

Concurrent Programming for the Control of Hexapod Walking

Bernard Thirion, Laurent Thiry
Groupe LSI, Laboratoire MIPS

ESSAIM, Ecole Supérieure des Sciences Appliquées pour l'Ingénieur-Mulhouse
12 rue des Frères Lumière
68093 Mulhouse Cedex, France
e-mail : {b.thirion, l.thiry}@uha.fr

Abstract

Ada95 is a powerful language with a great number of original constructions. Learning these constructions requires the finalization of projects that are both interesting and motivating for students, as well as the coverage of the different constructions during the project. Moreover, the field of mobile robotics is one that requires real-time programming and appropriate software architectures. More particularly, legged robots offer a real challenge as regards autonomy and the coordination of movements of the different legs. This field proves fruitful for the definition of projects on concurrent programming. The present paper describes such a project about an architecture for an omnidirectional legged robot. In a resolutely object-oriented approach, the project helps to teach the main constructions of the Ada language. Among others, it deals with child units, generics, tagged types and type extension, tasking, protected objects, family entries, asynchronous transfer of control, discriminants, etc. Numerous extensions can be considered within this project.

1. Introduction

Mobile robotics is a vast, multidisciplinary field of investigation which covers various domains as mechanics, electrical and software engineering, vision, etc. The renewed interest in this field is due to the fact that the robot can now be given great computation power at a low cost. From a software point of view, a robot needs an efficient, appropriate control architecture which allows the integration of the robot's numerous functions: movement of the platform, estimation of the position, perception of the environment, navigation, decision and planning, actions on the environment, vision, etc [1]. In general, these functions must occur jointly, in real-time. That is why the field of mobile robotics is an important source of inspiration for motivating projects that integrate concurrent programming and real-time aspects.

Moreover, mobile robotics helps to deal with a number of concepts linked to the control of systems, using either classic control methods or more advanced methods like fuzzy logic, neural networks, etc.

Ada is well suited to the teaching of the fundamental concepts of software engineering and concurrent programming. It is also starting to be used for projects about mobile robotics [2]. This paper will, more particularly, consider the case of legged locomotion. An interesting point to be studied is the coordination of the leg movements, so as to highlight the different walking gaits – tripod gait, slow gait, etc. The control of the walking algorithms is usually not centralized, which means that each leg is relatively independent in its movements. Such decentralized control will lead to interesting problems linked to the coordination and synchronization of movements which provide fair gaits and maintain the robot's stability. In particular, lifting one leg is concurrent with lifting others and can thus cause a conflict. This conflict is processed using the well-known algorithm of the dining philosophers, which is an interesting practical application of that algorithm.

The purpose of this paper is to illustrate the use of the different Ada constructions — in particular concurrent programming — with an example which interested students greatly. After giving some details on legged robots and walking, the paper will progressively describe the architecture of the whole system.

2. Legged robots

The understanding of walking mechanisms and the design of robust walking algorithms for legged robots remain a challenge. To try and take up this challenge, many laboratories have built walking machines [3-4]. There are two types of machines: the ones exhibiting dynamic stability and the ones exhibiting static stability. For robots with dynamic stability, the center of gravity can leave the support polygon; they are usually robots with a limited number of legs (1 to 4)

which have to keep their balance permanently. Robots with static stability maintain their center of gravity within the support polygon; they have at least four legs. The case of hexapods or octopods is interesting as they provide static stability and numerous walking gaits.

The walking algorithms are often decentralized and designed by assembling a multitude of small processes (or agents) which are executed concurrently [5]. The complexity of the computational aspects (kinematic computation, trajectory planning, etc) has led some roboticists [5-7] to distribute the processing over distributed architectures. For example the Robug IV robot [7] has 4 processors per leg and 8 legs, that is to say 32 processors linked by a CAN fieldbus. This kind of structure requires distributed algorithms; therefore the distributed philosophers algorithm will be used for the allocation of the privileges of leg lifting.

The author's team [8] has developed a hexagonal hexapod robot – called Bunny – to validate their software architectures concerning decentralized control (Fig. 1).



Figure 1. Bunny, the omnidirectional robot

Bunny is an omnidirectional robot with 18 degrees of freedom (3 degrees per leg). The platform is not directly used within student projects for reasons of mechanical fragility, but it is the inspiration for the definition of the problems. The following parts will more specifically consider the problem of leg coordination and the generation of walking gaits.

3. Fundamental Principles

On an ideal surface, a leg moves in a cyclic way between two extreme positions — AEP which is the anterior extreme position and PEP the posterior extreme position. A leg is said to be in *retraction* when it is on ground and pushes the platform forward. It is

said to be in *protraction* when it is lifted and tries to reach its AEP (Fig. 2).

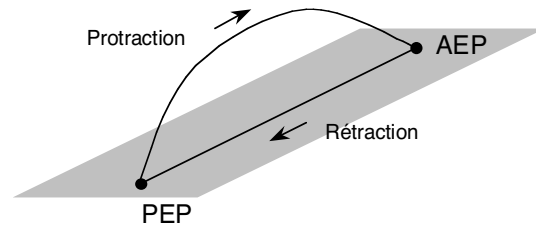


Figure 2. Basic cycle of a leg

For hexapods, static stability is maintained at all times by the configuration of the legs that are on the ground. The observation of insects shows that some specimens adapt their gait according to the speed at which they move. This is possible because the protraction speed of a leg is maximum (Max) while its retraction speed (S) varies and depends on the animal's speed, which results in different gaits. For example, for a hexapod moving at high speed, 3 legs are lifted and 3 legs are pushing; at an average speed 2 legs are lifted and 4 legs are pushing, and at a low speed (uneven ground or insect carrying a load), only 1 leg is lifted and the 5 others are pushing. Observations also show that the protraction of the legs moves like a wave that is propagated from the rear to the front of the animal. These movements are called wave gaits. It has been shown that these movements are stable and optimum and that they result in equal gaits for each leg (Fig. 3).

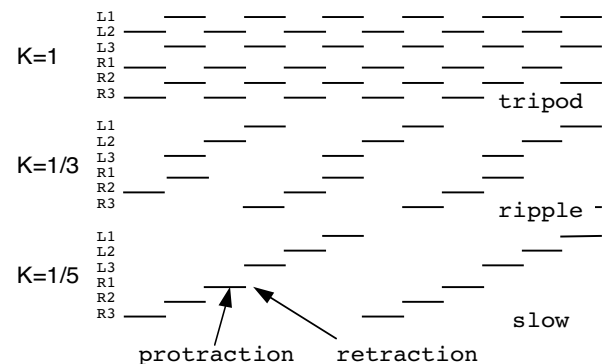
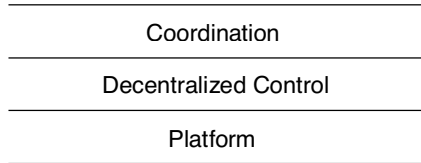


Figure 3. Wave gaits

Other studies have shown that the different gaits can be obtained using local synchronization rules. The robustness and flexibility of walking is then the result of the interaction and cooperation of several mechanisms. To obtain the emergence of those coordinated movements, a current approach is to use recurrent neural networks and an interconnection architecture obtained with evolutionary algorithms [9].

The present project gives the same results using a network of objects (Fig. 20) which allow the propagation of causal chains of events/actions and the interaction of several, more or less redundant, resynchronization mechanisms, which gives great robustness to the algorithm. The global architecture [10] of the project is divided into three main layers.



The *Platform* layer abstracts a hexapod which evolves and which can be controlled. The *Decentralized Control* layer contains 6 tasks for the control of the legs. These leg controllers are subject to constraints of conflict resolution imposed by the *Coordination* layer. This general system architecture is chosen in order to deal with the principal Ada constructions.

4. The Platform

The platform is an instance of the Façade pattern [11]; its role is to abstract the robot. So, the implementation can vary (3D rendering, real robot, etc.). To perform the simulation the hexapod is simplified. In particular, a leg movement occurs in an abstract space and consists of a simple position between AEP and PEP and a state (lifted or not). This model can be improved using a more precise geometry of the leg. Minimum graphics will help to draw the evolution of the legs (using AdaGraph for example). Fig.4 shows the class diagram in UML [12].

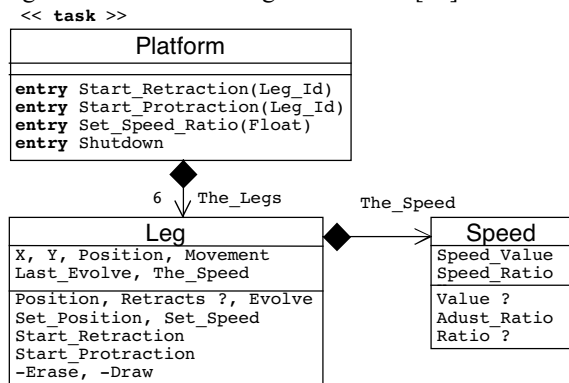


Figure 4. The Platform

The *Speed* class abstracts the fact that the retraction and protraction speeds are not the same. The speed is adjusted through a simple coefficient K which is the

ratio between the two speeds. The class is translated into Ada through a private type as follows.

```

Package Speed is
Maximum:    constant := 1.0;
Stopped:    constant := 0.0;
Full_Speed: constant := 1.0;
type object is private;
function Value(K: Float) return Object;
function Value(O: Object) return Float;
function Ratio(O: Object) return Float;
procedure Adjust_Ratio(O: in out Object;
                    K: Float);

private
type Object is record
    Speed_Ratio: Float := Stopped;
    Speed_Value: Float := 0.0;
end record;
end Speed;

```

This is the general design principle adopted for the translation of a class into Ada. The *Speed* body does not present any difficulties.

A leg is considered as a dynamical system which drives the position of the tip towards AEP or PEP. Once it has arrived in one of those positions, the leg stops moving. It will be the role of the leg controller to give it a cyclical behavior. The hybrid finite state machine in Fig. 5 specifies its functioning.

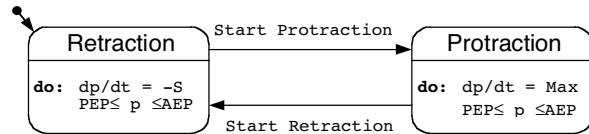


Figure 5. Basic leg behavior

X and Y are the graphic coordinates of the origin of the leg. The contract of the *Leg* class is defined using a *tagged type*, so as to allow the extension of this type. The specification of the package is the following:

```

with Calendar; with Speed;
package Leg is
AEP: constant Float := 1.0;
PEP: constant Float := -1.0;
type Object is tagged private;
type Class_Ref is access all Object'Class;

function Value(X, Y: Integer; P: Float)
return Class_Ref;
function Retracts(Leg: Object)
return Boolean;
function Position(Leg: Object)
return Float;
procedure Set_Position(Leg: in out Object;
                    To: Float);
procedure Set_Speed(Leg: in out Object;
                    To: Speed.Object);
procedure Start_Retraction(
    Leg: in out Object);

procedure Start_Protraction(
    Leg: in out Object);
procedure Evolve (Leg: in out Object);

```

```

private
type Movements is (
    Protraction,
    Retraction);
type Object is tagged record
    X, Y      : Integer := 0;
    Position  : Float   := AEP;
    Movement  : Movements := Retraction;
    The_Speed : Speed.Object;
    Last_Evolve: Calendar.Time ;
end record;
end Leg;

```

There is no particular problem about the package body. A few methods are given:

```

package body Leg is
procedure Draw (Leg: Object);
procedure Erase(Leg: Object);
function Value(X, Y: Integer;
    Position: Float) return Class_Ref is
begin
    return New Object'(X, Y, Position,
        Retraction,
        Speed.Value(0.0),
        Calendar.Clock);
end;

procedure Start_Retraction(
    Leg: in out Object) is
begin
    Erase (Leg);
    Leg.Movement := Retraction;
    Draw (Leg);
end;

-- etc

procedure Evolve (Leg: in out Object) is
    Step: Float; Now: Calendar.Time; S: Float;
    use Calendar ;
begin
    Erase (Leg);
    Now := Calendar.Clock;
    if Leg.Movement = Retraction then
        S := Speed.Value(Leg.The_Speed);
    else
        S := Speed.Maximum;
    end if;
    Step := S * Float(Now - Leg.Last_Evolve);
    case Leg.Movement is
        when Protraction => Leg.Position :=
            Float'Min(Leg.Position + Step, AEP);
        when Retraction => Leg.Position :=
            Float'Max(Leg.Position - Step, PEP);
    end case;
    Leg.Last_Evolve := Now;
    Draw(Leg);
end;
end Leg;

```

The platform is a task which ensures the motion of the 6 legs. The task accepts its Rendez-Vous and makes the legs move according to a sampling period. The task also has an *access discriminant* to an event notifier which will be described in § 5.

```

with Generic_Notifier;
package Platform is
type Leg_Ids is (L1, L2, L3, R3, R2, R1);
type Leg_Events is (
    PEP_Reached, Is_Late, AEP_Reached,
    Leg_Killed);
package Notifier is new
    Generic_Notifier(Leg_Ids, Leg_Events);

task type Object(
    Notifier: Platform.Notifier.Ref :=
        new Platform.Notifier.Object) is
    entry Shutdown;
    entry Start_Retraction (L: Leg_Ids);
    entry Start_Protraction (L: Leg_Ids);
    entry Set_Speed_Ratio (To: float);
end;
type Ref is access Object;
end Platform;

```

The *Platform* body exploits a private child unit *Platform.Legs* which manages the 6-leg collection.

```

with Calendar, Leg, Platform.Legs, Speed;
package body Platform is
    Period: constant := 0.1;
    procedure Notify_Shutdown (N: Notifier.Ref);
    task body Object is
        Alive: Boolean := True;
        The_Legs: Legs.Object := New_Legs (...);
        Next: Calendar.Time := Calendar.Clock;
        use Calendar; use Leg;
    begin
        while Alive loop
            select
                accept Start_Protraction(L: Leg_Ids) do
                    Start_Protraction(The_Legs(L).all);
                end;
            or
                accept Start_Retraction(L: Leg_Ids) do
                    Start_Retraction(The_Legs(L).all);
                end;
            or
                accept Set_Speed_Ratio(To: float) do
                    Legs.Set_Speed_Ratio(The_Legs, To);
                end;
            or
                accept Shutdown do Alive := False; end;
            or
                delay until Next;
                Legs.Evolve(The_Legs);
                Next := Next + Period;
            end select;
        end loop;
        Notify_Shutdown(Notifier);
        while Notifier.Has_Pendings loop --shutdown
            -- accept remaining Rendez-Vous
        end loop;
    end;
    ----- etc -----
end Platform;

```

To start a *Shutdown*, the platform exploits the *Notifier* to warn the leg controllers of the imminent end of the platform. In the shutdown phase, the task continues to accept Rendez-Vous as long as there are undelivered events (see § 5).

The *Platform.Legs* unit illustrates the possibilities of *private child units* and the *class wide types* for the creation of polymorphic arrays.

```
with Leg;
private package Platform.Legs is
  type Object is array(Leg_Ids) of
    Leg.Class_Ref;

  function New_Legs(X, Y, Bug_Size: Integer)
    return Object;
  procedure Evolve(Legs: Object);
  procedure Set_Speed_Ratio(Legs: Object;
    To: Float);
end;
```

5. Notification of Events

As the platform is a façade which ensures uncoupling according to the decentralized control layer, it must be able to notify the occurrence of important events to the upper layer. The mechanism used is that of an event notifier, as shown in Fig. 6.

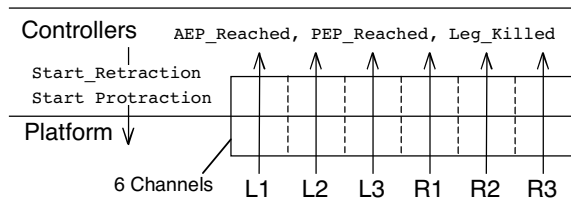


Figure 6. Event notification

The notifier has several channels. It will be built as an instance of a *generic unit*. It also allows the introduction of *protected objects* and *family entries*. The class diagram in Fig. 7 describes the situation.

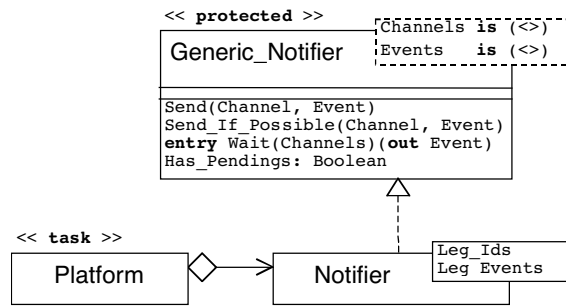


Figure 7. Generic Notifier

- The notifier helps to send :
- 1) memorized high-priority events which override possible undelivered events
 - 2) low-priority events which will be lost if they are not sent.

```
generic
  type Channels is (<>);
  type Events is (<>);
package Generic_Notifier is
  type Notification is record
    Event : Events;
    Arrived: Boolean := False;
  end record;
  type Notifications is array(Channels)
    of Notification;
  protected type Object is
  entry Wait(Channels)(E: out Events);
  procedure Send(C: Channels; E: Events);
  procedure Send_If_Possible(
    C: Channels; E: Events);
  function Has_Pendings return Boolean;
private
  The_Events: Notifications;
end;
type Ref is access Object;
end Generic_Notifier;

package body Generic_Notifier is
  protected body Object is
  entry Wait(for C in Channels)(E: out
    Events) when The_Events(C).Arrived is
  begin
    E := The_Events(C).Event;
    The_Events(C).Arrived := False;
  end;
  procedure Send (C: Channels; E: Events) is
  begin
    The_Events(C):= Notification'(E, True);
  end;
  procedure Send_If_Possible (
    C: Channels; E: Events) is
  begin
    if not The_Events(C).Arrived then
      Send (C, E);
    end if;
  end;
  function Has_Pendings return Boolean is
  begin
    for C in Channels loop
      if The_Events(C).Arrived then
        return True;
      end if;
    end loop;
    return False;
  end;
end;
end Generic_Notifier;
```

The *Notify_Shutdown* procedure of the Platform body notifies the *Leg_Killed* event in the 6 channels. Then the task exploits the *Has_Pendings* entry to know if those events have been delivered. When it receives a *Leg_Killed* event, a leg controller (which is a task) will start its own shutdown.

To illustrate inheritance and type extension in Ada and to generate the events of leg position, the *Leg* class is subclassed into a *Notifying_Leg* class. This class overloads the *Evolve* method to generate the *PEP_Reached* and *AEP_Reached* events and to notify them in the appropriate channel. Fig. 8 describes the situation.

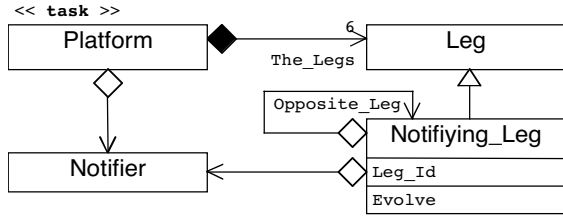


Figure 8. Leg with notifier

The call of *Evolve* from within the Platform task is a dispatching call because the type used is a class wide type. *Notifying_Leg* uses discriminants and is a child unit of *Leg*.

```

with Platform;
package Leg.Notifying_Leg is
  type Object(Leg_Id: Platform.Leg_Ids;
    Notifier: Platform.Notifier.Ref) is
    new Leg.Object with null record;
  type Class_Ref is access Object'Class;
  function Value(Leg_Id: Platform.Leg_Ids;
    Notifier: Platform.Notifier.Ref;
    X, Y: Integer; Position: Float
  ) return Class_Ref;
  procedure Evolve(O: in out Object);
end;

with Platform; use Platform;
package body Leg.Notifying_Leg is
  procedure Evolve(O: in out Object) is
    Super: Leg.Object renames Leg.Object(O);
    L: Leg_Ids renames O.Leg_Id;
    P1, P2: Float;
  begin
    P1 := O.Position;
    Leg.Evolve(Super);
    P2 := O.Position;
    if P1 < P2 and then P2 = AEP then
      O.Notifier.Send(L, AEP_Reached);
    elsif P1 > P2 and then P2 = PEP then
      O.Notifier.Send(L, PEP_Reached);
    end if;
  end;
  -- other methods
end;

```

At this stage, the platform layer is completed.

6. The control layer

A leg must move according to the following rules: 1) any leg resting on the ground moves in retraction; 2) when a leg arrives at its PEP, it must go into protraction; 3) a leg can only go into protraction if it does not compromise the static stability of the hexapod; 4) when a leg in protraction arrives at its AEP, it must go into retraction. The controller must generate this behavior permanently. Rule 3 stipulates that static stability must be maintained. The necessary condition to ensure this stability is that a leg can only

be lifted if its two neighbors are resting on the ground. Fig. 9 describes the neighborhood relation for the legs.

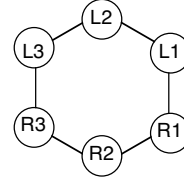


Figure 9. Neighborhood relation

As decentralized and concurrent control is to be implemented, there may be some conflict when making the decision of lifting a leg. This conflict must be solved. The problem is quite similar to the traditional problem of the dining philosophers, if a leg is considered a “philosopher” and if “to eat” means to “lift a leg”. To solve the problem, a leg that wishes to go into protraction must acquire a privilege and give it up when protraction is over. The class diagram in Fig. 10 describes the situation.

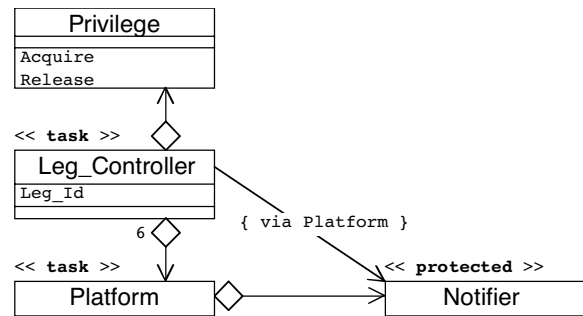


Figure 10. Controllers and privileges

The control layer consists of 6 instances of the *Leg_Controller* task type waiting for events from the channels of the notifier and generating the subsequent *Start_Retraction* or *Stop_Retraction* entry calls towards the platform. The transition diagram of the leg controller is shown in Fig. 11.

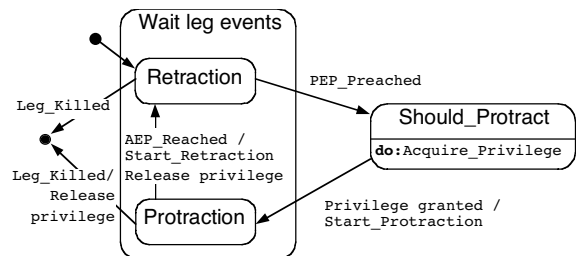


Figure 11. StateCharts of the leg controller

The specification and body of *Leg_Controller* are the following (*Is_Late* will be discussed later):

```

with Platform, Privilege ;
package Leg_Controller is
  task type Object(P: Platform.Ref;
    L: Platform.Leg_Ids;
    Protraction_Privilege: Privilege.Ref);
  type Ref is access Object;
end Leg_Controller;

with Platform; use Platform;
package body Leg_Controller is
  type States is (Retraction,
    Should_Protract, Protraction);
  task body Object is
    State: States := Retraction;
    Event: Platform.Leg_Events;
  begin
    loop
      P.Notifier.Wait(L)(Event);
      case Event is
        when Is_Late | PEP_Reached =>
          State := Should_Protract;
        when AEP_Reached =>
          State := Retraction;
          P.Start_Retraction (L);
          Protraction_Privilege.Release;
        when Leg_Killed =>
          if State = Protraction then
            Protraction_Privilege.Release;
          end if;
          exit;
        end case;
        if State = Should_Protract then
          Protraction_Privilege.Acquire;
          State := Protraction;
          P.Start_Protraction(L);
        end if;
      end loop;
    end;
end Leg_Controller;

```

Provided that the privileges are working, the controller ensures stable walking of the hexapod, whatever the speed. However, the gait is not fair and some legs may drag for a long while in PEP when waiting for a privilege. To reduce waiting time and improve the gait, other coordination mechanisms are necessary. Fig. 3 shows that, whatever the walking speed, a protraction wave runs from rear to front on both sides of the hexapod. A new control rule stipulates that the start of retraction for one leg stimulates the protraction of the preceding leg, i.e. L3 stimulates L2 and L2 stimulates L1 (it is the same for the right side). Such a (non-memorized) signal can be obtained with a protected object as shown in Fig. 12.

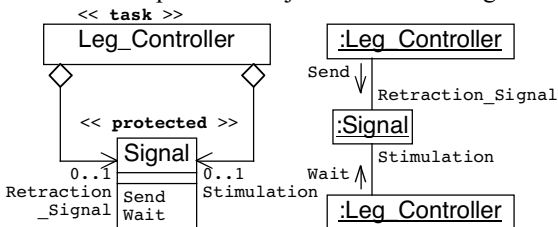


Figure 12. Stimulation signals

A leg controller must then be kept waiting for an event either from the notifier or from another leg controller, which requires a simultaneous call of two entries. This *selective entry call* is not available in Ada, but a solution can be obtained with the *asynchronous transfer of control* [13]. The following source gives the corrections of the controller.

```

package Leg_Controller is
  protected type Signal is
    procedure Send;
    entry Wait;
  private
    Arrived: Boolean := False;
  end;
  type Signal_Ref is access Signal;
  task type Object(P: Platform.Ref;
    L: Platform.Leg_Ids;
    Protraction_Privilege: Privilege.Ref);
    Stimulation: Signal_Ref ;
    Retraction_Signal: Signal_Ref);
  type Ref is access Object;
end Leg_Controller;

package body Leg_Controller is
  protected body Signal is
    procedure Send is
      begin
        Arrived := Wait'Count /= 0;
      end;
    entry Wait when Arrived is
      begin
        Arrived := False;
      end;
  end;
  task body Object is
    -- same as previous code
    loop
      if State = Retraction and then
        Stimulation /= null then
        Event := PEP_Reached;
        select
          P.Notifier.Wait(L)(Event);
        then abort
          Stimulation.Wait;
        end select;
      else
        P.Notifier.Wait(L)(Event);
      end if;
      case Event is
        -- same as previous code
        when AEP_Reached =>
          State := Retraction;
          P.Start_Retraction (L);
          Protraction_Privilege.Release;
          if Retraction_Signal /= null then
            Retraction_Signal.Send;
          end if;
        -- same as previous code
      end case;
    end loop;
  end Leg_Controller;

```

With this new mechanism, walking stabilizes rapidly in the form of tripod gait, even at low speeds. To obtain the wave gaits shown in Fig. 3 – notably slow walking ($K = 1/5$), a last resynchronization mechanism must be added. Fig. 3 shows that 2 Li-Ri legs are always in opposition of phase. In other terms,

the protraction of a leg starts in the middle of the cycle of the opposite leg. The phase is evaluated as shown in Fig. 13.

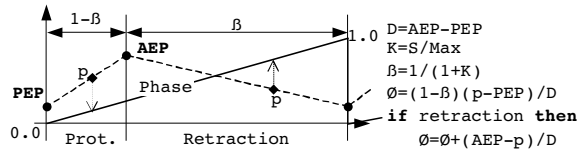


Figure 13. Phase calculation

A leg is linked to its opposite leg (Fig. 8) and a *Phase* method is added to the *Leg* class. To respect the phase opposition criterion the delay of a retracting leg is recovered by advancing its protraction. To do so, a leg compares its phase with that of the opposite leg and generates an *Is_Late* event if it is late. This event is trapped by the leg controller and processed as an event that is synonymous with *PEP_Reached*. Fig. 14 describes phase recovering.

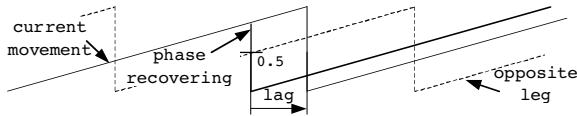


Figure 14. Phase recovering

A leg is considered to be late if its phase is greater than or equal to the phase of the opposite leg and if the phase of the opposite leg is higher than 0.5. So, *Notifying_Leg* must simply be completed with that law in order to generate the *Is_Late* event. *Is_Late* is considered to have no priority, as it is only used to resynchronize movements. It is posted using the *Send_If_Possible* method of the notifier.

At this stage, the control layer is complete and the typical wave gaits corresponding to $K = 1$, $K = 1/3$ and $K = 1/5$ are obtained. Moreover, any modification of the walking speed causes an automatic adaptation and resynchronization towards a new equitable gait.

7. Management of privileges

The coordination layer manages the privileges allocated to the different legs. It has been seen that the problem is similar to that of the dining philosophers. The problem can be tackled in a simple, centralized manner, or in a decentralized manner, which is more complex. Centralized management is possible with one protected *Privilege* object, shared between all the leg-controllers. This object acts like a kind of *Mediator* [11] to coordinate the leg controllers which do not know each other.

```

with Platform; use Platform;
package Privilege is
  type State is array (Leg_Ids)
    of Boolean;
  protected type Object is
    entry Acquire(Leg_Ids); -- family
    procedure Release(For_Leg: Leg_Ids);
  private
    Privileges: State := (others => False);
    function Can_Take_Privilege(L: Leg_Ids)
      return Boolean;
  end;
  type Ref is access Object;
end Privilege;

package body Privilege is
  function Right(L: Leg_Ids) return Leg_Ids is
  begin
    if L = Leg_Ids'Last then
      return Leg_Ids'First;
    else
      return Leg_Ids'Succ(L);
    end if;
  end;
  function Left(L: Leg_Ids) return Leg_Ids is
  begin
    if L = Leg_Ids'First then
      return Leg_Ids'Last;
    else
      return Leg_Ids'Pred(L);
    end if;
  end;
  protected body Object is
    entry Acquire(for L in Leg_Ids) when
      Can_Take_Privilege (L) is
    begin
      Privileges(L) := True;
    end;
    procedure Release (For_Leg: Leg_Ids) is
    begin
      Privileges(For_Leg) := False;
    end;
    function Can_Take_Privilege (L: Leg_Ids)
      return Boolean is
    begin
      return not (Privileges(Left(L)) or
        Privileges(Right(L)));
    end;
  end;
end Privilege;

```

As control is decentralized, it is pedagogically more interesting to study a decentralized algorithm for the allocation of privileges. In such a schema, all the synchronization is performed through passage of tokens (or messages). The present study will use part of K.M.Chandy and J.Misra's well-known algorithm [14]. In this algorithm Chandy and Misra tackle a more complex problem – the problem of the *drinking philosophers* – which is a generalization of the dining philosophers problem. Chandy and Misra describe a distributed variant of the dining philosophers problem as a first step in the solution of the drinking philosophers problem. This first part will be implemented in the present study.

The important elements of the algorithm are the following:

- When a philosopher becomes hungry, he tries to acquire the missing forks.
- When a hungry philosopher has the forks, he can eat.
- The forks are clean or dirty.
- As soon as a philosopher starts to eat, his forks become dirty.
- The forks can be used several times and so, they remain dirty.
- A request token is associated with each fork.
- A philosopher uses this token to ask his neighbor for a fork.
- Only the holder of a request token may ask his neighbor for a fork (passing of the token).
- To have the token then means that the neighbor has asked for or is in the possession of the fork.
- Before giving his fork to his neighbor, the philosopher cleans it.
- A clean fork is never given or given back. Indeed, a philosopher only asks for a fork when he is hungry. Consequently a fork is only given when the philosopher is not eating (even if he is hungry), when he has the token and when the fork is dirty.

The whole set must be initialized as follows:

- All forks are dirty.
- The tokens and forks are held by different philosophers. Moreover, for a couple of neighboring philosophers, one has a dirty fork and the other a request token.
- The precedence graph is acyclic. A philosopher is said to precede his neighbor if his neighbor has a dirty fork or if the fork is coming or if he already has a clean fork. Fig. 15 shows the initialization.

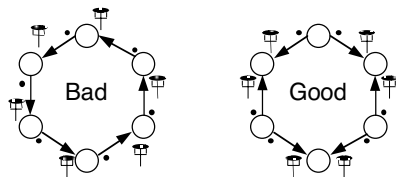


Figure 15. Initialization

The existence of a cycle may lead to a deadlock. Therefore, one of the aims of the algorithm is to always keep the precedence graph acyclic.

To apply the algorithm to the robot, the elements of the algorithm must be reformulated into the terms of the problem. A leg controller is considered a philosopher. *Retraction* corresponds to thinking, *Protraction* to eating and being hungry to *Should_Protract* (Fig. 11). A fork is replaced by a granted privilege and a dirty fork represents a privilege that has already been used. The notion of privilege is reified and each of the 6 leg controllers is linked to its own *Privilege* object. The privilege must be acquired

on the left and right sides. The scenario in Fig. 16 illustrates how privileges work.

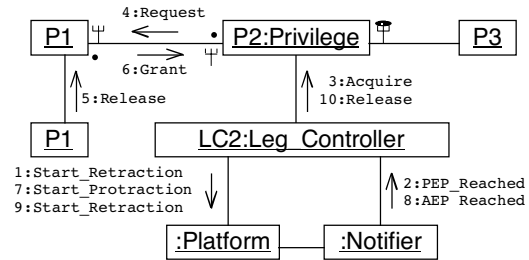


Figure 16. The Functioning of privileges

In this figure, the privilege on the right has already been acquired, used (dirty fork), but not given back. On the left, the privilege has not been acquired, which requires the sending of the *Request* message. P1 records the request, but does not give up his privilege, since it has not been used yet. When P1 has used his privilege, he will grant it (*Grant* message) to P2 who has asked for it. During all this phase, the LC2 controller is waiting. Once it is released, the controller is sure to hold the privilege and can now perform protraction, then give up the privilege at the end of protraction.

The privilege objects are shared and must be able to suspend the calling tasks. That is why protected objects become necessary. For easier management of privileges, each privilege object is assisted by 2 agents (or Brokers) that memorize the current privilege state and negotiate the privileges with the neighbors. The previous schema is thus improved (Fig. 17). The whole coordination layer acts like a ring of mediators.

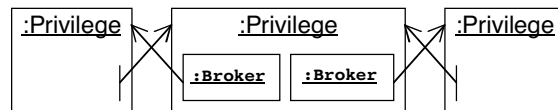


Figure 17. Chaining of brokers and privileges

The Ada specification is:

```

package Privilege is
  type Sides is (Left, Right);
  type States is (Used, Not_Granted,
                 Granted, In_Use);
  subtype Initial_States is States range
    Used..Not_Granted;
  type Object;
  type Ref is access Object;
  type Broker(Side: Sides; Initial_State:
              Initial_States) is record
    Needed      : Boolean := False;
    State       : States := Initial_State;
    Requested   : Boolean :=
      Initial_State = Not_Granted;
    Neighbour   : Privilege.Ref;
  end record;
end Privilege;

```

```

protected type Object(Initial_State:
  Initial_States) is
  entry Acquire;
  procedure Release;
  procedure Request(Side: Sides;
    Granted: out Boolean,
    Needed : out Boolean);
  procedure Grant(Side: Sides);
  procedure Link_To(Left, Right: Ref);
private
  entry Wait;
  Left_Broker: Broker(Left, Initial_State);
  Right_Broker: Broker(Right, Initial_State);
end;
end Privilege;

```

As for the state memorized in the Broker:

- *Needed* indicates that a privilege is needed, whether it has been obtained or not.

- *Requested* is the token; it is true when the neighbor has asked for a privilege, whether he has obtained it or not.

- *State* memorizes the privilege state associated with one side.

The discriminants allow correct initialization during the object construction phase. *Acquire* delegates the negotiation of privileges to the 2 brokers, then starts waiting at the *Wait* private entry. The *Request* method has 2 output parameters. Indeed, a request may be followed by an immediate allocation (*Granted* parameter). A privilege may also be given up because it has already been used while it is still needed; the *Needed* parameter encodes this fact. So, an allocation can be immediate (parameter) or postponed (*Grant* message). In the same way, a request for a privilege can occur when a privilege is lost (*Needed* parameter) or when a *Request* message is sent. This construction avoids an indirect entry call during the execution of a protected action (cf. ARM 9.5). Fig. 18 shows the finite state machine for privilege allocation.

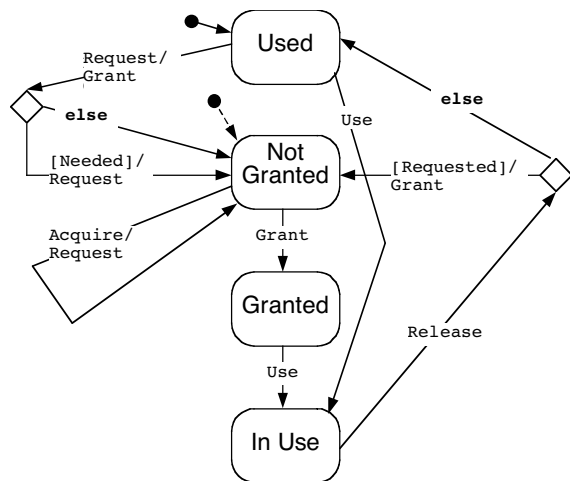


Figure 18. Privilege allocation FSM

The Privilege body is the following:

```

package body Privilege is
function Has_P(B: Broker) return Boolean is
begin
  return B.State /= Not_Granted;
end;
procedure Grant_P(B: in out Broker) is
begin
  B.State := Granted;
end;
procedure Need_P(B: in out Broker) is
  Granted, Requested: Boolean;
begin
  B.Needed := True;
  if B.State = Not_Granted then
    B.Requested := False; -- send token
    case B.Side is
      when Left =>
        B.Neighbour.Request(Right,
          Granted, Requested);
      when Right =>
        B.Neighbour.Request(Left,
          Granted, Requested);
    end case;
    if Granted then Grant_P(B); end if;
    if Requested then
      B.Requested := True;
    end if;
  end if;
end;
end;
procedure Use_P(B: in out Broker) is
begin
  B.State := In_Use;
end;
procedure Release_P (B: in out Broker) is
begin
  B.Needed := False;
  if B.Requested then
    case B.Side is
      when Left =>
        B.Neighbour.Grant(Right);
      when Right =>
        B.Neighbour.Grant(Left);
    end case;
    B.State := Not_Granted;
  else
    B.State := Used;
  end if;
end;
end;
procedure Request_P (B: in out Broker;
  Granted: out Boolean;
  Needed: out Boolean) is
begin
  Granted := False;
  Needed := B.Needed;
  B.Requested := True;
  if B.State = Used then
    B.State := Not_Granted;
    Granted := True;
    if Needed then
      B.Requested := False; -- token sent
    end if;
  end if;
end;
end;

protected body Object is
  procedure Link_To(Left, Right: Ref) is
  begin
    Left_Broker := Left;
    Right_Broker := Right;
  end;
  entry Acquire when True is
  begin

```

```

    Need_P(Left_Broker);
    Need_P(Right_Broker);
    request Wait with abort;
end;
entry Wait when Has_P(Left_Broker)
    and Has_P(Right_Broker) is
begin
    Use_P(Left_Broker);
    Use_P(Right_Broker);
end;
procedure Release is
begin
    Release_P(Left_Broker);
    Release_P(Right_Broker);
end;
procedure Request (Side: Sides;
    Granted: out Boolean,
    Needed : out Boolean) is
begin
    case Side is
    when Left =>
        Request_P(Left_Broker, Granted,
        Needed);
    when Right =>
        Request_P(Right_Broker, Granted,
        Needed);
    end case;
end;
procedure Grant (Side: Sides) is
begin
    case Side is
    when Left => Grant_P(Left_Broker);
    when Right => Grant_P(Right_Broker);
    end case;
end;
end Privilege;

```

8. Speed control

By continuously changing the reference speed, and despite the automatic transition between the different walking gaits when the reference speed changes, some of the legs may drag for a while in the PEP position, waiting for a privilege. To avoid such leg dragging, the speed of the robot is controlled and reduced until the stretched legs can start their protraction. The control law is simple: *If one or the other leg of the robot stretches too far, the speed must be reduced.* This law can be implemented using a small controller based on fuzzy logic [15, 16]. The source of this part will not be described in detail, but the general principle is the following: an *Is_Stretched* fuzzy predicate is added to the *Leg* class and a *Legs_Are_Stretched* fuzzy predicate is added to the *Platform* task. These predicates correspond to the fuzzification of the position of the legs. A small kinematic margin is provided for the PEP position. Thus, a leg can continue to move beyond PEP, but starts to stretch; *Is_Stretched* expresses this fact. The fuzzy predicate *Legs_Are_Stretched* is the “dilatation” of the “or” between the 6 *Is_Stretched* predicates of the legs. The fuzzy “or” operator is typically the maximum of the 2

operands and the “dilatation” operator is typically obtained using the square root function. Fig. 19 shows the fuzzy membership functions.

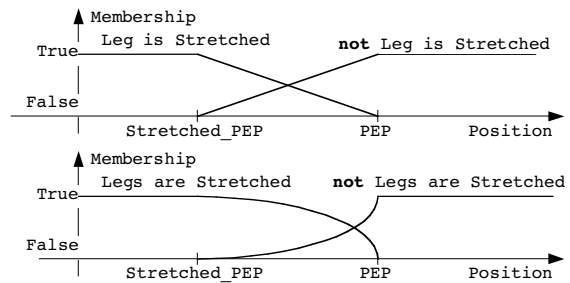


Figure 19. Fuzzy membership functions

The speed ratio is adjusted through the defuzzification of the control law, according to:

$$K = K_{reference} * (not\ Legs_Are_Stretched)$$

This control law is implemented through the addition of a periodic controller task which observes the stretching of the legs and adjusts the speed of the platform according to the desired speed.

Finally, all the objects and tasks that have been described are assembled using a *Hexapod* class.

9. Conclusion

This paper has described an example of coordination and synchronization of the leg movements of a hexapod robot, using several more or less redundant mechanisms. The whole set leads to a graph of objects which ensure an adaptive behavior. Fig. 20 shows the main points of this object graph.

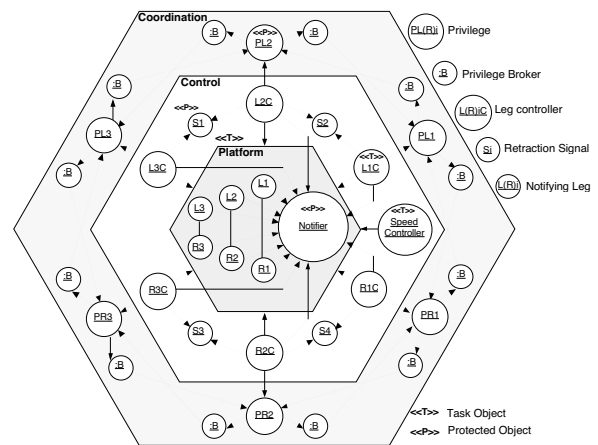


Figure 20. Complete object graph

This is a very comprehensive project insofar as it deals with most of the Ada language constructions, focuses on architectural aspects and takes great account of concurrent programming. It is complex enough to require an appropriate architecture. Numerous extensions can be imagined, including a more accurate leg model, a 3D graphic rendering using a binding to OpenGL, an off-line graphic rendering of walking sequences (e.g. using Pov-Ray, Fig. 21 and [8]), embedding the software on a real platform, the use of a distributed platform, more sophisticated movements which allow rotation, navigation and planning of trajectories, etc.

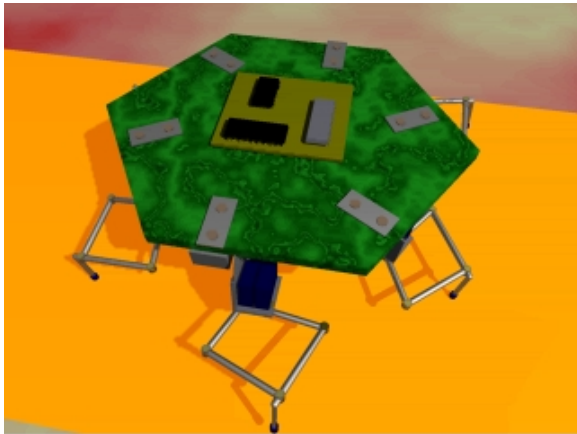


Figure 21. Pov-Ray rendering

The paper is also an incentive to explore non-traditional fields of software engineering. There is, indeed, much to be learnt from non-technical examples (in biology, for instance) as regards synchronization or coordination patterns, or as regards complex behaviors. In this respect, mobile robotics is an ideal field to learn how to integrate bio-inspired algorithms and advanced software technologies.

References

- [1] G.Dudek and M.Jenkin. Computational Principles of Mobile Robotics. *Cambridge University Press*. 2000.
- [2] P. Balbastre, S. Terrasa, J. Villa, and A. Crespo. Experiences using Ada in a Real-Time and Distributed Laboratory. *Ada-Letters*. XIX(3) :145-155. 1999.
- [3] The Walking Machine Catalog
http://www.fzi.de/ipt/WMC/preface/walking_machines_katalog.html
- [4] The Climbing and Walking Robots Home Page
<http://www.uwe.ac.uk/clawar/home.htm>
- [5] C. Ferrel. Robust agent control of an autonomous robot with many sensors and actuators. *Master's thesis, MIT, Dep. of Electrical Engineering and Computer Science*. 1993.
- [6] U. Schmucker, A. Schneider, and T. Ihme. Six-Legged robot for service operations. In *Proceedings of*

EUROBOT'96. Los Alamitos: IEEE Computer Society Press, p 135-142, 1996.

- [7] S. Galt, B. L. Luk, S. Chen, R. H. Istepanian, D. S. Cooke, and N. D. Hewer. Intelligent walking gait generation for legged robots. In *Proceedings of 2nd International Conference on Climbing and Walking Robots*, 605-613. 1999.
- [8] <http://www.lsi.crespim.uha.fr/>
- [9] R.D.Beer, R.D. Quinn, H.J. Chiel, and R.E. Ritzmann. Biologically Inspired Approaches to Robotics. *Communications of the ACM*. 40(3) :31-38, 1997.
- [10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, A Systems of Patterns*. John Wiley & Sons. 1996
- [11] E.Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design-Patterns – Elements of Reusable Object Oriented Software*. Addison-Wesley. 1995.
- [12] B.P. Douglass. *Real-Time UML, Developing Efficient Objects for Embedded Systems*. Addison Wesley. 1998.
- [13] A. Burns and A. Wellings. *Concurrency in Ada*. Cambridge University Press. 1995.
- [14] K.M. Chandi and J. Misra. The Drinking Philosophers Problem. *ACM Transactions on Programming Language and Systems*, 6(4) :632-646, 1984.
- [15] J.-S.R. Jang, C.-T. Sun, and E. Mizutani. *Neuro-Fuzzy and Soft Computing*. Prentice Hall. 1997.
- [16] N.Gaertner, and B.Thirion. A Framework for Fuzzy Knowledge Based Control. *Software Practice and Experience*. 30 :1-15. 2000.