Concurrencia en Ada: objetos protegidos

Lecturas:
Burns & Wellings, secciones 7.1, 7.2, 7.3, 7.5
Cohen, sección 17.4
Apuntes de la asignatura

Manuel Carro

Universidad Politécnica de Madrid

30 de noviembre de 2007

Mecanismos de concurrencia en Ada

- Vimos semáforos (pero son **externos** a Ada)
- Dos mecanismos principales:
  - Objetos protegidos (una variación de monitores)
    - ⋆ Aparecen con Ada 95
    - ⋆ P.O.O. (esp. encapsulación de estado) + exclusión mutua + sincronización
  - *Rendez-Vous* (paso de mensajes "domesticado")
    - ⋆ El inicial de Ada
    - ⋆ Basado en proceso que recibe peticiones

Un objeto protegido

- Refleja bastante bien un recurso:
  - Define un tipo
  - Operaciones públicas
  - Implementación y estado privados
- Similar a un paquete (puede formar parte de ellos: `Semaphores.{ads,adb}`)
- Operaciones actúan sobre variable del tipo definido:

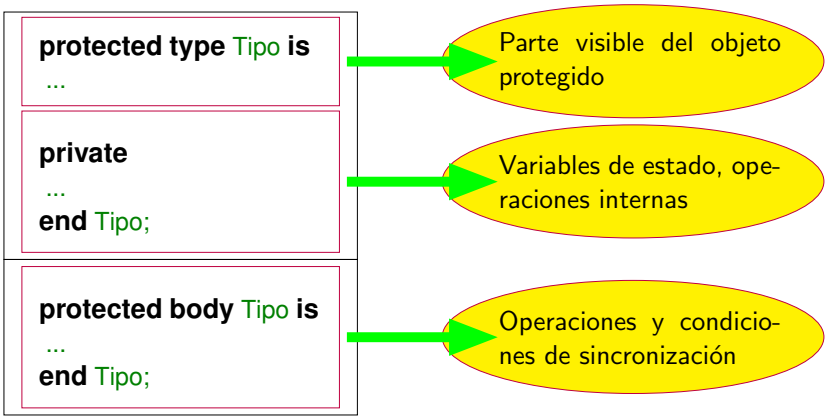
Notación procedimental:

```
Wait(Mi_Semaforo)
Signal(Mi_Semaforo)
```

Notación O.O.:

```
Mi_Semaforo.Wait
Mi_Semaforo.Signal
```

Esquema objeto protegido



Declaración visible

```
protected type Tipo is
   function F1(...);
   procedure P1(...);
   entry E1(...);
```

- Declaraciones similares a .ads
- Deben usar tipos **ya definidos**

- **procedure** y **entry**: exclusión mutua
- **function**: permite acceso concurrente
- **No usaremos** funciones
- Y **procedure** es una clase de **entry**

→ sólo usaremos **entries**

Estado (privado) del objeto

- No declaración tipos
- Variables de estado
  - No visibles desde el exterior

```
private
   Var_Estado: Tipo_Var;
   entry E_Priv(...);
end Tipo;
```

- **entry** privadas:
  - Aparecen en la implementación
  - No se pueden llamar desde afuera
  - Veremos su uso más adelante

Sección de implementación

```
protected body Tipo is
...
end Tipo;
```

- **Debe** contener código para los **function**, **procedure** y **entry** declaradas

- No declaraciones de tipos o variables locales
- **entry** además recoge condiciones sincronización

Un ejemplo simple

Declaracion de interfaz y estado

Código fuente en `shvar_protected.adb`

```
protected type Shared_Counter is
   -- Incrementa contador
   entry Increment;
   -- Devuelve valor contador
   entry Value(V: out Integer);
private
   Counter: Integer := 0;
end Shared_Counter;
```


Un ejemplo simple (Cont.)

Cuerpo de la implementación

```
protected body
  Shared_Counter is

  entry Increment
  when True is
  begin
   Counter:= Counter + 1;
  end Increment;

  entry Value
        (V: out Integer)
  when True is
  begin
   V:= Counter;
  end Value;

end Shared_Counter;
```

Todo ello implementa:

TIPO: $Tipo\_Contador = \mathbb{N}$

INVARIANTE: *cierto*

INICIAL(r): $r = 1$

CPRE: *cierto*
Increment(r)
POST: $r^{sal} = r^{ent} + 1$

CPRE: *cierto*
Value(r, v)
POST: $v^{sal} = r^{ent}$

Uso de un objeto protegido

```
Vueltas: Tipo_Contador;
...
   Vueltas.Incrementar;
```

- Valor del contador: estado interno
- Ej.: estado de semáforo, nº coches aparcamiento
- Exclusión mutua entre operaciones

Traducción inicial

Especificación		**Implementación**
Recurso	$\Longrightarrow$	Objeto protegido
Ops. CTADSOL	$\Longrightarrow$	Ops. públicas O.P.
Tipo recurso	$\Longrightarrow$	Tipos variables estado
Exclusión mutua	$\Longrightarrow$	Exclusión mutua
CPRE *cierto*	$\Longrightarrow$	**when** True
Postcondición	$\Longrightarrow$	Implementación

Sincronización condicional

```
entry Op(...)
   when Condicion is
begin
   ...
end Op;
```

- **when** *Condicion*: guarda de la operación

- *Condicion*: cualquier variable visible **excepto** argumentos llamada
- **Usar sólo variables estado**
- Llamada a *Op(...)* **suspende** si $\neg$*Condicion*

procedure Op(...) $\equiv$ **entry** Op(...) **when** True

Comportamiento en suspensión

```
entry Op(...)
   when Condicion is
begin
   ...
end Op;
```

- $\neg$*Condicion* → llamada suspende → abandona exclusión mutua
- Otra llamada puede *entrar*
- Estado cambia → puede cumplirse *Condicion*
- Rearrancar **alguna** llamada suspendida (Ada no define cuál)

- Orden de evaluación guardas no definido
- *Condición*$_i$ puede reevaluarse muchas veces
- Llamadas bloqueadas con guardas abiertas tienen **preferencia** sobre aceptación de nuevas llamadas
- Evaluación argumentos Op(...) espera aceptación
- Conserva cantidad *observable* de trabajo (no hay computación especulativa)

Un ejemplo: aparcamiento

```
protected type Tipo_Aparc is
   -- Espera sitio libre
   entry Entrar;
   -- Dec. número coches
   entry Salir;
private
   Vacios: Natural:= Max;
end Tipo_Aparc;
```

```
protected body Tipo_Aparc is

   entry Entrar
      when Vacios > 0 is
   begin
      Vacios:= Vacios - 1;
   end Entrar;

   entry Salir when True is
   begin
      Vacios:= Vacios + 1;
   end Salir;

end Tipo_Aparc;
```


Evolución aparcamiento

Tarea	Código	Vacíos
1:T_1	**entry** Entrar **when** Vacios > 0 **is**...	0
2:T_1	**entry** Entrar **when** Vacios > 0 **is**...	0
2:T_2	**entry** Salir **when** True **is**...	0
3:T_1	**entry** Entrar **when** Vacios > 0 **is**...	**1**
4:T_1	**entry** Entrar **when** Vacios > 0 **is**...	1
	entry Entrar **when** Vacios > 0 **is**...	0

Ejemplo: declaración semáforo

```
subtype Sem_Range_Type is Natural range 0..MaxValue;

protected type Sem_Type (Initial_Value: Sem_Range_Type) is

   entry Wait;     -- <await Val > 0 -> Val := Val - 1>
   entry Signal;   -- <Val < MaxVal ->
                   --       Val := Val + 1 else abort>

private -- Invariant: 0 <= Val <= MaxVal
   Value: Sem_Range_Type := Initial_Value;
end Sem_Type;
```


Ejemplo: una implementación de semáforos

```
protected body Sem_Type is

   entry Wait -- Esperar a que sea mayor que cero
      when Value > Sem_Range_Type'First is
   begin      -- Decreméntalo ahora
      Value := Sem_Range_Type'Pred(Value);
   end Wait;

   entry Signal
      when True is
   begin -- Incrementar incondicionalmente
      Value := Sem_Range_Type'Succ(Value);
   end Signal;

end Sem_Type;
```


Traducción de precondiciones

Especificación		**Implementación**	
Recurso	$\Longrightarrow$	Objeto protegido	
Ops. CTADSOL	$\Longrightarrow$	Ops. públicas O.P. (**entry**)	
Tipos recurso	$\Longrightarrow$	Tipo variables estado	
Exclusión mutua	$\Longrightarrow$	Exclusión Mutua	
CPRE $C(r)$	$\Longrightarrow$	**when** $C(r)$	**(⋆)**
Postcondición	$\Longrightarrow$	Implementación	

(⋆) Si no hay dependencia de variables de entrada

Dependencia de parametros de entrada

- Ej., buffer par/impar:
  - Consumidor de pares espera por (y consume) pares
  - Consumidor de impares espera por (y consume) impares
- Suspensión depende de un valor que no es parte del estado del recurso
- Ada no ofrece mecanismos de sincronización dependiente de los parámetros de entrada
- Pero ofrece:
  - **requeue**
  - Familia de **entries**
- En sentido general: almacenan información sobre los parámetros de entrada en el estado privado del objeto

Requeue (reencolado)

```
entry P(...)
    when C is
begin
    ...
    requeue P_1;
    ...
end P;

entry P_1(...)
    when C_1 is
begin
    ...
end P_1;
```

- Derivación a una **entry** privada
- **Diferente** a llamar op. pública desde interior objeto (¡causaría [auto]bloqueo!)
- Puede bloquear en P_1

→ Abandona exclusión mutua

- Puede rearrancar más tarde (otra operación puede hacer verdad C_1)

Requeue (Cont.)

```
entry P(...)
    when C is
begin
    ...
    requeue P_1;
    ...
end P;

entry P_1(...)
    when C_1 is
begin
    ...
end P_1;
```

- Al acabar P_1:
  - **No** retorna a P
  - Sale del objeto protegido
- **requeue** usa *cabeceras compatibles*
- ¿Falta de estructuración?
- Veremos reglas generales de uso

Buffer par-impar con *requeue*

Un ejemplo sencillo

```
entry Get
      (Req : in
Request_Type;
       What: out Item_Type)
when Item_Present is
begin
   if Req = Even then
      requeue Priv_Get_Even;
   else
      requeue Priv_Get_Odd;
   end if;
end Get;
```

```
entry Priv_Get_Even
      (Req : in
Request_Type;
       What: out Item_Type)
when Item_Present and
       Item mod 2 = 0 is
begin
   What:= Item;
   Item_Present:= False;
end Priv_Get_Even;
```


Esquema de redireccionado

Condición: $C \equiv C_1(E) \wedge C_2(E, P)$

E: estado, P: parámetros, $P \in \{p_1, p_2, \ldots p_n\}$

```
entry Op(P)
when C1 is — Opcional
   begin
      if P = p1 then
         requeue Op1;
      else if P =
p2 then
         requeue Op2;
      else ... then
         requeue Opn;
      end if;
end Get;
```

```
entry Opi(P)
when C1 and C2(E, pi) is
begin
      ...
end Opi;
```

C_2(E, p_i) puede escribirse sin parámetros de entrada — ya tenemos su valor

- ¿Multiplicación de **entries**?
- Bloqueo en dos fases: condición dependiente estado y condición completa (¿Sólo Item **mod** 2 = 0?)

Familia de **entries**

```
type Tipo_X is ...;

protected type Tipo_P is
   entry P (Tipo_X);
...
end Tipo_P;

protected body Tipo_P is
   entry P (for I in Tipo_X)
    when Estado > I is begin
    Q(I, ...);
...
end Tipo_P;
```

- *Tipo_X*: escalar
- Sincronización con estado e índice
- **Replica** condición para cada *I*
- Como si hubiese un fragmento de código diferente para cada $I \in$ *Tipo_P*

Conceptualmente:

P1 **when** Estado > 1 **is begin** Q(1,...); ... **end** P1;

P2 **when** Estado > 2 **is begin** Q(2,...); ... **end** P2;

Ejemplo familia: espera por un dato

Cola de la pescadería

ACCIÓN Deja: *Tipo_Espera[es]* $\times$ $\mathbb{N}$*[e]*
ACCIÓN Espera: *Tipo_Espera[es]* $\times$ $\mathbb{N}$*[e]*

TIPO: *Tipo_Espera* $=$ *(Dato:* $\mathbb{N}\times$ *Hay_Dato:* $\mathbb{B}$*)*

INICIAL(c**):** $c.Hay\_Dato = falso$

CPRE: *cierto*
Deja(e, d)
POST: $e^{sal}.Hay\_Dato \wedge e^{sal}.Dato = d$

CPRE: $e.Hay\_Dato \wedge e.Dato = d$
Espera(e, d)
POST: $\neg e^{sal}.Hay\_Dato$

Posible implementación

Declaraciones públicas

```
protected type Tipo_Espera is
   entry Deja(D: Tipo_Dato);
   entry Espera(Tipo_Dato);
private
   Dato: Tipo_Dato;
   Hay_Dato: Boolean:= False;
end Tipo_Espera;
```

(Si Tipo_Dato reducido puede usarse **requeue** sobre operaciones de espera con **if**–**then**–**elsif**–...)

Posible implementación (Cont.)

Cuerpo de la implementación

```
protected body Tipo_Espera is

   entry Deja(D: Tipo_Dato) when True is
     begin
       Dato:= D;
       Hay_Dato:= True;
     end Deja;

   entry Espera(for I in Tipo_Dato)
     when Hay_Dato and I = Dato is
     begin
       Hay_Dato:= False;
     end Espera;

end Tipo_Espera;
```

Parámetros

- Las familias de *entries* admiten parámetros

```
type Tipo_X is ...;
protected type Tipo_P is
   entry P (Tipo_X)(D: in Natural);
...
protected body Tipo_P is
   entry P (for I in Tipo_X)(D: in Natural)
     when Estado > I is
   begin
     Q(I, D, ...);
      ...
   end P;
```

(D no puede aparecer en la guarda)

Buffer par/impar con familias

Parte pública

```
type Req_Type is (Even, Odd);

protected type Buffer is
  entry Get(Req_Type)
          (What: out It_T);
  entry Put(What: in It_T);
private
  Item: It_T;
  Full: Boolean:= False;
end Buffer;
```

Parte privada

```
entry Get(for I in Req_Type)
      (What : out It_T)
   when Full and ((I = Even)
      = (Item mod 2 = 0)) is
begin
   What:= Item;
   Full:= False;
end Get;

entry Put
   (What : in It_T)
  when not Full is
begin
   Item:= What;
   Full:= True;
end Put;
```


Limitaciones de las familias de *entries*

- Necesidad de **linealizar** argumentos llamada:

Recurso	Cliente
entry P(X, Y) **when** Estado = X ∗ Y **is**...	P(X, Y)
⇓	⇓
entry P(**for** I **in** ...) **when** Estado = I **is**...	P(X * Y)

- Necesidad de adoptar sintaxis especial de llamada:

```
Buffer.Get(Even)(Out_Item);
```

- Posiblemente problema menor, pero ¿es tarea del cliente?
- ¡No debería depender de la implementación!
- ¿Puede linealizarse siempre?

Limitaciones de las familias de *entries* (Cont.)

- Evaluación de condiciones tras cambio estado
- Reevaluación de llamadas suspendidas

```
entry P (for I in Tipo_X)
  when F(Estado, I) ...
```

- Tiempo aumenta con tamaño rango
  (`fament_efi.adb` — comparar tiempos de ejecución)
- ¿Orden de rearranque?
  (siempre analizar si varias guardas abiertas)

Usar la técnica más adecuada

- Lo veremos en las siguientes sesiones
- Base: guardas, familias, reencolado
- Resolver:
  - Dependencia datos entrada
  - Selección entre distintas *entries*
  - Prioridad entre llamadas misma *entry*

  } Cuestiones relativas a la vivacidad
- Frecuentemente varias soluciones
- Considerar: corrección, legibilidad, generalidad, flexibilidad