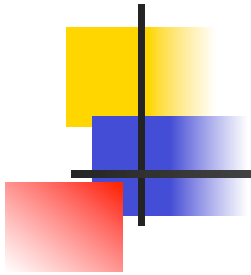
Programmation Concurrente en Ada

Laurent Pautet

Laurent.Pautet@enst.fr

Version 1.0



Ada
Plan

- Objet tâche et type tâche
- Gestion du temps
- Synchronisation directe par rendez-vous
- Synchronisation sélective
- Synchronisation indirecte par objet protégé
- Synchronisation différée
- Avortement et asynchronisme
- (tutorial Ada 95 – http://www.enst.fr/~pautet/Ada95)

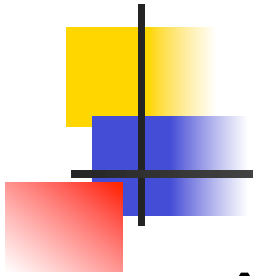
Ada
Définition des tâches

```
procedure Main is
begin
   Find_BD_Phone_Nbr;
   Find_JH_Phone_Nbr;
   Find_LP_Phone_Nbr;
end Main;
```

- Soit trois activités indépendantes, on peut les exécuter :
- En séquence (ci-dessus)
- En parallèle (ci-contre)

```
procedure Main is

   task BD_Phone_Nbr;
   task body BD_Phone_Nbr is
   begin
      Find_BD_Phone_Nbr;
   end BD_Phone_Nbr;

   task JH_Phone_Nbr;
   task body JH_Phone_Nbr is
   begin
      Find_JH_Phone_Nbr;
   end JH_Phone_Nbr;

begin
   Find_LP_Phone_Nbr;
end Main;
```

Ada
Tâches: Spécification et Implémentation



- Une tâche se compose
  - D'une spécification qui inclut
    - Un nom
    - Une partie visible
      - Avec déclarations d'entrées
    - *Une partie privée*
      - *Avec déclarations d'entrées*
  - D'une implémentation qui inclut
    - Un nom (voir spécification)
    - Une liste de déclarations
    - Une liste d'instructions

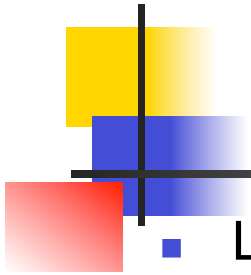
```
task T is
   entry E (P : Parameter);
   -- entrées optionnelles
end T;

task body T is
   D : Declarations;
   -- déclarations optionnelles
begin
   -- liste d'instructions
   Initialize (D);
   accept E (P : Parameter) do
     Modify (D, P);
   end E;
   Finalize (D);
end T;
```

Ada
Gestion du temps

```
task body Cyclic_Task is
   T1, T2 : Time;
   P : Duration := 0.5;
   -- Période
begin
   loop
      T1 := Clock;
      -- instructions
      T2 := Clock;
      if T2 - T1 > P then
         raise Time_Error;
      end if;
      delay T1 + P - T2;
      -- ou encore
      -- delay until T1 + P;
   end loop;
end Cyclic_Task;
```

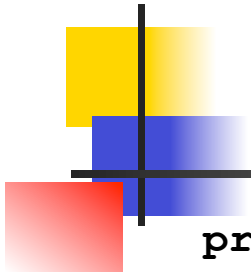
- Ada.Calendar et Ada.Real_Time fournissent des fonctions de gestion du temps :
  - Le type Time représente une date (01/01/03 20:54:37)
  - Le type Duration représente une durée (10s)
  - La fonction Clock retourne l'horloge de type Time
- Un délai peut être
  - absolu de type Time (**delay until**)
  - relatif de type Duration (**delay**)

Ada
Synchronisation par rendez-vous



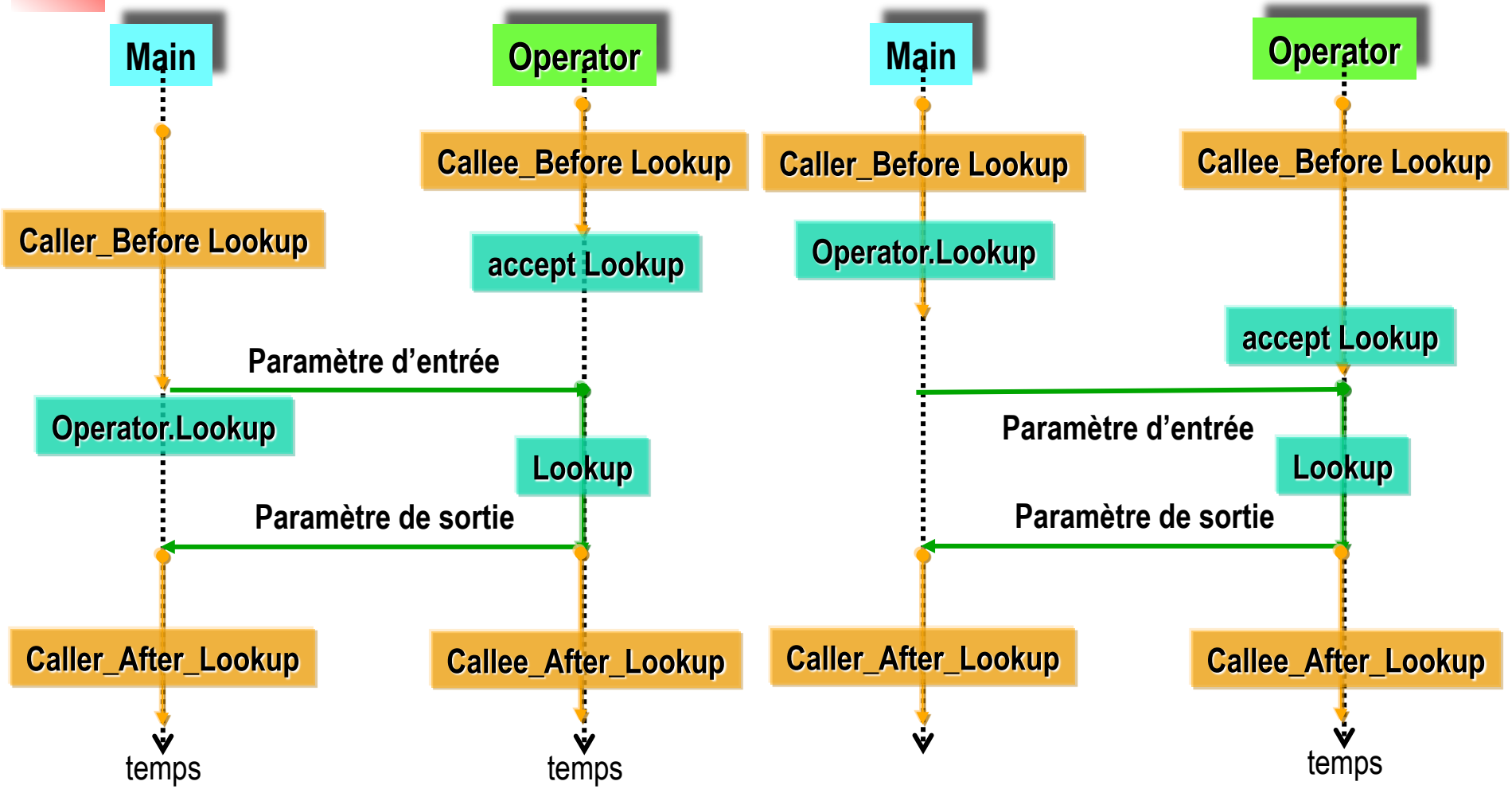
- Les tâches communiquent **directement** grâce à des entrées de tâche
- Une tâche publie ses entrées et leurs signatures dans sa spécification
- Une entrée correspond à une procédure appelable par une tâche appelante
- L'appelante invoque une entrée de l'appelée et se bloque en attendant que l'appelée accepte
- L'appelée accepte les appels sur ses entrées et se bloque en absence de demandes
- Lorsque appelante et appelée sont prêtes, l'appel se déroule dans le contexte de l'appelée

Ada
Spécification des entrées

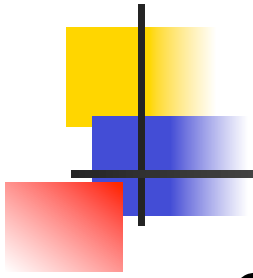


```
procedure Main is
   task Operator is
      entry Lookup(N, A : String; T : out Natural);
   end Operator;
   task body Operator is
   begin
      loop
         Callee_Before_Lookup;
         accept Lookup(N, A : String; T : out Natural) do
            T := Search_In_White_Pages (N, A);
         end Lookup;
         Callee_After_Lookup;
      end loop;
   end Operator;
   Phone : Natural;
begin
   Caller_Before_Lookup;
   Operator.Lookup (« pautet », « paris 13 », Phone);
   Caller_After_Lookup;
end Main;
```

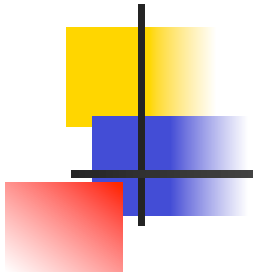
Ada
Deux scénarii - appelant et appelé



Ada
Compléments sur les entrées



- Chaque entrée de tâche dispose d'une file d'attente
- L'attribut *E'Count* retourne le nombre de tâche dans la file d'attente de l'entrée *E*
- Une exception levée dans un corps d'entrée est propagée à l'appelant **et** à l'appelé
- Une tâche peut dans un corps d'entrée accepter ou appeler une entrée
- Les familles d'entrées correspondent à des tableaux d'entrées
- Les entrées peuvent aussi être privées et leur visibilité se limitent aux tâches qui les déclarent



Ada
Synchronisation sélective

L'appelée accepte :

- De servir plusieurs entrées (alternatives)
- De servir des alternatives avec clauses de garde
- De servir pendant une période de temps
- De servir immédiatement si une demande est présente sinon d'exécuter un code alternatif
- De servir ou de terminer lorsque aucun appelant ne peut potentiellement la solliciter

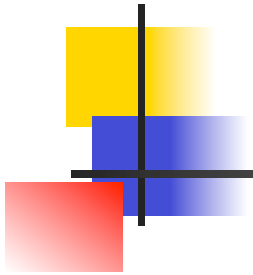
Ada
Alternatives simples

```
task Operatoris
   entry Lookup(N, A : String; T : out Natural);
   entry Remove (T : in Natural);
end Operator;
task body Operator is
begin
   loop
      select -- première alternative
         accept Lookup(N, A : String; T : out Natural) do
         -- cherche le numéro de téléphone
         end Lookup;
      or      -- seconde alternative
         accept Remove(T : in Natural) do
         -- retire le numéro de téléphone
         end Remove;
      end select;
   end loop;
end Operator;
```

Ada
Alternatives temporisées et gardes

```
task body Operator is
begin
   loop
      select -- 1ère alternative
         accept Lookup(N, A : String; T : out Natural) do
         -- cherche le numéro de téléphone dans la liste
         end Lookup;
      or -- 2ème alternative éligible si condition vraie
         when (Lookup'Count = 0) =>
         accept Remove(T : in Natural) do
         -- retire le numéro de téléphone dans la liste
         end Remove;
      or -- 3ème alternative élue si select dure + de 60s
         delay 60.0;
         -- trier la liste de numéros
      end select;
   end loop;
end Operator;
```

Ada
Alternatives immédiates



```
task body Operator is
begin
   select
      accept Lookup …;
   -- pas d'appel à Lookup
   else
      null; -- ne rien faire
   end select;
   select
      accept Lookup …;
   -- si timer 0.0 expire
   or
      delay 0.0;
   end select;
end Operator;
```

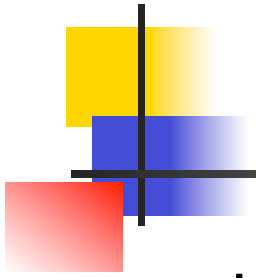
```
task body Operator is
begin
   select
      accept Lookup …;
   -- pas d'appel à Lookup
   -- ou Lookup'Count = 0
   else
      select
         accept Lookup …;
      or
         accept Remove …;
      end select;
   end select;
end Operator;
```

Ada
Compléments sur les rendez-vous

- Les clauses de garde sont évaluées une seule fois au début de l'instruction **select**
- Une alternative est éligible si sa garde est vraie
- L'exception Program_Error est levée en l'absence d'alternatives éligibles
- La tâche attend qu'une alternative soit élue
- Le choix entre plusieurs alternatives élues se fait selon la politique de gestion de file d'attente
- Une alternative **terminate** permet à la tâche de s'achever si aucune tâche ne peut l'appeler

Ada
Synchronisation par objet protégé

- Les tâches peuvent communiquer **indirectement** grâce à des objets *protégés* accessibles uniquement grâce à des fonctions, des procédures et des entrées
- Ces méthodes dotent les objets de mécanismes de synchronisation comme l'exclusion mutuelle ou les files d'attente au même titre que
  - les *mutexes* ou les *variables conditionnelles* de POSIX
  - les méthodes *synchronized* ou *wait/notify* de Java
  - les *fonctions et procédures* ou les *entrées* d'Ada



Ada
Fonction, procédure et entrée

- Une procédure accède à l'objet en lecture et écriture.
- Une procédure accède en exclusion mutuelle (un seul appel à la fois) en verrouillant le mutex lié à l'objet.
- Une fonction accède à l'objet uniquement en lecture.
- Plusieurs accès simultanés en lecture peuvent avoir lieu et seul le premier appel verrouille le mutex.
- Une entrée accède à l'objet en lecture et écriture dès lors que sa clause de garde associée devient vraie.
- Une entrée accède en exclusion mutuelle en verrouillant le mutex lié à l'objet et se suspend sur la clause de garde à l'aide de la variable conditionnelle liée à la clause.

Ada
Définition d'objets protégés (1/2)

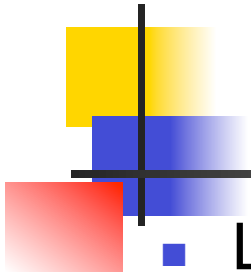
```
protected Semaphore is
   entry P;
   procedure V;
private
   N : Natural := 0;
end Semaphore;
protected body Semaphore is
   entry P when N > 0 is
   begin
      N := N - 1;
   end P;
   procedure V is
   begin
      N := N + 1;
   end V;
end Semaphore;
```

```
protected SharedBuffer is
   function Get return Item;
   procedure Put(V : Item);
private
   I : Item := InitialValue;
end SharedBuffer;
protected body SharedBuffer is
   function Get return Item is
   begin
      return I;
   end Get;
   procedure Put(V : Item) is
   begin
      I := V;
   end Put;
end SharedBuffer ;
```

Ada
Définition d'objets protégés (2/2)

```
type UnboundedBuffer is
   array (0 .. Length - 1)
      of Item;

protected BoundedBuffer is
   entry Get (I : out Item);
   entry Put(I : in Item);
private
   First  : Natural := 0;
   Last   : Natural := Length;
   Size   : Natural := 0;
   Buffer : UnboundedBuffer;
end BoundedBuffer;
```

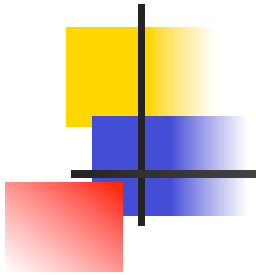
```
protected body BoundedBuffer is
   entry Get (I : out Item)
      when Size /= 0 is
   begin
      I      := Buffer (First);
      First :=
        (First + 1) mod Length;
      Size := Size - 1;
   end Get;
   entry Put(I : in Item)
      when Size /= Length is
   begin
      Last :=
        (Last + 1) mod Length;
      Buffer (Last) := I;
      Size := Size + 1;
   end Put;
end BoundedBuffer;
```



Ada
Complément sur les objets protégés

- Les objets protégés offrent une communication asynchrone entre tâches fondée sur les données
- Les clauses de garde sont ré-évaluées lors de l'exécution de procédures ou d'entrées qui peuvent modifier l'objet
- Dès lors, il est fortement déconseillé de faire dépendre les clauses de garde de variables globales
- Le code des méthodes d'accès ne doit pas faire appel à des opérations bloquantes (code court)
- En cas de levée d'exception, la restitution du mutex est effectuée de manière transparente
- La clause de garde ne peut dépendre des paramètres passés lors de l'appel à une entrée en raison d'un coût prohibitif de mise en oeuvre

Ada
Appel immédiat ou temporisé

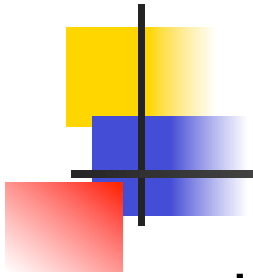


- Une tâche peut effectuer un appel immédiat ou temporisé à une entrée de tâche ou d'objet protégé
- Un appel immédiat permet d'abandonner une demande si cela entraînerait une attente
- Un appel temporisée permet d'abandonner une demande après un délai
- On ne peut appeler qu'une entrée à la fois

```
select
   Operator.Remove (0145817322);
else
   null; -- ne rien faire
end select;
```

```
select
   BoundedBuffer.Get (Item);
or
   delay until Clock + 60.0;
   Put_Line (« timeout »);
end select;
```

Ada
Limites des objets protégés

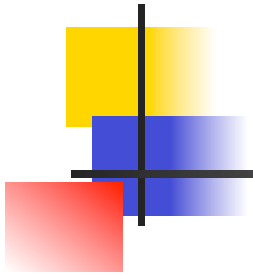


- La clause de garde ne peut dépendre des paramètres passés lors de l'appel à une entrée en raison d'un coût prohibitif d'implémentation
- Dans le cas contraire, il faudrait évaluer la clause pour chacune des tâches appelantes et non pas une seule fois
- Une méthode pour contourner cette limitation consiste à fournir une première entrée puis en fonction des paramètres et l'objet soit de sortir de cette entrée soit de re-diriger l'appel vers une autre entrée
- L'instruction **requeue** permet de faire passer une tâche d'une file d'attente d'entrée à une autre file d'attente d'entrée de même signature

Ada
Acquisition de plusieurs ressources (1/2)

- On veut extraire exactement N ressources à stocker dans un tableau *I*
  (N vaut *I'Length* c-à-d la longueur du tableau *I*)
- Si N ressources ne sont pas disponibles, la tâche passe en attente sur la seconde entrée *Wait* en attendant que d'autres ressources soient libérées.

```
type Items is array (Natural) of Item;
-- type de tableau de taille variable
-- que l'on obtient par I'Length
protected BoundedBuffer is
   entry Get (I : out Items);
   -- extrait I'Length ressources et
   -- bloque si indisponibles
   procedure Set (I : Items);
   -- fournit I'Length ressources
private
   entry Wait (I : out Items);
   Updating : Boolean := False;
   First    : Natural := 0;
   Last     : Natural := Length;
   Size     : Natural := 0;
   Buffer   : Items (0 .. Length - 1);
end Shared_Items;
```

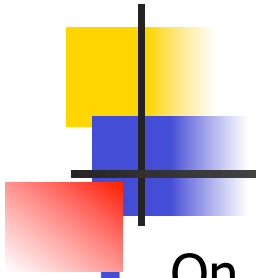


Ada
Acquisition de plusieurs ressources (2/2)

```
entry Get (I : out Items)
  when not Updating is
begin
   if I'Length <= Size then
      Size := Size - I'Length;
      -- retire les ressources
   else
      requeue Wait;
   end if;
end Get;
```

```
procedure Set (I : Items) is
begin
   Size := Size + I'Length;
   -- ajoute les ressources
   if Wait'Count > 0 then
      Updating := True;
   end if;
end Set;
```

```
entry Wait (I : out Items)
  when Updating is
begin
   if Wait'Count = 0 then
      Updating := False;
   end if;
   requeue Get;
end Wait;
```



Ada
Types de tâches - Objets protégés

- On peut définir des types d'objets tâches ou d'objets protégés
- On peut alors définir une instance d'un type d'objets tâches (resp d'objets protégés) pour créer une tâche (resp un objet protégé)
- La spécification d'un type d'objet tâche (resp d'objet protégé) débute par **task type** (resp **protected type**) et comporte
  - Un nom
  - Un discriminant éventuel (constructeur pour paramètrer l'objet)
  - Une partie visible
  - Une partie privée

- La définition d'un objet tâche (resp objet protégé) correspond à
  - La définition implicite d'un type anonyme d'objet tâche (resp d'objet protégé)
  - La définition d'une instance de ce type anonyme d'objet tâche (resp d'objet protégé)

Ada
Définition de types tâches

```
task type
   Server(D : Discriminant) is
   -- discriminant optionnel
entry Service (R : Request);
   -- entrées optionnelles
end Server;
task body Server is
   C : Context;
begin
   Initialize (C, D);
   accept Service (R : Request)
   do
      Modify (C, R);
   end Service;
   Finalize (C);
end Server;
S : Server (X); -- objet tâche
```

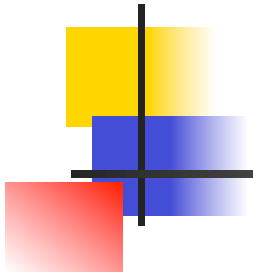
```
task S is
   entry Service (R : Request);
   -- entrées optionnelles
end S;
task body S is
   C : Context;
begin
   Initialize (C, X);
   accept Service (R : Request)
   do
      Modify (C, R);
   end Service;
   Finalize (C);
end S;
-- S : unique objet tâche d'un
-- type tâche anonyme
```

Ada
Définition d'objets protégés

```
protected type
   Shared_Item (D : Item)
is
   function Get return Item;
   procedure Set (V : Item);
private
   I : Item := D;
end Shared_Item;

protected body Shared_Item is
   function Get return Item is
   begin return I; end Get;
   procedure Set (V : Item) is
   begin I := V; end Set;
end Shared_Item;

SI : Shared_Item (X);
```

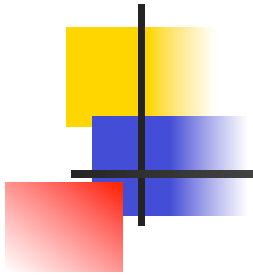
```
protected SI is
   function Get return Item;
   procedure Set (V : Item);
private
   I : Item := X;
end SI;

protected body SI is
   function Get return Item is
   begin return I; end Get;
   procedure Set (V : Item) is
   begin I := V; end Set;
end SI;
```

Ada
Tâches statique et dynamique



```
procedure Main is
   task type Phone_Operator is
      entry Lookup(N, A : String; T : out Natural);
   end Phone_Operator;
   My_Operator   : Phone_Operator;
   -- alloue et active une tâche
   Six_Operators : array (1 .. 6) of Phone_Operator;
   -- alloue et active six tâches en parallèle
   type Operator_Ptr is access Phone_Operator;
   An_Operator : Operator_Ptr;
   -- déclare une référence vers une tâche
begin
   An_Operator := new Phone_Operator;
   -- alloue et active la tâche
end Main;
```

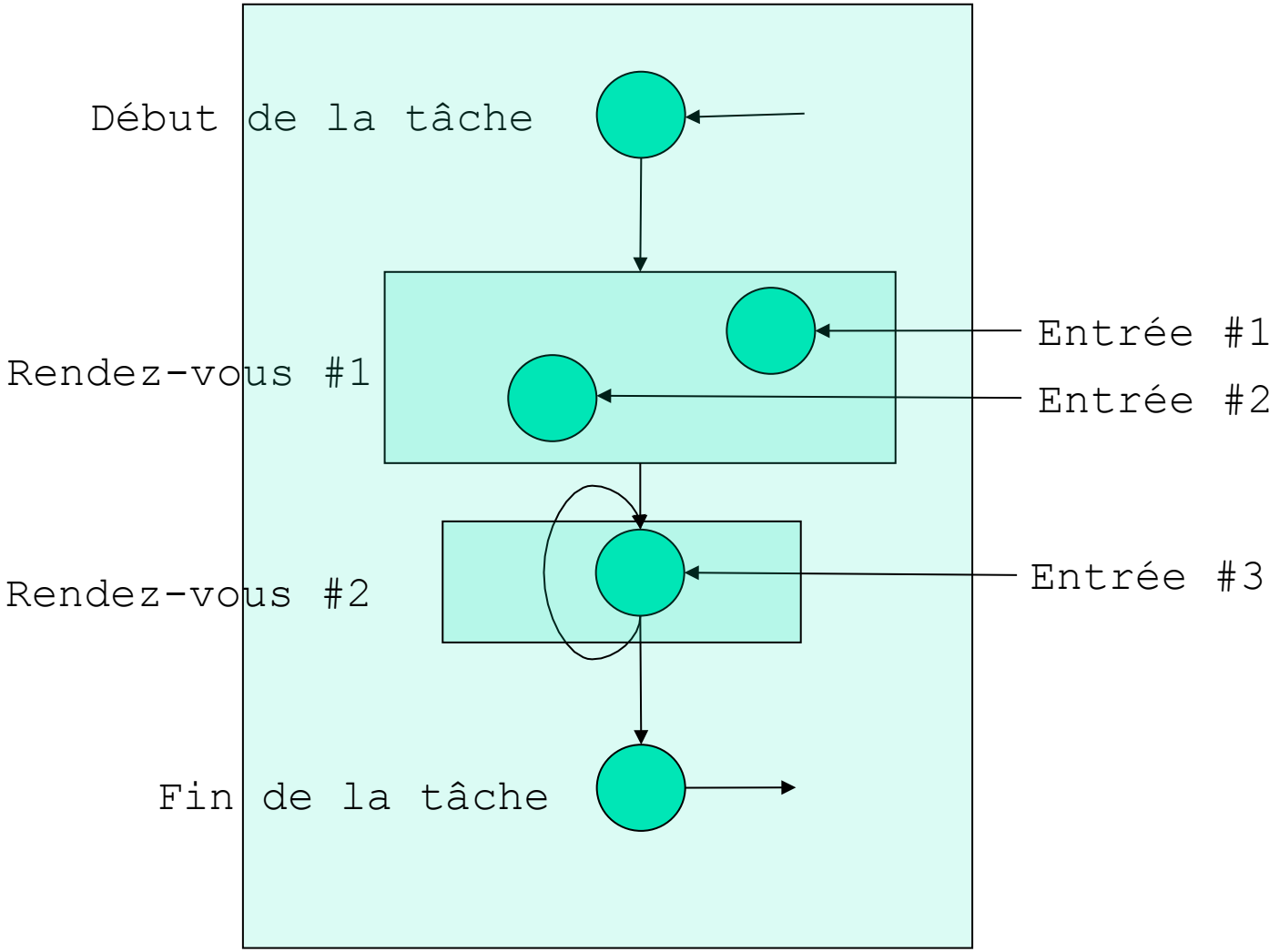
Ada
Exemple du crible d'Erathostène

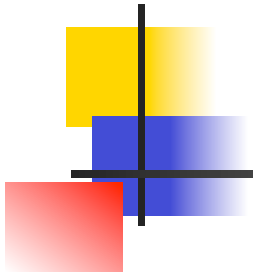


```
procedure Main is
   task type Cell (P : Integer) is
      entry Test (I : Integer);
   end Cell;
   type Cell_Ptr is access Cell;

   function New_Cell (I : Integer)
      return Cell_Ptr is
   begin
      return new Cell (I);
   end New_Cell;

   Two : Cell_Ptr := New_Cell (2);
begin
   for I in 3 .. 1000 loop
      Two.Test (I);
   end loop;
end Main;
```

```
task body Cell is
   N : Cell_Ptr; --  next cell
begin
   loop
      select
         accept Test (I : Integer) do
            if I mod P = 0 then
               return;
            end if;
            if N = null then
               N := New_Cell (I);
            else
               N.Test (I);
            end if;
         end Test;
      or
         terminate;
      end select;
   end loop;
end Cell;
```

Ada
Tâches et automates à états



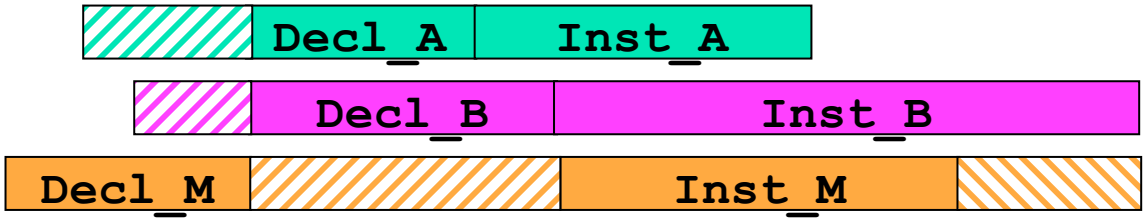


Ada
États d'un objet tâche

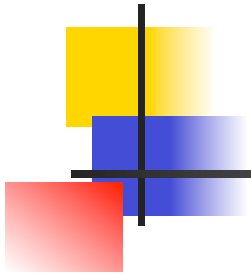
- Elaboration des objets de la partie déclarative puis activation de la tâche
- Exécution ou déroulement de la séquence d'instruction du corps de la tâche
- Terminaison de la tâche puis destructions des objets de la partie déclarative

Ada
Règles d'activation de tâches

```
procedure M is
    task A;
    task body A is <<Decl_A>> begin <<Inst_A>> end A;
    task B;
    task body B is <<Decl_B>> begin <<Inst_B>> end B;
    -- A et B exécutent leur partie déclarative (activation)
    -- dès que la partie déclarative de M est achevée
    -- sinon elles pourraient avoir visibilité
    -- sur des déclarations corrompues.
    X : Matrix := Read (« matrix.data »);
begin
    -- M exécute sa partie instruction dès que
    -- les parties déclaratives de A et B sont achevées
end M;
```



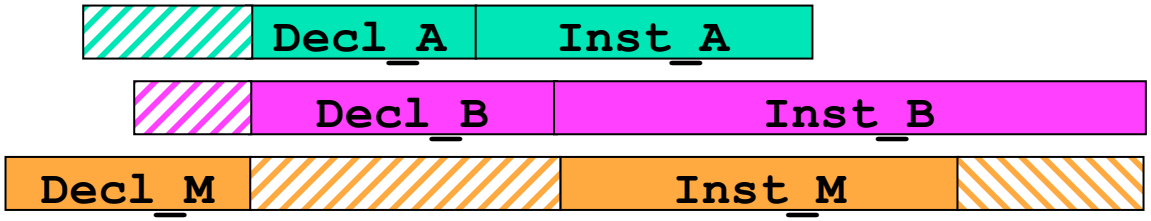
Ada
Règles d'activation de type tâche



```
procedure M is
   task type T;
   type P is access T;
   A : P;
   task body T is <<Decl_T>> begin <<Inst_T>> end T;
begin
   A := new T;
   -- l'opérateur new est bloqué tant que A et ses filles
   -- n'ont pas terminé leur activation.
   -- si l'activation de A échoue, elle devient terminée.
   -- L'opérateur new lève l'exception Tasking_Error
end D;
```

Ada
Règles de terminaison de tâches

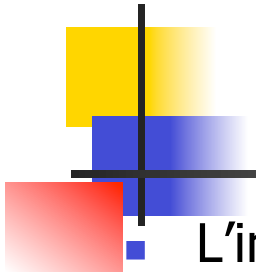
```
procedure M is
   task A;
   task body A is <<Decl_A>> begin <<Inst_A>> end A;
   task B;
   task body B is <<Decl_B>> begin <<Inst_B>> end B;
   X : Matrix := Read (« matrix.data »);
begin
   -- La tâche supportant l'exécution de M s'achève
   -- lorsque la fin des instructions est atteinte.
   -- Elle se termine (destruction de la pile) dès
   -- lors qu'elle s'achève et que toutes les tâches
   -- qui dépendent d'elle sont terminées (A et B)
end M;
```



Ada
Règles de terminaison de types tâches

```
procedure M is
   task type T; type T_Ptr is access T;
   A : T_Ptr;
   task body T is <<Decl_T>> begin <<Inst_T>> end T;
   procedure P is
      B : T;
      C : P := new T;
   begin
      A := C;
   end P;
   -- P se termine si B se termine (peu importe A et C)
   -- car B se trouve dans la pile de P.
begin
   P;
end M;
-- M se termine lorsque les tâches T se terminent car
-- T a visibilité sur la pile de M.
```

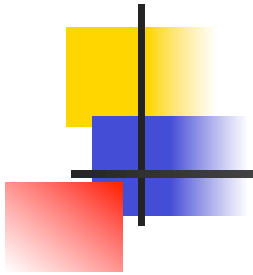
Ada
Avortement d'exécution



- L'instruction **abort** interrompt l'exécution d'une tâche
- L'arrêt d'une tâche survient dès que possible
  - certaines opérations peuvent le différer
    - Exécuter un destructeur
    - Sortir d'une section critique
  - il doit survenir avant toute nouvelle synchronisation
- Les tâches qui en dépendent sont également avortées
- **abort** constitue une opération de dernier recours

```
task type TT;
type TA is access TT;
task T0;
T1 : TT;
T2 : TA := new TT;
abort T0, T1, T2.all;
```

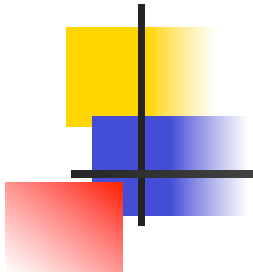
Ada
Avortement de synchronisation



- Exécution d'un code en restant réactif à la réception d'événement
- Le code pouvant être avorté démarre
- La tâche prend sa place dans une file d'entrée
- Si le code se termine, la tâche est retirée de la file d'attente
- Si l'entrée est acceptée, le code est avorté
- L'exécution de la tâche se poursuit après la construction

```
select
   delay 1.0;
then abort
   Compute;
   -- Calcul interrompu
   -- si exécution > 1s
end select;
```

```
select
   S.P;
then abort
   Compute;
   -- Calcul interrompu dès que
   -- le sémaphore P est prêt
end select;
```

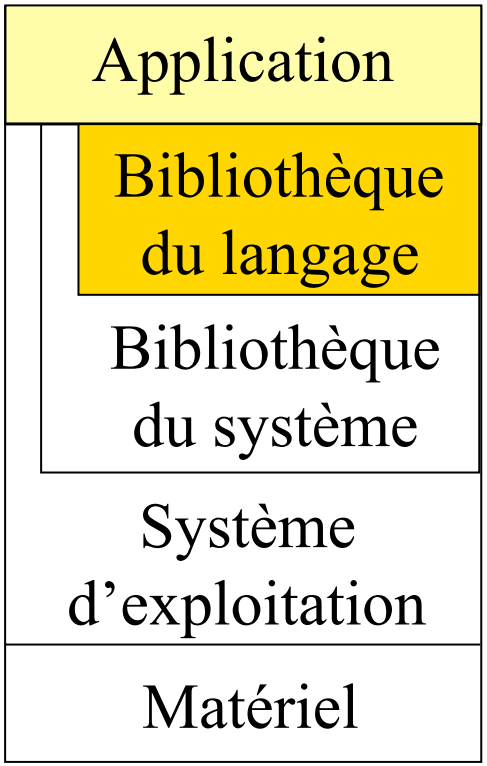


Compilation
Du langage à l'exécutif

- Architecture globale
- Traduction des tâches Ada vers POSIX
- Traduction des objets protégés Ada vers POSIX

Ada
Ada -> Biblio Ada -> Biblio POSIX

- Chaque construction Ada se trouve traduite sous forme d'une séquence d'appels vers la bibliothèque du langage
- La bibliothèque du langage fait le lien avec celle du système (comme une machine virtuelle vers une bibliothèque POSIX)
- Tâche
  - -> N x Appels à une bibliothèque du langage
  - -> Thread (POSIX)
- Objet Protégé
  - -> N x Appels à une bibliothèque du langage
  - -> 1 Mutex + M x Var Cond (POSIX)



Ada
Traduction des spécs de tâches

1. La tâche t se traduit sous forme d'un type tâche tTK et d'une variable t de ce type
2. tTK se traduit ensuite sous forme d'une structure tTKV contenant un thread, du programme principal tTKB et d'une procédure d'initialisation _init_proc
3. _init_proc appelle create_task de la bibliothèque du langage en lui indiquant le sous programme principal tTKB du thread et son nombre d'entrées donc de files d'attente (ici **2** pour e1 et e2).

```
task t is
   entry e1;
   entry e2 (x : in natural);
end t;
```

```
task type tTK is [...]
end tTK;
t : tTK;
```

```
type tTKV is record
    _task_id : task_id;
end record;
procedure tTKB
   (_task : access tTKV);
procedure _init_proc
   (_init : in out tTKV) is [...]
  create_task (tTKB'address, 2);
end _init_proc;
```

Ada
Traduction des impls de tâches

- Le code de t est extrait pour former le sous-programme tTKB
- Le transfert des paramètres de la pile de l'appelant vers celle de l'appelé s'effectue par indirection
- tTK__A6s pointe vers l'ensemble des paramètres de e2
- tTK__A4s pointe vers le paramètre x en mode in out
- accept_call met à disposition la 2 ème entrée de t et récupère les paramètres dans A4b

```
task body t is begin
   accept e2  (x : in out natural)
   do x := x + 1; end e2;
 end t;
```

```
type tTK__A4s is access natural;
type u__tTK__P5s is
record x : tTK__A4s; end record;
type tTK__A6s is access tTK__P5s;
```

```
procedure tTKB (_task : access tTKV) is [...]
   A4b : tTK_A6s;
begin
   accept_call (2, A4b);
   A4b.x.all := A4b.x.all + 1;
   complete_rendezvous;
exception  when others =>
   exceptional_complete_rendezvous ([...]);
end tTKB;
```

Ada
Traduction des spécs d'objets protégés

- Le PO s se traduit sous forme d'un type PO et d'une variable de ce type
- Ce type sTV contient un mutex _controller, une table d'entrées _object et une variable privée n
- sPT__E3s contient le code de l'entrée p et sPT__B4s celui de sa clause de garde
- sPT__vN contient le code de v et sPT__vP un appel à sPT_vN encadré par un mutex avec gestion des exceptions

```
protected s is
   entry p;
   procedure v;
private
   n : natural := 0;
end s;
```

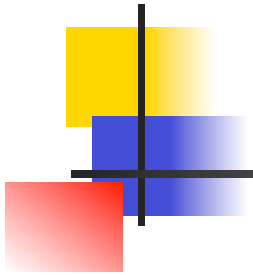
```
type sTV is limited record
   _controller : limited_record_controller;
   n : natural := 0;
   _object : protection_entries (1);
end record;
procedure sPT__E3s ([...]);
function sPT__B4s ([...]) return boolean;
procedure sPT__vN (_object : in out sTV);
procedure sPT__vP (_object : in out sTV);
sTA : protected_entry_body_array  (1 .. 1)
   := ((sPT__B4s'address, sPT__E3s'address));
```

Ada
Traduction des impls d'objets protégés

```
protected body s is
   entry p when n > 0 is
   begin n := n – 1; end p;
   procedure v is
   begin n := n + 1; end v;
end s;
```

- p ou v exécuté, on recalcule sPT__E3s pour reprendre éventuellement les tâches bloquées sur e2
- Un appel à s.p se traduit par un appel à sPT__vP

```
function sPT__B4s (_object : sTV) [...] is
   return _object.n  >  0;
end sPT__B4s;
procedure sPT__E3s (_object : in out sTV) is
   _object.n := _object.n - 1;
   complete_entry_body ([...]);
end sPT__E3s;
procedure sPT__vN (_object : in out sTV) is
   _object.n := _object.n + 1;
end sPT__vN;
procedure sPT__vP (_object : in out sTV) is
   lock_entries (_object);
   sPT__vN (_object);
   unlock_entries (_object);
exception when E8b: others =>
   unlock_entries (_object);
   reraise_exception (E8b);
end sPT__vP;
```



Ada
Conclusions

- La concurrence induit des besoins complexes en fonctionnalités que seul le langage peut offrir en assurant clarté et sémantique
- La concurrence comme l'approche structurée ou objet répond à des besoins de génie logiciel et doit introduire un niveau suffisant d'abstraction
- La concurrence devient une propriété indispensable aux nouvelles applications mais la communauté informatique reste mal préparée à ses difficultés et à ses exigences