Real-Time Programming Languages

Labtorial 9: Ada Data Structures, Threads & Synchronization

Martin Becker

TU München
Institute for Real-Time Computer Systems (RCS)

December 8, 2015

Lehrstuhl für
Realzeit-Computersysteme

Technische Universität München

Outline

Section 1

General Features

Control Constructs

The most important ones:

```
if condition then statement1; else statement2; end if;
```

```
case X is
  when 1 => statements1;
  when 2 => statements2;
  when others => statements3;
end case;
```

```
loop statements; end loop;
```

```
while condition loop statements; end loop;
```

See `http://en.wikibooks.org/wiki/Ada_Programming/Control`

Attributes

- useful to get/set information about an object
- denoted by a single quote, e.g.:
  - `myInteger'First`. Attribute "First" returns lowest possible Integer value
  - `myInteger'Last`. Returns highest possible Integer value
- attribute "Range" can be used with arrays:

```
myArray : array(1..5) of Integer := (2,4,6,8,10);
for i in myArray'Range loop
  Put_Line("element " & i'img & "=" & myArray(i)'img);
end loop;
```

- Attribute "img" yields the string representation of the object:

```
Put_Line("Value of myInteger is " & myInteger'img);
```

For more see `http://en.wikibooks.org/wiki/Ada_Programming/Attributes`

Function/Procedure Parameters (1)

- arguments can be handed over in one of the following modes:
  - `in`: default mode. The callee gets a *constant* copy, i.e., unlike in *C* value cannot be changed in callee

```
function hello( x : in Integer, y: out Integer) is
begin
  x := 5; -- error! x is constant
  y := x+1; -- that's ok
end hello;
```

  - `in out`: callee gets a reference; i.e., modifications of parameter is reflected in caller
  - `out`: callee does *produce* a value for argument, value is passed to caller.

Function/Procedure Parameters (2)

- named parameters prevent from switching arguments:
  - in your main...:

```
distance_to_home := getHomeDist(11.4812, 45.77192);
```

  - ...and the specification of `getHomeDist`:

```
function getHomeDist(latitude : float, longitude :
     float) return float;
```

What happens if the developer decides to switch arguments?

Function/Procedure Parameters (2)

- named parameters prevent from switching arguments:
  - in your main...:

```
distance_to_home := getHomeDist(11.4812, 45.77192);
```

  - ...and the specification of `getHomeDist`:

```
function getHomeDist(latitude : float, longitude :
     float) return float;
```

    What happens if the developer decides to switch arguments? Code runs incorrectly w/o noticing since latitude and longitude have the same data type
  - *named parameters* make it much safer:

```
distance_to_home := getHomeDist(longitude =>
     11.3512,  latitude => 48.877192);
```

Records

- you can build composite data types, like with `struct` in C:

```
type Car is record
    Identity         : Long_Long_Integer;
    Paint            : Color;
    Horse_Power_kW   : Float range 0.0 .. 2_000.0;
    Consumption      : Float range 0.0 .. 100.0;
 end record;
```

The contents (e.g., `Paint`) are called *components*. Now a new variable with this type can be declared:

```
   -- ...
   mycar : Car;
begin
   mycar.Consumption := 10.1;
   -- ...
```

Is there a convenient way to assign all components at once?

Aggregates

Is there a convenient way to assign all fields at once? **Yes.**
This is called an *aggregate:*

```
BMW : Car := (2007_752_83992434, Blue, 190.0, 10.1);
```

But like this we could accidentally switch horsepower (190.0) and consumption (10.1). Similar to *named parameters*, we can do this:

```
BMW : Car :=
    (Identity        => 2007_752_83992434,
     Horse_Power_kW  => 190.0,
     Consumption     => 10.1,
     Paint           => Blue);
```

Ada's *full coverage rule* forces you to specify all fields.

Discriminants

- sometimes you might want to parametrize a record type, e.g.:

```
type card_deck (Size : Positive Integer) is
record
   A : array (1 .. Size) of Integer;
end record;
```

- here this allows to have a record `card_deck` with a user-defined (array) size
- usage:

```
  -- ...
  skat : card_deck (10); -- array of length 1...10
  patience : card_deck (32); -- array of length 1...32
begin
  -- ...
```

Variant Record

- sometimes not all values in a record are meaningful
- they can be “turned off” depending on a discriminant:

```
type Vehicle_Type is (Car, Bike);

 type Vehicle (Option : Vehicle_Type) is
    record
       Number_Wheels : Positive Integer;
       Max_People: Positive Integer;
       case Option is
          when Car =>
             Horse_Power_kW : Float;
          when Bike =>
             Number_of_Gears : Positive Integer;
       end case;
    end record;
```

- asking for `Horse_Power_kW` in a bike will throw an exception

Variant Record

- sometimes not all values in a record are meaningful
- they can be “turned off” depending on a discriminant:

```
type Vehicle_Type is (Car, Bike);

 type Vehicle (Option : Vehicle_Type) is
    record
       Number_Wheels : Positive Integer;
       Max_People: Positive Integer;
       case Option is
          when Car =>
             Horse_Power_kW : Float;
          when Bike =>
             Number_of_Gears : Positive Integer;
       end case;
    end record;
```

discriminant

variant 1

- asking for `Horse_Power_kW` in a bike will throw an exception

(Pool) Access

- *access types*=“pointers” in Ada, should be used rarely
- with them a new object can be *allocated* during run-time
- they have their own type, but no arithmetic ~~(“ptr++”)~~

Ada:

```
type Int_Access is access
     Integer;
ptrFoo, ptrBar : Int_Access;
ptrFoo := new Integer;
ptrBar := ptrFoo;
ptrBar.all := ptrFoo.all;
ptrFoo.all := 5;
```

C equivalent:

```
typedef int* int_access;

int_access ptrFoo, ptrBar;
ptrFoo = malloc(sizeof(int));
ptrBar = ptrFoo;
*ptrBar = *ptrFoo;
*ptrFoo = 5;
```

Warning: Not all Ada compilers implement garbage collection!
Allocated objects may have to be free’d manually

(General) Access

difference to C: access types by default only can point to dynamically allocated objects (i.e., not to declared variables)

- to obtain access to a declared variable we need to:
  1. allow the variable to be accessed with the `aliased` keyword (alias = "s.th. can be addressed with a different name"):

     ```
     myInt : aliased Integer;
     ```

  2. define an access type that can point to *all* integers:

     ```
     type Int_Access is access all Integer;
     ```

- Now we can assign the pointer to the declared variable

```
ptrFoo : Int_Access;
ptrFoo := myInt'Access;
```

(General) Access (2)

One can restrict the accesses further:

- `access all`: read and write access
- `access constant`: read-only access

```
type pointer_rw is access all      Integer;
type pointer_to is access constant Integer;

var_const: aliased constant Integer := 10;
var: aliased Integer;
var_noalias: Integer; -- no alias means no access!

p1: pointer_rw := var_const'Access;   -- illegal
p2: pointer_ro := var_const'Access;   -- OK, read only
p3: pointer_rw := var'Access;   -- OK, read and write
p4: pointer_ro := var'Access;   -- OK, read only
p5: pointer_ro := var_noalias'Access; -- illegal
p6: constant pointer_rw := var'Access; -- r/w only on
    var
```

Generics

- Ada is made for large-scale systems; reusability required
- *generics* provide that (similar to *templates* in C++)

```
-- this is a normal procedure;
-- only works for Integer
procedure Swap(Left, Right :
      in out Integer);

procedure Swap(Left, Right :
      in out Integer) is

  Temporary : Integer;
begin
  Temporary := Left;
  Left := Right;
  Right := Temporary;
end Swap;
```

```
generic
type Element_Type is private;
procedure Generic_Swap(Left,
      Right : in out
      Element_Type);
procedure Generic_Swap(Left,
      Right : in out
      Element_Type) is
  Temporary : Element_Type;
begin
  Temporary := Left;
  Left := Right;
  Right := Temporary;
end Generic_Swap;
```

- now it works for float, unsigned, hex, ...

Generics

- Ada is made for large-scale systems; reusability required
- *generics* provide that (similar to *templates* in C++)

```
-- this is a normal procedure;
-- only works for Integer
procedure Swap(Left, Right :
      in out Integer);

procedure Swap(Left, Right :
      in out Integer) is

  Temporary : Integer;
begin
  Temporary := Left;
  Left := Right;
  Right := Temporary;
end Swap;
```

```
generic
type Element_Type is private;
procedure Generic_Swap(Left,
      Right : in out
      Element_Type);
procedure Generic_Swap(Left,
      Right : in out
      Element_Type) is
  Temporary : Element_Type;
begin
  Temporary := Left;
  Left := Right;
  Right := Temporary;
end Generic_Swap;
```

- now it works for float, unsigned, hex, ...

Generics (2)

How do we “call” a specific version of `Generic_Swap`?

- we have to *instantiate* a real version from the generic

```
with Generic_Swap;
procedure main is
  procedure Swap is new Generic_Swap(Integer);
  A, B : Integer;
begin
  A := 5;
  B := 7;
  Swap(A, B);
  -- Now A=7 and B=5.
end main;
```

Section 2

Tasking

Tasks

- a *task* runs concurrently to the rest of the Ada program (=thread)
- main program is also a task
- each task (also called *server*):
  - declaration and body
  - depends on a *master* = its surrounding block
  - terminates only after all its *dependents* terminate

```
task Single is
  -- declaration of exported identifiers
end Single;
-- ...
task body Single is
  -- declaration of locals and statements
end Single;
```

Task Interactions

tasks can

1. exchange messages (“rendezvous”)
2. wait for other tasks to complete (“join”)
3. use *protected objects* (“mutex”)
4. set global variables $\Rightarrow$ hands away, dangerous!

Details in the next slides...

Rendezvous and Entries

How can tasks exchange messages?

- tasks cannot have functions or procedures, but they can have something similar called *entries*
- parameters of entries are the messages

```
procedure main is
  task type My_Task_Type is
    entry Print  (whatever : in  integer);
  end My_Task_Type;
  task body My_Task_Type is begin
    loop
      accept Print (whatever : in integer) do
        Put_Line("message with value=" & whatever'img);
      end Print;
    end loop;
  end My_Task_Type;
  task1 : My_Task_Type;
begin
  task1.Print(5); delay (5.0); task1.Print(10);
end main;
```

Rendezvous and Entries

How can tasks exchange messages?

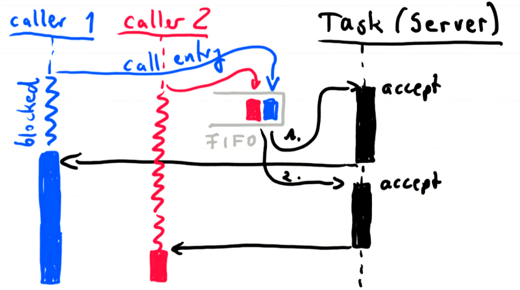
- tasks cannot have functions or procedures, but they can have something similar called *entries*
- parameters of entries are the messages

```
procedure main is
  task type My_Task_Type is
    entry Print  (whatever : in  integer);
  end My_Task_Type;
  task body My_Task_Type is begin
    loop
      accept Print (whatever : in integer) do
        Put_Line("message with value=" & whatever'img);
      end Print;
    end loop;
  end My_Task_Type;
  task1 : My_Task_Type;
begin
  task1.Print(5); delay (5.0); task1.Print(10);
end main;
```

entry

Rendezvous and Entries

- entry is executed in the thread of the task, not by the caller (major difference to threads in C[++])
  - the caller blocks until the server completed the call:



  - if `Task1` is not ready to process the call immediately, then `main` waits in an associated FIFO
  - note that entries are therefore never executed concurrently

Rendezvous and Entries (2)

The task which contains the entries (*server*) is blocking every time the control reaches an `accept` statement. How about this?

```
task body My_Task_Type is begin
  loop
    accept Print (whatever : in integer) do
      Put_Line("message with value=" & whatever'img);
    end Print;
    accept Scan (whatever : out integer) do
      whatever := Integer'value(Get_Line);
    end Scan;
  end loop;
end My_Task_Type;
```

Rendezvous and Entries (2)

The task which contains the entries (*server*) is blocking every time the control reaches an `accept` statement. How about this?

```
task body My_Task_Type is begin
  loop
    accept Print (whatever : in integer) do
      Put_Line("message with value=" & whatever'img);
    end Print;
    accept Scan (whatever : out integer) do
      whatever := Integer'value(Get_Line);
    end Scan;
  end loop;
end My_Task_Type;
```

Since `accept` blocks every time it is reached, this task enforces alternating calls to `Print` and `Scan`. **What if we do not want this blocking, e.g., to serve multiple entries in any order?**

What if we do not want blocking, e.g., to serve multiple entries in any order? This is called a *selective wait*:

```
task body My_Task_Type is begin
  loop
    select
      accept Print (whatever : in integer) do
        Put_Line("message with value=" & whatever'img);
      end Print;
    or
      accept Scan (whatever : out integer) do
        whatever := Integer'value(Get_Line);
      end Scan;
    end select;
  end loop;
end My_Task_Type;
```

What if we do not want blocking, e.g., to serve multiple entries in any order? This is called a *selective wait*:

```
task body My_Task_Type is begin
  loop
    select
      accept Print (whatever : in integer) do
        Put_Line("message with value=" & whatever'img);
      end Print;
    or
      accept Scan (whatever : out integer) do
        whatever := Integer'value(Get_Line);
      end Scan;
    end select;
  end loop;
end My_Task_Type;
```

alternative 1

alternative 2

Selective Wait

What if we do not want blocking, e.g., to serve multiple entries in any order? This is called a *selective wait*:

```
task body My_Task_Type is begin
  loop
    select
      accept Print (whatever : in integer) do
        Put_Line("message with value=" & whatever'img);
      end Print;
    or
      accept Scan (whatever : out integer) do
        whatever := Integer'value(Get_Line);
      end Scan;
    end select;
  end loop;
end My_Task_Type;
```

alternative 1

alternative 2

- if only one of alternatives has pending entry call ⇒ accepted this
- if multiple calls pending ⇒ choose freely ⇒ non-determinism
- If no alternative is viable ⇒ error

Timeout

Selective wait allows a non-blocking server task, but what about the caller?

- as mentioned before, caller waits in a FIFO until server can handle the call
- **can we avoid such blocking of the caller?** *Yes.*

Timeout

Selective wait allows a non-blocking server task, but what about the caller?

- as mentioned before, caller waits in a FIFO until server can handle the call
- **can we avoid such blocking of the caller?** *Yes.* We can specify a *timeout* as follows:

```
task body My_Task_Type is begin
  loop
    select
      accept Print ( whatever : in integer ) do
        Put_Line("message with value ="& whatever'img);
      end Print ;
    or
      delay 5.0; -- select ends latest after 5 sec
    end select;
  end loop;
end My_Task_Type;
```

Conditional Entry Call / Nonblocking Call

Can we have a timeout with zero delay (a.k.a. “nonblocking call”)?

- `select` statement shall return or do s.th. else, if it cannot be served immediately
- **how about that:**

```
select
  accept
  -- ...
or
  delay 0.0;
  Do_something_else;
end select;
```

Conditional Entry Call / Nonblocking Call

Can we have a timeout with zero delay (a.k.a. “nonblocking call”)?

- `select` statement shall return or do s.th. else, if it cannot be served immediately
- **how about that:**

```
select
  accept
  -- ...
or
  delay 0.0;
  Do_something_else;
end select;
```

not what we want:

- on a real processor, rendezvous takes (a nonzero) time
- thus `delay 0.0` is impossible
- always a timeout

Conditional Entry Call / Nonblocking Call

Can we have a timeout with zero delay (a.k.a. ‘‘nonblocking call’’)?

- `select` statement shall return or do s.th. else, if it cannot be served immediately
- for ‘‘as fast as possible’’ delay, Ada offers a *conditional entry call*:

```
select
  accept
  -- ...
else
  Do_something_else;
end select;
```

- entry call is not done if the rendezvous is not ‘‘immediately’’ achieved
- `else` branch is executed instead

Guards

Sometimes one might want to disable alternatives. Ada provides *guards* for this:

```
task body My_Task_Type is
  print_ready : Boolean := true;
begin
  loop
    select
      when print_ready =>
        accept Print ( whatever : in integer ) do
          Put_Line("message with value ="& whatever'img);
        end Print ;
    or
      delay 5.0; -- select ends latest after 5 sec
    end select;
  end loop;
end My_Task_Type;
```

- note guards are not parameters, but internal states of the task

Termination

A of now, our tasks do not terminate, therefore their *master* (surrounding block) cannot terminate. **How to fix this?**

- we add a *terminate alternative* to the task:

```
task body My_Task_Type is begin
  loop
    select
      accept Print (whatever : in integer) do
        Put_Line("message with value=" & whatever'img);
      end Print;
    or
      terminate; -- this is what we added
    end select;
  end loop;
end My_Task_Type;
```

Termination (2)

`terminate` is executed when *all* of the following conditions hold true:

1. terminate alternative is reachable AND
2. no pending calls to entries AND
3. all other tasks of the same master are in the same state AND
4. master has completed (i.e., control reached end of statements).

Example: Task from previous slide

main:

```
delay 1.0;
task1.Print(5);
-- task still running
4 delay 1.0;
task1.Print(10);
Put_Line("master complete"
      );
-- task terminates
```

output::

```
message with value = 5
message with value = 10
3 master complete

[2014-12-08 15:49:54]
      process terminated
      successfully, elapsed
       time: 02.16s
```

Requeue Statement

- **recall:** calling an entry on a task means putting yourself in a FIFO queue
- server may decide to redirect the call to another entry (“requeue”)
- the entry which we redirect to must have same parameters

```
loop
  select
    accept Print (whatever : in integer) do
      -- probably do some stuff here
      requeue Real_Print;
    end Print;
  or
    accept Real_Print (whatever : in integer) do
      Put_Line("value=" & whatever'img);
    end Other_Print;
  end select;
end loop;
```

Protected Types (“‘mutex’’)

We have seen that entries serve sequentially (…FIFO)

- Ada requires no explicit mutex to protect the data in a task
- but sometimes it might be too heavy to create a task just for the purpose of protecting data
- for that, Ada provides data objects with protection against data inconsistency (e.g., race condition)
- like a fancy form of a semaphore or mutex
- the protected data can only be accessed through *protected operations:*
  1. **protected functions:** provide read-only access
  2. **protected procedures:** exclusive read-write access
  3. **protected entries:** like procedure, but guarded
- very efficient $\Rightarrow$ any protected operation should be short and fast

Protected Types (“mutex”) (2)

```
protected type Protected_Buffer_Type is
  entry Insert (An_Item : in  Item);
  entry Remove (An_Item : out Item);
private
   Buffer : Item;
   Empty  : Boolean := True;
end Protected_Buffer_Type;
protected body Protected_Buffer_Type is
  entry Insert (An_Item : in  Item) when Empty is
  begin
    Buffer := An_Item;
    Empty := False;
  end Insert;
  entry Remove (An_Item : out Item) when not Empty is
  begin
    An_Item := Buffer;
    Empty := True;
  end Remove;
end Protected_Buffer_Type;
```

guarded

If guard evaluates to *false*, the caller waits in the FIFO

Protected Types (“mutex”) (3)

```
procedure main is
  mybuf : Protected_Buffer_Type;
  task type OtherTask is end OtherTask;
  task body OtherTask is
    i : Integer := 0;
  begin
    loop
     delay 1.0;
     mybuf.Insert(i);
     Put_Line("insert " & i'img);
     i := i + 1;
    end loop;
  end OtherTask;
  task1 : OtherTask;
  o : Integer;
begin
  loop
    mybuf.Remove(o);
    Put_Line("remove " & o'img);
  end loop;
end main;
```

these two tasks are running concurrently without any mutex

Abort

- avoid using this (rather use asynchronous `select`)

Timing Events

- reacting to the arrival of a point in time – like an interrupt
- e.g., execute some task starting at exactly 7.00am
- usually you would need an appropriate `delay` statement
  - concurrency overhead unnecessary and inefficient
- **secondary in this lecture**, if you are interested study the blinker example (`Set_Handler()` in `Timer_Setup::Pulser`)

Dynamically Created Tasks

- need a task type
- need an access type (cmp. with C: for creating threads you need a pointer, and pointers are types in Ada)
- use `new` keyword

```
procedure main is
  task type T is
    -- ...
  end T;
  task body T is -- implementation of the task of task
    -- ...
  end T;
  type dynTask is access T;
  Task_3 : dynTask; -- "pointer" to task
begin
  Task_3 := new T; -- starts running immediately
  -- procedure terminates if all its task terminate
end main;
```

Dynamically Created Tasks

- need a task type
- need an access type (cmp. with C: for creating threads you need a pointer, and pointers are types in Ada)
- use `new` keyword

```
procedure main is
  task type T is
    -- ...
  end T;
  task body T is -- implementation of the task
    -- ...
  end T;
  type dynTask is access T;
  Task_3 : dynTask; -- "pointer" to task
begin
  Task_3 := new T; -- starts running immediately
  -- procedure terminates if all its task terminate
end main;
```

type declaration

access type declara-
tion

dynamic instantiation

Section 3

Exercises

Exercises for Today (1)

1. finish tasks from last lab (calendar, stop watch)
2. explore and understand the *blinker example* (code on Moodle)

Section 4

References

References

- *Ada Glossary*, Bard S. Crawford, `http://www.cs.uni.edu/~mccormic/AdaEssentials/glossary.htm`
- *Ada95 Lovelace tutorial*, David A. Wheeler (who wrote that in his free time), `http://www.adahome.com/Tutorials/Lovelace/lovelace.htm`
- *GNAT book*, J. Miranda and E. Schonberg, `https://www2.adacore.com/gap-static/GNAT_Book/html/aarm/AA-TOC.html`
- *Concurrent and Real-Time Programming in Ada*, A. Burns and A. Wellings, Cambridge Univ. Press, 2007.
- *Ada Wikibook*, `http://en.wikibooks.org/wiki/Ada_Programming`

All online resources as of December 2014.