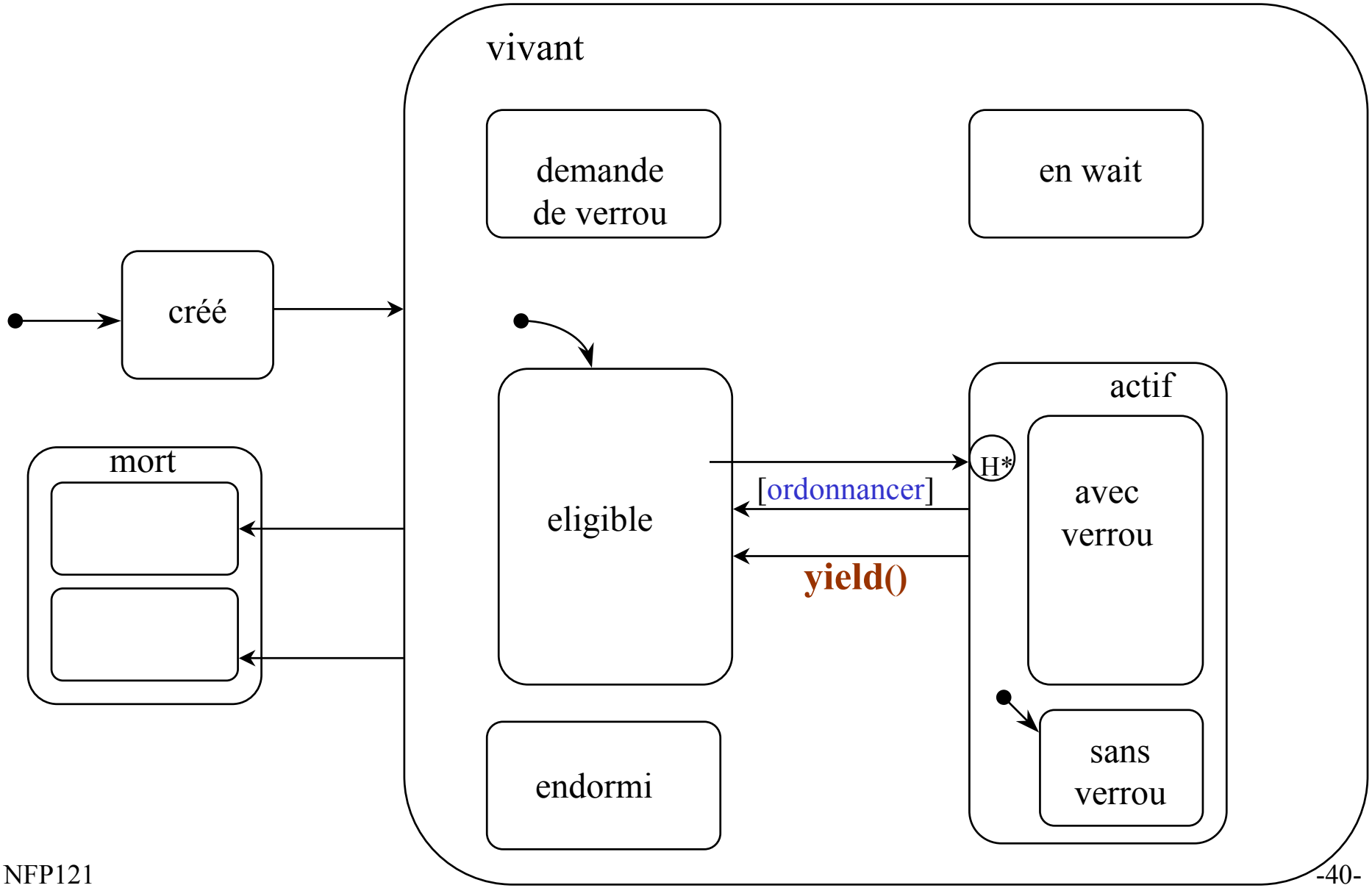
acquisition d'un verrou



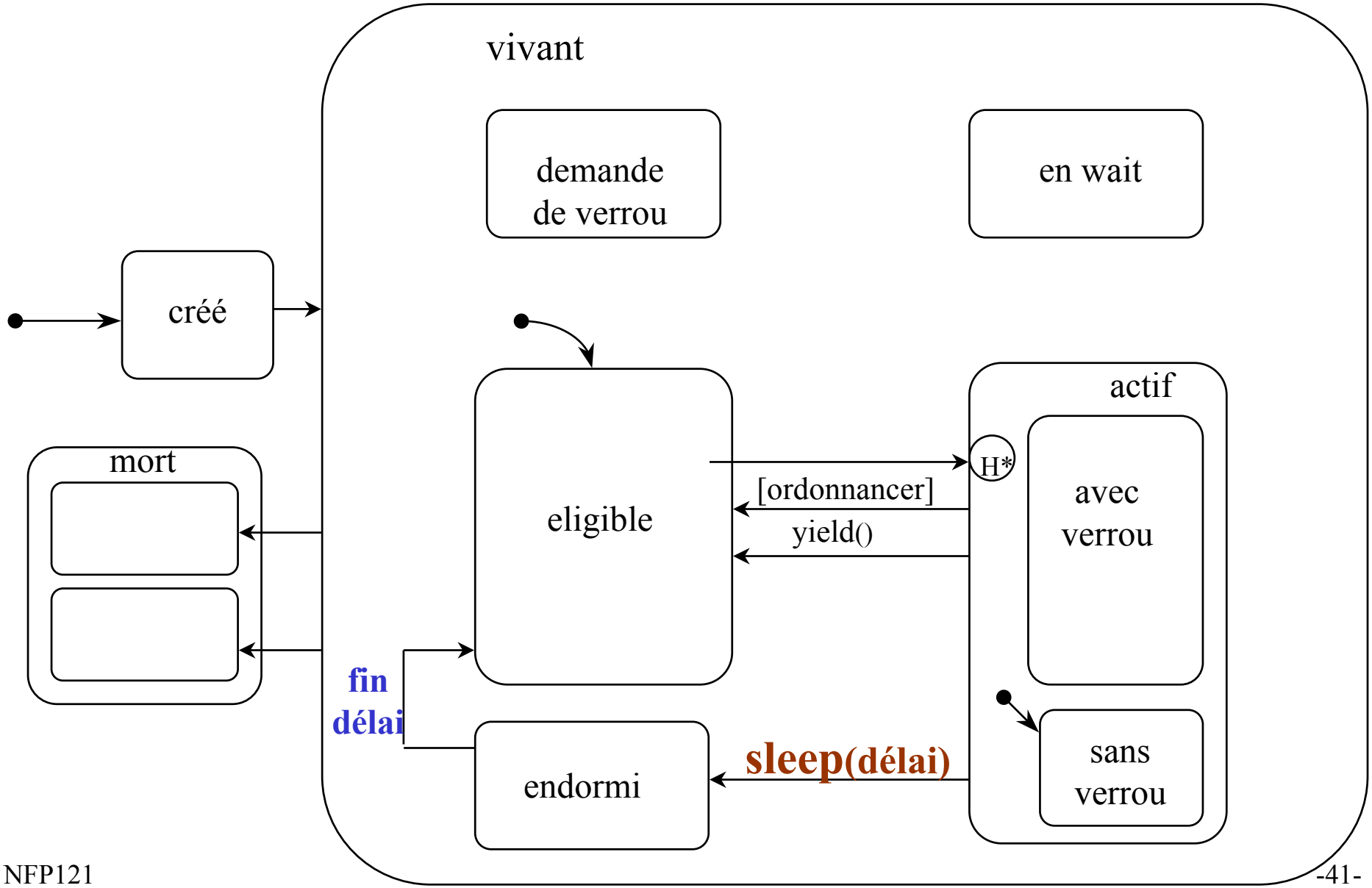
Ordonnancement des threads



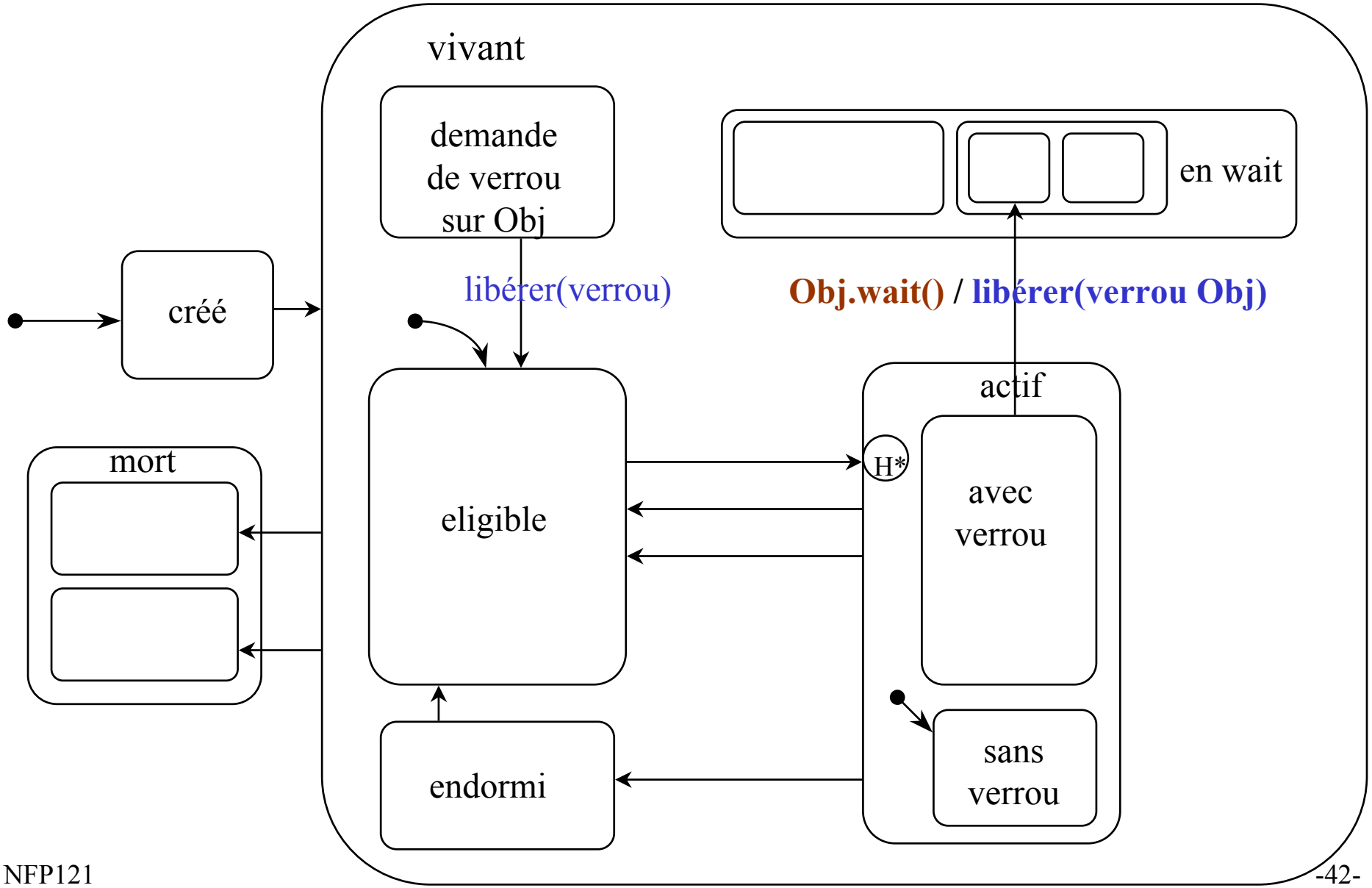
Forcer l'ordonnancement



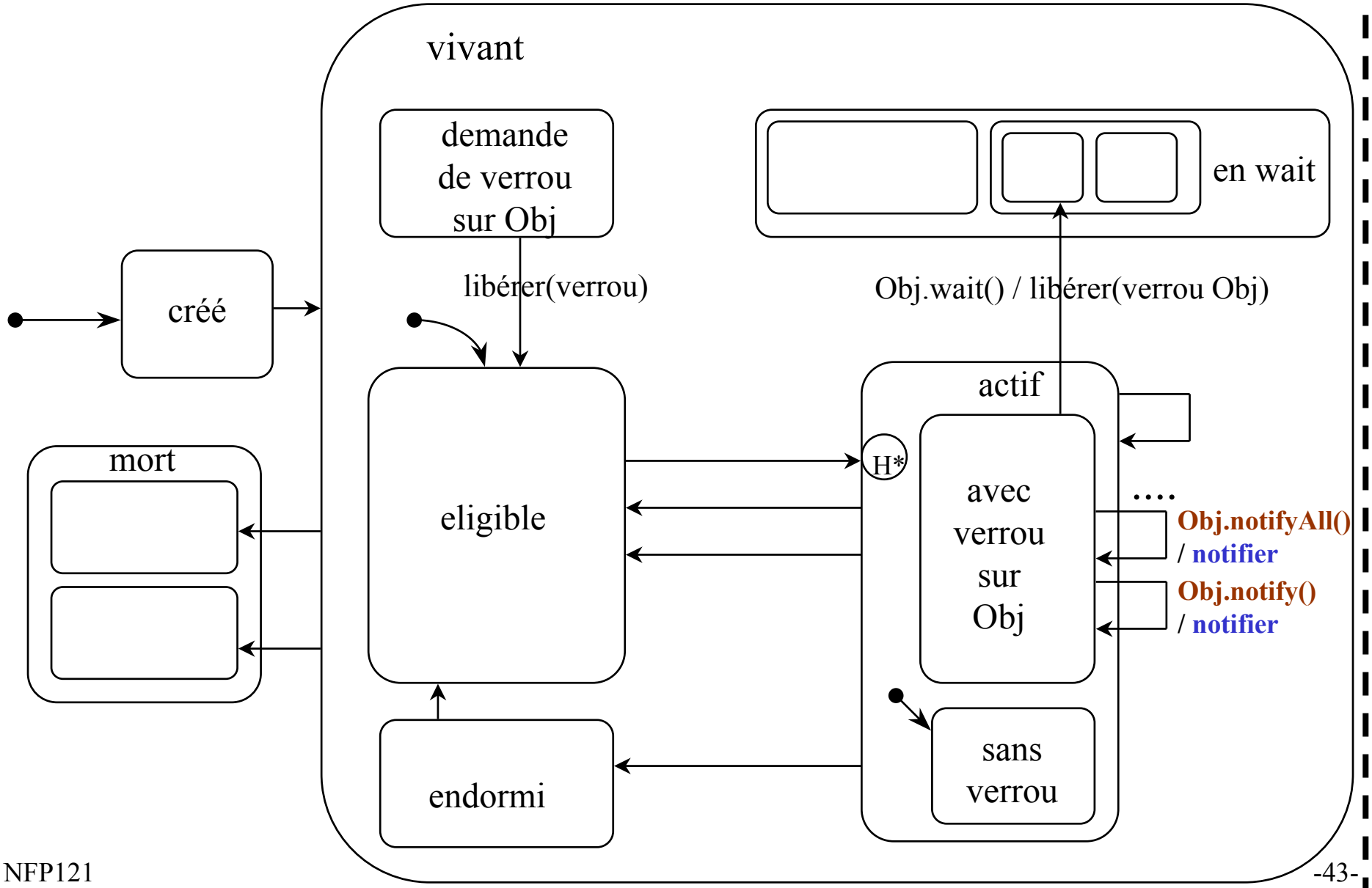
Endormir un Thread



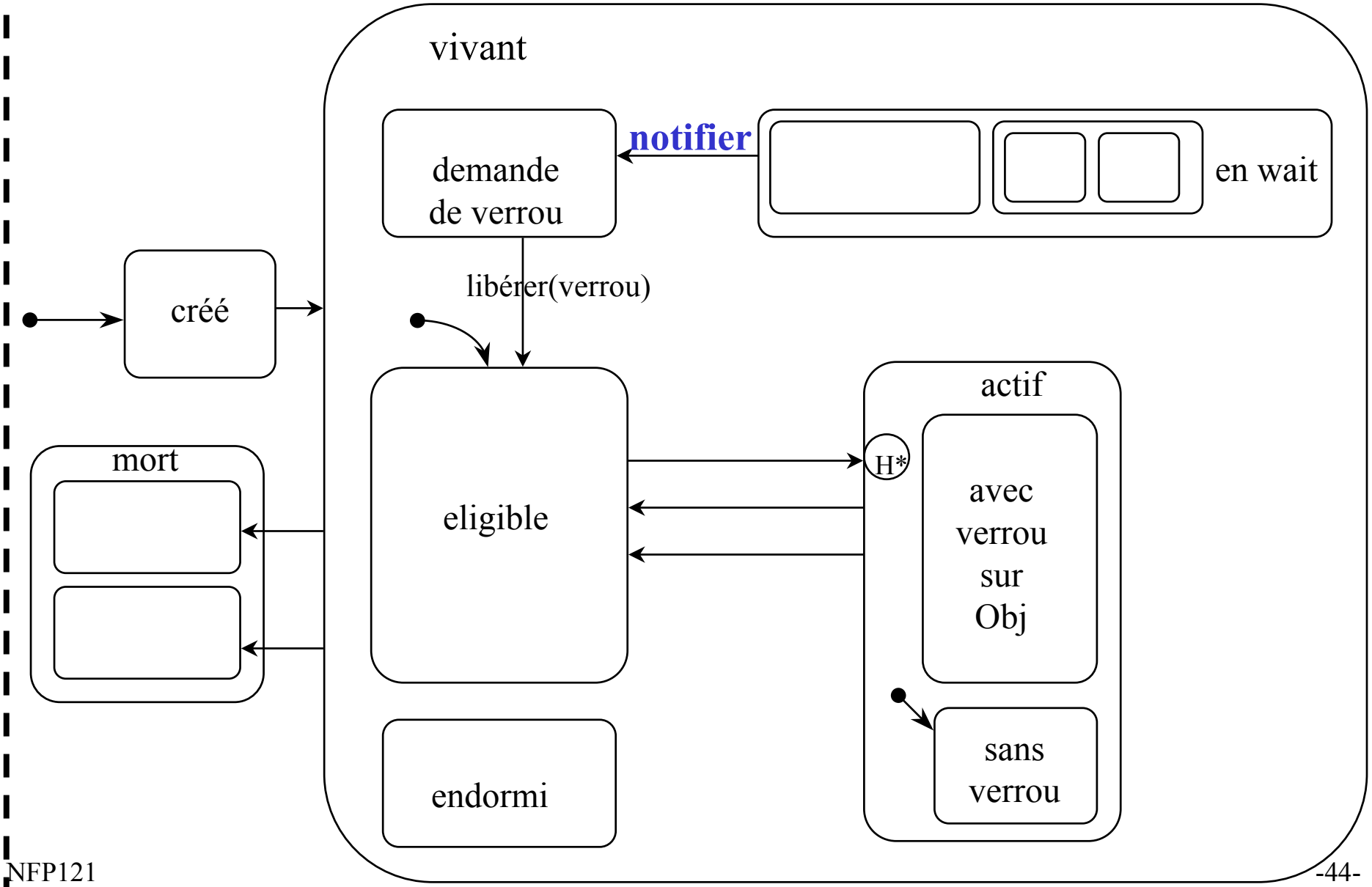
Attente sur une condition



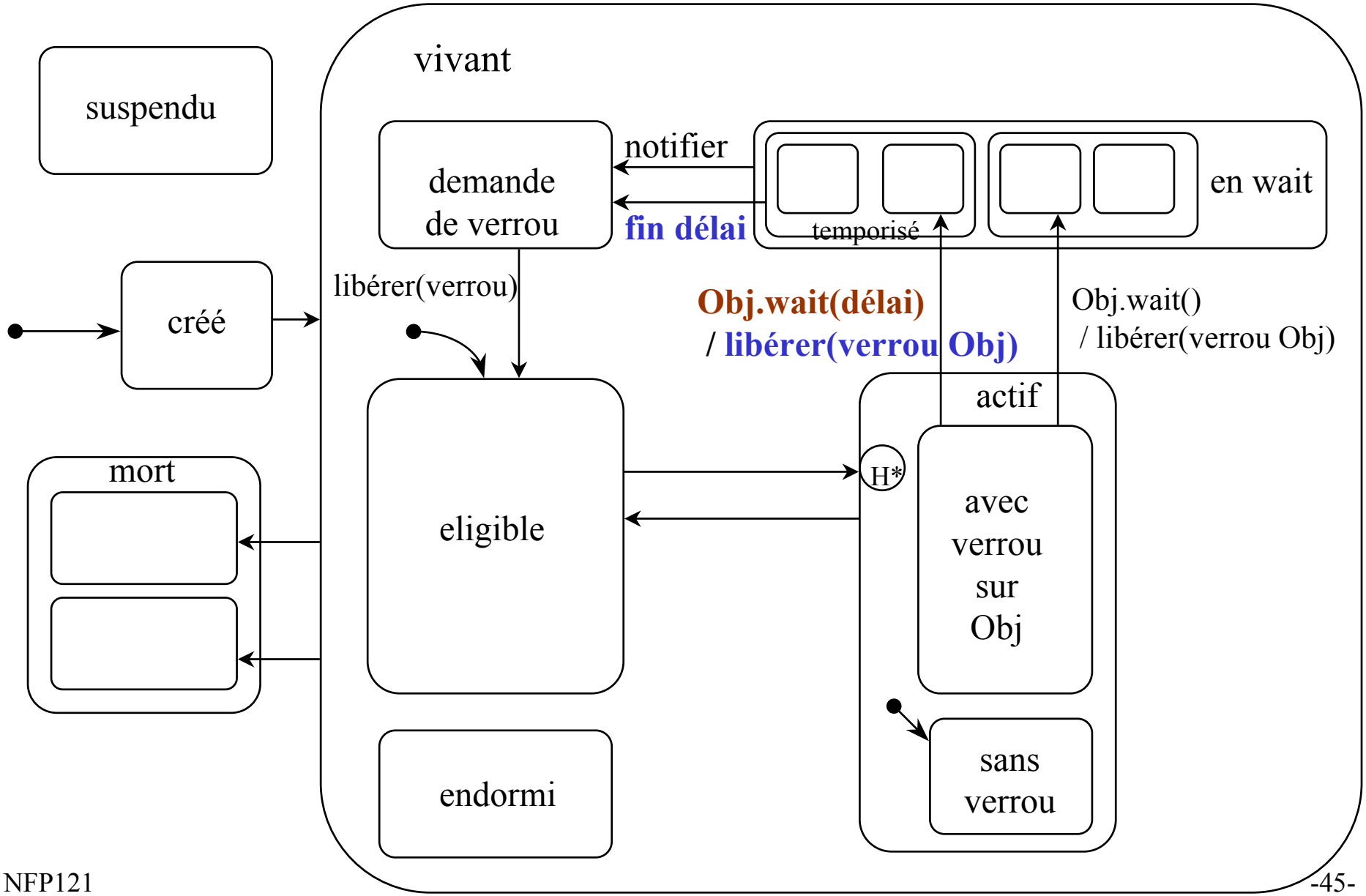
Libérer un thread en attente



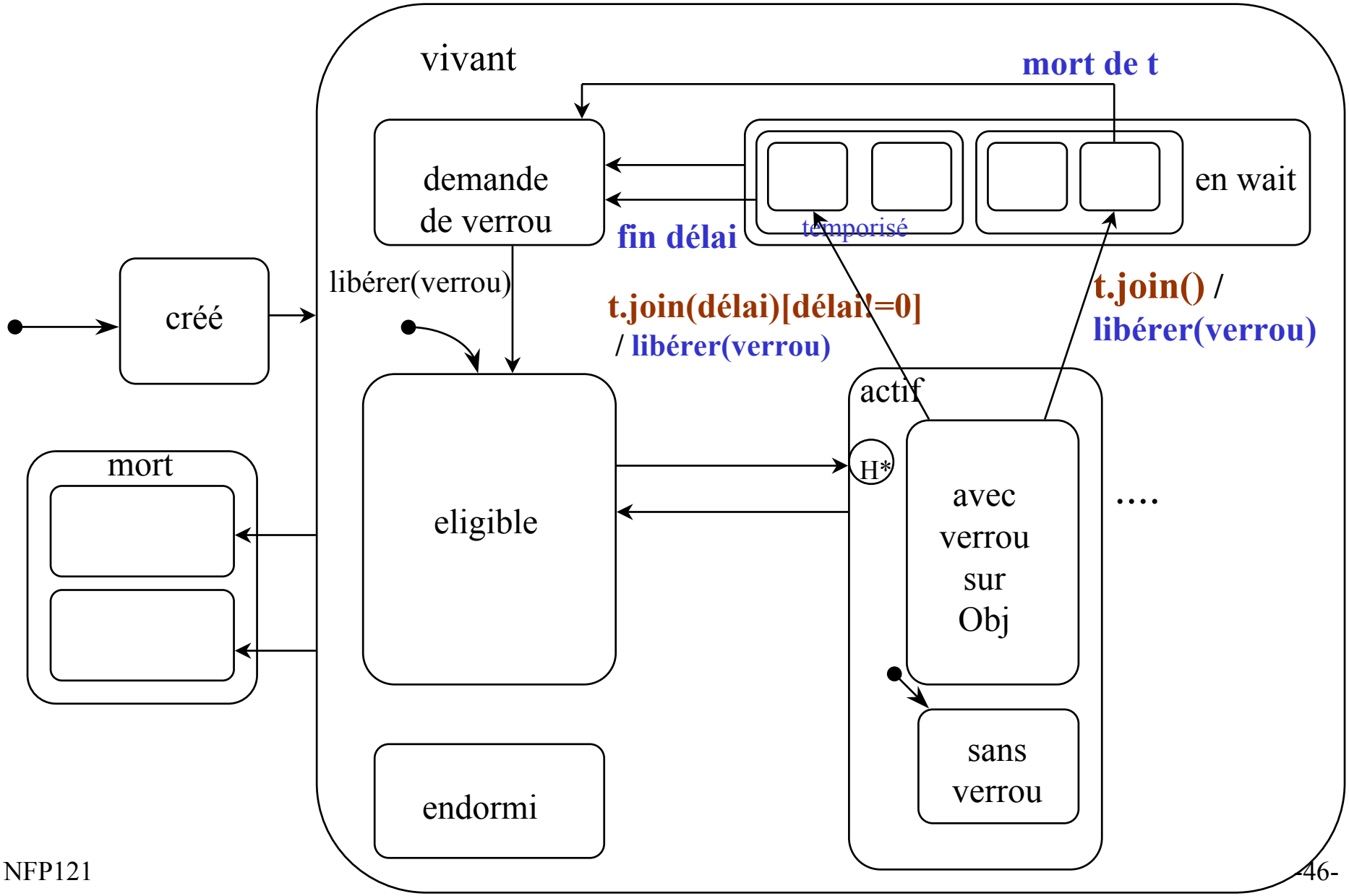
libération


vivant
demande
de verrou
notifier
en wait
libérer(verrou)
créé
mort
eligible
actif
H*
avec
verrou
sur
Obj
sans
verrou
endormi

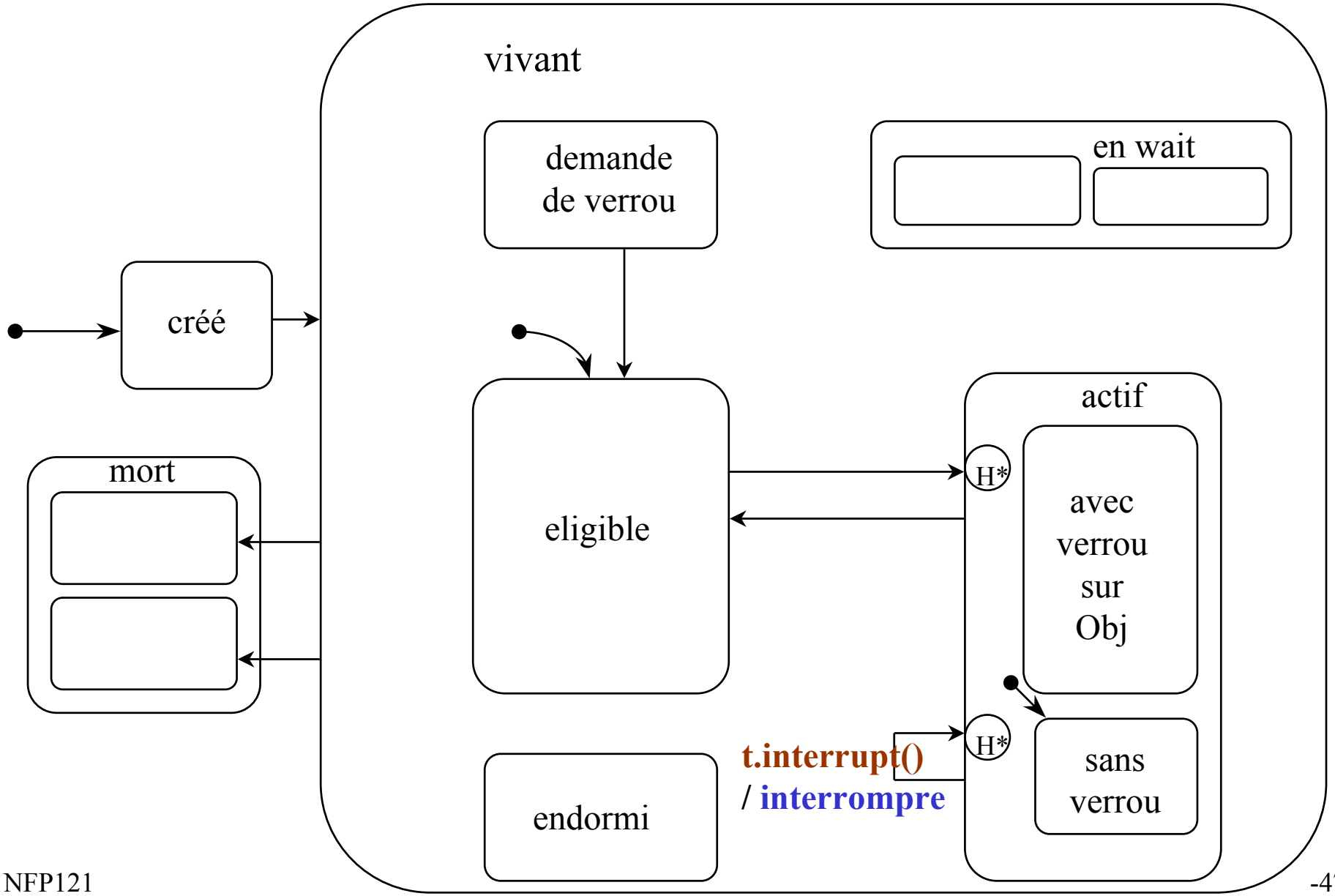
Attente conditionnelle limitée



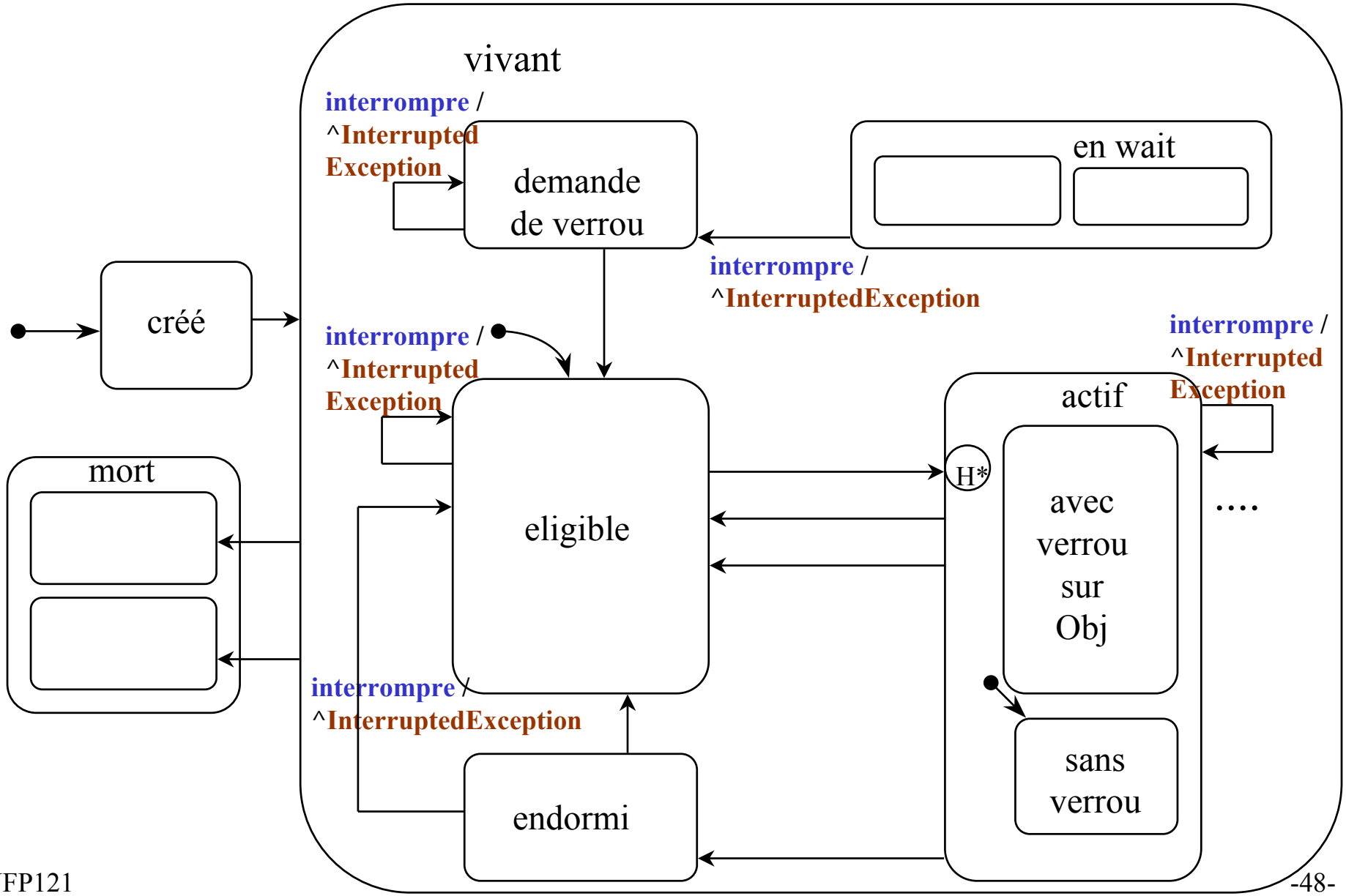
attente morbide



interrupt()


vivant
demande
de verrou
en wait
créé
mort
eligible
actif
H*
avec
verrou
sur
Obj
H*
sans
verrou
endormi
t.interrupt()
/ interrompre

Interrompu


vivant
interrompre /
^InterruptedException
demande de verrou
en wait
interrompre /
^InterruptedException
créé
interrompre /
^InterruptedException
interrompre /
^InterruptedException
actif
mort
H*
avec verrou sur Obj
....
eligible
interrompre /
^InterruptedException
sans verrou
endormi

ThreadGroup

Tous les threads appartiennent à un groupe

Les groupes forment un arbre dont le groupe racine est «system»

Les groupes permettent de gérer un ensemble de threads et de structurer la sécurité

Un thread appartient normalement au groupe du thread qui le crée

Créer un thread dans un autre groupe exige un contrôle du Security Manager (voir méthode checkAccess de la classe ThreadGroup)

De même créer un groupe est contrôlé par le SecurityManager

Créer un thread dans un groupe

t= new Thread();

t= new Thread(nom); // avec nom de classe String

t= new Thread(runnableObject);

t= new Thread(runnableObject, nom);

t= new Thread(threadGoup, nom);

t= new Thread(threadGoup, runnableObject);

t= new Thread(threadGoup, runnableObject, nom);