What is Ada?


HPI
Hasso
Plattner
Institut
| Engineering · Universität Potsdam

What we are telling today

This is not yet another Ada tutorial

Ada origins and history

Ada overview
Some Ada concepts

Ada runtime systems
Ravenscar
SPARK → Appendix

A personal journey with Ada

Getting started with Ada
Projects of interest for Railergy

This is not yet another Ada tutorial

The objective of this course is to provide some basic concepts and to spark some interest in Ada.

We might be off-topic sometimes, if so, it is fully intentional…

Ada is a very versatile language that was designed from scratch to cover (among others)

- Safety Critical Systems
- Secure Systems
- Realtime Applications
- Non- linear programming
- Distributed systems
- Very large and complex systems

Ada has been designed by a large industrial team against a given set of requirements, which was novel at the time.

We will provide some starting points for self- studying or systematic learning and teaching in academia at the end of the session.

RAILERGY

Who is the lady?

Augusta Ada King, Countess of Lovelace

(1815-1852)

English mathematician and writer, chiefly known for her work on Charles Babbage's proposed mechanical general-purpose computer, the Analytical Engine. She is believed by some to be the first to recognise that the machine had applications beyond pure calculation, and to have published the first algorithm intended to be carried out by such a machine. As a result, she is often regarded as the first to recognise the full potential of computers and as one of the first to be a computer programmer.

"The Analytical Engine might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine...Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent"

Above text from Wikipedia

Number of Operation.	Nature of Operation.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data.			Working Variables.									Result Variables.			
						$^1V_1$ ○ 0 0 1 [1]	$^1V_2$ ○ 0 0 2 [2]	$^1V_3$ ○ 0 0 4 [n]	$^0V_4$ ○ 0 0 0 [ ]	$^0V_5$ ○ 0 0 0 [ ]	$^0V_6$ ○ 0 0 0 [ ]	$^0V_7$ ○ 0 0 0 [ ]	$^0V_8$ ○ 0 0 0 [ ]	$^0V_9$ ○ 0 0 0 [ ]	$^0V_{10}$ ○ 0 0 0 [ ]	$^0V_{11}$ ○ 0 0 0 [ ]	$^0V_{12}$ ○ 0 0 0 [ ]	$^0V_{13}$ ..... ○ 0 0 0 [ ]	$^1V_{21}$ ○ B_1 in a decimal fraction. [B_1]	$^1V_{22}$ ○ B_3 in a decimal fraction. [B_3]	$^1V_{23}$ ○ B_5 in a decimal fraction. [B_5]	$^0V_{24}$... ○ 0 0 0 [B_7]
1	×	$^1V_2 \times {}^1V_3$	$^1V_4, {}^1V_5, {}^1V_6$	$\left\{\begin{matrix} {}^1V_2 = {}^1V_2 \\ {}^1V_3 = {}^1V_3 \end{matrix}\right\}$	$= 2n$	...	2	n	$2n$	$2n$	$2n$											
2	−	$^1V_4 - {}^1V_1$	$^2V_4$	$\left\{\begin{matrix} {}^1V_4 = {}^2V_4 \\ {}^1V_1 = {}^1V_1 \end{matrix}\right\}$	$= 2n-1$	1	...	...	$2n-1$													
3	+	$^1V_5 + {}^1V_1$	$^2V_5$	$\left\{\begin{matrix} {}^1V_5 = {}^2V_5 \\ {}^1V_1 = {}^1V_1 \end{matrix}\right\}$	$= 2n+1$	1	...	...	...	$2n+1$												
4	÷	$^2V_5 \div {}^2V_4$	$^1V_{11}$	$\left\{\begin{matrix} {}^2V_5 = {}^0V_5 \\ {}^2V_4 = {}^0V_4 \end{matrix}\right\}$	$= \dfrac{2n-1}{2n+1}$	...	...	...	0	0	...	...	...	...	...	$\dfrac{2n-1}{2n+1}$						
5	÷	$^1V_{11} \div {}^1V_2$	$^2V_{11}$	$\left\{\begin{matrix} {}^1V_{11} = {}^2V_{11} \\ {}^1V_2 = {}^1V_2 \end{matrix}\right\}$	$= \dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1}$	...	2	...	...	...	...	...	...	...	...	$\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1}$						
6	−	$^0V_{13} - {}^2V_{11}$	$^1V_{13}$	$\left\{\begin{matrix} {}^2V_{11} = {}^0V_{11} \\ {}^0V_{13} = {}^1V_{13} \end{matrix}\right\}$	$= -\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} = A_0$	...	...	...	...	...	...	...	...	...	...	0	.....	$-\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} = A_0$				
7	−	$^1V_3 - {}^1V_1$	$^1V_{10}$	$\left\{\begin{matrix} {}^1V_3 = {}^1V_3 \\ {}^1V_1 = {}^1V_1 \end{matrix}\right\}$	$= n-1\ (=3)$	1	...	n	...	...	...	...	...	...	$n-1$							
8	+	$^1V_2 + {}^0V_7$	$^1V_7$	$\left\{\begin{matrix} {}^1V_2 = {}^1V_2 \\ {}^0V_7 = {}^1V_7 \end{matrix}\right\}$	$= 2 + 0 = 2$	...	2	...	...	...	...	2										
9	÷	$^1V_6 \div {}^1V_7$	$^3V_{11}$	$\left\{\begin{matrix} {}^1V_6 = {}^1V_6 \\ {}^0V_{11} = {}^3V_{11} \end{matrix}\right\}$	$= \dfrac{2n}{2} = A_1$	...	...	...	...	...	$2n$	2	...	...	...	$\dfrac{2n}{2} = A_1$						
10	×	$^1V_{21} \times {}^3V_{11}$	$^1V_{12}$	$\left\{\begin{matrix} {}^1V_{21} = {}^1V_{21} \\ {}^3V_{11} = {}^3V_{11} \end{matrix}\right\}$	$= B_1 \cdot \dfrac{2n}{2} = B_1 A_1$	...	...	...	...	...	...	...	...	...	...	$\dfrac{2n}{2} = A_1$	$B_1 \cdot \dfrac{2n}{2} = B_1 A_1$	.....	B_1			
11	+	$^1V_{12} + {}^1V_{13}$	$^2V_{13}$	$\left\{\begin{matrix} {}^1V_{12} = {}^0V_{12} \\ {}^1V_{13} = {}^2V_{13} \end{matrix}\right\}$	$= -\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} + B_1 \cdot \dfrac{2n}{2}$ ...	...	...	...	...	...	...	...	...	...	...	.....	0	$\left\{-\dfrac{1}{2} \cdot \dfrac{2n-1}{2n+1} + B_1 \cdot \dfrac{2n}{2}\right\}$				
12	−	$^1V_{10} - {}^1V_1$	$^2V_{10}$	$\left\{\begin{matrix} {}^1V_{10} = {}^2V_{10} \\ {}^1V_1 = {}^1V_1 \end{matrix}\right\}$	$= n-2\ (=2)$	1	...	...	...	...	...	...	...	...	$n-2$							
13	−	$^1V_6 - {}^1V_1$	$^2V_6$	$\left\{\begin{matrix} {}^1V_6 = {}^2V_6 \\ {}^1V_1 = {}^1V_1 \end{matrix}\right\}$	$= 2n-1$	1	...	...	...	...	$2n-1$											
14	+	$^1V_1 + {}^1V_7$	$^2V_7$	$\left\{\begin{matrix} {}^1V_1 = {}^1V_1 \\ {}^1V_7 = {}^2V_7 \end{matrix}\right\}$	$= 2 + 1 = 3$	1	...	...	...	...	...	3										
15	÷	$^2V_6 \div {}^2V_7$	$^1V_8$	$\left\{\begin{matrix} {}^2V_6 = {}^2V_6 \\ {}^2V_7 = {}^2V_7 \end{matrix}\right\}$	$= \dfrac{2n-1}{3}$	...	...	...	...	...	$2n-1$	3	$\dfrac{2n-1}{3}$									
16	×	$^1V_8 \times {}^3V_{11}$	$^4V_{11}$	$\left\{\begin{matrix} {}^1V_8 = {}^0V_8 \\ {}^3V_{11} = {}^4V_{11} \end{matrix}\right\}$	$= \dfrac{2n}{2} \cdot \dfrac{2n-1}{3}$	...	...	...	...	...	...	...	0	...	...	$\dfrac{2n}{2} \cdot \dfrac{2n-1}{3}$						
17	−	$^2V_6 - {}^1V_1$	$^3V_6$	$\left\{\begin{matrix} {}^2V_6 = {}^3V_6 \\ {}^1V_1 = {}^1V_1 \end{matrix}\right\}$	$= 2n-2$	1	...	...	...	...	$2n-2$											
18	+	$^1V_1 + {}^2V_7$	$^3V_7$	$\left\{\begin{matrix} {}^2V_7 = {}^3V_7 \\ {}^1V_1 = {}^1V_1 \end{matrix}\right\}$	$= 3 + 1 = 4$	1	...	...	...	...	...	4										
19	÷	$^3V_6 \div {}^3V_7$	$^1V_9$	$\left\{\begin{matrix} {}^3V_6 = {}^3V_6 \\ {}^3V_7 = {}^3V_7 \end{matrix}\right\}$	$= \dfrac{2n-2}{4}$	...	...	...	...	...	$2n-2$	4	...	$\dfrac{2n-2}{4}$	...	$\left\{\dfrac{2n}{2} \cdot \dfrac{2n-1}{3} \cdot \dfrac{2n-2}{3} = A_3\right\}$						
20	×	$^1V_9 \times {}^4V_{11}$	$^5V_{11}$	$\left\{\begin{matrix} {}^1V_9 = {}^0V_9 \\ {}^4V_{11} = {}^5V_{11} \end{matrix}\right\}$	$= \dfrac{2n}{2} \cdot \dfrac{2n-1}{3} \cdot \dfrac{2n-2}{4} = A_3$	...	...	...	...	...	...	...	...	0								
21	×	$^1V_{22} \times {}^5V_{11}$	$^0V_{12}$	$\left\{\begin{matrix} {}^1V_{22} = {}^1V_{22} \\ {}^0V_{12} = {}^2V_{12} \end{matrix}\right\}$	$= B_3 \cdot \dfrac{2n}{2} \cdot \dfrac{2n-1}{3} \cdot \dfrac{2n-2}{3} = B_3 A_3$	...	...	...	...	...	...	...	...	...	...	0	$B_3 A_3$	.....	.....	B_3		
22	+	$^2V_{12} + {}^2V_{13}$	$^3V_{13}$	$\left\{\begin{matrix} {}^2V_{12} = {}^0V_{12} \\ {}^2V_{13} = {}^3V_{13} \end{matrix}\right\}$	$= A_0 + B_1 A_1 + B_3 A_3$	...	...	...	...	...	...	...	...	...	...	.....	0	$\left\{A_3 + B_1 A_1 + B_3 A_3\right\}$			
23		$^2V_{10} - {}^1V_1$	$^3V_{10}$	$\left\{\begin{matrix} {}^2V_{10} = {}^3V_{10} \end{matrix}\right\}$	$= n-3\ (=1)$	1									$n-3$							

Note G by Ada Lovelace

Bernoulli algorithm

Ada origins

1815-1835	Augusta Ada Byron is born, the daughter of Lord Byron, the poet, and Anne Isabella Millbanke, the mathematician. Ada marries and becomes the Countess of Lovelace.
1833-1852	Ada Lovelace meets and then collaborates with Charles Babbage, the inventor of the Difference Engine and the Analytical Engine. She writes mathematical treatises on how computing machines might be used. (See "Ada, the Enchantress of Numbers" [Toole92].)
1975-1978	The U.S. Department of Defense (DoD) confronts the "software crisis" and sponsors a series of studies leading to a decision to create a standard language for use in military embedded or mission-critical systems. The [Steelman78] document is issued, stating requirements to be satisfied by the new language. A request for proposals is issued.
1978-1979	Four competitors are selected (Blue, Green, Red, Yellow) for the first phase of a design competition. All four select Pascal as the base language upon which to build, and submit initial design proposals. The competition is narrowed to the Green and Red teams, who then refine and expand their design proposals.
1980	The name, Ada, is selected for the new language, honoring the person who is now considered to be the world's first computer programmer.
1979-1983	The Green team (headed by Jean Ichbiah of France, from Bull) wins and produces document MIL-STD-1815A, the Ada Language Reference Manual. [ARM83] (Note use of the year of Ada's birth.)
1995, 2005, 2012	The language has evolved as an ANSI standard in several versions (Ada95, Ada2005, Ada2012)

Ada main features

Object orientated programming
Strong typing
Abstractions to fit program domain
Generic programming/templates
Exception handling
Facilities for modular organization of code
Standard libraries for I/O, string handling, numeric computing, containers
Systems programming
Concurrent programming
Real-time programming
Distributed systems programming
Numeric processing
Interfaces to other languages (C, COBOL, Fortran)

RAILERGY

Hello World!

```
1  with Ada.Text_IO;
2
3  procedure Greet is
4  begin
5     --  Print "Hello, World!" to the screen
6     Ada.Text_IO.Put_Line ("Hello, World!");
7  end Greet;
8
```

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Greet is
4  begin
5     --  Print "Hello, World!" to the screen
6     Put_Line ("Hello, World!");
7  end Greet;
8
```

A subprogram in Ada can be either a procedure or a function.
A procedure, as illustrated above, does not return a value when called.
with is used to reference external modules that are needed in the procedure.

This version utilizes an Ada feature known as a use clause, which has the form use *package-name*. As illustrated by the call on Put_Line, the effect is that entities from the named package can be referenced directly, without the *package-name.* prefix

Ada Packages

Packages

Packages let you make your code modular, separating your programs into semantically significant units. Additionally the separation of a package's specification from its body (which we will see below) can reduce compilation time.

While the with clause indicates a dependency, you can see in the example above that you still need to prefix the referencing of entities from the Week package by the name of the package. (If we had included a use Week clause, then such a prefix would not have been necessary.)

Accessing entities from a package uses the dot notation, A.B, which is the same notation as the one used to access record fields.

A with clause can only appear in the prelude of a compilation unit (i.e., before the reserved word, such as procedure, that marks the beginning of the unit). It is not allowed anywhere else. This rule is only needed for methodological reasons: the person reading your code should be able to see immediately which units the code depends on.

week.ads — ACAD

```
1
2  package Week is
3
4     Mon : constant String := "Monday";
5     Tue : constant String := "Tuesday";
6     Wed : constant String := "Wednesday";
7     Thu : constant String := "Thursday";
8     Fri : constant String := "Friday";
9     Sat : constant String := "Saturday";
10    Sun : constant String := "Sunday";
11
12 end Week;
13
```

Line: 14 Ada Tab Size: 4 Week

main.adb — ACAD

```
1
2  with Ada.Text_IO; use Ada.Text_IO;
3  with Week;
4  --  References the Week package, and adds a
       dependency from Main
5  --  to Week
6
7  procedure Main is
8  begin
9     Put_Line ("First day of the week is " &
Week.Mon);
10 end Main;
11
```

Line: 10:10 Ada Tab Size: 4 Symbols

Ada packages vs. C/C++ header files

Packages look similar to, but are semantically very different from, header files in C/C++.

The first and most important distinction is that packages are a language-level mechanism. This is in contrast to a #include'd header file, which is a functionality of the C preprocessor.

An immediate consequence is that the with construct is a semantic inclusion mechanism, not a text inclusion mechanism. Hence, when you with a package, you are saying to the compiler "I'm depending on this semantic unit", and not "include this bunch of text in place here".

The effect of a package thus does not vary depending on where it has been withed from. Contrast this with C/C++, where the meaning of the included text depends on the context in which the #include appears.

This allows compilation/recompilation to be more efficient. It also allows tools like IDEs to have correct information about the semantics of a program. In turn, this allows better tooling in general, and code that is more analyzable, even by humans.

An important benefit of Ada with clauses when compared to #include is that it is stateless. The order of with and use clauses does not matter, and can be changed without side effects.

Some aspects of the Ada type system

Strong typing

Ada's type system allows the programmer to construct powerful abstractions that represent the real world, and to provide valuable information to the compiler, so that the compiler can find many logic or design errors before they become bugs. It is at the heart of the language, and good Ada programmers learn to use it to great advantage. Four principles govern the type system:

Strong typing: types are incompatible with one another, so it is not possible to mix apples and oranges. There are, however, ways to convert between types.

Static typing: type checked while compiling, this allows type errors to be found earlier.

Abstraction: types represent the real world or the problem at hand; not how the computer represents the data internally. There are ways to specify exactly how a type must be represented at the bit level, but we will defer that discussion to another chapter.

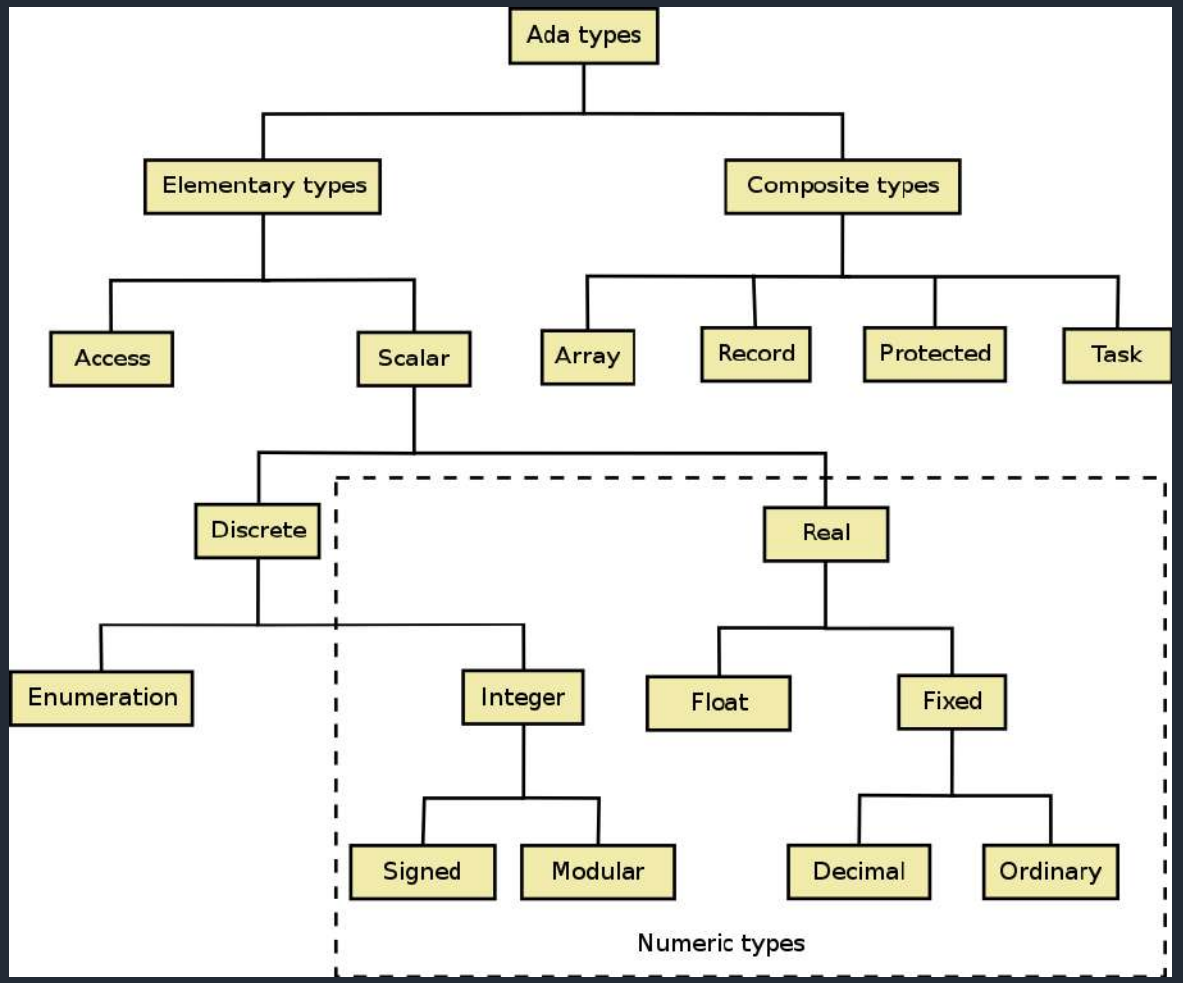
Name equivalence, as opposed to structural equivalence used in most other languages. Two types are compatible if and only if they have the same name; not if they just happen to have the same size or bit representation. You can thus declare two integer types with the same ranges that are totally incompatible, or two record types with exactly the same components, but which are incompatible.

Types are incompatible with one another. However, each type can have any number of subtypes, which are compatible with their base type and may be compatible with one another. See below for examples of subtypes which are incompatible with one another.

RAILERGY™

Ada type hierarchy

Types are organized hierarchically. A type inherits properties from types above it in the hierarchy. For example, all scalar types (integer, enumeration, modular, fixed-point and floating-point types) have operators "<", ">" and arithmetic operators defined for them, and all discrete types can serve as array indexes.



RAILERGY

Classification of types (some examples…)

Constrained vs. Unconstrained

```
1   type I is range 1 .. 10;           -- constrained
2   type AC is array (1 .. 10) of ...  -- constrained
3
4
5   type AU is array (I range <>) of ...          -- unconstrained
6   type R (X: Discriminant [:= Default]) is ...  -- unconstrained
```

By giving a constraint to an unconstrained subtype, a subtype or object becomes constrained:

```
5   subtype RC is R (Value);  -- constrained subtype of R
6   OC: R (Value);            -- constrained object of anonymous constrained subtype of R
7   OU: R;                    -- unconstrained object
8
```

RAILERGY

Defining new types and subtypes

```
type T is..
```

followed by the description of the type, as explained in detail in each category of type.

Formally, the above declaration creates a type and its first subtype named T. The type itself, correctly called the "type of T", is anonymous; the RM refers to it as *T* (in italics), but often speaks sloppily about the type T. But this is an academic consideration; for most purposes, it is sufficient to think of T as a type. For scalar types, there is also a base type called T'Base, which encompasses all values of T.

For signed integer types, the type of T comprises the (complete) set of mathematical integers. The base type is a certain hardware type, symmetric around zero (except for possibly one extra negative value), encompassing all values of T.

As explained above, all types are incompatible; thus:

```
type Integer_1 is range 1 .. 10;
type Integer_2 is range 1 .. 10;
A : Integer_1 := 8;
B : Integer_2 := A; -- illegal!
```

is illegal, because Integer_1 and Integer_2 are different and incompatible types. It is this feature which allows the compiler to detect logic errors at compile time, such as adding a file descriptor to a number of bytes, or a length to a weight. The fact that the two types have the same range does not make them compatible: this is name equivalence in action, as opposed to structural equivalence.

RAILERGY™

Defining new types and subtypes

```
10  type Integer_1 is range 1 .. 10;
11  subtype Integer_2 is Integer_1      range 7 .. 11;  -- bad
12  subtype Integer_3 is Integer_1'Base range 7 .. 11;  -- OK
13  A : Integer_1 := 8;
14  B : Integer_3 := A; -- OK
```

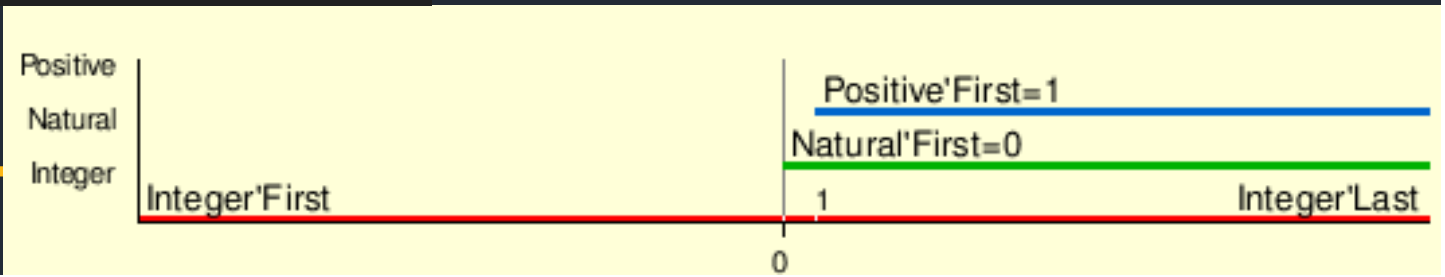
The declaration of Integer_2 is bad because the constraint 7 .. 11 is not compatible with Integer_1; it raises Constraint_Error at subtype elaboration time.

Integer_1 and Integer_3 are compatible because they are both subtypes of the same type, namely Integer_1'Base.

It is not necessary that the subtype ranges overlap, or be included in one another. The compiler inserts a run-time range check when you assign A to B; if the value of A, at that point, happens to be outside the range of Integer_3, the program raises Constraint_Error.

There are a few predefined subtypes which are very useful:

```
16  subtype Natural  is Integer range 0 .. Integer'Last;
17  subtype Positive is Integer range 1 .. Integer'Last;
18
```



Derived types

A derived type is a new, full-blown type created from an existing one. Like any other type, it is incompatible with its parent; however, it inherits the primitive operations defined for the parent type.

```
18
19  type Integer_1 is range 1 .. 10;
20  type Integer_2 is new Integer_1 range 2 .. 8;
21  A : Integer_1 := 8;
22  B : Integer_2 := A; -- illegal!
23
```

Here both types are discrete; it is mandatory that the range of the derived type be included in the range of its parent. Contrast this with subtypes. The reason is that the derived type inherits the primitive operations defined for its parent, and these operations assume the range of the parent type.

Inheritance of primitive operations

More Object Oriented concepts (tagged types, classwide types, dispatching operations, and others are available. But for safety critical systems, dynamic OO concepts are very restricted in use)

Here we use the type Weekend_Days derived from the type Week in order to inherit primitive behaviour from the related subprogram

A subprogram will only become a primitive of the type if:

The subprogram is declared in the same scope as the type and

The type and the subprogram are declared in a package

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Primitives is
4    package Week is
5      type Days is (Monday, Tuesday, Wednesday, Thursday,
6                    Friday, Saturday, Sunday);
7
8       --  Print day is a primitive of the type Days
9      procedure Print_Day (D : Days);
10   end Week;
11
12   package body Week is
13     procedure Print_Day (D : Days) is
14     begin
15        Put_Line (Days'Image (D));
16     end Print_Day;
17   end Week;
18
19   use Week;
20   type Weekend_Days is new Days range Saturday .. Sunday;
21
22   --  A procedure Print_Day is automatically inherited here. It is as if
23   --  the procedure
24   --
25   --  procedure Print_Day (D : Weekend_Days);
26   --
27   --  has been declared with the same body
28
29   Sat : Weekend_Days := Saturday;
30 begin
31    Print_Day (Sat);
32 end Primitives;
```

Limited Types

Limiting a type means disallowing assignment Programmers can define their own types to be limited, too, like this:

```
1
2   type T is limited …;
3
```

(The ellipsis stands for private, or for a record definition, these concepts will not be discussed in detail in this session)

A limited type also doesn't have an equality operator unless the programmer defines one.

The “concurrency types” described on the next slides are always limited.

Concurrency Types

The Ada language uses types for one more purpose in addition to classifying data + operations. The type system integrates concurrency (threading, parallelism). Programmers will use types for expressing the concurrent threads of control of their programs.

The core pieces of this part of the type system are the **task types** and the **protected types.**

Task types are limited, i.e. they are restricted in the same way as limited private types, so assignment and comparison are not allowed.

RAILERGY™

Task types and Tasks

Tasks

```
1
2  task Single is
3       declarations of exported identifiers
4    end Single;
5    ...
6    task body Single is
7       local declarations and statements
8    end Single;
9
```

A *task unit* is a program unit that is obeyed concurrently with the rest of an Ada program. The corresponding activity, a new locus of control, is called a *task* in Ada terminology, and is similar to a *thread*, for example in Java Threads. The execution of the main program is also a task, the anonymous environment task. A task unit has both a declaration and a body, which is mandatory. A task body may be compiled separately as a subunit, but a task may not be a library unit, nor may it be generic. Every task depends on a *master*, which is the immediately surrounding declarative region - a block, a subprogram, another task, or a package. The execution of a master does not complete until all its dependent tasks have terminated. The environment task is the master of all other tasks; it terminates only when all other tasks have terminated.

Task units are similar to packages in that a task declaration defines entities exported from the task, whereas its body contains local declarations and statements of the task.

Tasks example

```
2  procedure Housekeeping is
3
4      task Check_CPU;
5      task Backup_Disk;
6
7      task body Check_CPU is
8         ...
9      end Check_CPU;
10
11     task body Backup_Disk is
12        ...
13     end Backup_Disk;
14     -- the two tasks are automatically created and begin execution
15  begin -- Housekeeping
16     null;
17     -- Housekeeping waits here for them to terminate
18  end Housekeeping;
```

RAILERGY™

Task types

```
1
2  task type T is
3     ...
4  end T;
5  ...
6  Task_1, Task_2 : T;
7  ...
8  task body T is
9     ...
10 end T;
11
```

It is possible to declare task types, thus allowing task units to be created dynamically, and incorporated in data structures

Rendezvous

This task type implements a single-slot buffer, i.e. an encapsulated variable that can have values inserted and removed in strict alternation. Note that the buffer task has no need of state variables to implement the buffer protocol: the alternation of insertion and removal operations is directly enforced by the control structure in the body of Encapsulated_Buffer_Task_Type which is, as is typical, a loop.

```
1
2  task type Encapsulated_Buffer_Task_Type is
3     entry Insert (An_Item : in  Item);
4     entry Remove (An_Item : out Item);
5  end Encapsulated_Buffer_Task_Type;
6  ...
7  Buffer_Pool : array (0 .. 15) of Encapsulated_Buffer_Task_Type;
8  This_Item   : Item;
9  ...
10 task body Encapsulated_Buffer_Task_Type is
11    Datum : Item;
12 begin
13    loop
14       accept Insert (An_Item : in  Item) do
15          Datum := An_Item;
16       end Insert;
17       accept Remove (An_Item : out Item) do
18          An_Item := Datum;
19       end Remove;
20    end loop;
21 end Encapsulated_Buffer_Task_Type;
22 ...
23 Buffer_Pool(1).Remove (This_Item);
24 Buffer_Pool(2).Insert (This_Item);
```

RAILERGY

Rendezvous explained

```
1
2  task type Encapsulated_Buffer_Task_Type is
3     entry Insert (An_Item : in  Item);
4     entry Remove (An_Item : out Item);
5  end Encapsulated_Buffer_Task_Type;
6  ...
7  Buffer_Pool : array (0 .. 15) of Encapsulated_Buffer_Task_Type;
8  This_Item   : Item;
9  ...
10 task body Encapsulated_Buffer_Task_Type is
11    Datum : Item;
12 begin
13    loop
14       accept Insert (An_Item : in  Item) do
15          Datum := An_Item;
16       end Insert;
17       accept Remove (An_Item : out Item) do
18          An_Item := Datum;
19       end Remove;
20    end loop;
21 end Encapsulated_Buffer_Task_Type;
22 ...
23 Buffer_Pool(1).Remove (This_Item);
24 Buffer_Pool(2).Insert (This_Item);
```

The only entities that a task may export are entries.

An entry looks much like a procedure. It has an identifier and may have in, out or in out parameters.

Ada supports communication from task to task by means of the entry call. Information passes between tasks through the actual parameters of the entry call. We can encapsulate data structures within tasks and operate on them by means of entry calls, in a way analogous to the use of packages for encapsulating variables.

The main difference is that an entry is executed by the called task, not the calling task, which is suspended until the call completes. If the called task is not ready to service a call on an entry, the calling task waits in a (FIFO) queue associated with the entry.

This interaction between calling task and called task is known as a rendezvous. The calling task requests rendezvous with a specific named task by calling one of its entries. A task accepts rendezvous with any caller of a specific entry by executing an accept statement for the entry.

If no caller is waiting, it is held up. Thus entry call and accept statement behave symmetrically.

Selective Wait (1)

To avoid being held up when it could be doing productive work, a server task often needs the freedom to accept a call on any one of a number of alternative entries. It does this by means of the *selective wait* statement, which allows a task to wait for a call on any of two or more entries.

If only one of the alternatives in a selective wait statement has a pending entry call, then that one is accepted. If two or more alternatives have calls pending, the implementation is free to accept any one of them. For example, it could choose one at random. This introduces *bounded non-determinism* into the program. A sound Ada program should not depend on a particular method being used to choose between pending entry calls.

```
2  task type Encapsulated_Variable_Task_Type is
3      entry Store (An_Item : in  Item);
4      entry Fetch (An_Item : out Item);
5    end Encapsulated_Variable_Task_Type;
6    ...
7    task body Encapsulated_Variable_Task_Type is
8       Datum : Item;
9    begin
10      accept Store (An_Item : in Item) do
11         Datum := An_Item;
12      end Store;
13      loop
14         select
15            accept Store (An_Item : in Item) do
16               Datum := An_Item;
17            end Store;
18         or
19            accept Fetch (An_Item : out Item) do
20               An_Item := Datum;
21            end Fetch;
22         end select;
23      end loop;
24   end Encapsulated_Variable_Task_Type;
```

RAILERGY™

Selective Wait (2)

```
2  task type Encapsulated_Variable_Task_Type is
3      entry Store (An_Item : in  Item);
4      entry Fetch (An_Item : out Item);
5  end Encapsulated_Variable_Task_Type;
6  ...
7  task body Encapsulated_Variable_Task_Type is
8     Datum : Item;
9  begin
10    accept Store (An_Item : in Item) do
11       Datum := An_Item;
12    end Store;
13    loop
14       select
15          accept Store (An_Item : in Item) do
16             Datum := An_Item;
17          end Store;
18       or
19          accept Fetch (An_Item : out Item) do
20             An_Item := Datum;
21          end Fetch;
22       end select;
23    end loop;
24 end Encapsulated_Variable_Task_Type;
```

```
26
27  x, y : Encapsulated_Variable_Task_Type;
28
```

creates two variables of type Encapsulated_Variable_Task_Type. They can be used thus:

```
29
30  it : Item;
31  ...
32  x.Store(Some_Expression);
33  ...
34  x.Fetch (it);
35  y.Store (it);
```

Guards

Depending on circumstances, a server task may not be able to accept calls for some of the entries that have accept alternatives in a selective wait statement. The acceptance of any alternative can be made conditional by using a *guard*, which is *Boolean* precondition for acceptance. This makes it easy to write monitor-like server tasks, with no need for an explicit signaling mechanism, nor for mutual exclusion. An alternative with a True guard is said to be *open*. It is an error if no alternative is open when the selective wait statement is executed, and this raises the Program_Error exception.

```
3
4  task Cyclic_Buffer_Task_Type is
5      entry Insert (An_Item : in  Item);
6      entry Remove (An_Item : out Item);
7  end Cyclic_Buffer_Task_Type;
8  ...
9  task body Cyclic_Buffer_Task_Type is
10     Q_Size : constant := 100;
11     subtype Q_Range is Positive range 1 .. Q_Size;
12     Length : Natural range 0 .. Q_Size := 0;
13     Head, Tail : Q_Range := 1;
14     Data : array (Q_Range) of Item;
15 begin
16     loop
17        select
18           when Length < Q_Size =>
19              accept Insert (An_Item : in  Item) do
20                 Data(Tail) := An_Item;
21              end Insert;
22              Tail := Tail mod Q_Size + 1;
23              Length := Length + 1;
24        or
25           when Length > 0 =>
26              accept Remove (An_Item : out Item) do
27                 An_Item := Data(Head);
28              end Remove;
29              Head := Head mod Q_Size + 1;
30              Length := Length - 1;
31        end select;
32     end loop;
33 end Cyclic_Buffer_Task_Type;
34
```

Ada Exceptions

RAILERGY

Exception declaration

Ada uses exceptions for error handling. Unlike many other languages, Ada speaks about *raising*, not *throwing*, an exception and *handling*, not *catching*, an exception.

Ada exceptions are not types, but instead objects, which may be peculiar to you if you're used to the way Java or Python support exceptions.

RAILERGY

Raising an exception

exceptions.ads — ACAD

```
1  package Exceptions is
2      My_Except : exception;
3      --  Like an object. *NOT* a type !
4  end Exceptions;
5
```

Line: 5 | Ada | Tab Size: 4 | Excep...

main.adb — ACAD

```
1   with Exceptions; use Exceptions;
2
3   procedure Main is
4   begin
5      raise My_Except;
6      --  Execution of current control flow abandoned; an exception of kind
7      --  "My_Except" will bubble up until it is caught.
8
9      raise My_Except with "My exception message";
10     --  Execution of current control flow abandoned; an exception of
11     --  kind "My_Except" with associated string will bubble up until
12     --  it is caught.
13  end Main;
14
15
```

Line: 13:10 | Ada | Tab Size: 4 | Symbols

Ada Runtime Library

Ada Runtime Library

The run-time library for a language is typically the standard library of functions responsible for implementing the interface with the underlying functionality exposed by the system. These are normally statically linked against the executable at compile-time.

The Ada run-time library is responsible for the implementation of the standard library defined in annexes A-H of the Ada Language Reference Manual. Not all annexes defined in the standard are required to be implemented for a specific platform.

See also: http://ada-auth.org/standards/12rm/html/RM-TOC.html

Appendix A: Predefined Language Environment

Appendix B: Interface to Other Languages

Appendix C: Systems Programming

Appendix D: Real-Time Systems

Appendix E: Distributed Systems

Appendix F: Information Systems

Appendix G: Numerics

Appendix H: High Integrity Systems


RAILERGY
Ravenscar

Ravenscar profile

The **Ravenscar profile** is a subset of the Ada tasking features designed for safety-critical hard real-time computing. It was defined by a separate technical report in Ada 95; it is now part of the Ada 2012 Standard. It has been named after the English village of Ravenscar, the location of the 8th International Real-Time Ada Workshop (IRTAW 8).

A Ravenscar Ada application uses the following compiler directive:

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

This is the same as writing the following set of configuration pragmas:

```
1  pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
2  pragma Locking_Policy (Ceiling_Locking);
3  pragma Detect_Blocking;
4  pragma Restrictions (
5                  No_Abort_Statements,
6                  No_Dynamic_Attachment,
7                  No_Dynamic_Priorities,
8                  No_Implicit_Heap_Allocations,
9                  No_Local_Protected_Objects,
10                 No_Local_Timing_Events,
11                 No_Protected_Type_Allocators,
12                 No_Relative_Delay,
13                 No_Requeue_Statements,
14                 No_Select_Statements,
15                 No_Specific_Termination_Handlers,
16                 No_Task_Allocators,
17                 No_Task_Hierarchy,
18                 No_Task_Termination,
19                 Simple_Barriers,
20                 Max_Entry_Queue_Length => 1,
21                 Max_Protected_Entries  => 1,
22                 Max_Task_Entries       => 0,
23                 No_Dependence => Ada.Asynchronous_Task_Control,
24                 No_Dependence => Ada.Calendar,
25                 No_Dependence => Ada.Execution_Time.Group_Budget,
26                 No_Dependence => Ada.Execution_Time.Timers,
27                 No_Dependence => Ada.Task_Attributes);
```

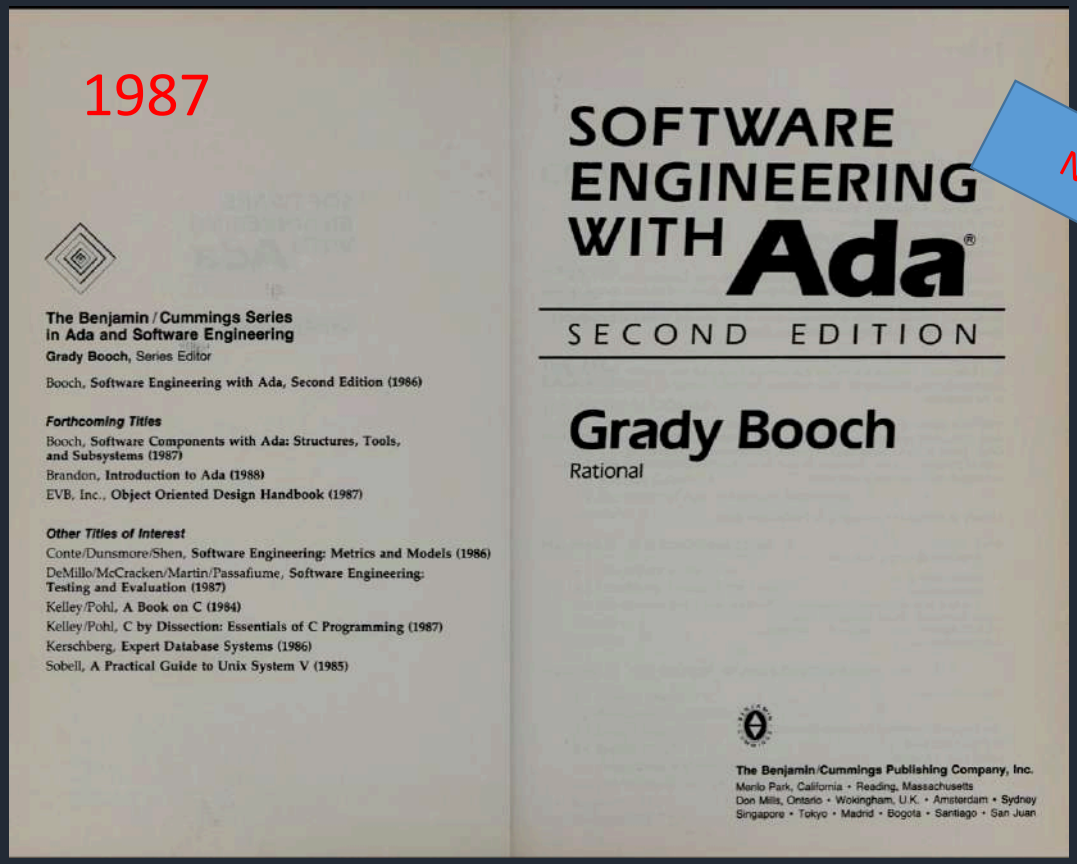
RAILERGY™

Ada and Jakob 

RAILERGY

A personal voyage with Ada (1987- ….)

1987



The Benjamin / Cummings Series
in Ada and Software Engineering
Grady Booch, Series Editor

Booch, Software Engineering with Ada, Second Edition (1986)

Forthcoming Titles

Booch, Software Components with Ada: Structures, Tools, and Subsystems (1987)
Brandon, Introduction to Ada (1988)
EVB, Inc., Object Oriented Design Handbook (1987)

Other Titles of Interest

Conte/Dunsmore/Shen, Software Engineering: Metrics and Models (1986)
DeMillo/McCracken/Martin/Passafiume, Software Engineering: Testing and Evaluation (1987)
Kelley/Pohl, A Book on C (1984)
Kelley/Pohl, C by Dissection: Essentials of C Programming (1987)
Kerschberg, Expert Database Systems (1986)
Sobell, A Practical Guide to Unix System V (1985)

SOFTWARE ENGINEERING WITH Ada®

SECOND EDITION

Grady Booch
Rational

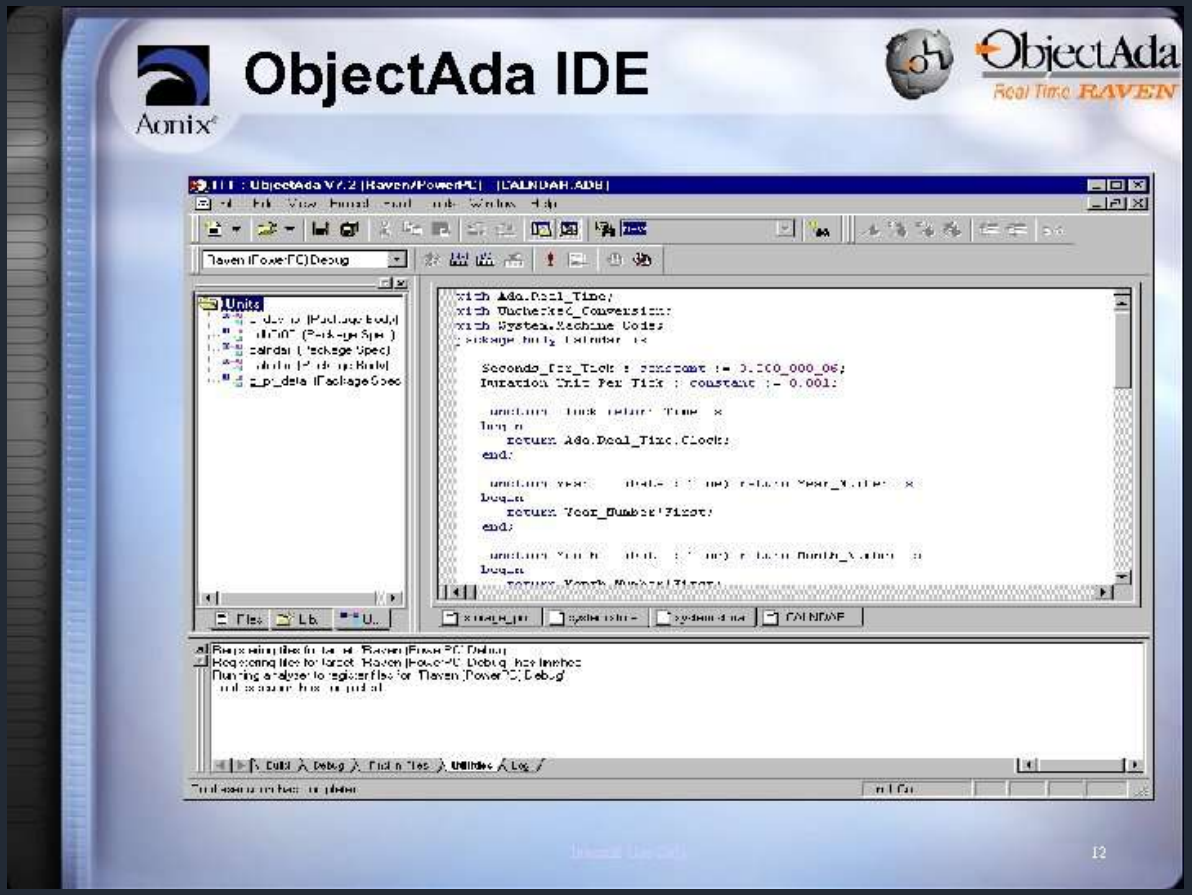
The Benjamin/Cummings Publishing Company, Inc.
Menlo Park, California • Reading, Massachusetts
Don Mills, Ontario • Wokingham, U.K. • Amsterdam • Sydney
Singapore • Tokyo • Madrid • Bogota • Santiago • San Juan

NO WAY

1990

RAILERGY™

A personal voyage with Ada (1995- ….)



Windows NT !!!

RAILERGY

A personal voyage with Ada (2010- ….)

Esterel Technologies SCADE KCG is an automatic code generator, which is qualifiable to DO-178C DAL A, IEC 61508 SIL, EN50128 SIL3/4, ISO26262 ASIL D (and others)

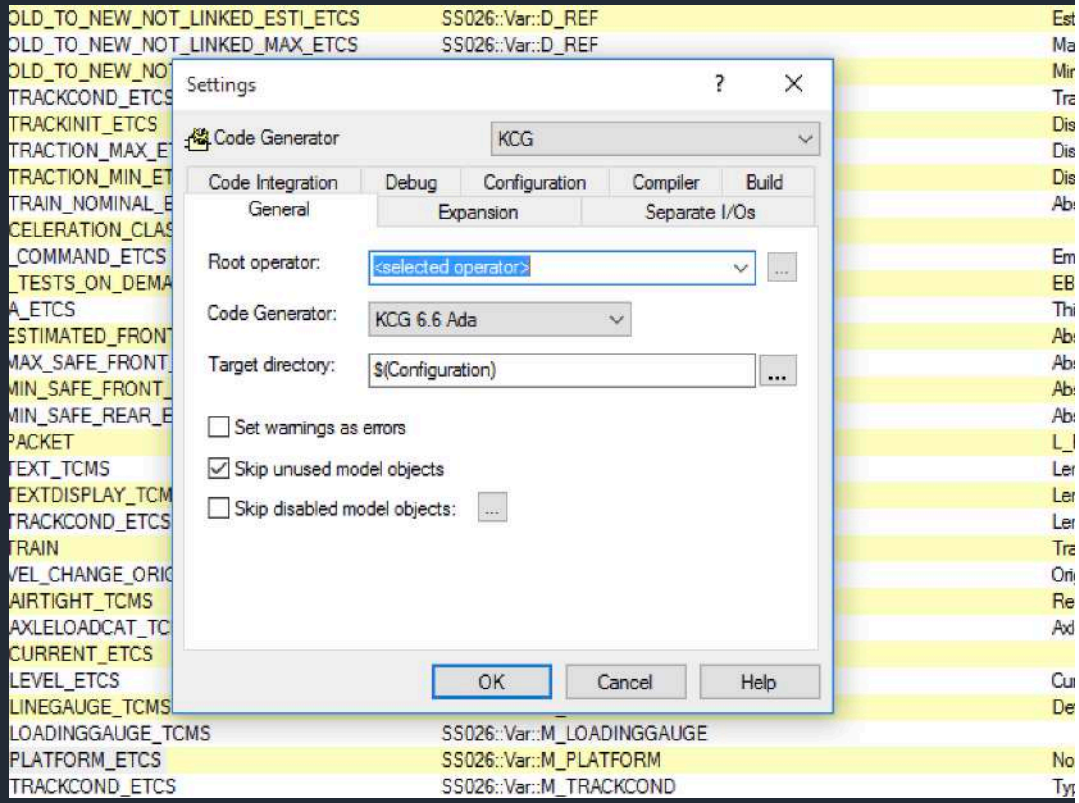
Since 2012, there is an Ada / SPARK backend.

The requirements were developed with the following SCADE + Ada users:

- Rolls Royce Aero Engines, Derby UK (FADEC for commercial turbofans)
- BAE Systems, Rochester UK (Eurofighter Typhoon)
- Alstom, Villeurbanne F (Railway/ Signalling)

and partner companies:

- Praxis (now Altran), Bath UK, developers of SPARK language
- AdaCore, Paris F, maintaining GNAT

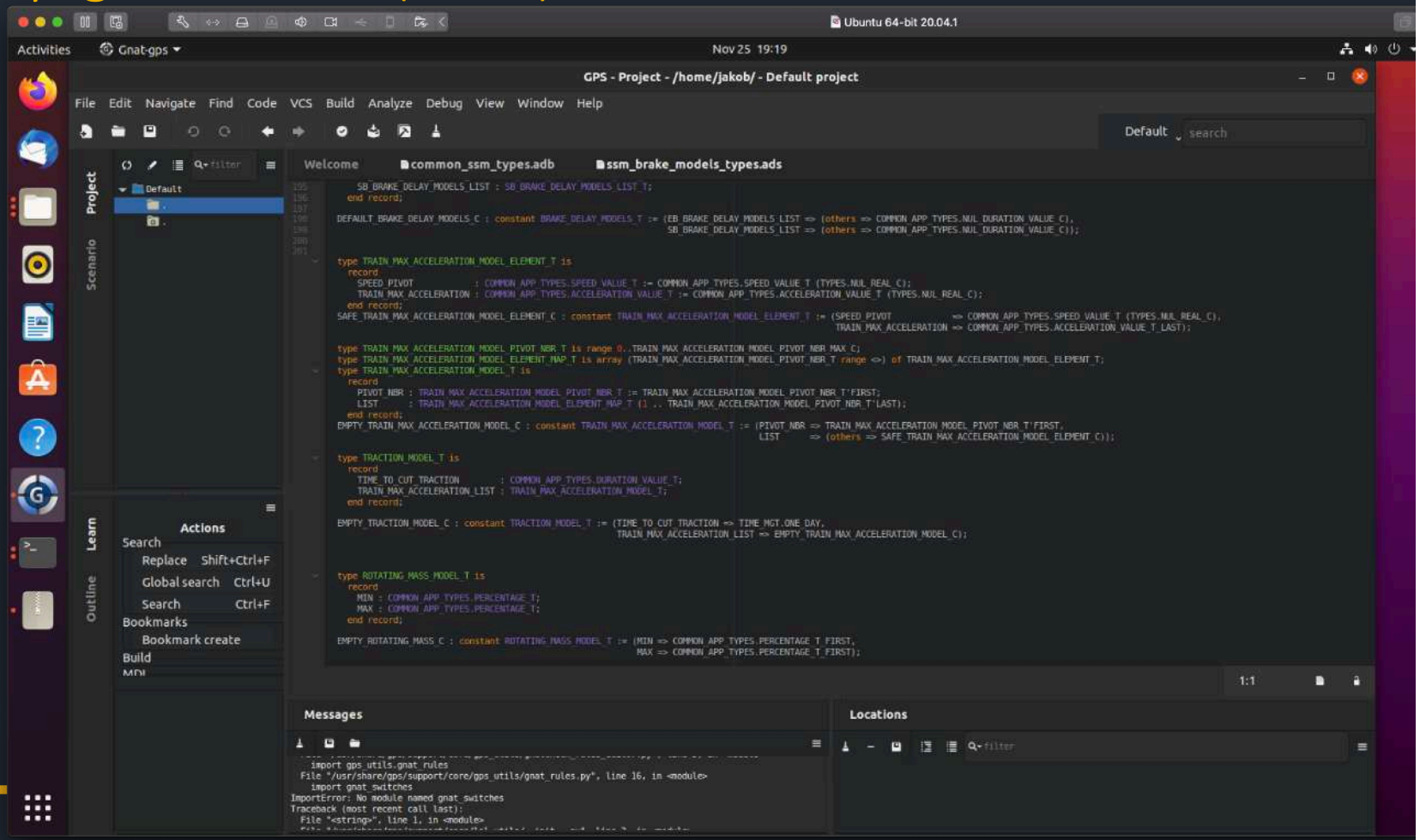


RAILERGY

A personal voyage with Ada (2020)

Analysis of
ALSTOM ETCS Core
Software for DB

(retrofit for
ICE T/ 1/ 3)



Getting started with Ada

Learning Ada

A good starting point is the *Ada information clearinghouse*

https://www.adaic.org/learn/materials/ for a list of books and online tutorials

The *Ada information clearinghouse* also maintains a list of free and commercial tools and libraries

Universities in Germany teaching Ada

Karlsruhe Institute of Technology – Karlsruhe
Anhalt University of Applied Science – Köthen
Frankfurt University of Applied Science – Frankfurt
Chemnitz University of Technology – Chemnitz
Munich Technical University – Garching
Regensburg University of Applied Science – Regensburg
Rosenheim University of Applied Sciences – Rosenheim
Technical University of Applied Sciences – Berlin
University of Stuttgart – Stuttgart
University of Duisburg-Essen – Duisburg
University of Jena – Jena
University of Bremen – Bremen
University of Weimar – Weimar
Wiesbaden University of Applied Sciences – Wiesbaden
Universität der Bundeswehr München – Munich
Thüringer Landessternwarte Tautenburg – Tautenburg

Getting started with Ada

Ada IDEs and compilers

Most Ada tools and compilers are proprietary and very expensive

(and there has been a lot of consolidation of the market)

Vendor	Name	Win	Linux	other	License	Target	Origin
PTC	ObjectAda	yes	yes		Comm	Host	Aonix
PTC	APEX Ada	No	Yes		Comm	Embedded	Rational
GreenHills	Ada Optimizing Compiler	Yes	yes		Comm	Embedded	GreenHills
DDC-I	SCORE Ada	Yes	Yes		Comm	Embedded	DDC-I
AdaCore	GNAT	Yes	Yes	Mac	GPLv3+	Host & Embedded	GNU

RAILERGY

Getting started with Ada

GNAT Community

AdaCore is maintaining the community version of GNAT (GNU Ada Translator)

Get GNAT:

https://www.adacore.com/download (for Windows, Linux and Mac; the Windows and Linux versions also include GPS (GNAT Programming Studio IDE))

On Ubuntu:

```
sudo apt-get update –y
sudo apt-get install -y gnat-gps
```

GAP (GNAT Academic Program)

AdaCore provides the GAP Package to members at no cost.

Membership is open to teachers, as well as graduate students using Ada or SPARK technologies in the context of a master's thesis or Ph.D.

https://www.adacore.com/academia

Appendix: SPARK

SPARK

SPARK is a formally defined computer programming language based on the Ada programming language, intended for the development of high integrity software used in systems where predictable and highly reliable operation is essential. It facilitates the development of applications that demand safety, security, or business integrity.

The SPARK language consists of a well-defined subset of the Ada language that uses contracts to describe the specification of components in a form that is suitable for both static and dynamic verification.

In SPARK83/95/2005, the contracts are encoded in Ada comments (and so are ignored by any standard Ada compiler), but are processed by the SPARK "Examiner" and its associated tools.

SPARK 2014, in contrast, uses Ada 2012's built-in "aspect" syntax to express contracts, bringing them into the core of the language. The main tool for SPARK 2014 (GNATprove) is based on the GNAT/GCC infrastructure, and re-uses almost the entirety of the GNAT Ada 2012 front-end.

SPARK Overview

SPARK aims to exploit the strengths of Ada while trying to eliminate all its potential ambiguities and insecurities. SPARK programs are by design meant to be unambiguous, and their behavior is required to be unaffected by the choice of Ada compiler. These goals are achieved partly by omitting some of Ada's more problematic features (such as unrestricted parallel tasking) and partly by introducing contracts which encode the application designer's intentions and requirements for certain components of a program.

- The combination of these approaches is meant to allow SPARK to meet its design objectives, which are:
- logical soundness
- rigorous formal definition
- simple semantics
- security
- expressive power
- verifiability
- bounded resource (space and time) requirements.
- minimal runtime system requirements

RAILERGY

Spark contract examples

Consider:

```
4
5   procedure Increment (X : in out Counter_Type);
6
```

What does this subprogram actually do? In pure Ada, it could do virtually anything – it might increment the X by one or one thousand; or it might set some global counter to X and return the original value of the counter in X; or it might do absolutely nothing with X at all.

With SPARK 2014, contracts are added to the code to provide additional information regarding what a subprogram actually does. For example, we may alter the above specification to say:

```
9   procedure Increment (X : in out Counter_Type)
10    with Global => null,
11         Depends => (X => X);
```

This specifies that the Increment procedure does not use (neither update nor read) any global variable and that the only data item used in calculating the new value of X is X itself.

```
15  procedure Increment (X : in out Counter_Type)
16    with Global  => (In_Out => Count),
17         Depends => (Count  => (Count, X),
18                     X      => null);
```

This specifies that Increment will use the global variable Count in the same package as Increment, that the exported value of Count depends on the imported values of Count and X,
and that the exported value of X does not depend on any variables at all (it will be derived from constant data only).

SPARK Verification conditions

GNATprove can also generate a set of verification conditions or VCs. VCs are used to attempt to establish certain properties hold for a given subprogram. At a minimum, the GNATprove will generate VCs attempting to establish that all run-time errors cannot occur within a subprogram, such as

- array index out of range
- type range violation
- division by zero
- numerical overflow.

If a postcondition or other assertions are added to a subprogram, GNATprove will also generate VCs that require the user to show that these properties hold for all possible paths through the subprogram.

Under the hood, GNATprove uses the Why3 intermediate language and VC Generator, and the CVC4, Z3, and Alt-Ergo theorem provers to discharge VCs. Use of other provers (including interactive proof checkers) is also possible through other components of the Why3 toolset.

Notes and references

[Toole92]: B. A. Toole, "Ada, the Enchantress of Numbers," Strawberry Press, 1992.

[Steelman78]: "Department of Defense Requirements for High Order Computer Programming Languages - 'Steelman'," Defense Advanced Research Projects Agency, 1978.

[ARM83] [ARM83] "Reference Manual for the Ada Programming Language," ANSI/MIL- STD-1815A-1983, U.S. Department of Defense, February 1983.

Current Ada Reference Manual: http://ada-auth.org/standards/12rm/html/RM-TOC.html