Bakkelaureatsarbeit im Rahmen des Bakkelaureatsseminars "Concurrent Programming"

Concurrent Programming in Ada

Andreas Rottmann
Department of Computer Science
University of Salzburg, Austria
arott@cosy.sbg.ac.at

24. Jänner 2003

Contents

1 Introduction **3**

1.1 Operating systems defined versus language defined concurrency 3

1.2 Program Entities 4

1.3 A bit of history 4

2 Clocks and Time **4**

2.1 The `delay` statement 5

3 Tasks **5**

3.1 Task hierarchies 6

3.2 Task Identification 7

4 The Rendezvous **8**

4.1 Primes by sieve example 8

4.2 Synchronization without communication 10

4.3 Private entries 10

4.4 Exceptions and the rendezvous 10

5 The `select` statement and the rendezvous **11**

5.1 Selective accept 11

5.1.1 Waiting for more than one rendezvous 11

5.2 Guarded alternatives 12

5.3 Delay alternative 12

5.4 The else part 13

5.5 The terminate alternative 14

5.5.1 Primes by sieve, revisited 14

6 Protected Objects **16**

6.1 Protected Object Entries 16

6.2 Protected Objects as Building Blocks 17

6.2.1 Semaphores 17
6.2.2 Bounded Buffers 17

1 Introduction

A modern computer consists of one or more CPUs and many I/O devices, all operating in parallel. In programming embedded systems, one thus must deal with this inherent parallelism. A language suited for real-time use must therefore provide some facility for multi-programming. This can either be achieved by using a standard interface to a multi-processing operating system (e.g. POSIX threads) or by allowing to express concurrency in the language itself.

Ada provides for the direct programming of parallel activities. Within an Ada program there may be a number of *tasks*, each having its own thread of control. Languages whose conceptual framework includes parallelism are called *concurrent programming languages*. Besides Ada, there are several other such languages including Modula, CHILL and Mesa.

1.1 Operating systems defined versus language defined concurrency

There has been (and still is) a lot of debate whether it is appropriate to implement concurrency as a language feature; Here are some arguments for including concurrency as a language feature:

- It leads to more readable and maintainable programs
- There are many types of operating systems; defining the concurrency in the language increases portability.
- An embedded computer may not have a operating system

Some arguments against:

- Different languages use different concurrency models; composing programs of different languages is easier if they all use the operating system model

- A languages model may not map well onto the operating system model, making it difficult or inefficient to implement
- Operating system standards emerge, making programs more portable

1.2 Program Entities

In a concurrent program, there are two basic types of entities: *active* and *passive*. Active entities have a own thread of control and are expressed in Ada by tasks. Passive entities are reactive: They can act upon requests (calls), but have no own thread of control. Passive entities are expressed in Ada by *protected objects*.

1.3 A bit of history

Ada 83 was the first International standard to include concurrency as an intrinsic language feature. This was a big step, and not without problems. Ada 83, while being useful for a broad range of concurrent applications, had some deficiencies, most notably the lack of a abstraction for shared data access.

These deficiencies were addressed with Ada 95, which provides the protected object abstraction for concurrent access to shared data. Additionally, Ada 95 introduced *preference control* with the **`requeue`** statement, better timing (**`delay until`**).

Ada 95 also introduced OOP (object oriented programming) and is structured differently: the standard defines a core language, plus (optional) annexes.

2 Clocks and Time

Ada provides access to the systems time via two packages: the compulsory `Ada.Calendar` package, providing an abstraction of "wall clock time" and `Ada.Real_Time`, defined in the Real-Time-Systems Annex, providing a monotonic (i.e. non-decreasing) regular clock.

2.1 The `delay` statement

The delay statement in Ada has two forms; the first is used to delay for a time span, the second one can be used to delay *until* an absolute time has been reached. The use is illustrated by the following listing:

```
-- delay for (at least) 10 seconds
delay 10.0

-- delay until 10 seconds after the start of First_Action
Start := Clock;
First_Action;
delay until Start + 10.0;
Second_Action;

-- Note that this is not the same as:
Start := Clock;
First_Action;
delay (Start + 10.0) - Clock;
Second_Action;
```

Important to understand is that **`delay until`** can *not* be "implemented" by the use of **`delay`**, since the **`delay`** statement is interruptible, so if the timeout is calculated and the task is suspended thereafter, the calculated time out will take effect after the task is resumed, which will cause the task to delay longer than desired.

3 Tasks

As already mentioned, Ada provides *tasks* as abstraction of concurrently executing parts of the program. An Ada compiler is free to implement them however it likes, however, a task conceptually very similar to a thread. A task is introduced by declaring a *task type*, consisting of an identifier, a discriminant, a visible and a private part as well as a task body. It is also possible to use an *anonymous task type* for convenience. A basic syntax template is shown in the following listing:

```
-- task type
task type My_Task(Discriminant1 : Disc1_Type; D2 : Disc2_Type) is
```

```
   -- visible part
   entry Action1(Parameter : Par_Type);
   entry Action2(Parameter : Par_Type);
private
   --private part
   ...;
end My_Task;

-- task body
task body My_Task is
begin
   -- sequence of statements
   ...;
end My_Task;

-- declaration
T : My_Task;

-- anonymous tasks
task A_Task;

task body A_Task is
begin
   ...;
end A_Task;
```

The visible and private parts both contain entry and representation clauses. Representation clauses are used only for interrupt handling and are not covered in this paper. The entries define the interface to the objects of the task type, i.e. how to communicate with them.

An object of task type consists of the entries, the values of the discriminants and the execution of the task body.

3.1 Task hierarchies

As Ada is a structured language, consisting of blocks that may be nested in each other, also tasks may be nested. Before taking a look at the semantics of nested tasks, we need some terminology on the lifetime of a task. A task is *activated* before it starts executing and *finalized* before it is terminated.

The activation of a task happens after its body's **begin** statement, but before any statements following that **begin** are executed. Of special importance here is that

the declarative part (if present) has been elaborated before the activation. After its activation, these body's statements are executed. If the body is done executing, either by reaching its **end** statement, or by having thrown an unhandled exception, the task is finalized. This basically means all the objects declared for the task body are finalized.

Now with task hierarchies, there are additional rules: Before a task can be activated, all of its children must have been activated, because all errors in during the activation must be reported to the parent task.

To illustrate this, consider this example:

```
task Parent; -- parent task, using anonymous task type

task body Parent is

   Child1 : A_Task_Type;
   Child2 : Another_Task_Type;

begin
   -- the next line will be executed *after* Child1 and Child2 have
   --  been activated
   Some_Statement;
end;
```

As the task `Parent` is declared, its declarative part is elaborated. Here, two tasks are declared. Thus, before the `Parent` tasks' body is executed, it has to wait for `Child1` and `Child2` to become activated before itself is activated. After the body of `Parent` has been executed (before the **end** statement) it waits for the child tasks to terminate.

3.2 Task Identification

For some applications it might be necessary to associate tasks with unique identifier, for example, a server might want to keep some state information about each client task and process requests from its client based on this information. Ada's systems programming annex provides the `Ada.Task_Identification` package.

4 The Rendezvous

A synchronization point where two concurrent entities (tasks) "meet" is called a rendezvous. In Ada, a rendezvous is accomplished by task entries and the **accept** statement. A task entry declares a rendezvous point, and the **accept** statement specifies the actions to be performed when the entry is called.

A accept statement has the following form:

```
accept Entry_Name(Family_Index)(P : Parameter) do
   -- sequence of statements
exception
   --exception handling part
end Entry_Name;
```

The `Family_Index` part is for entry families, which provide an easy way of declaring a bunch of similar entries. For further information, see [1], Chap. 5.4. The formal part (i.e. parameter types) must match the corresponding entry.

4.1 Primes by sieve example

The next listing illustrates the use of task entries and the **accept** statement. It is an implementation of the "primes by sieve" algorithm (otherwise known as the Sieve of Eratosthenes). The main program generates a continuous stream of odd integers. These are passed down a pipeline of `Sieve` tasks. Each `Sieve` task receives a stream of integers, the first one being a prime number, which it keeps in a local variable `Prime`. The task then processes the rest of the numbers, checking each one if it can be divided exactly by `Prime`. If it can, it is thrown away; if it cannot, it is passed down the pipeline to the next task.

The implementation uses dynamic task creation via Ada's **new** operator. The actual task creation must be done in the procedure `Get_New_Sieve`, since a task cannot create a new task of its own type with **new**. Note that this implementation does not terminate correctly; a modified version with correct termination is given in Sec. 5.5.1, after the **terminate** statement has been introduced.

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Sieve1 is
```

```
task type Sieve is
   entry Pass_On(Int : Integer);
end Sieve;

type Sieve_Ptr is access Sieve;

function Get_New_Sieve return Sieve_Ptr is
begin
   return new Sieve;
end Get_New_Sieve;

task body Sieve is
   Next_Sieve : Sieve_Ptr;
   Prime, Num : Natural;

begin
   accept Pass_On(Int : Integer) do
      Prime := Int;
   end Pass_On;
   Put(Prime); -- output prime number
   New_Line;
   loop
      accept Pass_On(Int : Integer) do
         Num := Int;
      end Pass_On;
      exit when Num rem Prime /= 0;
   end loop;

   -- pass the number on to a new sieve task
   Next_Sieve := Get_New_Sieve;
   Next_Sieve.Pass_On(Num);
   loop
      accept Pass_On(Int : Integer) do
         Num := Int;
      end Pass_On;
      if Num rem Prime /= 0 then
         Next_Sieve.Pass_On(Num);
      end if;
   end loop;
end Sieve;

Limit : constant Positive := 1000;
```

```
   Num : Positive;
   S : Sieve_Ptr := Get_New_Sieve;

begin
   Num := 3;
   while Num < Limit loop
      S.Pass_On(Num);
      Num := Num + 1;
   end loop;
end Sieve1;
```

4.2 Synchronization without communication

Where the sequence of statements in an **accept** statement is precisely null, then there is nothing to do inside the rendezvous, and the **accept** statement can be terminated with an semicolon after the formal part, like this:

```
accept Synchronization_Point;
```

4.3 Private entries

In contrast to the examples seen so far, a task may also have private entries. There are several reasons why one might wish to do this:

1. The task has several tasks declared internally; these have access to the private entries.
2. The entry is used internally by the task for requeueing purposes. For a information on the **requeue** statement, see [1].
3. The task is an interrupt entry, and the entry should not be called by any software tasks. For a discussion of interrupt handling, see [1], Chapter 11.

4.4 Exceptions and the rendezvous

Exceptions raised in an **accept** statement and not handled there, will cause a termination of the rendezvous and the exception is re-raised in *both* the server (called) and client (caller) tasks.

5 The `select` statement and the rendezvous

In the basic rendezvous model discussed in the last section, a server task can only wait for a single rendezvous at any time. It also has to wait for a client to actually arrive at the rendezvous point once it has issued the **accept** statement. The same also holds for the client. Now we will discuss how this restrictions can be lifted the use of the **select** statement.

The select statement has four forms: selective accept, conditional entry call, timed entry call and asynchronous select. In this paper, only the first form is discussed; a more comprehensive discussion can be found in [1], Chapter 6.

5.1 Selective accept

The selective accept form of the **select** statement allows a server task to wait for more than a single rendezvous at a time, time out if no rendezvous happened in specified time period, check if a rendezvous is immediately possible and to terminate if no clients can possibly call its entries.

5.1.1 Waiting for more than one rendezvous

Consider the following example:

```
task type Server is
   entry Service1(...);
   entry Service2(...);
end Server;

task body Server is
begin
   loop
      select
         accept Service1(...) do
            -- handle request for Service1
         end Service1;
      or
         accept Service2(...) do
            -- handle request for Service2
         end Service2;
      end select;
   end loop;
```

```
end Server;
```

On each execution of the loop, one of the two **accept** statements will be executed. Without the **select** statement, the server could only wait for one of the services at a time. This is clearly not sensible in many cases — the server could for instance only handle requests for `Service2` after a client has used `Service1` and would not service request for `Service2` at all if no client ever requested `Service1`.

5.2 Guarded alternatives

Selective accept alternatives can be guarded. The guard is a boolean expression which is evaluated when the select statement is executed. If the guard evaluates to true, the alternative is considered, else it is ignored, even if clients are waiting on the associated entry. A guarded accept alternative looks like this:

```
select
   when Boolean_Expression =>
      accept Service1(...) do
         -- handle request for Service1
      end Service1;
or
   ...
end select;
```

5.3 Delay alternative

Often it is necessary to time-out when waiting for a rendezvous; for instance a server might have to do some periodic task, unless otherwise requested. The next listing shows the structure for such a server.

```
task body Server is

   Period := Time_Span := To_Time_Span(10.0);
   Next_Cycle : Time := Clock + Period;

begin
   loop
```

```
      -- do periodic maintainance
      select
         accept Do_Maintaince_Now;
      or
         delay until Next_Cycle;
         Next_Cycle := Next_Cycle + Period;
      end select;
   end loop;
end Server;
```

Delay alternatives may be guarded, too.

5.4 The else part

If a server task only wants to engage in a rendezvous if there are entry calls on it, then the **else** part of the **select** statement can be used:

```
task body Server is
begin
   loop
      select
         accept Service1(...) do
            -- handle request for Service1
         end Service1;
      else
         -- do something in reaction that Service1 was not requested
         ...;
      end select;
      accept Service2(...) do
         -- handle request for Service2
      end Service2;
   end loop;
end Server;
```

With this code, if any client has called `Service1`, it will be served before the next client is served `Service2`. However, when no client was present, the server will block until `Service2` has been requested.

5.5 The terminate alternative

In general, server tasks need only exist while there are clients that can possibly call their entries. However, since a server generally doesn't know the identities of its clients, it is difficult for it to decide when to terminate. The condition for termination could be explicitly expressed within the programs logic, for example by using a special entry that, when invoked, causes the server to terminate. But since this is a common situation, Ada provides a special select alternative: the terminate alternative consists only of the single statement **terminate**, which can be guarded.

A server task suspended at a **select** statement with an open terminate alternative will become completed, if they are no task that can possibly call its entries (for a more detailed discussion how this is determined, see [1]).

5.5.1 Primes by sieve, revisited

Now that the terminate alternative has been discussed, a correctly terminating version of the "primes by sieve" example can be formulated:

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Sieve2 is

   task type Sieve is
      entry Pass_On(Int : Integer);
   end Sieve;

   type Sieve_Ptr is access Sieve;

   function Get_New_Sieve return Sieve_Ptr is
   begin
      return new Sieve;
   end Get_New_Sieve;

   task body Sieve is
      Next_Sieve : Sieve_Ptr;
      Prime, Num : Natural;

   begin
      accept Pass_On(Int : Integer) do
         Prime := Int;
```

```
      end Pass_On;
      Put(Prime); -- output prime number
      New_Line;
      loop
         select
            accept Pass_On(Int : Integer) do
               Num := Int;
            end Pass_On;
         or
            terminate;
         end select;
         exit when Num rem Prime /= 0;
      end loop;

      -- pass the number on to a new sieve task
      Next_Sieve := Get_New_Sieve;
      Next_Sieve.Pass_On(Num);
      loop
         select
            accept Pass_On(Int : Integer) do
               Num := Int;
            end Pass_On;
         or
            terminate;
         end select;
         if Num rem Prime /= 0 then
            Next_Sieve.Pass_On(Num);
         end if;
      end loop;
   end Sieve;

   Limit : constant Positive := 1000;
   Num : Positive;
   S : Sieve_Ptr := Get_New_Sieve;

begin
   Num := 3;
   while Num < Limit loop
      S.Pass_On(Num);
      Num := Num + 1;
   end loop;
end Sieve2;
```

The only changes to the program were to surround the second and third **accept** with a **select** with a terminate alternative.

6 Protected Objects

The two essential requirements for sharing resources between concurrent entities are *mutual exclusion* (ensuring only one task accesses a resource at a time) and *condition synchronization* (waiting for some condition) can be expressed in Ada with resources encapsulated in server tasks and using the rendezvous. Since Ada 95, however, the language provides a more lightweight and efficient mechanism.

A protected object in Ada encapsulates data items and allows access to them only via protected subprograms or protected entries. The language guarantees that these subprograms and entries will be executed in a manner that ensures the data is updated under mutual exclusion. Thus, they are similar to monitors (Hoare, 1974) found in previous concurrent programming languages.

A protected unit may be declared either as a type or a single object:

```
-- A protected type
protected type A_Protected_Type is
   procedure Operation1(P : Parameter);
   entry Entry1(P : Parameter);
private
   -- place here private operations and items here
end A_Protected_Type;
```

6.1 Protected Object Entries

The subprograms (functions and procedures) declared for a protected object are simply guaranteed to be executed in mutual exclusion. However, there may be more than one function call on a protected object at a time, since functions are not allowed to alter the state of the object.

A protected entry, however, is guarded by a boolean expression (called a barrier) inside the body of the protected object. If the barrier evaluates to false when an entry call is made, the calling task is suspended until the barrier evaluates to true and there are no other task currently active in the protected object.

Protected entries provide a means to achieve condition synchronization.

6.2 Protected Objects as Building Blocks

Protected objects can be used to easily build several well-known synchronization and communication primitives, such as semaphores, signals, and bounded buffers. In the following, implementations of a semaphore and a bounded buffer are given, for more examples, see [1].

6.2.1 Semaphores

```
protected type Semaphore(Initial : Natural := 0) is
   entry Wait; -- P operation
   procedure Signal; -- V operation
private
   Value : Natural := Initial;
end Semaphore;

protected body Semaphore is
   entry Wait when Value > 0 is
   begin
      Value := Value - 1;
   end Wait;

   procedure Signal is
   begin
      Value := Value + 1;
   end Signal;
end Semaphore;
```

This implementation is really straightforward; the semantics of the semaphore operations can be directly implemented with an entry `Wait`, which can only be entered when the semaphore value is greater than zero, and a procedure `Signal` that increases the value. So when one or more callers are blocked in `Wait`, and another task executes `Signal`, one of the tasks is allowed into wait after the task calling `Signal` has left the procedure.

6.2.2 Bounded Buffers

```
Buffer_Size : constant Integer := 10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0..Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is
   entry Get(Item : out Data_Item);
   entry Put(Item : in Data_Item);
private
   First : Index := Index'First;
   Last : Index := Index'Last;
   Number_In_Buffer : Count := 0;
   Buf : Buffer;
end Bounded_Buffer;

protected body Bounded_Buffer is
   entry Get(Item : out Data_Item) when Number_In_Buffer /= 0 is
   begin
      Item := Buf(First);
      First := First + 1;
      Number_In_Buffer := Number_In_Buffer - 1;
   end Get;

   entry Put(Item : in Data_Item) when Number_In_Buffer /= Buffer_Size is
   begin
      Last := Last + 1;
      Buf(Last) := Item;
      Number_In_Buffer := Number_In_Buffer + 1;
   end Put;
end Bounded_Buffer;
```

Here, there are two entries for the protected objects, which can only be entered when the buffer is not empty (`Number_In_Buffer /= 0`), for `Get`, or when the buffer is not full (`Number_In_Buffer /= Buffer_Size`), for `Put`. This is exactly what one would expect for a bounded buffer implementation.

References

[1] Burns, Wellings: Concurrency in Ada

[2] The Ada 95 Reference Manual