

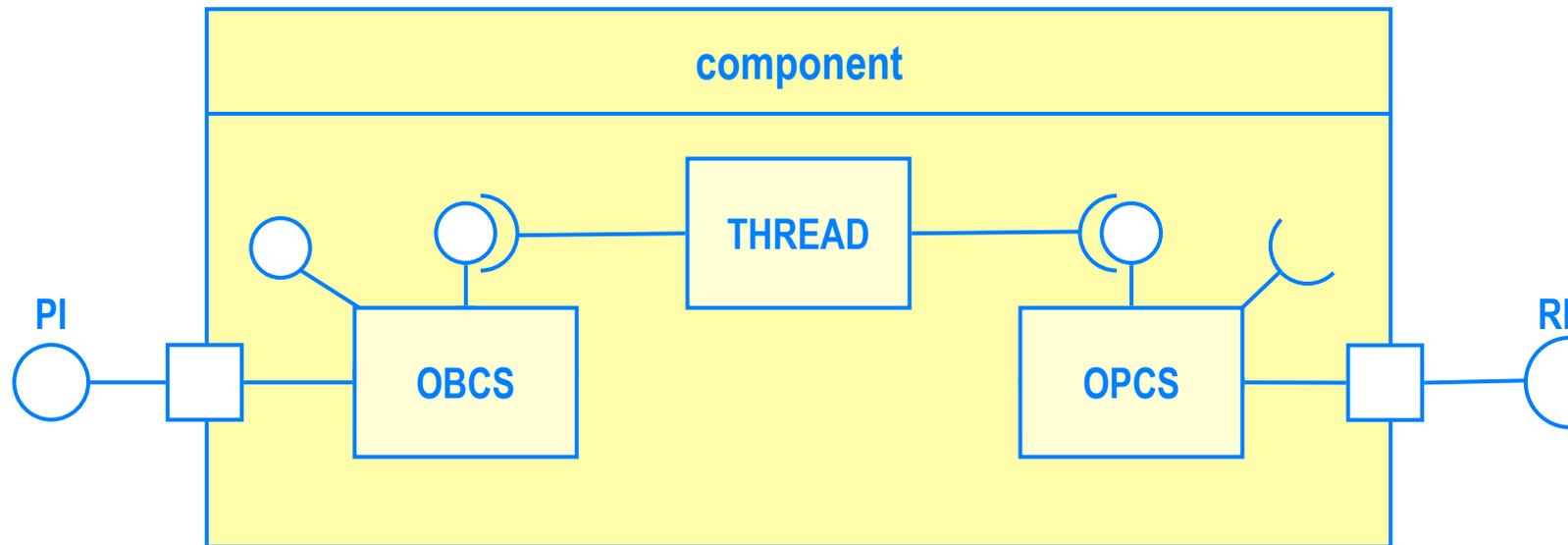
Esquemas de tareas de tiempo real

Juan Antonio de la Puente
DIT/UPM

Objetivos

- ◆ Reconocer un conjunto de esquemas básicos para construir sistemas de tiempo real
 - Tareas periódicas
 - Tareas esporádicas
 - Recursos compartidos
 - Sincronización
- ◆ Veremos esquemas basados en UML
 - También algunos esquemas más generales

Realización de un componente de tiempo real

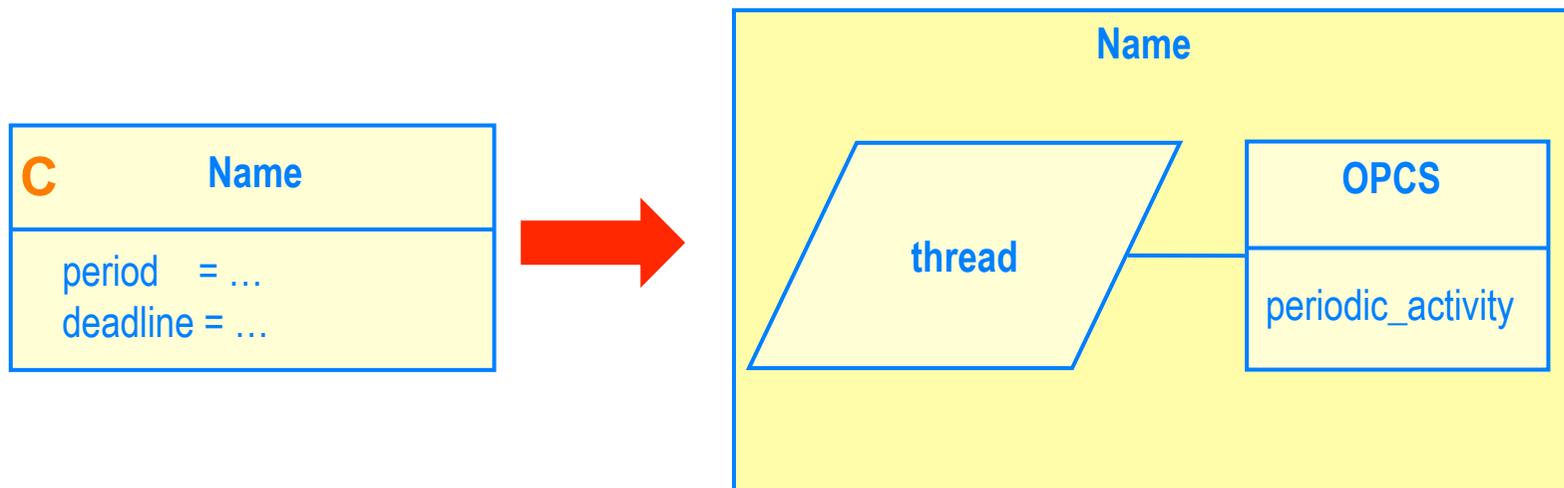


© Juan Antonio de la Puente 2000-2007

- ◆ OBCS : *object control structure* → objeto protegido
- ◆ THREAD → tarea
- ◆ OPCS : *operation control structure* → objeto pasivo

Un componente puede tener sólo algunos de estos elementos

Componente cíclico



Componente cíclico: realización

```
with Ada.Real_Time; use Ada.Real_Time;
with System;       use System;
package Name is
  Start_Time       : Time       := ... ;
  Period           : Time_Span := ... ;
  Thread_Priority  : Priority    := ... ;
  pragma Elaborate_Body (Name);
end Name;
```

```
with Ada.Real_Time; use Ada.Real_Time;
-- other context clauses
package body Name is

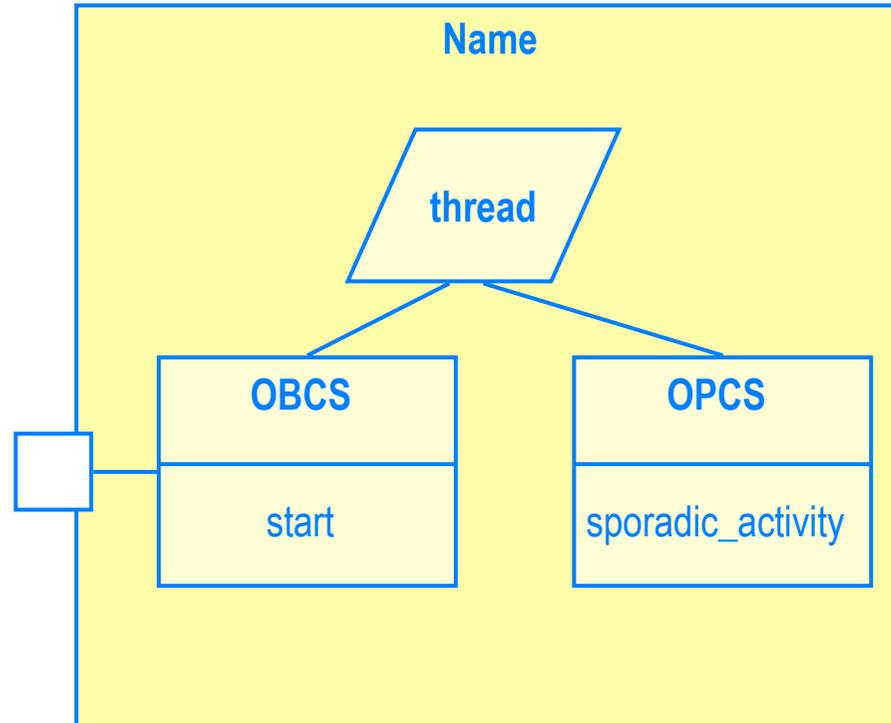
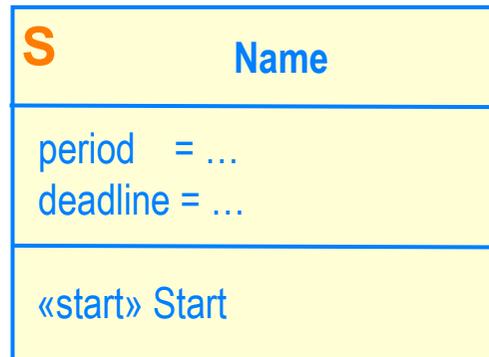
  task Thread is
    pragma Priority (Thread_Priority);
  end Thread;

  task body Thread is
    Next_Time : Time := Start_Time ;
  begin
    -- thread initialization
    loop
      delay until Next_Time;
      OPCS.Periodic_Activity;
      Next_Time := Next_Time + Period;
    end loop;
  end Thread;

begin
  -- package initialization
end Name;
```

- ◆ No hay operaciones visibles
 - Elaborate_Body indica que el paquete tiene cuerpo
- ◆ Atributos temporales
 - Start_Time
 - Period
 - Thread_Priority
- ◆ No se detecta si se excede el plazo de ejecución o el tiempo de cómputo

Componente esporádico



Componente esporádico: realización

```
with System; use System;
package Name is

    Thread_Priority : Priority := ... ;

    procedure Start;

end Name;
```

```
with Ada.Synchronous_Task_Control;
use  Ada.Synchronous_Task_Control;
-- other context clauses
package body Name is

    task Thread is
        pragma Priority (Thread_Priority);
    end Thread;

    Control : Suspension_Object;

    task body Thread is
    begin
        -- thread initialization
        loop
            Suspend_Until_True(Control);
            OPCS.Sporadic_Activity;
        end loop;
    end Thread;

    procedure Start is
    begin
        Set_True (Control);
    end Start;

begin
    -- package initialization
end Name;
```

- ◆ La única operación visible es la operación de arranque (*Start*)
- ◆ El OBCS se realiza con un objeto de suspensión
- ◆ No se vigila el cumplimiento de la separación mínima entre sucesos
- ◆ No se detecta si se excede el plazo de ejecución o el tiempo de cómputo

Componente esporádico: Sincronización con objeto protegido (1)

```
with System; use System;
package Name is

  Thread_Priority : Priority := ... ;
  Control_Priority: Priority := ... ;

  procedure Start;
```

```
private

  protected OBCS is
    pragma Priority (Control_Priority);
    procedure Start;
    entry      Wait;
  private
    Started : Boolean := False;
  end OBCS;

  procedure Start
    renames OBCS.Start;
end Name;
```

- ◆ El objeto de sincronización se sustituye por un objeto protegido
- ◆ Funcionalmente es equivalente, aunque menos eficiente
- ◆ Es fácil de extender para tener en cuenta la separación mínima entre sucesos
- ◆ Permite poner parámetros en las operaciones

Componente esporádico: Sincronización con objeto protegido (2)

```
with System; use System;
package body Name is
  task Thread is
    pragma Priority (Thread_Priority);
  end Thread;

  task body Thread is
  begin
    -- thread initialization
    loop
      OBCS.Wait;
      OPVS.Sporadic_Activity;
    end loop;
  end Thread;
```

```
protected body OBCS is

  procedure Start is
  begin
    Started := True;
  end Start;

  entry Wait when Started is
  begin
    Started := False;
  end Wait;

end OBCS;

begin
  -- package initialization
end Name;
```

Componente esporádico: Separación mínima (1)

```
with Ada.Real_Time; use Ada.Real_Time;
with System; use System;
package Name is

    Separation      : Time_Span := ... ;
    Thread_Priority : Priority   := ... ;
    Control_Priority: Priority   := ... ;

    procedure Start;
```

```
private

    protected OBCS is
        pragma Priority (Control_Priority);
        procedure Start;
        entry Wait(Start_Time : out Time);
    private
        Started      : Boolean := False;
        Event_Time   : Time;
    end OBCS;

    procedure Start
        renames OBCS.Start;
end Name;
```

- ◆ El objeto **OBCS** registra el tiempo en que se hace **Start** (no es necesariamente el mismo en que se ejecuta la acción esporádica)

Componente esporádico: Separación mínima (2)

```
with Ada.Real_Time; use Ada.Real_Time;
with System; use System;
package body Name is
  task Thread is
    pragma Priority (Thread_Priority);
  end Thread;

  task body Thread is
    Start_Time : Time;
  begin
    -- thread initialization
    loop
      OBCS.Wait (Start_Time);
      OPCS.Sporadic_Activity;
      delay until
        Start_Time + Separation;
    end loop;
  end Thread;
```

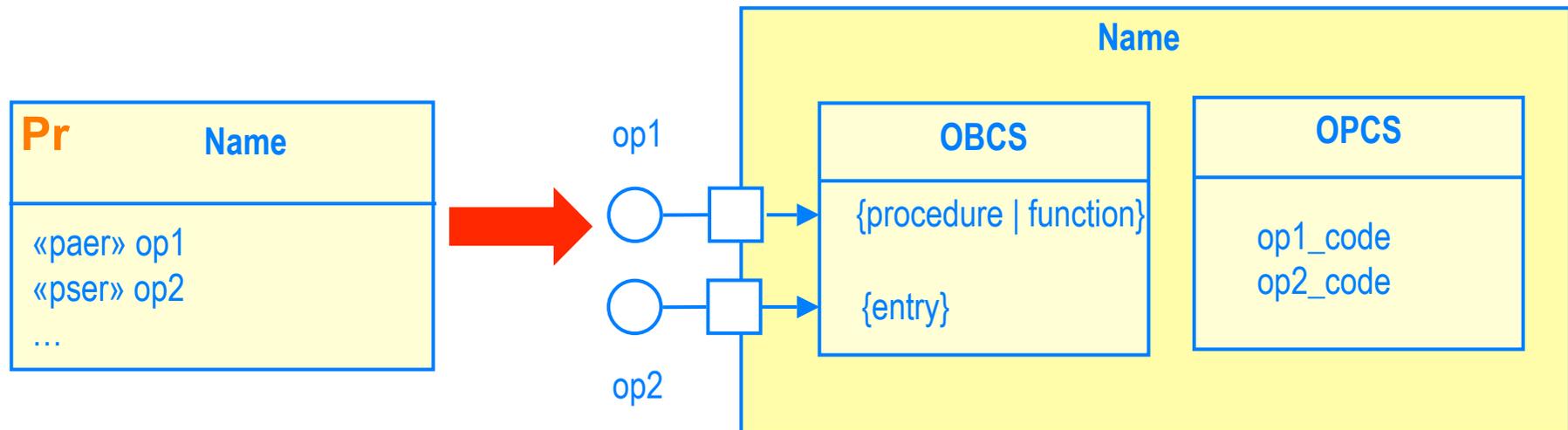
```
protected body OBCS is
  procedure Start is
  begin
    Started := True;
    Event_Time := Ada.Real_Time.Clock;
  end Start;

  entry Wait
    (Start_Time : out Time)
  when Started is
  begin
    Started := False;
    Start_Time := Event_Time;
  end Wait;
end OBCS;

begin
  -- package initialization
end Name;
```

- ◆ Inconveniente: dos cambios de contexto
 - si se está seguro de la separación mínima es mejor el esquema anterior

Componente protegido



Componente protegido: realización (1)

```
with System; use System;
package Name is
  Ceiling_Priority : Priority
  := ... ;

  procedure op1 (...);
  procedure op2 (...);
  ...
```

```
private

  protected OBCS is
    pragma Priority(Ceiling_Priority);

    procedure op1 (...);
    entry op2 (...);
  private
    ...
  end OBCS;

  procedure op1 renames OBCS.op1;
  procedure op2 renames OBCS.op2;

end Name;
```

- ◆ Las operaciones visibles son las especificadas en UML
 - ◆ pser : síncronas (entradas)
 - ◆ paser : asíncronas (procedimientos y funciones)
- ◆ Atributos temporales
 - Ceiling_Priority

Componente protegido: realización (2)

- ◆ Las operaciones síncronas se implementan como entradas
- ◆ Las operaciones asíncronas corresponden a procedimientos y funciones

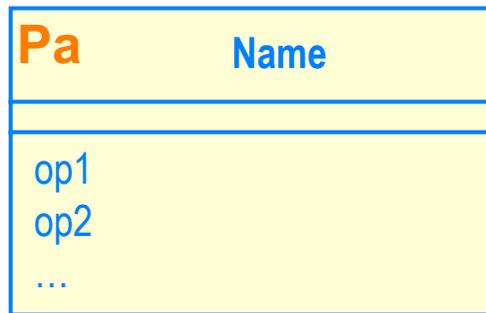
```
package body Name is
  protected body OBCS is
    procedure op1 (...) is
      ...
    begin
      OPCS.op1_code;
    end op1;

    entry op2 when ... is
      ...
    begin
      OPCS.op2_code;
    end op2;

  end OBCS;

begin
  -- package initialization
end Name;
```

Componente pasivo



- ◆ No es un objeto de tiempo real
- ◆ Se implementa mediante un paquete de Ada, sin más

Componente pasivo: realización

```
package Name is
```

```
  procedure op1 (...);
```

```
  procedure op2 (...);
```

```
  ...
```

```
  -- otros procedimientos o funciones
```

```
end Name;
```

```
package body Name is
```

```
  procedure op1 (...) is
```

```
    ...  
  begin
```

```
    ...  
  end op1;
```

```
  procedure op2 is
```

```
    ...  
  begin
```

```
    ...  
  end op2;
```

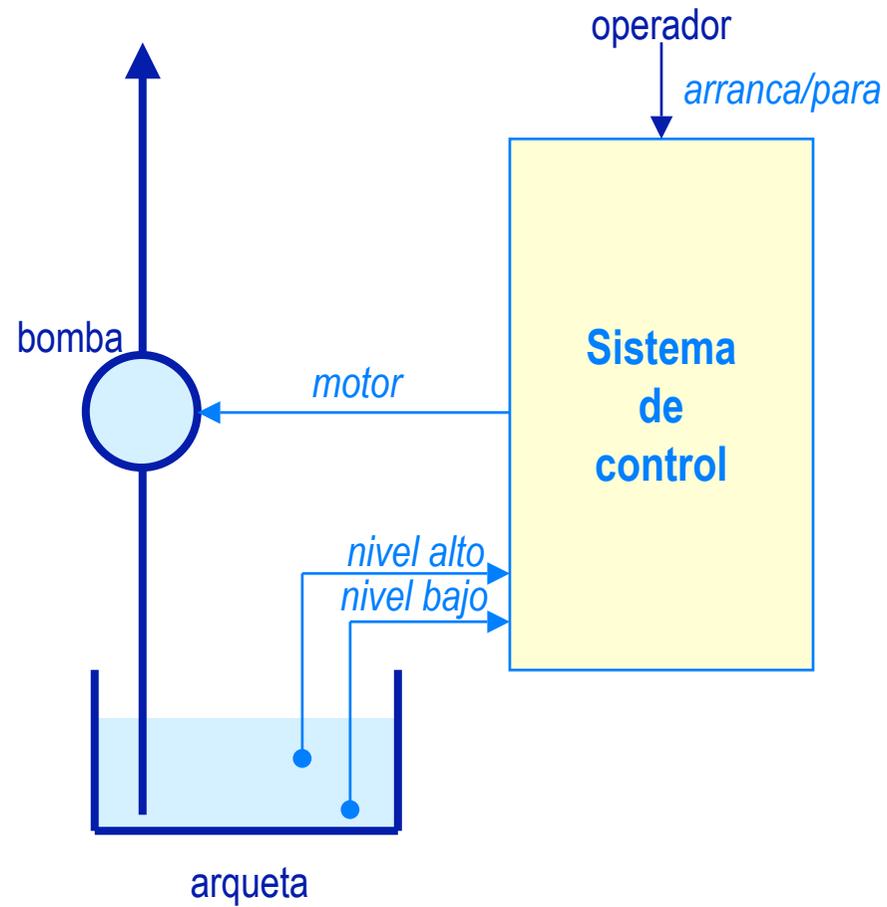
```
begin
```

```
  -- package initialization
```

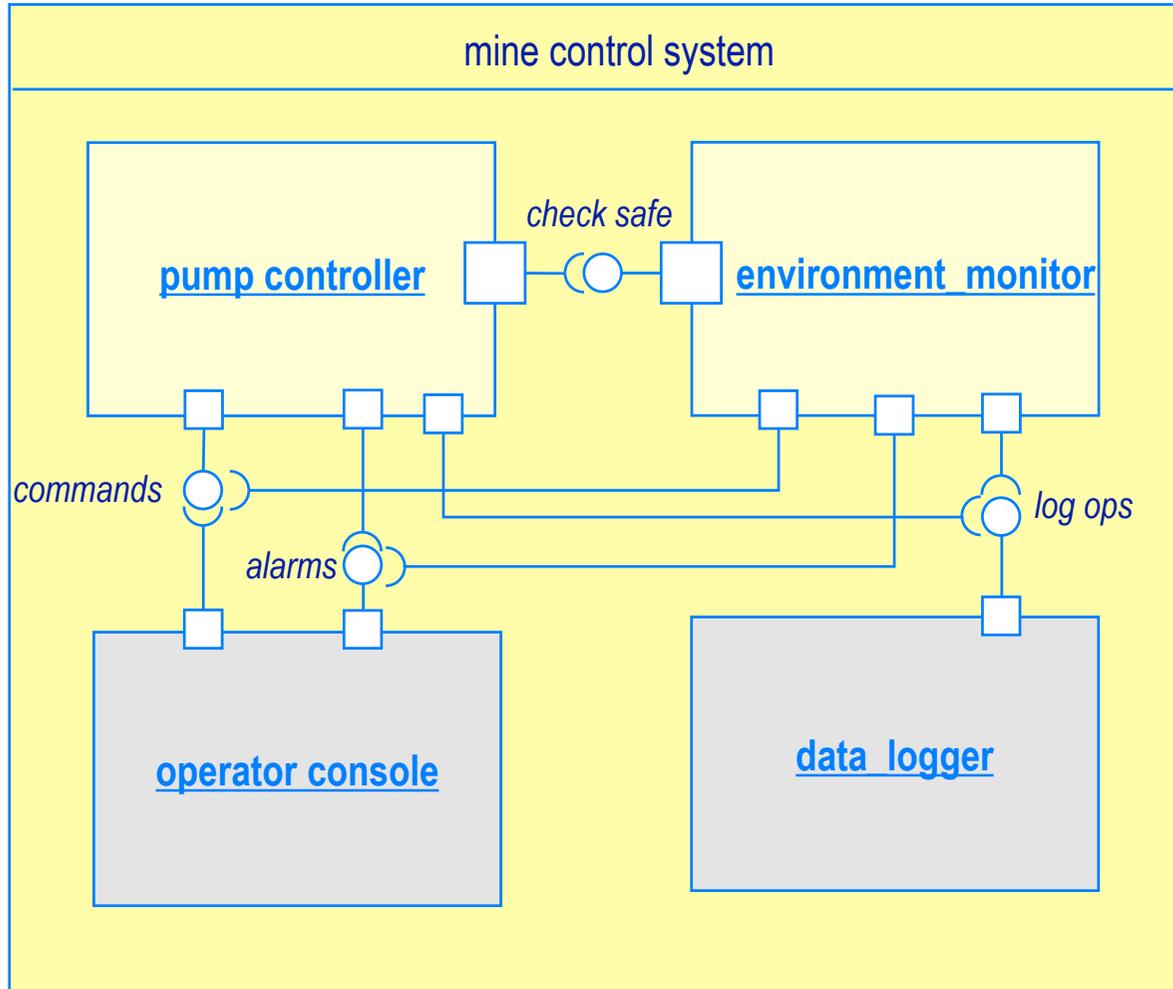
```
end Name;
```

- ◆ Las operaciones visibles son las especificadas en UML
 - ◆ son todas asíncronas (no bloquean a la tarea que llama)

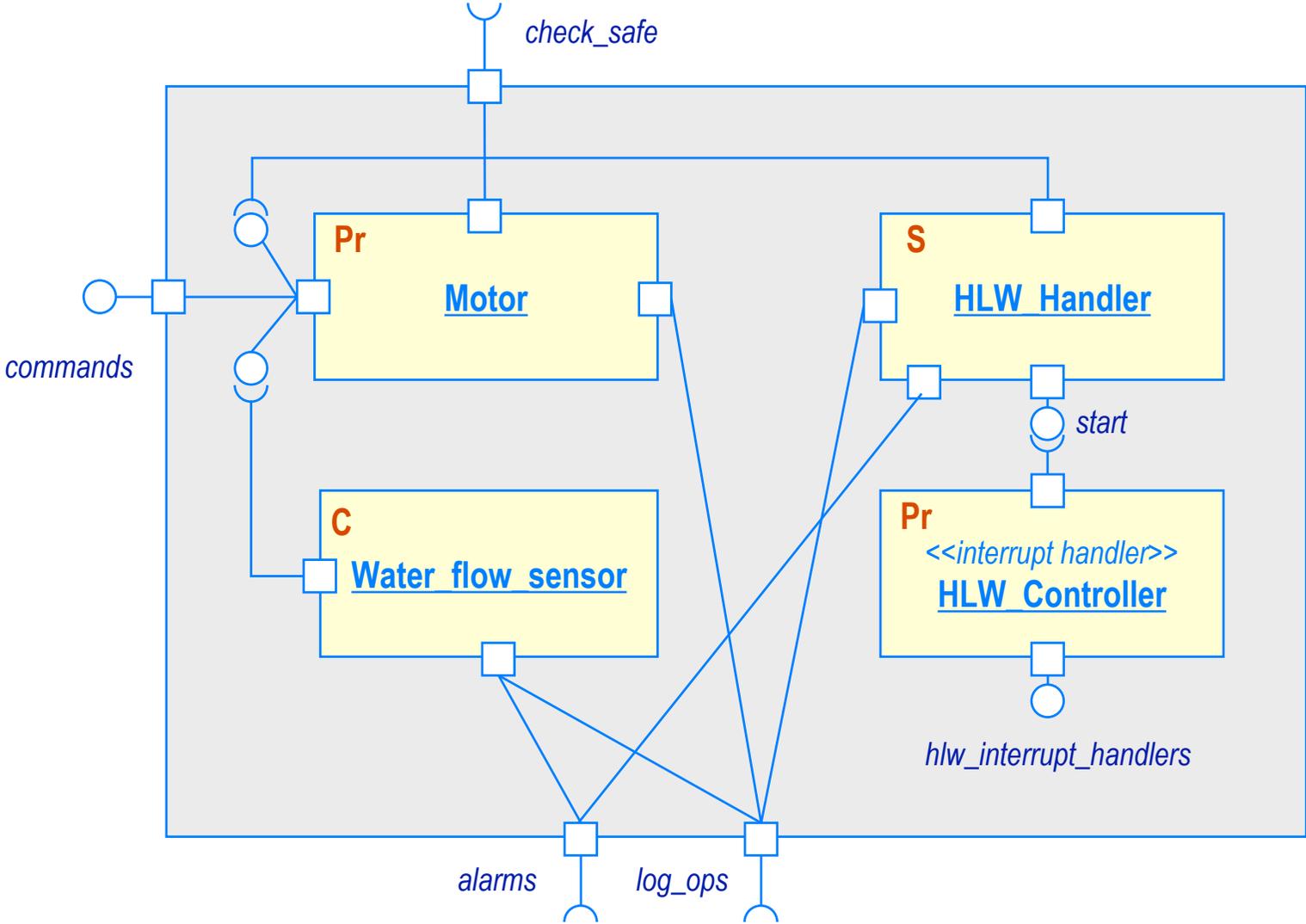
Ejemplo: control de una bomba de agua



Arquitectura lógica

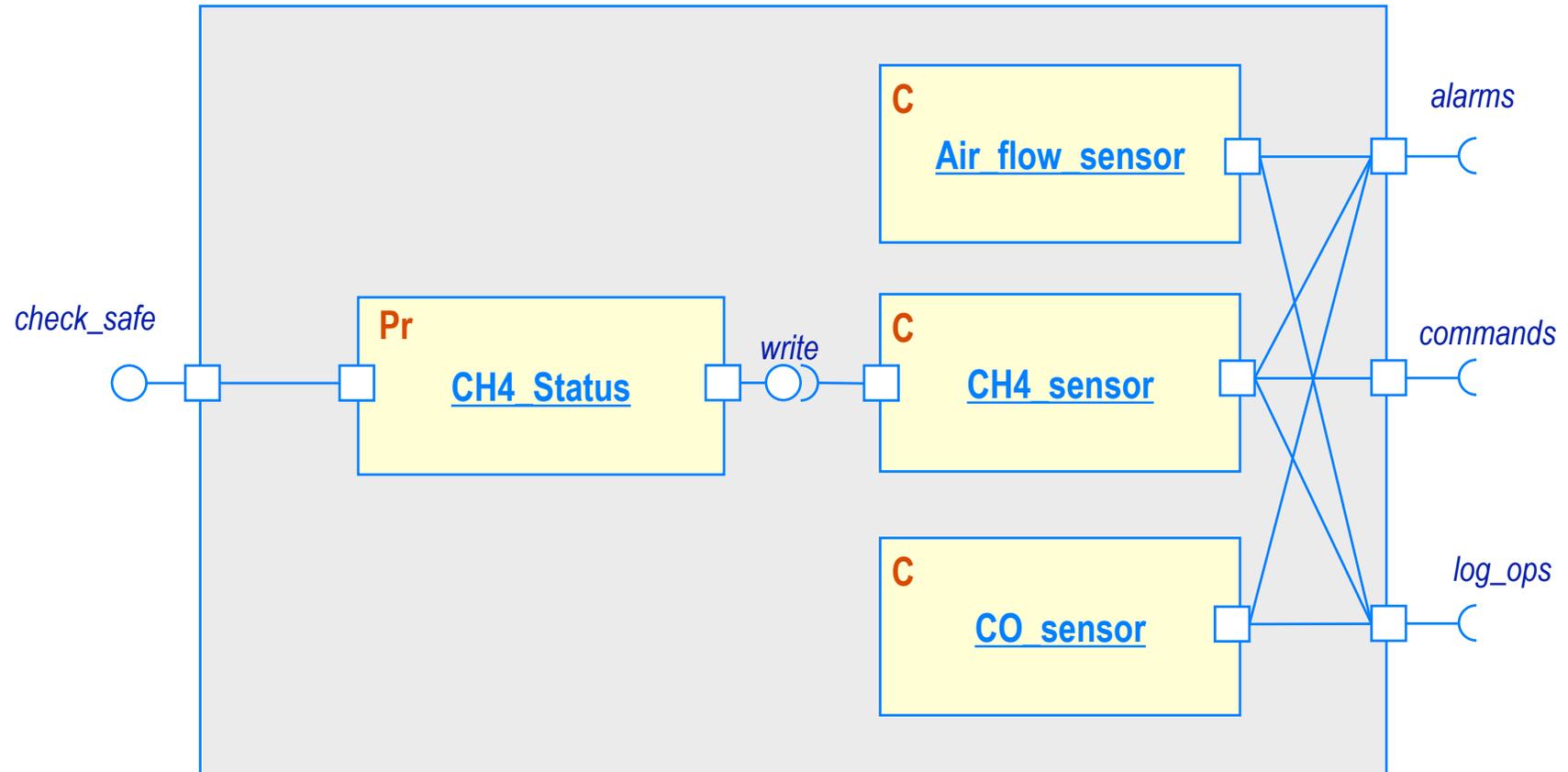


Pump_controller



© Juan Antonio de la Puente 2000-2007

Environment_Monitor



© Juan Antonio de la Puente 2000-2007

Realización: Motor (1)

```
package Motor is -- protected
  type Pump_Status      is (On, Off);
  type Pump_Condition  is (Enabled, Disabled);
  type Operational_Status is
    record
      Status      : Pump_Status;
      Condition   : Pump_Condition;
    end;

  Pump_Not_Safe : exception;

  procedure Is_Safe;
  procedure Not_Safe;
  function  Request_Status return Operational_Status;
  procedure Start_Pump;
  procedure Stop_Pump;
end Motor;
```

Realización: Motor (2)

```
with Motor.HW;
package body Motor is -- protected object

    protected Control is
        pragma Priority(Motor.RTATT.Ceiling_Priority);
        procedure Is_Safe;
        procedure Not_Safe;
        function Request_Status return Operational_Status;
        procedure Set_Pump (To : Pump_Status);
    private
        Motor_Status      : Pump_Status := Off;
        Motor_Condition   : Pump_Condition := Disabled;
    end Control;

    ...
end;
```

Realización: Motor (3)

```
...  
procedure Is_Safe is  
begin  
    Control.Is_Safe;  
end Is_Safe;  
  
procedure Not_Safe is  
begin  
    Control.Not_Safe;  
end Not_Safe;  
  
procedure Start_Pump is  
begin  
    Control.Set_Pump (On);  
end Start_Pump;  
  
procedure Stop_Pump is  
begin  
    Control.Set_Pump (On);  
end Stop_Pump;
```

Realización: Motor (4)

```
...
protected body Control is
  procedure Is_Safe is
  begin
    if Motor_Status = On then
      Motor.HW.Start;
    end if;
    Motor_Codition = Enabled;
  end Is_Safe;

  procedure Not_Safe is
  begin
    if Motor_Status = On then
      Motor.HW.Stop;
    end if;
    Motor_Codition = Disabled;
  end Not_Safe;
  ...
```

Realización: Motor (5)

```
procedure Set_Pump (To : Pump_Status) is
begin
  if To = On then
    if Motor_Status = Off then
      if Motor_Condition = Disabled then
        raise Pump_Not_Safe;
      end if;
      if Environment_Monitor.Check_Safe then
        Motor_Status := On;
        Motor.HW.Start;
      else
        raise Pump_Not_Safe;
      end if;
    end if; -- nothing done if motor was already on
  else -- To = Off
    if Motor_Status = On then
      Motor_Status = Off;
      if Motor_Condition = Enabled then
        Motor.HW.Stop;
      end if;
    end if;
  end if;
end Set_Pump;
end Control;
end Motor;
```

Realización: HLW Handler (1)

```
package HLW_Handler is -- sporadic
  type Water_Mark is (High,Low);
  procedure Start (Level : Water_Mark);
end HLW_Handler;
```

```
package body HLW_Handler is -- sporadic

  procedure Sporadic_Code (Level : Water_Mark) is
  begin
    case Level is
      when High => Motor.Set_Pump(On);
      when Low  => Motor.Set_Pump(Off);
    end case;
  end Sporadic_Code;

  task Thread is
    pragma Priority (HLW_Handler.RTATT.Thread_Priority);
  end Thread;
  ...
```

Realización: HLW Handler (2)

```
...
protected Control is
  procedure Start      (Level :      Water_Mark);
  entry      Wait_Start (Level : out Water_Mark);
private
  Started : Boolean := False;
  Water   : Water_Mark;
end Control;

task body Thread is
  Level : Water_Mark;
begin
  loop
    Control.Wait_Start(Level);
    Sporadic_Code(Level);
  end loop;
end Thread;
...
```

Realización: HLW Handler (3)

```
protected body Control is  
  procedure Start (Level :      Water_Mark) is  
  begin  
    Water := Level;  
    Started := True;  
  end Start;  
  
  entry Wait_Start (Level : out Water_Mark) when Started is  
  begin  
    Level := Water;  
    Started := False;  
  end Wait_Start;  
end Control;  
  
end HLW_Handler;
```

Realización: procedimiento principal

```
with Pump_Controller, Environment_Monitor,...;  
procedure Pump_Control_System is  
begin  
    null;  
end Pump_Control_System;
```

- ◆ ¡El procedimiento principal no hace nada!
- ◆ La tarea de entorno elabora los paquetes
 - al elaborar **HLW_Handler** arranca la tarea esporádica y luego llama al procedimiento principal
- ◆ ¡Una tarea de Ada no termina hasta que han terminado todos sus descendientes!
 - la tarea de entorno espera que termine la tarea esporádica

Resumen

- ◆ Los componentes de tiempo real básicos admiten una realización sencilla en Ada
 - algunos objetos tienen una hebra de ejecución interna
 - algunos objetos tienen una estructura de control interna
- ◆ Todos se pueden implementar con las restricciones de Ravenscar
 - los objetos protegidos se restringen a una entrada con barrera simple y sin colas
- ◆ Se pueden realizar esquemas más complicados
 - cambios de modo en objetos cíclicos y esporádicos
 - detección de desbordamiento de plazos
 - detección de desbordamiento de tiempo de ejecución