REAL-TIME HIGH-LEVEL CONCURRENT PROGRAMMING LANGUAGES AND SOME FEATURES OF ADA AND JAVA

Historical Development of RT languages

- Early systems were programmed in assembly language - high-level languages were not well supported and were not efficient → but led to high development costs, maintenance problems and portability problems
- High-level languages were pressed into service, e.g. FORTRAN, PASCAL, C, etc
- Due to the deficiencies, specialised languages were developed for embedded systems programming, e.g. in the U.K. - CORAL 66 (defence) and RTL/2 (industry), in the U.S. - JOVIAL (defence). All these languages are sequential and require operating system support for real-time features → portability was still a problem area.
- Current generation RT languages, e.g. Ada 95, Modula-3, Real-Time Euclid, ESTREL, PEARL, CHILL, Mesa, Occam-3 all provide explicit support for concurrency and inter-process communication but do not consider timing constraints and fail to provide predictable temporal behaviour of programs.

Performance Criteria for RT languages

- *Security* - automatic detection of programming errors, preferably by the compiler or at the least by run-time support.
- *Readability* - influenced by choice of keywords, ability to define types and support for program modularization. Important for maintainability of the code, but usually increases program length.

- *Flexibility* - all required operations must be expressed in a clear and coherent manner to limit operating system calls or assembly language calls.
- *Simplicity* - useful to minimise compiler complexity, programmer training cost and programmer errors - important for expressive power (with flexibility) and useability.
- *Portability* - hardware independence is a useful feature → the language must be able to isolate machine dependent parts of the program.
- *Efficiency* - mechanisms leading to unpredictable run-time overheads should be avoided.

The final language design is necessarily a balance between these criteria and any application specific criteria

Languages to consider

Most studies of RT languages have focussed on Ada, Modula, Occam, or Concurrent-C. With performance enhancements, Java is only relatively recently been a candidate for addition to this list.

The Ada language supports the principles of modern software engineering - modularity, strong typing, data abstraction, information hiding, overloading, exception handling, and structured control statements.

In addition to the programming features also supported by other languages, such as control structures, arrays and records, subprograms, types, and access types (pointers), when introduced (and then enhanced with Ada 95) Ada had some features which were fairly unique:

- *Attributes* - predefined parameters that yield the characteristics of various items, e.g. INTEGER'LAST on a 16-bit machine would yield 32,767 → the aim is to improve software portability.
- *Packages* - a grouping of related resources (often a compilation unit) which allows separation of interface information to users of the package from the implementation details → encapsulation.

- *Generics* - templates from which procedures, functions and packages can be created → purpose is to minimise code duplication.
- *Exceptions* - a specific mechanism is provided to allow the software to identify the type of exception and handle run-time errors e.g. CONSTRAINT_ERROR is raised when an array index leaves a defined range.
- *Tasking* - support for concurrent execution by allowing code units (called *tasks*) to be effectively executed in the same time period. Inter-task synchronization and communication is handled via a rendezvous mechanism.
- *Low-level programming* - support for embedded systems through bit manipulation, hardware interrupts, machine specific instructions and addressing.
- Hierarchical Libraries: Ada 83 libraries had a mostly 'flat' structure, Ada 95 allows packages to have child units which can be added without affecting the parent of the parent's clients.
- Programming by Extension: new features can be added without requiring modification or recompilation of any of the existing software.
- Classwide Programming: abstraction to group the common properties of related types in a class of types - addition of a "tagged" type.
- Protected Objects: allowing concurrent tasks to share data in a protected manner, i.e. only one task at a time can update shared data very efficiently.

Reference Text: Naiditch, D. J. *Rendezvous with Ada 95, 2nd ed.*: Wiley 1995

On-line Tutorial: Lovelace by David A. Wheeler (see the unit web page)

FEATURES OF ADA TO SUPPORT REAL-TIME DISTRIBUTED PROCESSING

TASKING

Concurrent processing in Ada is called *tasking* and units of code that execute concurrently are called *tasks*. On single processors, interleaved concurrency is used to produce the effect that all tasks are executing simultaneously. Overlapped concurrency can also be supported on multiple processors, but in either case, tasks are used whenever several activities are to be performed in the same time period.

Non-interfacing tasks:

Tasks that do not communicate with each other have the following form:

```
task task name;                 -- specification
task body task name is          -- body
    declarations
begin
    statements
end task name;
```

As an example consider the following two tasks:

```
procedure Activity is

    task Exercise_Jaw;
    task body Exercise_Jaw is
    begin
        Chew_Gum;
    end Exercise_Jaw;

    task Exercise_Legs;
    task body Exercise_Legs is
    begin
        Run_Fast;
    end Exercise_Legs;

begin
    null;       -- must have at least one executable statement
end;
```

Tasks must always be embedded in the declarative part of another unit - they cannot be a separate compilation unit (although a task body can be a separately compiled subunit). The unit in which tasks are embedded is referred to as a *master unit* or *environment task* and the tasks are dependent on it.

Interfacing tasks without data transfer:

Such tasks are used when only synchronization is required and these tasks require two additional clauses:

```
task task name is
    entry entry name (parameter definitions);
end task name;
```

i.e. the task specification has an **entry** clause with a name and optional parameters (used for data transfer). This clause associates an entry identifier with one or more entry points in the task body. The entry points are defined by **accept** statements and the entry identifier, with the following form:

```
accept entry name (parameter definitions) do
    statements
end entry name;
```

Taking a simple example to illustrate this:

```
procedure Get_Data is
    task See_If_Data is              -- task specification
        entry Data_Check;
    end See_If_Data;
    task body See_If_Data is         -- task body
    begin
        accept Data_Check do
            Go_Get_Data;
        end Data_Check;
    end See_If_Data;
            •
            •
```

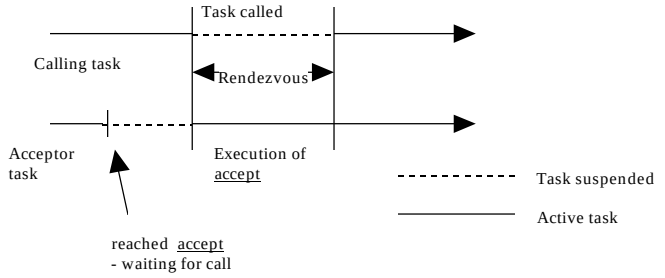


```
task Process_Data;
task body Process_Data is
begin
    See_If_Data.Data_Check
end Process_Data;
            •
            •
```

The *calling task* (Process_Data) must identify the *acceptor task* (See_If_Data) and the entry point name (Data_Check). The *dot* notation is used to specify an entry point (which cannot be circumvented with a **use** clause), but a task entry point can be renamed as a procedure, e.g:

```
task body Process_Data is
    procedure Test_Data renames
        See_If_Data.Data_Check
begin
    Test_Data;
end;
```

Note that an **accept** statement can only be placed in a task body and cannot appear in the statement part of a procedure. In Ada, the meeting of two tasks in this way is known as a *rendezvous*. Where an acceptor task is prepared to accept a call before the call is made the following diagram is applicable:



The other possible rendezvous situations are:

- where the call is made before the acceptor task is prepared to accept it, in which case the calling task just waits until its call is accepted.
- where the call is made at the same instant that the acceptor task is prepared to accept it, both tasks rendezvous immediately and neither task waits.

In the event that a task reaches its **accept** statement only once, yet is called twice, a Tasking_Error exception is raised since the task with the accept statement terminates after the first rendezvous and the second call to that task finds no task to rendezvous with. To avoid this error, multiple **accept** statements, or looping on the **accept** statement, can be used.

A handler for the Tasking_Error exception can also be added to the task body, e.g:

```
task body Process_Data is
begin
    See_If_Data.Data_Check;
exception
    when Tasking_Error => Dont_Panic;
end Process_Data;
```

where Dont_Panic is a procedure to handle the exception.

Exception propagation during rendezvous can be complex (consult the LRM for full details). For example, if a rendezvous was in progress, and an exception is unhandled by the calling task, then propagation is to a point immediately after the **accept** statement in the called task.

As a task entry point can be called several times, and the called task cannot accept all calls at once, a FIFO queue of calls for each task entry is maintained. The calling task with a call in the entry queue is suspended until the rendezvous is completed. So that an acceptor task can determine the number of queued entry calls, a Count attribute is available e.g. Task_Name'Count returns an integer call count.

Interfacing tasks with data transfer:

Where data transfer between tasks is involved, the structure of the **entry** and **accept** statements is the same as before, only now the optional parameter definitions are used. The following example also illustrates multiple **entry** and **accept** statements:

```
task Service_Station is                    -- task specification
    entry Pumps (Car : Car_Id);
    entry Garage (Car : Car_Id);
end Service_Station;

task body Service_Station is
begin
    loop
        accept Pumps (Car : Car_ID) do
            Fill_With_Petrol (Car);
        end Pumps;
        accept Garage (Car : Car_ID) do
            Service_Car (Car);
        end Garage;
    end loop;
end Service_Station;
```

Where a typical task call for the above tasks would be:

```
Service_Station.Pumps (Car_ 123);
Service_Station.Garage (Car_678);
```

In the previous Service_Station task example, the task is suspended waiting on the first **accept** whereas we would prefer it to accept either **accept** as they become active through an entry call - this is achieved with the **select** statement.

Selective wait:

The basic selective wait has the following form:

```
select
     accept statement
or
     accept statement
          •
          •
or
     accept statement
end select;
```

where the task waits for *any* task entry before continuing execution. For multiple simultaneous entries the entry selected is not specified (but one entry is selected by the run-time environment).

A **when** clause can also be added before the **accept** statement which acts as a *guard* on the **accept**. Consider the previous example with a selective wait and a guard (the boolean Petrol_Available):

```
     •
     •    (as before)
     •
loop
     select
          when Petrol_Available =>
               accept Pumps (Car: Car_Id) do
                    Fill_With_Petrol (Car);
               end Pumps;
     or
          accept Garage (Car: Car_Id) do
               Service_Car (Car);
          end Garage;
     end select;
end loop;
```


Caution is required where all possible alternatives in a select statement are guarded, as if after evaluating all guards no open accept statement is found, the predefined exception **Program_Error** is raised. To avoid this situation an **else** clause can be used in the select statement, e.g:

```
     •
     •
select
     when ...
          accept ...
               •
               •
or
     •
     •
else
     Sleep;
end select;
```

The **else** clause executes if all accepts are guarded and all their associated task entries have not been called - i.e. no task entry is waited on. As an intermediate alternative, a **delay** statement can be used to provide a timed waiting period, e.g:

```
select
     accept ...
or
     accept ...
or
     delay 1_000.0;    -- wait for 1,000 seconds before sleeping
     Sleep;
end select;
```

In place of a **delay** statement, a **terminate** statement can also be used to terminate the task - this can only occur when there are no calls in the task entry queues waiting to be accepted. This statement can be contrasted with the **abort** statement which causes immediate task shutdown.

The abort statement has the following form:

abort *task name list*;

which not only aborts tasks specified in the list but also aborts all dependent tasks and active subprograms as well.

There are restrictions placed on the use of the **select** statement alternatives: **else, delay** and **terminate.** No more than one of these alternatives can be contained in a single **select** statement.

Conditional entry calls and timed entry calls:

The select statement can also be used in the calling task. Firstly in a conditional entry call an **else** clause is used, e.g:

```
select
    Service_Station.Pumps (Car_123);
else
    Try_Another_ Service_Station;
end select;
```

which allows an alternative procedure to be activated if the call is not accepted immediately.

The timed entry call allows a **delay** statement to be executed if the call is not immediately accepted by the called task, e.g:

```
select
    Service_Station.Pumps (Car_123);
or
    delay 60.0;     -- wait 60 seconds
    Try_Another_ Service_Station;
end select;
```

In the above, if the rendezvous occurs before the delay of 60 seconds expires, then the procedure Try_Another_ Service_Station is not executed and the **select** statement is exited.

Asynchronous Select Statements:

Ada95 supports another form of selective entry call which uses a **then abort** clause. This causes a task in a selective wait to exit the wait when an alternative action becomes feasible. Consider a simple ATM example:

```
begin
    Read_Card (Card_data);
    select  -- asynchronous select
        Keyboard.Cancel_Pressed;     -- user may cancel
        raise Transaction_Cancelled;
    then abort
        Validate_Card (Card_Data);  -- query message on card
    end select;
    Perform_Transaction (Card_Data);
exception
    when Transaction_Cancelled =>
        Display_Cancellation_Notice;
        Return_card;
end;
```

Task attributes:

Since a task can have four states: running, completed, terminated or abnormal, there are two attributes provided that can be used to monitor the state of a task:

- Task_Name'Terminate returns True if Task_Name has terminated and False otherwise
- Task_Name 'Callable returns True if Task_Name is running and False otherwise

The abnormal state is only caused by an **abort** and is not otherwise distinguished from the terminated state.

Entry Families:

It is possible to use one-dimensional arrays as entry point families for conciseness, e.g:

```
task Multiple_Entries is
     entry  Point (1..5) (On: in  Boolean);
     entry Emergency (Boolean);
end  Multiple_Entries;
```

and the task body could have the following **accept** statements:

```
accept Point (3) (On: in  Boolean); -- 3rd member of family
accept Emergency (True);                    -- 2nd member of family
```

and other tasks can call these entries with:

```
Multiple_Entries.Point (3) (On => True);
Multiple_Entries.Emergency (True);
```

Note the difference between the entry family index, which appears first, and the entry parameters which follow.

Task priorities:

A compiler pragma is provided to assign task priorities in the task specification, i.e. the compiler prioritizes tasks according to the priority level, e.g:

```
pragma  Priority (priority level);
```

where the range of *priority level* is defined by the compiler.

Task types:

All the tasks defined so far have belonged to anonymous task types, but these can be defined explicitly, e.g:

```
task type Resource is  -- task type declaration
     entry  Seize;
     entry Release;
end  Resource;
```


Once a task type is declared then objects can be declared to belong to that type, e.g:

```
Laser_Printer, Ink_Jet_Printer : Resource;
```

which declares two tasks having the same entries (hence the use of dot notation for calling tasks) and performing the same processing. Task types are limited private types so task objects cannot be compared to each other, although they can be used as subprogram parameters.

Task interference:

This can occur where two or more tasks concurrently use the same resource, and a possible solution is to make use of the Resource task type defined above. Tasks obtain control of a resource by calling the task entry Seize and release it by calling the task entry Release.

As an example the following is taken from the Ada LRM:

```
package Resource_Handler is
     •
     •      -- task type Resource declaration
end Resource_Handler;
package body  Resource_Handler  is
     task body Resource is
          In_Use : Boolean := False;
     begin
          loop
               select
                    when not  In_Use =>
                         accept  Seize do
                              In_Use := True;
                         end  Seize;
               or
                    accept  Release do
                         In_Use := False;
                    end Release;
               or
                    terminate;
               end select;
          end loop;
     end  Resource;
end  Resource_Handler;
```

To use the Resource_Handler package it is made visible to the unit using it, e.g:

```
with Text_IO, Resource_Handler;
use  Text_IO, Resource_Handler;
procedure  Print is
      Screen : Resource;
      task type  Message_Type
      Copy_1, Copy_2 : Message_Type;
      task body  Message_Type
           is separate;
begin
     null;
end  Print;
```

```
separate  (Print)
task body Message_Type  is
begin
      Screen.Seize;
      Put ("message text");
      New_Line;
      Screen.Release;
end  Message_Type;
```

In this example, as soon as elaboration is complete and execution begins (after the **begin** keyword), the three tasks Screen, Copy_1 and Copy_2 are activated.

Dynamically created tasks:

All the examples examined so far have static task creation, i.e. tasks are created at the start of execution after elaboration. However, dynamically created tasks can be brought into existence using access types - the same way dynamically created objects are created, e.g:

```
•
•      procedure Print as previously
•
type Screen_Task is access Resource;
Screen : Screen_Task := new Resource;
•
•      procedure Print as previously
•
```

In this example, the task Screen is activated when it is created with the allocator **new**. The tasks Copy_1 and Copy_2 are activated as before after elaboration of the Print procedure.

Real-Time Extensions

- Time management: Ada 83 provided a Calendar package to represent time points (Calendar.Time) and intervals (Standard.Duration). In Ada 95, a Monotonic package is provided to give a guaranteed non-decreasing clock, and a **delay until** keyword is added to provide precise periodic execution.
- Task scheduling: In Ada 95, task priorities can be varied dynamically, alternative scheduling policies can be implemented, control over selection priorities for alternatives in selective waits, entries from entry queues, and interrupt handling tasks, is provided. A universal type for the task class, System.Task_Class can be used as an access type for a task, e.g:

```
with Dynamic_Priority_Support.Implementation_Extensions;
package body Mode_Change_Management is
      use  Dynamic_Priority_Support.Implementation_Extensions;
      Num_Managed: constant:= Number_Of_Tasks;
      Tasks_Being_Managed: Task_List(1..Num_Managed):=
            (T1'Access, T2'Access, T3'Access,...);
                        .
      Priority_In_Mode: array (System_Mode) of
            Priority_List(1..Num_Managed):=
                  (Normal => (T1_Norm, T2_Norm, ...),
                   Mode2 => (T1_Mode2, T2_Mode2, ...),
                   Mode3 => ...);

      task body  Manager_Task  is
            Manager_Mode : System_Mode := Normal;
            New_Mode : System_Mode;
      begin
            loop
                  Mode_Manager.Wait_For_Mode_Change
                        (Manager_Mode, New_Mode);
                  Set_Priority(Tasks_Being_Managed,
                        Priority_In_Mode(New_Mode));
                  Manager_Mode: = New_Mode;
            end loop;
      end Manager_Task;
end Mode_Change_Management;
```

- Interrupts: Ada 95 provides an Interrupt_Management package which provides for:
  - an Interrupt_ID type to identify interrupts
  - a sub-package Interrupt_Names with implementation dependent constants
  - operations for dynamically attaching and detaching interrupts

Distributed Processing Extensions

The primary addition to Ada 95 to support the distribution of software modules is the *partition*. The mapping of partitions to physical nodes is determined by a user provided configuration in a *virtual nodes* approach.

The same result can be indirectly achieved with separate programs in Ada 83, but in Ada 95 the software for all nodes is one logical program so the strong type-checking applies across partition boundaries.

Communication between partitions is achieved via a **Remote_Call_Interface** package or (if shared memory is supported) through variables declared in a **Shared_Passive** package.

Ada 95 specifies a standard interface to the communications subsystem which is user or third-party supplied (e.g. RPC or REC).

As an example, consider a distributed database application, where the clients have their own address spaces, and there is some globally shared memory for holding messages between the clients and the database server.

The example also illustrates the use of the **Pure** pragma which allows the package in which it is used to be shared by all active partitions. First, define the types to be used in inter-partition communication:


```
package Comm_Types is
    pragma Pure;
    type File_Mode is (In_Db, Out_Db, Inout_Db);
    type Byte is range 0..255;
    Record_Size: constant:= 4096;
    type Byte_Index is range 0..Record_Size-1;
    type Record_Key is String (1..32);
    type Record_Buffer_Type is array (Byte_Index) of Byte;
    pragma Pack (Record_Buffer_Type);
    type System_Time is range 0..2**31-1;
end Comm_Types;
```

Now the interface to the passive partition is defined corresponding to the global shared buffer 'pool':

```
with Comm_Types;
package Global_Memory is
    pragma Shared_Passive;
    Global_Timer: atomic Comm_Types.System_Time;
    Num_Buffers: constant := 100;
    type Buffer_Index is range 0..Num_Buffer: = 0;
    Null_Buffer_Index: constant Buffer_Index: = 0;
    Buffer_Array: array(Buffer_Index range 1..Num_Buffers) of
        Comm_Types.Record_Buffer_Type;
    type Index_Array is array (Buffer_Index range
        1..Num_Buffers) of Buffer_Index;          -- free list array
    protected Buffer_Pool_Manager is
        procedure Get_Buffer (Buffer: out Buffer_Index);
        procedure Release_Buffer (Buffer: in out Buffer_Index);
    private record
        Last_Used_Buffer: Buffer_Index: = 0;
        First_Free_Buffer: Buffer_Index: = 0;
        Next_Free_Buffer: Index_Array;
    end Buffer_Pool_Manager;
end Global_Memory;
```

The **Shared_Passive** pragma is used to place the package in a shared passive partition accessible to all active partitions. The protected record is used to synchronize access to the list of free buffers.