Ada: A concurrent programming language with real-time support

IL2206 Embedded Systems

Ingo Sander

KTH Royal Institute of Technology
Stockholm, Sweden
ingo@kth.se

Outline

Outline

History

Ada is the result of an initiative of the US Department of Defense (DoD) in order to develop a language for embedded and real-time systems

- Hundreds of programming languages were used in 1983 by DoD
- Less than forty were used in 1996 by DoD

Several standardised versions of Ada exist

- ANSI standard in 1983 (Ada 83)
- ANSI/ISO standard in 1995 (Ada 95)
- ISO standard in 2005 (Ada 2005)
- ISO standard in 2012 (Ada 2012)

Features

- strong typing
- modularity mechanisms (packages)
- run-time checking
- exception handling
- generics
- object-oriented
- parallel processing (tasks)
- real-time

Application Area

Ada's main application area is the area of safety-critical systems.

- avionics
- military systems
- space applications

Thus there exist a large number of both compile-time and run-time checks to allow to design very large and robust software systems.

Documentation

The Ada standard is freely available from

http://www.ada-auth.org/standards/ada12.html

The following documents can be very useful for the programmer

- Ada 2012 Language Reference Manual (LRM)
- Annotated Ada 2012 Language Reference Manual (AARM)
- Ada 2012 Rationale

Outline

Structure of an Ada Program

An Ada program consists of several parts.

Packages Ada uses packages as modularity mechanism. A package consists of a

- package specification defining the interface of the package
- package body defining the implementation of the package

Main Program The main program can use functions, objects or data types defined in packages.

Specification of a Package (greetings.ads)

```
1  package Greetings is
2     procedure Hello;
3     procedure Goodbye;
4  end Greetings;
```

The package specification defines the interface of the package. Package specification files shall have the suffix ads.

Body of a Package (greetings.adb)

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  package body Greetings is
4     procedure Hello is
5     begin
6        Put_Line ("Hello WORLD!");
7     end Hello;
8
9     procedure Goodbye is
10    begin
11       Put_Line ("Goodbye WORLD!");
12    end Goodbye;
13 end Greetings;
```

The package body defines the details of the implementation of the package. Package body files and main programs shall be saved with the suffix adb.

Main Program (gmain.adb)

```
1  with Greetings;  -- "include Package 'Greetings'"
2
3  procedure Gmain is
4  begin
5     Greetings.Hello;
6     Greetings.Goodbye;
7  end Gmain;
```

Outline

GNAT

GNAT is a free compiler and software development toolset for the full Ada programming language. It is integrated into the gcc compiler system.

```
1 $ gcc -c gmain.adb
2 $ gcc -c greetings.adb
3 $ gnatbind gmain
4 $ gnatlink gmain
```

or

```
1 $ gnatmake gmain.adb
```

Outline

Tasks

Support for tasks is directly build into the Ada language.

Parallel activities are defined by means of tasks. A task consists of a specification and a body.

```
1  task type T is    -- Specification
2     ...
3  end T;
4
5  task body of T    -- Body
6     ...
7  end T;
```

The exact scheduling policy can be defined using the real-time annex.

Task Example

```
1  procedure SimpleTasks is
2     N : Integer := 0;
3     task type Simple;
4     T1, T2 : Simple; -- two tasks defined, both
5                      -- have the same body
6     task body Simple is
7     begin
8        for I in 1..20 loop
9           N := N + 1;
10       end loop;
11    end Simple;
12 begin
13    delay 2.0; -- Wait 2 seconds, so that
14               -- both tasks have completed
15    Put("N = " & Integer'Image(N));
16 end SimpleTasks;
```


Quick Test

```
1  with Ada.Text_IO;
2  use Ada.Text_IO;
3
4  procedure Concurrency1 is
5     N : Integer := 0;
6     task Task1;
7     task body Task1 is
8     begin
9        for I in 1..2 loop
10          N := N + 1;
11       end loop ;
12    end Task1;
```

```
1     task Task2;
2     task body Task2 is
3     begin
4        for I in 1..2 loop
5           N := N * 2;
6        end loop ;
7     end Task2;
8  begin
9     delay 1.0; -- Wait 1 sec
10    Put_Line("N = " &
11       Integer'Image ( N ));
12 end Concurrency1;
```

What is the sum of all possible results?

1) 10; 2) 20; 3) 23; 4) Something Else

Outline

Protected Objects

The concept of protected object is implemented in Ada.

- The specification provides the access protocol (entries, procedures)
- The body gives the details of the implementation

```
1  protected Object is -- Specification
2     -- Access Protocol
3  end Object;
4
5  protected body Object is -- Body
6     -- Implementation details
7  end Object;
```

A Box with Money...

Assume the following situation:

- There is a box, which is used to store coins
- People take money from the box
- People put money into the box

How can we model such a system?

How about a protected object?

Let's model the system using a protected object that implements a shared variable.

- We would like to have the following access methods:
  - Add, Subtract, Read and Write

How about a protected object?

Let's model the system using a protected object that implements a shared variable.

- We would like to have the following access methods:
  - Add, Subtract, Read and Write
- The protected object shall ensure that only one of the methods Add, Subtract and Write is executed and finished before another of these methods is started
- But several readers can access the protected object simultaneously

Specification of Shared_Variable

```
1  protected type Shared_Variable(Initial_Value : Integer)
2  is
3     procedure Add(Number : Integer);
4     procedure Subtract(Number : Integer);
5     procedure Write(Number : Integer);
6     function Read return Integer;
7  private
8     X : Integer := Initial_Value;
9  end Shared_Variable;
```

Behaviour

- A protected procedure provides mutually exclusive read/write access to the data encapsulated.
- A protected function provides concurrent read-only access to the encapsulated data.

Body of Shared_Variable

```
1  protected body Shared_Variable is
2     procedure Add(Number : Integer) is
3     begin
4        X := X + Number;
5     end Add;
6     ...
7     function Read return Integer is
8     begin
9        return X;
10    end Read;
11 end Shared_Variable;
```


Access to a protected object

A protected object is accessed using 'dot'-notation.

```
1  procedure Box_App is
2     -- Box includes initially 2 coins
3     Box : Shared_Variable(2);
4
5     task type Put_Money_Task;
6
7     task body Put_Money_Task is
8     begin
9        for I in 1..5 loop
10          Box.Add(1);
11       end loop;
12    end Put_Money_Task;
13    ...
```

However, something is missing...

The model of the shared variable does not ensure that

- coins cannot be taken, if the box is empty
- coins cannot be put into the box, if the box is full

Condition Synchronisation

Instead of a protected procedure a protected entry, which is guarded by a Boolean expression can be used.

```
1  protected type Shared_Variable(Initial_Value : Integer;
2                                 Min_Value : Integer;
3                                 Max_Value : Integer) is
4     entry Add(Number : Integer);
5     entry Subtract(Number : Integer);
6     procedure Write(Number : Integer);
7     function Read return Integer;
8  private
9     X : Integer := Initial_Value;
10    Max : Integer := Max_Value;
11    Min : Integer := Min_Value;
12 end Shared_Variable;
```

Condition Synchronisation

The statement `when` is used to implement the guard.

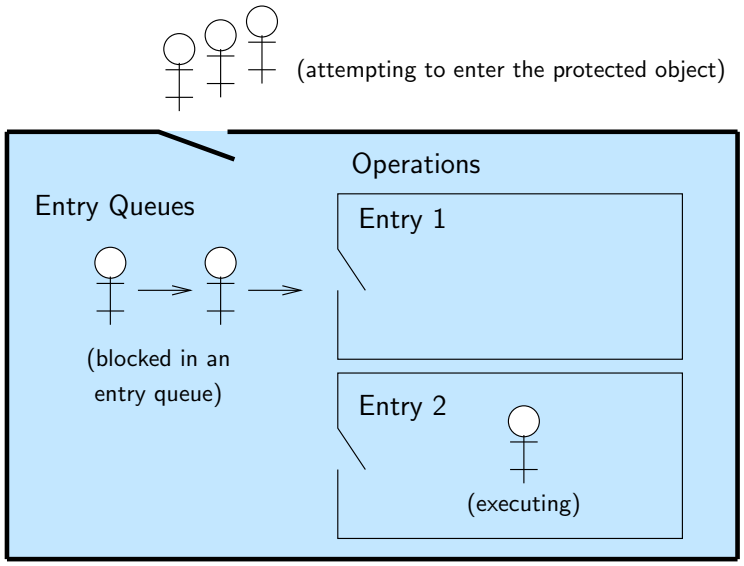
```
1  entry Add(Number : Integer)
2     when X < Max is -- Guard
3  begin
4     X := X + Number;
5  end Add;
```


Condition Synchronisation

Behaviour - Protected Entry

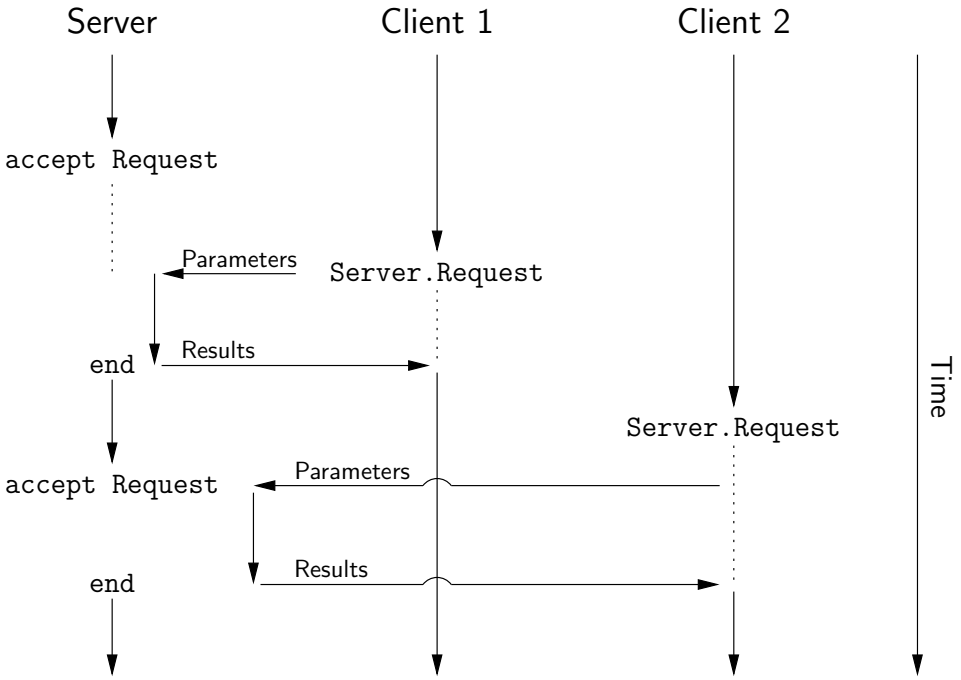
- A protected entry is similar to a protected procedure in that it is guaranteed to execute in mutual exclusion and has read/write access to the encapsulated data.
- A protected entry is guarded by a Boolean expression (called a barrier) inside the body of the protected object.
- If this barrier evaluates to false when the entry call is made, the calling task is suspended until the barrier evaluated to true and no other tasks are currently active in the protected object.

Illustration of Protected Objects



Calls that have already passed the outer shell have preference over calls that are outside the protected object.

Communication and Synchronisation during a Rendezvous



Rendezvous (1/2)

Ada implements the rendezvous mechanism. Tasks can communicate with each other directly by sending messages to each other.

The calling task calls an entry in the called task. The entry is specified as follows.

```
1  task CalledTask is
2     entry E(...);
3  end CalledTask;
```

Each entry has an associated queue of tasks waiting to call the entry. Usually this queue is processed in a first-in-first-out manner.

Rendezvous (2/2)

The calling task calls the entry in the called task

```
1  CalledTask.E(...)
```

Each entry is accompanied by an accept-statement in the body of the called task, which specifies the code that shall be executed during the rendezvous.

```
1  accept E(...) do
2    ...  -- sequence of statements
3  end E;
```

Guards (when) and select-statements can be used to express more complex conditions for a rendezvous.

Rendezvous: Box with Money

The box is modelled as a server task, which contains a set of services, which are offered to the other client tasks.

```
1  task type Box(Initial_Value : Integer;
2                Min_Value : Integer;
3                Max_Value : Integer) is
4     entry Add(Number : Integer);
5     entry Subtract(Number : Integer);
6     entry Read;
7  end Box;
```


The accept statement

- The accept statement specifies the actions to be performed when an entry is called.
- For each and every entry defined in a task there must be at least one accept statement in the corresponding task body.

What will happen here?

```
1  task body Box is
2     X : Integer := Initial_Value;
3  begin
4     loop
5        accept Add(Number : Integer) do
6           X := X + Number;
7        end Add;
8        accept Subtract(Number : Integer) do
9           X := X - Number;
10       end Subtract;
11       accept Read do
12          Put("Coins:");
13          Put_Line(Integer'Image(X));
14       end Read;
15    end loop;
16 end Box;
```


Selective Accept

The `select` statement allows a server to

- wait for more than a single rendezvous at any time
- time out if no client request is issued with in a specified period (`delay`)
- terminate if no client can possibly call its entries (`terminate`)
- withdraw its offer to communicate if no rendezvous is immediately available (not discussed here)

A selective accept can be combined with guards (`when`-statements)

Selective Accept

A selective accept can be combined with guards (`when`-statements)

Execution of select statement

- only one accept alternative is executed
- If none of the accept alternatives can be taken, the task is suspended

Rendezvous with selective accept

```
1   loop
2      select
3         when X < Max_Value => -- Guard
4            accept Add(Number : Integer) do
5               X := X + Number;
6            end Add;
7      or
8         when X > 0 => -- Guard
9            accept Subtract(Number : Integer) do
10              X := X - Number;
11           end Subtract;
12     or
13        accept Read do
14           Put("Coins:");
15           Put_Line(Integer'Image(X));
16        end Read;
17     or -- Timeout
18        delay 5.0;
19        Put_Line("No request received for 5 seconds");
20     end select;
21  end loop;
```

The terminate-alternative

A server task terminates, if all remaining tasks that can call the server task are completed or will not be able to make a future call to the server.

```
1  loop
2     select
3        accept Service1(...) do ... end;
4     or
5        accept Service2(...) do ... end;
6     ...
7     or -- Terminate
8        terminate;
9     end select;
10 end loop;
```


Outline

Real-Time Annex

The real-time annex defines additional semantics and facilities for real-time applications. It has been significantly extended in the Ada 2005 standard.

The real time annex covers among others

- scheduling policies
- support for priority ceiling protocol
- real-time clock
- timers
- ...

Ada's Time Facilities

Ada defines two different packages that deal with time

- `Ada.Calendar` provides an abstraction for wall clock time
- `Ada.Real_Time` specifies a high-resolution, monotonic (non-decreasing) clock package. It is defined in the real-time annex

The current time is returned by the function `Clock`.

The package Ada.Real_Time (1/2)

```
1  package Ada.Real_Time is
2    type Time is private;
3    ...
4    type Time_Span is private;
5    ...
6    function Clock return Time;
7    ...
8    function "+" (Left : Time; Right : Time_Span)
9       return Time;
10   function "+" (Left : Time_Span; Right : Time)
11      return Time;
12   function "+" (Left, Right : Time_Span)
13      return Time_Span;
14   ...
```


The package Ada.Real_Time (2/2)

```
1    ...
2    function To_Duration (TS : Time_Span)
3       return Duration;
4    function To_Time_Span (D : Duration)
5       return Time_Span;
6    ...
7    function Microseconds (US : Integer)
8       return Time_Span;
9    function Milliseconds (MS : Integer)
10      return Time_Span;
11   function Seconds      (S  : Integer)
12      return Time_Span;
13   ...
14 end Ada.Real_Time;
```

Delay Primitives

The statement `delay` can be used in two different ways:

- `delay 5.0` delays the task for five seconds
- `delay until` will delay a task until a certain absolute time.

Periodic Tasks

Periodic tasks can be implemented with priority and period.

```
1  task type T(Id: Integer; Period_Ms : Integer) is
2     pragma Priority(System.Default_Priority + Id);
3  end;
4
5  task body T is
6     Interval : Time_Span := Milliseconds(Period_Ms);
7     Next : Time;
8  begin
9     Next := Clock;
10    loop
11       -- Do something
12       Next := Next + Interval;
13       delay until Next; -- Wait until start of period
14    end loop;
15 end T;
```

Scheduling Policies (1/4)

Different scheduling policies can be selected by

```
1  pragma Task_Dispatching_Policy(policy)
```

The following policies are available

- `FIFO_Within_Priorities`
- `Non_Preemptive_FIFO_Within_Priorities`
- `Round_Robin_Within_Priorities`
- `EDF_Across_Priorities`

Scheduling Policies (2/4)

FIFO Within Priorities Within each priority level tasks are scheduled on a first-in-first-out basis. A task may preempt another task with lower priority.

Non-Preemptive FIFO Within Priorities A task runs to completion and cannot be preempted by a task with higher priority.

Scheduling Policies (3/4)

Round-Robin Within Priorities Within each priority level tasks are time-sliced with an interval that can be specified.

EDF Across Priorities Each task has relative deadline (given by pragma Relative_Deadline). Tasks within a range of priorities are scheduled according to the earliest-deadline-first algorithm.

Scheduling Policies (4/4)

Scheduling policies can be combined in order to divide a set of tasks into hard and soft real-time tasks.

```
1  pragma Priority_Specific_Dispatching
2            (Round_Robin_Within_Priorities, 1, 4);
3  pragma Priority_Specific_Dispatching
4            (FIFO_Within_Priorities, 5, 20);
```

Priority levels 5 to 20 are reserved for hard real-time tasks, while levels 1 to 4 are used for soft-real time tasks.

Quick Test

The following code uses the real-time annex. S1 and S2 are binary semaphores (implemented with protected objects) and initialised to 1. T1 has the highest priority and T4 the lowest. The dispatching policy used is FIFO_Within_Priorities.

```
task body T1 is
begin
  loop
    S1.Wait;
    Put_Line(Integer'Image(1));
    S2.Wait;
   end loop;
end;
task body T2 is
begin
  loop
    S2.Wait;
    Put_Line(Integer'Image(2));
    S1.Wait;
  end loop;
end;
```

```
task body T3 is
begin
  loop
    Put_Line(Integer'Image(3));
    S2.Signal;
    S2.Wait;
  end loop;
end;
task body T4 is
begin
  loop
    Put_Line(Integer'Image(4));
    S1.Signal;
    S2.Wait;
  end loop;
end;
```


Priority-Ceiling Protocol

It is possible to give a protected object a priority.

```
protected Object is
   pragma Priority(20);
   entry E;
   ...
```

The use of the priority-ceiling protocol can be specified by

```
pragma Locking_Policy(Ceiling_Locking);
```

A task calling the entry E will inherit the ceiling priority (20) while executing the protected operation.

The Ravenscar Profile

The Ravenscar profile is a restricted subset of the Ada tasking model in order to meet requirements on

- determinism
- schedulability analysis
- memory boundness

and to allow for a small and efficient run-time system that supports task synchronisation and communication.

The Ravenscar profile is defined in Ada 2005 and can be used by

```
1    pragma Profile(Ravenscar);
```

The rendezvous mechanism is not supported by the Ravenscar profile!

GNAT and the Real-Time Annex

- GNAT supports the full real-time annex
- However, it depends on the underlying architecture of all GNAT features are implemented for a given operating system/architecture.

Outline

Summary

- Ada has been designed as a language for embedded real-time systems.
- It comes with a strong support for
  - concurrency
  - real-time
- In contrast to other programming languages concurrency and real-time mechanisms are directly built into the language
- A free compiler (GNAT) supporting the full Ada standard exists

Outline

Further Reading

John Barnes. Programming in Ada 2012. Cambridge University Press, 2014.

Alan Burns and Andy Wellings. Concurrent and Real-Time Programming in Ada. Cambridge University Press, 2007.