1

Introduction

Designing, implementing and maintaining software for large systems is a non-trivial exercise and one which is fraught with difficulties. These difficulties relate to the management of the software production process itself, as well as to the size and complexity of the software components. Ada is a mature general-purpose programming language that has been designed to address the needs of large-scale system development, especially in the embedded systems domain. A major aspect of the language, and the one that is described comprehensively in this book, is its support for concurrent and real-time programming.

Ada has evolved over the last thirty years from an object-based concurrent programming language into a flexible concurrent and distributed object-oriented language that is well suited for high-reliability, long-lived applications. It has been particularly successful in high-integrity areas such as air traffic control, space systems, railway signalling, and both the civil and military avionics domains. Ada success is due to a number of factors including the following.

- Hierarchical libraries and other facilities that support large-scale software development.
- Strong compile-time type checking.
- Safe object-oriented programming facilities.
- Language-level support for concurrent programming.
- A coherent approach to real-time systems development.
- High-performance implementations.
- Well-defined subsetting mechanisms, and in particular the SPARK subset for formal verification.

The development and standardisation of Ada have progressed through a number of definitions, the main ones being Ada 83 and Ada 95. Ada 2005 now builds on this success and introduces a relatively small number of language changes to provide:

- Better support for multiple inheritance through the addition of Java-like interfaces.
- Better support for OO style of programming by use of the `Object.Operator` notation.
- Enhanced structure and visibility control via the introduction of 'limited with clauses' that allow types in two library packages to refer to each other.
- More complete integration between object-oriented and concurrent programming by the introduction of synchronised interfaces.
- Added flexibility to the support for real-time systems development via alternative scheduling mechanisms and more effective resource monitoring.
- A well-defined subset of the tasking model, called the Ravenscar profile, for high-integrity applications.
- New library packages, for example an extensive 'containers' package for lists, maps, vectors etc.

Ada 2005 attempts, successfully, to have the safety and portability of Java and the efficiency and flexibility of C/C++. It also has the advantage of being an international standard with clear well-defined semantics.

The reference manual for Ada (ARM) differentiates between the 'core' language and a number of annexes. Annexes do not add to the syntax of the language but give extra facilities and properties, typically by the introduction of language-defined library packages. For the focus of this book, the key elements of the reference manual are Chapter 9 which deals with tasking and the Real-Time Systems Annex (Annex D). In terms of presentation, this book does not draw attention to core language or annex-defined facilities. All are part of the Ada language.

The remainder of this chapter provides an introduction to concurrency and real-time. The book then considers, in depth, the role that Ada can play in the construction of concurrent and real-time systems. It gives particular emphasis to the new features of Ada 2005. However, prior knowledge of the earlier language definitions is not required as a complete description of these features is provided. But the reader is assumed to be familiar with object-oriented programming in sequential Ada. For a more detailed discussion on the sequential aspects of Ada, the reader should see the further reading section at the end of this chapter.

1.1 Concurrency

Support for concurrency in a language allows the programmer to express (potential) parallelism in application programs. There are three main motivations for wanting to write concurrent programs.

- To fully utilise the processor – Modern processors run at speeds far in excess of

the input and output devices with which they must interact. Concurrency allows the programmer to express other activities to be performed while the processor is waiting for IO to complete.

- To allow more than one processor to solve a problem – A sequential program can only be executed by one processor (unless the compiler has transformed the program into a concurrent one). A concurrent program is able to exploit true parallelism and obtain faster execution.
- To model parallelism in the real world – Real-time and embedded programs have to control and interface with real-world entities (robots, conveyor belts etc.) that are inherently parallel. Reflecting the parallel nature of the system in the structures of the program makes for a more readable, maintainable and reliable application.

One of the major criticisms of concurrent programming is that it introduces overheads and therefore results in slower execution when the program is running on a single-processor system. Nevertheless, the software engineering issues outweigh these concerns, just as the efficiency concerns of programming in a high-level sequential language are outweighed by its advantages over programming with an assembly language. Chapter 2 further explores these concerns, and discusses in detail the nature and use of concurrent programming techniques.

Writing concurrent programs introduces new problems that do not exist in their sequential counterparts. Concurrent activities need to coordinate their actions, if they are to work together to solve a problem. This coordination can involve intricate patterns of communication and synchronisation. If not properly managed, these can add significant complexity to the programs and result in new error conditions arising. These general problems and their solutions are explained in Chapter 3. Chapters 4–11 then discuss in detail the facilities that Ada provides to create concurrent activities (called **tasks** in Ada) and to manage the resulting communication and synchronisation needs.

1.2 Real-time systems

A major application area of concurrent programming is real-time systems. These are systems that have to respond to externally generated input stimuli (including the passage of time) within a finite and specified time interval. They are inherently concurrent because they are often embedded in a larger engineering system, and have to model the parallelism that exists in the real-world objects that they are monitoring and controlling. Process control, manufacturing support, command and control are all example application areas where real-time systems have a major role. As computers become more ubiquitous and pervasive, so they will be embedded

in a wide variety of common materials and components throughout the home and workplace – even in the clothes we wear. These computers will need to react with their environment in a timely fashion.

It is common to distinguish between *hard* and *soft* real-time systems. Hard real-time systems are those where it is absolutely imperative that responses occur within the specified deadline. Soft real-time systems are those where response times are important, but the system will still function correctly if deadlines are occasionally missed. Soft systems can be distinguished from interactive ones in which there are no explicit deadlines. For example, the flight control system of a combat aircraft is a hard real-time system because a missed deadline could lead to a catastrophic situation and loss of the aircraft, whereas a data acquisition system for a process control application is soft, as it may be defined to sample an input sensor at regular intervals but to tolerate intermittent delays. A text editor is an example of an interactive system. Here, performance is important (a slow editor will not be used); however, the occasional poor response will not impact on the overall system's performance. Of course, many systems will have both hard and soft real-time subsystems along with some interactive components. What they have in common is that they are all concurrent.

Time is obviously a critical resource for real-time systems and must be managed effectively. Unfortunately, it is very difficult to design and implement systems that will guarantee that the appropriate outputs will be generated at the appropriate times under all possible conditions. To do this and make full use of all computing resources at all times is often impossible. For this reason, real-time systems are usually constructed using processors with considerable spare capacity, thereby ensuring that 'worst-case behavior' does not produce any unwelcome delays during critical periods of the system's operation. Given adequate processing power, language and run-time support are required to enable the programmer to

- specify times at which actions are to be performed,
- specify times at which actions are to be completed,
- respond to situations where all the timing requirements cannot be met,
- respond to situations where the timing requirements are changing dynamically.

These are called real-time control facilities. They enable the program to synchronise with time itself. For example, with digital control algorithms, it is necessary to sample readings from sensors at certain periods of the day, for example, 2pm, 3pm and so on, or at regular intervals, for instance, every 10 milliseconds (with analogue-to-digital converters, sample rates can vary from a few hundred hertz to several hundred megahertz). As a result of these readings, other actions will need to be performed. In order to meet response times, it is necessary for a system's

behaviour to be predictable. Providing these real-time facilities is one of the main goals of Ada and its Real-Time Systems Annex.

As well as being concurrent, real-time systems also have the following additional characteristics:

Large and complex. Real-time systems vary from simple single-processor embedded systems (consisting of a few hundred lines of code) to multi-platform, multi-language distributed systems (consisting of millions of lines of code). The issue of engineering large and complex systems is an important topic that Ada and its support environments do address. However, consideration of this area is beyond the scope of this book.

Extremely reliable and safe. Many real-time systems control some of society's critical systems such as air traffic control or chemical/power plants. The software must, therefore, be engineered to the highest integrity, and programs must attempt to tolerate faults and continue to operate (albeit perhaps providing a degraded service). In the worst case, a real-time system should make safe the environment before shutting down in a controlled manner. Unfortunately, some systems do not have easily available safe states when they are operational (for example, an unstable aircraft) consequently, continued operation in the presence of faults or damage is a necessity. Ada's design goals facilitate the design of reliable and robust programs. Its exception handling facility allows error recovery mechanisms to be activated. The Real-Time Systems Annex extends the core language to allow the flexible detection of common timing-related problems (such as missed deadlines).

Interaction with hardware interfaces. The nature of embedded systems requires the computer components to interact with the external world. They need to monitor sensors and control actuators for a wide variety of real-world devices. These devices interface to the computer via input and output registers, and their operational requirements are device and computer dependent. Devices may also generate interrupts to signal to the processor that certain operations have been performed or that error conditions have arisen. In the past, the interfacing to devices either has been left under the control of the operating system or has required the application programmer to resort to assembly language inserts to control and manipulate the registers and interrupts. Nowadays, because of the variety of devices and the time-critical nature of the associated interactions, their control must often be direct, and not through a layer of operating system functions. Furthermore, reliability requirements argue against the use of low-level programming techniques. Ada's representation items allow memory-mapped device registers to be accessed and interrupts to be handled by protected procedures.

Efficient implementation and a predictable execution environment. Since real-time systems are time-critical, efficiency of implementation will be more important than in other systems. It is interesting that one of the main benefits of using

a high-level language is that it enables the programmer to abstract away from implementation details and to concentrate on solving the problem at hand. Unfortunately, embedded computer systems programmers cannot afford this luxury. They must be constantly concerned with the cost of using particular language features. For example, if a response to some input is required within a microsecond, there is no point in using a language feature whose execution takes a millisecond! Ada makes predictability a primary concern in all its design trade-offs.

Chapters 12–17 discuss Ada's support for real-time systems in general, and also, where appropriate, how it facilitates the programming of efficient and reliable embedded systems.

1.3 Ada's time and clock facilities

Time values and clocks are used throughout this book to help manage interactions between concurrent activities and communication with the external environment. Consequently, this chapter concludes with a detailed discussion of the Ada facilities in this area.

To coordinate a program's execution with the natural time of the environment requires access to a hardware clock that approximates the passage of real time. For long-running programs (that is, years of non-stop execution), this clock may need to be resynchronised to some external standard (such as International Atomic Time) but from the program's point of view, the clock is the source of *real* time.

Ada provides access to this clock by providing several packages. The main section of the ARM (Ada Reference Manual) defines a compulsory library package called `Ada.Calendar` that provides an abstraction for 'wall clock' time that recognises leap years, leap seconds and other adjustments. Child packages support the notion of time zones, and provide arithmetic and formatting functions. In the Real-Time Systems Annex, a second representation is given that defines a monotonic (that is, non-decreasing) regular clock (package `Ada.Real_Time`). Both these representations should map down to the same hardware clock but cater for different application needs.

First consider package `Ada.Calendar`:

```
package Ada.Calendar is

  type Time is private;

  subtype Year_Number is Integer range 1901..2399;
  subtype Month_Number is Integer range 1..12;
  subtype Day_Number is Integer range 1..31;
  subtype Day_Duration is Duration range 0.0..86_400.0;
```

```
   function Clock return Time;

   function Year(Date:Time) return Year_Number;
   function Month(Date:Time) return Month_Number;
   function Day(Date:Time) return Day_Number;
   function Seconds(Date:Time) return Day_Duration;

   procedure Split(Date:in Time; Year:out Year_Number;
             Month:out Month_Number; Day:out Day_Number;
             Seconds:out Day_Duration);

   function Time_Of(Year:Year_Number; Month:Month_Number;
            Day:Day_Number;
            Seconds:Day_Duration := 0.0) return Time;

   function "+"(Left:Time;Right:Duration) return Time;
   function "+"(Left:Duration;Right:Time) return Time;
   function "-"(Left:Time;Right:Duration) return Time;
   function "-"(Left:Time;Right:Time) return Duration;

   function "<"(Left,Right:Time) return Boolean;
   function "<="(Left,Right:Time) return Boolean;
   function ">"(Left,Right:Time) return Boolean;
   function ">="(Left,Right:Time) return Boolean;

   Time_Error:exception;
   -- Time_Error may be raised by
   -- Time_Of, Split, Year, "+" and "-"

private
   ... -- not specified by the language
end Ada.Calendar;
```

A value of the private type `Time` is a combination of the date and the time of day, where the time of day is given in seconds from midnight. Seconds are described in terms of a subtype `Day_Duration` which is, in turn, defined by means of `Duration`. The fixed point type `Duration` is one of the predefined `Scalar` types and has a range which, although implementation dependent, must be at least –86_400.0 .. +86_400.0. The value 86_400 is the number of seconds in a day. The accuracy of `Duration` is also implementation dependent but the smallest representable value (`Duration'Small`) must not be greater than 20 milliseconds (it is recommended in the ARM that it is no greater than 100 microseconds).

The current time is returned by the function `Clock`. Conversion between `Time` and program accessible types, such as `Year_Number`, is provided by subprograms `Split` and `Time_Of`. In addition, some arithmetic and boolean operations are specified. Package `Calendar`, therefore, defines an appropriate structure for an abstract data type for time.

New to Ada 2005 is a child package of `Calendar` that provides further support

for arithmetic on time values. It is now possible to add and subtract a number of days to and from a time value (rather than express the interval as a duration).

```
package Ada.Calendar.Arithmetic is
   -- Arithmetic on days:
   type Day_Count is range
     -366*(1+Year_Number'Last - Year_Number'First) ..
     366*(1+Year_Number'Last - Year_Number'First);

   subtype Leap_Seconds_Count is Integer range -999..999;
   procedure Difference (Left, Right : in Time;
             Days : out Day_Count;
             Seconds : out Duration;
             Leap_Seconds : out Leap_Seconds_Count);

   function "+" (Left : Time; Right : Day_Count)
                 return Time;
   function "+" (Left : Day_Count; Right : Time)
                return Time;
-- similarly for "-"
end Ada.Calendar.Arithmetic;
```

Other new clock-related packages in Ada 2005 include a package to support the formatting of time values for input and output, and rudimentary support for time zones.

```
with Ada.Calendar.Time_Zones;
package Ada.Calendar.Formatting is
   type Day_Name is (Monday, Tuesday, Wednesday, Thursday,
       Friday, Saturday, Sunday);

   function Day_of_Week (Date : Time) return Day_Name;

   subtype Hour_Number is Natural range 0 .. 23;
   subtype Minute_Number is Natural range 0 .. 59;
   subtype Second_Number is Natural range 0 .. 59;
   subtype Second_Duration is Day_Duration range 0.0 .. 1.0;

   function Hour(Date : Time;
                 Time_Zone : Time_Zones.Time_Offset := 0)
                 return Hour_Number;
   ... -- similarly for Minute, Second, Sub_Second

   function Seconds_Of(Hour : Hour_Number; Minute : Minute_Number;
            Second : Second_Number := 0;
            Sub_Second : Second_Duration := 0.0)
                return Day_Duration;

   procedure Split(Seconds : in Day_Duration;
             Hour : out Hour_Number;
             Minute : out Minute_Number;
             Second : out Second_Number;
             Sub_Second : out Second_Duration);
```

```
   ... --  other variations

   function Image(Date : Time;
            Include_Time_Fraction : Boolean := False;
            Time_Zone  : Time_Zones.Time_Offset := 0)
                return String;

   function Value(Date : String;
            Time_Zone  : Time_Zones.Time_Offset := 0)
                return Time;
   function Image (Elapsed_Time : Duration;
            Include_Time_Fraction : Boolean := False)
                return String;
   function Value (Elapsed_Time : String) return Duration;
end Ada.Calendar.Formatting;
```

```
package Ada.Calendar.Time_Zones is

   -- Time zone manipulation:

   type Time_Offset is range -1440 .. 1440;

   Unknown_Zone_Error : exception;

   function UTC_Time_Offset (Date : Time := Clock)
           return Time_Offset;

end Ada.Calendar.Time_Zones;
```

The `Ada.Real_Time` package has a similar form to the `Ada.Calendar` package:

```
package Ada.Real_Time is
  type Time is private;
  Time_First: constant Time;
  Time_Last: constant Time;
  Time_Unit: constant := implementation-defined-real-number;
  Time_Unit : constant := 1.0;

  type Time_Span is private;
  Time_Span_First: constant Time_Span;
  Time_Span_Last: constant Time_Span;
  Time_Span_Zero: constant Time_Span;
  Time_Span_Unit: constant Time_Span;

  Tick: constant Time_Span;
  function Clock return Time;

  function "+" (Left: Time; Right: Time_Span) return Time;
  function "+" (Left: Time_Span; Right: Time) return Time;
  function "-" (Left: Time; Right: Time_Span) return Time;
  function "-" (Left: Time; Right: Time) return Time_Span;
```

```
  function "<" (Left, Right: Time) return Boolean;
  function "<="(Left, Right: Time) return Boolean;
  function ">" (Left, Right: Time) return Boolean;
  function ">="(Left, Right: Time) return Boolean;

  function "+" (Left, Right: Time_Span) return Time_Span;
  function "-" (Left, Right: Time_Span) return Time_Span;
  function "-" (Right: Time_Span) return Time_Span;
  function "/" (Left,Right : Time_Span) return Integer;
  function "/" (Left : Time_Span; Right : Integer)
           return Time_Span;
  function "*" (Left : Time_Span; Right : Integer)
           return Time_Span;
  function "*" (Left : Integer; Right : Time_Span)
           return Time_Span;

  function "<" (Left, Right: Time_Span) return Boolean;
  function "<="(Left, Right: Time_Span) return Boolean;
  function ">" (Left, Right: Time_Span) return Boolean;
  function ">="(Left, Right: Time_Span) return Boolean;

  function "abs"(Right : Time_Span) return Time_Span;

  function To_Duration (Ts : Time_Span) return Duration;
  function To_Time_Span (D : Duration) return Time_Span;

  function Nanoseconds  (NS: Integer) return Time_Span;
  function Microseconds (US: Integer) return Time_Span;
  function Milliseconds (MS: Integer) return Time_Span;

  type Seconds_Count is range implementation-defined;
  procedure Split(T : in Time; SC: out Seconds_Count;
                  TS : out Time_Span);
  function Time_Of(SC: Seconds_Count; TS: Time_Span)
           return Time;
private
  ... -- not specified by the language
end Ada.Real_Time;
```

The `Real_Time.Time` type represents time values as they are returned by `Real_Time.Clock`. The constant `Time_Unit` is the smallest amount of time representable by the `Time` type. The value of `Tick` must be no greater than one millisecond; the range of `Time` (from the epoch that represents the program's start-up) must be at least fifty years. Other important features of this time abstraction are described in the Real-Time Systems Annex; it is not necessary, for our purposes, to consider them in detail here.

To illustrate how the above packages could be used, consider the following code which tests to see if some sequence of statements executes within 1.7 seconds: