1 Objetivo general

La presente práctica se pretende diseñar un sistema de control para un robot autónomo que debe gestionar su comportamiento en un entorno dinámico. El robot debe realizar varias tareas concurrentes, como moverse, evitar obstáculos, y cargar su batería cuando esta esté baja. Estas tareas deben ejecutarse en función de diferentes estados del robot y del entorno. El objetivo es utilizar concurrencia, prioridades fijas y estructuras de planificación jerárquica y determinista.

2 Descripción del Problema

El robot cuenta con varios sensores que le permiten detectar el estado de su entorno y tomar decisiones basadas en la información que obtiene. Entre los posibles estados del robot se incluyen:

- Batería baja: El robot debe cargar su batería.
- Obstáculo detectado: El robot debe evitar el obstáculo (por ejemplo, girando).
- Camino libre: El robot puede continuar su movimiento sin problemas.

3 Desafío

El desafío principal radica en diseñar un sistema de planificación que permita al robot tomar decisiones en tiempo real, respetando ciertos criterios de prioridad. Las decisiones no deben ser tomadas de forma aislada; se deben gestionar de manera jerárquica para que el robot dé prioridad a las acciones más críticas (como cargar la batería) antes de proceder con otras tareas.

Abordaremos el problema en varias fases de desarrollo:

- Fase 1: Máquinas de estados finito (FSD)

  Comienza con un diseño basado en una máquina de estados finitos, donde el robot sigue un conjunto de transiciones simples y ejecuta acciones basadas en condiciones.

- Fase 2: Priorización simple de acciones

  Introduces la capacidad de gestionar prioridades para las tareas. El planificador elige qué acción ejecutar en función de las prioridades, asegurando que las tareas más críticas se ejecuten antes.

- Fase 3: Evolución con priorización y tareas concurrentes

  Se introduce un sistema de tareas concurrentes, donde el robot puede realizar varias acciones al mismo tiempo, gestionando prioridades de manera más avanzada.

- Fase 4: Planificación jerárquica

  En esta fase, el sistema pasa a un modelo jerárquico. El robot toma decisiones estratégicas de alto nivel y luego las desglosa en tareas tácticas más pequeñas y específicas. Este enfoque permite una mayor flexibilidad y control sobre el comportamiento del robot.

4 Fase 1: Máquinas de Estados Finito (FSD)

A continuación se muestra el código asociado a este robot:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Robot_Fase_1 is

   type Estado_Tipo is record
      Hay_Obstaculo : Boolean := False;
      Camino_Libre  : Boolean := False;
      Bateria_Baja  : Boolean := False;
   end record;

   function Leer_Sensores(Paso: Integer) return Estado_Tipo is
      -- Simula lectura de sensores
   begin
      case Paso mod 6 is
         when 0 => return (True, False, False);   -- Obstáculo
         when 1 => return (False, True, False);   -- Camino libre
         when 2 => return (False, False, True);   -- Batería baja
         when 3 => return (True, True, False);    -- Obstáculo + libre
         when 4 => return (True, False, True);    -- Obstáculo + batería baja
         when others => return (False, True, True); -- Camino libre + batería baja
      end case;
   end Leer_Sensores;

   procedure Planificar(E: Estado_Tipo) is
   begin
      Put_Line("Estado detectado:");
      Put_Line("  Obstáculo: " & Boolean'Image(E.Hay_Obstaculo));
      Put_Line("  Camino libre: " & Boolean'Image(E.Camino_Libre));
      Put_Line("  Batería baja: " & Boolean'Image(E.Bateria_Baja));

      -- Lógica determinista
      if E.Bateria_Baja then
         Put_Line("→ Acción: Cargar");
      elsif E.Hay_Obstaculo then
         Put_Line("→ Acción: Girar");
      elsif E.Camino_Libre then
         Put_Line("→ Acción: Avanzar");
      else
         Put_Line("→ Acción: Esperar");
      end if;
   end Planificar;

begin
   for I in 1 .. 10 loop
      declare
         Estado : Estado_Tipo := Leer_Sensores(I);
      begin
         Planificar(Estado);
         delay 1.0; -- Simula tiempo entre ciclos
      end;
   end loop;
end Robot_Fase_1;
```

Este es un ejemplo básico de un **Máquina de Estados Finita** (FSM). El robot lee sensores que simulan

el estado de su entorno, luego ejecuta una acción determinada basada en ese estado. La lógica de planificación es sencilla, siguiendo un flujo de decisiones basado en condiciones.

En este ejemplo, el robot tiene tres posibles estados de interés:

- Batería baja: Si la batería está baja, el robot cargará.
- Obstáculo detectado: Si se detecta un obstáculo, el robot girará.
- Camino libre: Si no hay obstáculos y la batería está bien, el robot avanzará.

En función de estos estados, el robot ejecuta una acción simple y clara. El uso de la estructura `if-elsif-else` en el procedimiento Planificar es una forma directa de implementar un FSM, donde cada transición entre estados depende de las condiciones del entorno.

Este enfoque es adecuado para entornos simples, pero si deseas manejar tareas más complejas y decisiones jerárquicas (como que el robot cargue primero la batería antes de girar, etc.), sería necesario introducir prioridades o una planificación jerárquica para manejar múltiples condiciones concurrentes, como hemos discutido en fases posteriores de la práctica.

5 Fase 2: Priorización simple de acciones

En esta fase, el objetivo es mejorar la planificación del robot mediante la priorización de acciones. En lugar de decidir únicamente sobre el estado del entorno, como en la Fase 1, se introduce una forma de ordenar y priorizar las acciones de acuerdo con la situación detectada.

El robot, al igual que en la fase anterior, debe realizar ciertas acciones como avanzar, girar, y cargar la batería, dependiendo de los sensores. Sin embargo, en esta fase se introducen prioridades para las acciones, de modo que se puede decidir qué acción ejecutar primero, en caso de que varias condiciones sean verdaderas al mismo tiempo.

Se utiliza un planificador que organiza las acciones en función de sus prioridades. El planificador asigna prioridades (Cargar $= 3$, Girar $= 2$, Avanzar $= 1$) y las coloca en una lista. Después, la lista de acciones se ordena por prioridad en orden descendente. El robot luego ejecuta la acción más prioritaria.

Las acciones se representan como tuplas que contienen el nombre de la acción y su prioridad correspondiente. En este caso, la priorización es simple y se resuelve con un algoritmo de ordenación (en este caso, un ordenamiento de burbuja).

5.1 Características:

- Priorización de acciones: El robot no ejecuta las acciones de forma arbitraria, sino que prioriza lo más urgente, como cargar la batería si es necesario.
- Planificación concurrente: Se usan tareas concurrentes en Ada con prioridades fijas para ejecutar las acciones, de forma que siempre se garantiza que las acciones más importantes se ejecuten primero.

5.2 ¿Cómo funciona?

1. El robot detecta el estado mediante los sensores.
2. El planificador asigna prioridades a las acciones.
3. Las acciones se ordenan por prioridad.
4. El robot ejecuta la acción con la mayor prioridad.
5. El ciclo se repite en un entorno simulado, donde los sensores cambian su valor en cada iteración.

Este enfoque añade un mecanismo de prioridades simples, lo que es un paso importante hacia un sistema más flexible y robusto. Esta fase prepara el terreno para etapas más complejas, como la planificación jerárquica de acciones, que pueden considerar condiciones más dinámicas y priorizaciones más complejas.

El código es el siguiente:

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

procedure Robot_Fase_2 is

   type Estado_Tipo is record
      Hay_Obstaculo : Boolean := False;
      Camino_Libre  : Boolean := False;
      Bateria_Baja  : Boolean := False;
   end record;

   type Accion_Tupla is record
      Nombre    : Unbounded_String;
      Prioridad : Integer;
   end record;

   Max_Acciones : constant Integer := 3;
   type Acciones_Array is array (1 .. Max_Acciones) of Accion_Tupla;

   protected type Acciones_Protegidas is
      procedure Guardar(Acciones : in Acciones_Array);
      procedure Obtener(Acciones : out Acciones_Array);
      procedure Resetear;
   private
      Internas : Acciones_Array := (others => (To_Unbounded_String("Esperar"), 0));
   end Acciones_Protegidas;

   protected body Acciones_Protegidas is
      procedure Guardar(Acciones : in Acciones_Array) is
      begin
         Internas := Acciones;
      end Guardar;

      procedure Obtener(Acciones : out Acciones_Array) is
           Temp : Acciones_Array := Internas;
           Tmp  : Accion_Tupla;
      begin
           -- Ordenar por prioridad descendente (burbuja)
           for I in Temp'First .. Temp'Last - 1 loop
              for J in I + 1 .. Temp'Last loop
                 if Temp(J).Prioridad > Temp(I).Prioridad then
                    Tmp := Temp(I);
                    Temp(I) := Temp(J);
                    Temp(J) := Tmp;
                 end if;
              end loop;
           end loop;
```

```
         Acciones := Temp;
   end Obtener;
   procedure Resetear is
   begin
      for I in Internas'Range loop
         Internas(I) := (To_Unbounded_String("Esperar"), 0);
      end loop;
   end Resetear;
end Acciones_Protegidas;

task type Planificador(Acc : access Acciones_Protegidas) is
   entry Planear(Estado : in Estado_Tipo);
end Planificador;

task body Planificador is
   Estado_Actual : Estado_Tipo;
   Acciones : Acciones_Array := (others => (To_Unbounded_String("Esperar"), 0));
begin
   loop
      accept Planear(Estado : in Estado_Tipo) do
         Estado_Actual := Estado;
         Put_Line("Planeando...");

         -- Vaciar Acciones
         for I in Acciones'Range loop
            Acciones(I) := (To_Unbounded_String("Esperar"), 0);
         end loop;

         -- Prioridades: Cargar (3), Girar (2), Avanzar (1)
         if Estado_Actual.Bateria_Baja then
            Acciones(1) := (To_Unbounded_String("Cargar"), 3);
         end if;
         if Estado_Actual.Hay_Obstaculo then
            Acciones(2) := (To_Unbounded_String("Girar"), 2);
         end if;
         if Estado_Actual.Camino_Libre then
            Acciones(3) := (To_Unbounded_String("Avanzar"), 1);
         end if;

         Acc.all.Guardar(Acciones);
      end Planear;
   end loop;
end Planificador;

Acc : aliased Acciones_Protegidas;
P   : Planificador(Acc'Access);

-- Tareas concurrentes con prioridades fijas
task Avanzar is
   pragma Priority(10);
   entry Ejecutar;
end Avanzar;
```

```
task body Avanzar is
   begin
      loop
         accept Ejecutar do
            delay 1.0;
            Put_Line("Ejecutando: Avanzar");
            delay 1.0;
         end Ejecutar;
      end loop;
   end Avanzar;

   task Girar is
      pragma Priority(20);
      entry Ejecutar;
   end Girar;

   task body Girar is
   begin
      loop
         accept Ejecutar do
            delay 1.0;
            Put_Line("Ejecutando: Girar");
            delay 1.0;
         end Ejecutar;
      end loop;
   end Girar;

   task Cargar is
      pragma Priority(30);
      entry Ejecutar;
   end Cargar;

   task body Cargar is
   begin
      loop
         accept Ejecutar do
            delay 1.0;
            Put_Line("Ejecutando: Cargar");
            delay 1.0;
         end Ejecutar;
      end loop;
   end Cargar;

   task Robot;
   task body Robot is
      Estado : Estado_Tipo;
      Acciones : Acciones_Array;
   begin
      for I in 1 .. 5 loop
         -- Simular sensores
         declare
            V : Integer := I mod 4;
         begin
            case V is
               when 0 => Estado := (True, False, False);     -- Obstáculo
               when 1 => Estado := (False, True, False);     -- Camino libre
               when 2 => Estado := (False, False, True);     -- Batería baja
               when others => Estado := (True, True, True);  -- Todo a la vez
            end case;
         end;
```

```
Put_Line("Detectando estado...");
         Put_Line("  Obstáculo: " & Boolean'Image(Estado.Hay_Obstaculo));
         Put_Line("  Camino libre: " & Boolean'Image(Estado.Camino_Libre));
         Put_Line("  Batería baja: " & Boolean'Image(Estado.Bateria_Baja));

         P.Planear(Estado);
         delay 0.5;
         Acc.Obtener(Acciones);

         -- Ejecutar acción más prioritaria
         declare
            Nombre : constant String := To_String(Acciones(1).Nombre);

         begin
            Put_Line("  " & Nombre);
            if Nombre = "Avanzar" then
               Avanzar.Ejecutar;
            end if;
            if Nombre = "Girar" then
               Girar.Ejecutar;
            end if;
            if Nombre = "Cargar" then
               Cargar.Ejecutar;
            end if;
            if Nombre = "Esperar" then
               Put_Line("Esperando...");
            end if;
         end;
         Acc.Resetear;
         delay 0.5;
      end loop;
   end Robot;
begin
   null;
end Robot_Fase_2;
```

6 Fase 3: Ejecución de acciones prioritarias concurrentes

En esta fase, se mejora aún más la planificación y ejecución de acciones del robot, introduciendo un mecanismo de ejecución de múltiples acciones prioritarias de manera concurrente. El código se enfoca en ejecutar las acciones con las más altas prioridades en un orden secuencial, lo que permite que el robot responda de forma más eficiente a las condiciones del entorno.

6.1 Características:

- Prioridad de acciones: Similar a la fase anterior, pero ahora el robot puede ejecutar varias acciones en orden de prioridad. Si una acción tiene una prioridad mayor, se ejecuta antes que las de menor prioridad.
- Ejecución concurrente: Las acciones (como avanzar, girar y cargar) se implementan mediante tareas concurrentes en Ada. Estas tareas tienen prioridades fijas y se ejecutan de forma independiente, lo que permite al robot realizar varias acciones simultáneamente sin bloquearse.

- Evaluación y ejecución por prioridad: Las acciones se ordenan por prioridad y se ejecutan en orden descendente de prioridad. La acción más prioritaria se ejecuta primero. Si hay más de una acción con la misma prioridad, se ejecutan en el orden que se les asignó.
- Comportamiento más flexible: Ahora el robot puede tomar decisiones más complejas al poder ejecutar múltiples acciones prioritarias dependiendo del estado del entorno, lo que hace que el sistema sea más flexible y eficiente.

6.2 Mejoras con respecto a la fase 2:

- Ejecución de múltiples acciones: A diferencia de la fase anterior, donde solo se ejecutaba una acción a la vez, ahora el robot puede ejecutar múltiples acciones en función de las prioridades. Esto permite una ejecución más dinámica y eficaz, sobre todo en situaciones donde varias condiciones del entorno se cumplen simultáneamente.
- Manejo más preciso de los estados: El robot tiene más control sobre su ejecución. Las tareas concurrentes permiten que el robot avance, gire o cargue la batería de forma simultánea si es necesario, siempre respetando las prioridades.
- Planificación jerárquica: Aunque no es completamente jerárquico aún, se hace un primer paso hacia un sistema más complejo de planificación, en el que el robot toma decisiones secuenciales basadas en prioridades. Esto también ayuda a asegurar que las acciones críticas (como cargar la batería) se ejecuten antes de otras acciones menos importantes (como avanzar).

6.3 ¿Cómo Funciona?

- Detección de estado: El robot detecta el estado del entorno mediante los sensores.
- Planificación: El planificador asigna prioridades a las acciones, las ordena, y las guarda.
- Ejecución concurrente: Las tareas del robot ejecutan las acciones ordenadas por prioridad en orden descendente.
- Repetición: El ciclo se repite, lo que permite que el robot se adapte dinámicamente a las condiciones cambiantes del entorno.

Esta fase prepara el camino para un sistema más robusto y eficiente que puede realizar acciones de manera concurrente y priorizada. Aunque no es completamente jerárquico, introduce un enfoque más flexible en la ejecución de tareas, permitiendo que el robot responda a su entorno de forma más efectiva y con una mayor autonomía en la toma de decisiones.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

procedure Robot_Fase_3 is

   type Estado_Tipo is record
      Hay_Obstaculo : Boolean := False;
      Camino_Libre  : Boolean := False;
      Bateria_Baja  : Boolean := False;
   end record;

   type Accion_Tupla is record
      Nombre    : Unbounded_String;
      Prioridad : Integer;
   end record;

   Max_Acciones : constant Integer := 3;
   type Acciones_Array is array (1 .. Max_Acciones) of Accion_Tupla;

   protected type Acciones_Protegidas is
      procedure Guardar(Acciones : in Acciones_Array);
      procedure Obtener(Acciones : out Acciones_Array);
      procedure Resetear;
   private
      Internas : Acciones_Array := (others => (To_Unbounded_String("Esperar"), 0));
   end Acciones_Protegidas;

   protected body Acciones_Protegidas is
      procedure Guardar(Acciones : in Acciones_Array) is
      begin
         Internas := Acciones;
      end Guardar;

      procedure Obtener(Acciones : out Acciones_Array) is
           Temp : Acciones_Array := Internas;
           Tmp  : Accion_Tupla;
        begin
           -- Ordenar por prioridad descendente (burbuja)
           for I in Temp'First .. Temp'Last - 1 loop
              for J in I + 1 .. Temp'Last loop
                 if Temp(J).Prioridad > Temp(I).Prioridad then
                    Tmp := Temp(I);
                    Temp(I) := Temp(J);
                    Temp(J) := Tmp;
                 end if;
              end loop;
           end loop;

           Acciones := Temp;
        end Obtener;

      procedure Resetear is
      begin
         for I in Internas'Range loop
            Internas(I) := (To_Unbounded_String("Esperar"), 0);
         end loop;
      end Resetear;
   end Acciones_Protegidas;
```

```
task type Planificador(Acc : access Acciones_Protegidas) is
   entry Planear(Estado : in Estado_Tipo);
end Planificador;

task body Planificador is
   Estado_Actual : Estado_Tipo;
   Acciones : Acciones_Array := (others => (To_Unbounded_String("Esperar"), 0));
begin
   loop
      accept Planear(Estado : in Estado_Tipo) do
         Estado_Actual := Estado;
         Put_Line("Planeando...");

         -- Vaciar Acciones
         for I in Acciones'Range loop
            Acciones(I) := (To_Unbounded_String("Esperar"), 0);
         end loop;

         -- Prioridades: Cargar (3), Girar (2), Avanzar (1)
         if Estado_Actual.Bateria_Baja then
            Acciones(1) := (To_Unbounded_String("Cargar"), 3);
         end if;
         if Estado_Actual.Hay_Obstaculo then
            Acciones(2) := (To_Unbounded_String("Girar"), 2);
         end if;
         if Estado_Actual.Camino_Libre and not Estado_Actual.Hay_Obstaculo then
            Acciones(3) := (To_Unbounded_String("Avanzar"), 1);
         end if;

         Acc.all.Guardar(Acciones);
      end Planear;
   end loop;
end Planificador;

Acc : aliased Acciones_Protegidas;
P   : Planificador(Acc'Access);

-- Tareas concurrentes con prioridades fijas
task Avanzar is
   pragma Priority(10);
   entry Ejecutar;
end Avanzar;

task body Avanzar is
begin
   loop
      accept Ejecutar do
         delay 1.0;
         Put_Line("Ejecutando: Avanzar");
         delay 1.0;
      end Ejecutar;
   end loop;
end Avanzar;

task Girar is
   pragma Priority(20);
   entry Ejecutar;
end Girar;
```

```
task body Girar is
begin
   loop
      accept Ejecutar do
         delay 1.0;
         Put_Line("Ejecutando: Girar");
         delay 1.0;
      end Ejecutar;
   end loop;
end Girar;

task Cargar is
   pragma Priority(30);
   entry Ejecutar;
end Cargar;

task body Cargar is
begin
   loop
      accept Ejecutar do
         delay 1.0;
         Put_Line("Ejecutando: Cargar");
         delay 1.0;
      end Ejecutar;
   end loop;
end Cargar;

task Robot;
task body Robot is
   Estado : Estado_Tipo;
   Acciones : Acciones_Array;
begin
   for I in 1 .. 5 loop
      -- Simular sensores
      declare
         V : Integer := I mod 4;
      begin
         case V is
            when 0 => Estado := (True, False, False);     -- Obstáculo
            when 1 => Estado := (False, True, False);     -- Camino libre
            when 2 => Estado := (False, False, True);     -- Batería baja
            when others => Estado := (True, True, True);  -- Todo a la vez
         end case;
      end;

      Put_Line("Detectando estado...");
      Put_Line("  Obstáculo: " & Boolean'Image(Estado.Hay_Obstaculo));
      Put_Line("  Camino libre: " & Boolean'Image(Estado.Camino_Libre));
      Put_Line("  Batería baja: " & Boolean'Image(Estado.Bateria_Baja));

      P.Planear(Estado);
      delay 0.5;

      Acc.Obtener(Acciones);
```

```
         -- Ejecutar acción más prioritaria
         -- Ejecutar varias acciones en orden de prioridad
        for I in Acciones'Range loop
           declare
              Nombre : constant String := To_String(Acciones(I).Nombre);
           begin
              exit when Nombre = "Esperar";  -- Fin de acciones relevantes

              Put_Line("Ejecutando acción: " & Nombre);
              if Nombre = "Avanzar" then
                 Avanzar.Ejecutar;
              elsif Nombre = "Girar" then
                 Girar.Ejecutar;
              elsif Nombre = "Cargar" then
                 Cargar.Ejecutar;
              end if;
           end;
        end loop;

         Acc.Resetear;
         delay 0.5;
      end loop;
   end Robot;

begin
   null;
end Robot_Fase_3;
```

7 Fase 4: Planificación jerárquica con tareas concurrentes

En esta fase, se introduce un modelo jerárquico de planificación, donde las tareas de mayor nivel (estratégicas) y las de menor nivel (tácticas) trabajan juntas para lograr que el robot tome decisiones más complejas y eficientes.

7.1 Características:

- Planificación jerárquica: Se implementan dos niveles de planificación:
  - Planificador estratégico: Establece metas basadas en el estado del robot (evitar obstáculos, cargar batería, moverse).
  - Planificador táctico: Traducirá esas metas en acciones concretas que el robot puede ejecutar.
- Metas en lugar de acciones: Las metas son evaluadas y asignadas primero por el planificador estratégico y luego se convierten en acciones por el planificador táctico. Esto facilita la toma de decisiones complejas y permite que el robot tenga una visión más global y estructurada de sus tareas.
- Ejecución concurrente: Como en fases anteriores, las acciones (moverse, girar, cargar) se ejecutan concurrentemente, pero ahora con una jerarquía de metas que las guía.

7.2 Mejoras con respecto a la fase 3:

- Descentralización de la toma de decisiones: La división de la planificación en estratégica y táctica permite al robot manejar las metas a un nivel abstracto antes de traducirlas en acciones concretas,

mejorando la claridad y organización del sistema.

- Mayor flexibilidad: El sistema jerárquico hace que el robot sea más flexible al manejar situaciones complejas, como cuando debe decidir entre moverse, evitar obstáculos o cargar la batería, priorizando las metas.
- Modularidad: Al dividir las responsabilidades entre los planificadores estratégico y táctico, se facilita la extensión del sistema para incorporar más tipos de metas y acciones en el futuro.

7.3 ¿Cómo Funciona?

- Evaluación de estado y metas: El planificador estratégico evalúa el estado del robot y establece las metas (evitar obstáculos, moverse, cargar la batería).
- Planificación de acciones: El planificador táctico traduce estas metas en acciones concretas (como girar, avanzar o cargar).
- Ejecución concurrente: Las acciones se ejecutan en paralelo respetando la prioridad y la secuencia de ejecución.

Esta fase introduce un enfoque jerárquico, organizando la planificación en niveles de abstracción (metas y acciones), lo que mejora la toma de decisiones y hace que el robot pueda abordar tareas complejas de forma más organizada y flexible.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

procedure Robot_Jerarquico is

   type Estado_Tipo is record
      Hay_Obstaculo : Boolean := False;
      Camino_Libre  : Boolean := False;
      Bateria_Baja  : Boolean := False;
   end record;

   -- Cambiar Meta_Tipo a un array de metas
   type Meta_Tipo is (Evitar_Obstaculo, Cargar_Bateria, Moverse, Ninguna);

   -- Usar un array de metas
   type Metas_Array is array (1 .. 3) of Meta_Tipo;

   type Accion_Tupla is record
      Nombre    : Unbounded_String;
      Prioridad : Integer;
   end record;

   Max_Acciones : constant Integer := 3;
   type Acciones_Array is array (1 .. Max_Acciones) of Accion_Tupla;

   protected type Acciones_Protegidas is
      procedure Guardar(Acciones : in Acciones_Array);
      procedure Obtener(Acciones : out Acciones_Array);
      procedure Resetear;
   private
      Internas : Acciones_Array := (others => (To_Unbounded_String("Esperar"), 0));
   end Acciones_Protegidas;

   protected body Acciones_Protegidas is
      procedure Guardar(Acciones : in Acciones_Array) is
      begin
         Internas := Acciones;
      end Guardar;
```

```
   procedure Obtener(Acciones : out Acciones_Array) is
        Temp : Acciones_Array := Internas;
        Tmp  : Accion_Tupla;
     begin
        for I in Temp'First .. Temp'Last - 1 loop
           for J in I + 1 .. Temp'Last loop
              if Temp(J).Prioridad > Temp(I).Prioridad then
                 Tmp := Temp(I);
                 Temp(I) := Temp(J);
                 Temp(J) := Tmp;
              end if;
           end loop;
        end loop;
        Acciones := Temp;
     end Obtener;

   procedure Resetear is
   begin
      for I in Internas'Range loop
         Internas(I) := (To_Unbounded_String("Esperar"), 0);
      end loop;
   end Resetear;
end Acciones_Protegidas;

task type Planificador_Estrategico is
   entry Evaluar(Estado : in Estado_Tipo; Metas : out Metas_Array);
end Planificador_Estrategico;

task body Planificador_Estrategico is
begin
   loop
     select
      accept Evaluar(Estado : in Estado_Tipo; Metas : out Metas_Array) do
         -- Comenzamos con las metas vacías
         Metas := (Ninguna, Ninguna, Ninguna);  -- Inicializamos las metas en "Ninguna"

         -- Prioridad más alta: cargar si la batería está baja
         if Estado.Bateria_Baja then
            Metas(1) := Cargar_Bateria;
         end if;

         -- Si hay un obstáculo, debemos evitarlo
         if Estado.Hay_Obstaculo then
            if Metas(1) = Ninguna then
               Metas(1) := Evitar_Obstaculo;  -- Si no hay meta asignada, asignamos "Evitar Obje
            else
               Metas(2) := Evitar_Obstaculo;  -- Si ya hay una meta, asignamos "Evitar Objeto" a
            end if;
         end if;

         -- Si el camino está libre, nos podemos mover
         if Estado.Camino_Libre then
            if Metas(1) = Ninguna then
               Metas(1) := Moverse;
            elsif Metas(2) = Ninguna then
               Metas(2) := Moverse;
            end if;
         end if;
      end Evaluar;
```

```
      or
         terminate;
      end select;
   end loop;
end Planificador_Estrategico;

task type Planificador_Tactico(Acc : access Acciones_Protegidas) is
   entry Planear(Metas : in Metas_Array);
end Planificador_Tactico;

task body Planificador_Tactico is
   Acciones : Acciones_Array := (others => (To_Unbounded_String("Esperar"), 0));
begin
   loop
     select
      accept Planear(Metas : in Metas_Array) do
         Acciones := (others => (To_Unbounded_String("Esperar"), 0));

         -- Asignar las acciones de acuerdo con las metas
         for I in Metas'Range loop
            case Metas(I) is
               when Cargar_Bateria =>
                  Acciones(1) := (To_Unbounded_String("Cargar"), 3);
               when Evitar_Obstaculo =>
                  Acciones(2) := (To_Unbounded_String("Girar"), 2);
               when Moverse =>
                  Acciones(3) := (To_Unbounded_String("Avanzar"), 1);
               when others =>
                  null;
            end case;
         end loop;

         -- Guardar las acciones ordenadas por prioridad
         Acc.all.Guardar(Acciones);
      end Planear;
     or
         terminate;
     end select;
   end loop;
end Planificador_Tactico;

Acc : aliased Acciones_Protegidas;
PE  : Planificador_Estrategico;
PT  : Planificador_Tactico(Acc'Access);

task Avanzar is
   --pragma Priority(10);
   entry Ejecutar;
end Avanzar;

task body Avanzar is
begin
   loop
     select
      accept Ejecutar do
         Put_Line("Ejecutando: Avanzar"); delay 1.0;
      end Ejecutar;
     or
         terminate;
     end select;
   end loop;
end Avanzar;
```

```
task Girar is
   --pragma Priority(20);
   entry Ejecutar;
end Girar;

task body Girar is
begin
   loop
     select
      accept Ejecutar do
         Put_Line("Ejecutando: Girar"); delay 1.0;
      end Ejecutar;
     or
         terminate;
     end select;
   end loop;
end Girar;

task Cargar is
   --pragma Priority(30);
   entry Ejecutar;
end Cargar;

task body Cargar is
begin
   loop
     select
      accept Ejecutar do
         Put_Line("Ejecutando: Cargar"); delay 1.0;
      end Ejecutar;
     or
         terminate;
     end select;
   end loop;
end Cargar;

task Robot;
task body Robot is
   Estado : Estado_Tipo;
   Metas  : Metas_Array;
   Acciones : Acciones_Array;
begin
   for I in 1 .. 5 loop
      declare
         V : Integer := I mod 4;
      begin
         case V is
            when 0 => Estado := (True, False, False);
            when 1 => Estado := (False, True, False);
            when 2 => Estado := (False, False, True);
            when others => Estado := (True, True, True);
         end case;
      end;

      Put_Line("Detectando estado...");
      Put_Line("  Obstáculo: " & Boolean'Image(Estado.Hay_Obstaculo));
      Put_Line("  Camino libre: " & Boolean'Image(Estado.Camino_Libre));
      Put_Line("  Batería baja: " & Boolean'Image(Estado.Bateria_Baja));
```

```
         PE.Evaluar(Estado, Metas);
         PT.Planear(Metas);
         delay 0.2;

         Acc.Obtener(Acciones);

         -- Ejecutar las acciones según su prioridad
         for I in Acciones'Range loop
            declare
               Nombre : constant String := To_String(Acciones(I).Nombre);
            begin
               exit when Nombre = "Esperar";
               if Nombre = "Avanzar" then
                  Avanzar.Ejecutar;
               elsif Nombre = "Girar" then
                  Girar.Ejecutar;
               elsif Nombre = "Cargar" then
                  Cargar.Ejecutar;
               end if;
            end;
         end loop;

         Acc.Resetear;
         delay 0.5;
      end loop;
   end Robot;

begin
   null;
end Robot_Jerarquico;
```

ENTREGABLE: Partiendo de la fase 3 o 4, se pide:

- Implementar más sensores y metas para enriquecer el comportamiento del robot.
- Explorar la incorporación de un planificador a nivel de misión, que coordine las metas estratégicas del robot en un nivel más alto.