A3. Programming Languages for Writing Safety-Critical Software

(a) Overview.

(b) SPARK Ada.
(a) Overview

Important Factors for Programming Languages in Safety Critical Areas

- **Logical soundness.**
  - Is there a sound, unambiguous definition of the language?

- **Complexity of definition.**
  - Are there simple, formal definitions of the language features?
  - Too high complexity results in high complexity in compilers, support tools and therefore in errors there.

- **Expressive power.**
  - Can program features be expressed easily and efficiently?
  - The easier the program on has written, the easier it is to verify it.
Important Factors for Programming Languages in Safety Critical Areas (Cont.)

- **Security.**
  - Can violations of the language definitions be detected before execution?

- **Verifiability.**
  - Is there support for verifying that program code meets the specification?

- **Bounded space and time requirements.**
  - Can it be shown that time and memory constraints are not exceeded?
Common Reasons for Program Errors

• **Subprogram side-effects.**
  – Variables in the calling environment are unexpectedly changed.

• **Aliasing.**
  – Two or more distinct names refer to the same storage location. Changing one variable changes a seemingly different one.

• **Failure to initialize.**
  – Variable is used before it is initialized.

• **Expression evaluation errors.**
  – Eg. out-of-range array subscript, division by zero, arithmetic overflow.
  – Different behaviour of compilers of the same language in case of arithmetic errors.
Cullyer, Goodenough, Wichman have compared suitability of programming languages for high integrity software by using the following criteria:

- **Wild jumps.**
  - Can it be guaranteed that a program cannot jump to an arbitrary memory location?

- **Overwrites.**
  - Can a language overwrite an arbitrary memory location?

- **Semantics.**
  - Is semantics defined sufficiently so that the correctness of the code can be analyzed?

- **Model of mathematics.**
  - Is there a rigorous definition of integer and floating point arithmetic (overflow, errors)?
• **Operational arithmetic.**
  – Are there procedures for checking that the operational program obeys the model of arithmetic when running on the target processor?

• **Data typing.**
  – Are there means of data typing that prevent misuse of variables?

• **Exception handling.**
  – Is there an exception handling mechanism in order to facilitate recovery if malfunction occurs?

• **Exhaustion of memory.**
  – Are there facilities to guard against running out of memory?
• **Safe subsets.**
  
  – Is there a safe subset of the language that satisfies requirements more adequately than the full language?

• **Separate compilation.**
  
  – Is it possible to compile modules separately, with type checking against module boundaries?

• **Well-understood.**
  
  – Will designers and programmers understand the language sufficiently to write safety critical software?

Next slide:
+ means protection available,
? means partial protection,
- means no protection.
## Comparison of Programming Languages

<table>
<thead>
<tr>
<th>Feature</th>
<th>Structured assembler</th>
<th>CO-RAL 66</th>
<th>ISO Pascal</th>
<th>Modula-2</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wild jumps</td>
<td>+</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>+</td>
</tr>
<tr>
<td>Overwrites</td>
<td>?</td>
<td>-</td>
<td>-</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Semantics</td>
<td>?</td>
<td>-</td>
<td>?</td>
<td>?</td>
<td>+</td>
</tr>
<tr>
<td>Model of maths</td>
<td>?</td>
<td>-</td>
<td>?</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Oper. arith.</td>
<td>?</td>
<td>-</td>
<td>-</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Typing</td>
<td>?</td>
<td>-</td>
<td>?</td>
<td>?</td>
<td>+</td>
</tr>
<tr>
<td>Exception handling</td>
<td>-</td>
<td>?</td>
<td>-</td>
<td>?</td>
<td>+</td>
</tr>
<tr>
<td>Safe subsets</td>
<td>?</td>
<td>-</td>
<td>-</td>
<td>?</td>
<td>-</td>
</tr>
<tr>
<td>Exhaustion of mem.</td>
<td>+</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>-</td>
</tr>
<tr>
<td>Separate compil.</td>
<td>-</td>
<td>-</td>
<td>?</td>
<td>?</td>
<td>+</td>
</tr>
<tr>
<td>Well understood</td>
<td>+</td>
<td>?</td>
<td>?</td>
<td>+</td>
<td>?</td>
</tr>
</tbody>
</table>
Analysis

- C most unsuitable language.
- Module-2 most suitable.
  - Problem: limited industrial use.
  - Therefore lack of tools, compilers.
    * Industrial use contributes to reliability of compilers.
- One solution: development of new languages for high integrity software.
  - Same problem as for Modula-2: limited industrial use.
- Better solution: introduction of safe subsets.
  - Rely on standard compilers and support tools.
  - Only additional checker, which verifies that the program is in the subset.
  - Add annotations to the language.
  - Problem: Are compilers safe?
    * Case study revealed:
      Compiler faults are equivalent to one undetected fault in 50 000 lines of code.
    * Especially problem of optimization.
<table>
<thead>
<tr>
<th>Feature</th>
<th>CORAL subset</th>
<th>SPADE-Pascal</th>
<th>Modula2-subset</th>
<th>Ada subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wild jumps</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Overwrites</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Semantics</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>?</td>
</tr>
<tr>
<td>Model of maths</td>
<td>?</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Oper. arith.</td>
<td>?</td>
<td>+</td>
<td>?</td>
<td>+</td>
</tr>
<tr>
<td>Typing</td>
<td>?</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Exception handling</td>
<td>-</td>
<td>-</td>
<td>?</td>
<td>+</td>
</tr>
<tr>
<td>Safe subsets</td>
<td>?</td>
<td>+</td>
<td>+</td>
<td>?</td>
</tr>
<tr>
<td>Exhaustion of mem.</td>
<td>+</td>
<td>+</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Separate compil.</td>
<td>?</td>
<td>?</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Well understood</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Programming Languages Used

• **Aerospace.**
  
  – Trend towards Ada.
  – Use of languages like FORTRAN, Jovial, C, C++.
  – 140 languages used in the development of the Boeing 757/767.
  – 75 languages used in development of the Boeing 747-400.
  – E.g. C++ for the seat back entertainment system of Boeing 777.
  – Northrup B2 bomber control system: C++

• **Spacecraft.**
  
  – Space shuttle: Hal/s and Ada plus other languages.
  – Air traffic control systems in US, Canada, France: Ada.
Programming Languages Used

- **Automotive systems:**
  - Much assembler. Also C, C++, Modula-2

- **Railway industry:**
  - Ada as de-facto standard.

- Denver Airport baggage system written in C++, but initial problems probably not directly related to the use of that C++.

- **In general:**
  - Trend towards Ada for the high-integrity parts of the software.
  - Use of assembler where necessary.
Motivation for Developing Ada

- Original problem of Department of Defense in USA (DOD):
  - Too many languages used and created for military applications (>450).
    * Languages largely incompatible and not portable.
    * Often minimal software available.
    * Competition restricted, since only one vendor.
  - Existing languages too primitive.
    * No modularity.
    * Hard to reuse.
  - Problems particularly severe in embedded systems.
    * 56% of the software cost of DOD in 1973 for embedded systems.
    * Most money spent on maintaining software, not developing it.
Ada

- Decision by DOD: Development of new standard programming language for military applications.
  - Name Ada = name of Ada Lovelace (1815-1852).
    * Wrote programs for Babbage’s computer.
    * Therefore called “the first computer programmer”.

- First release: Ada 83
  (1983 – same year C++ was released).

• SPARK Ada

  • Subset of Ada.
    – Several revisions carried out by Praxis Critical Systems Ltd.
    – Adapted to Ada95
    – Commercial tools available from Praxis Critical Systems.

  • Annotations to Ada.
    – Some required for data and information flow analysis.
    – Others allow to generate and prove verification conditions.
    – It is as well possible to integrate unchecked or even unimplemented Ada code.

  • SPARK Ada code compiles with standard Ada compilers.
Factors for Programming Languages Addressed by SPARK Ada

• **Logical Soundness.**
  - Problem: statement like \( Y := F(X) + G(X) \):
    Order of evaluation not specified.
    Problem if \( F(X) \) and \( G(X) \) have side effects.
    * E.g. \( F(X) \) has effect \( Z := 0 \),
      \( G(X) \) has effect \( Z := 1 \).
    * Solution in many languages: define order of evaluation.
      Not possible, if SPARK Ada should compile on standard compilers.
    * Solution in SPARK Ada:
      Functions are not allowed to have side-effects.

• **Simplicity of language definition.**
  - Omission of too complex principles.
    * No variant records.
      * (Dependent types, but no complete compile time checking).
    * No tasks (concurrency).
    * No generic types.
Factors for Programming Languages Addressed by SPARK Ada

- **Expressive power.**
  - Hiding of variables allowed.
  - Allows to specify strong assertions about variables

- **Security.**
  - Array bound checks.
  - Programs does not stray outside the computational model
  - Both guaranteed by Ada.
  - In order to be verifiable at compile-time:
    * Constraints (array bounds, ranges) have to be static (determined at compile time).
Factors for Programming Languages Addressed by SPARK Ada

- **Verifiability**
  - Extra annotations
    * control of data flow,
    * control of information flow,
    * proof annotations.
  (more below)
  - Every fragment of code has a single entry point and limited exit points.

- **Bounded space and time requirements.**
  - Recursion disallowed.
  - No arrays without bounds
    * Can be declared, but only subtypes of it can be used.
  - No pointers (called access types in Ada).
  - The above guarantees bounded space.
    Bounded time difficult to guarantee.
Factors for Programming Languages Addressed by SPARK Ada

- **Language should be as explicit as possible.**
  - **No polymorphism** (ie. that an operation is defined for different types):
    - *No overloading* of functions.
    - *No array sliding*:
      Assignment, comparison, operations on arrays only allowed on arrays with same array index sets.
      - As well no concatenation of arrays.
      - However, for strings allowed.
    - *No default parameters, default record components.*
    - However standard +, * are overloaded.
Factors for Programming Languages Addressed by SPARK Ada

• (Language should be as explicit as possible, cont.)
  – No anonymous subtypes.
    * Instead of:
      type Vector is array (0..100) of Integer;
      one has to write
      type Vector_index is range 0..100;
      type Vector is array (Vector_index) of Integer;
    * Exception: loop variables can be elements of an anonymous range.
  – Unique names of entities at a given place:
    * Package variables have to used explicitly:
      A variable X of a package Mypackage has to be referenced as Mypackage.X
Factors for Programming Languages Addressed by SPARK Ada

- **No structures which are too complex to grasp.**
  - **No type hierarchies.**
    * No derived types (essentially a copy of a type).
    * No type extension (subtypes in object-oriented progr.)
    * No class-wide types (classes which have subclasses).
      Therefore no inheritance.
      - Problem of inheritance: properties are inherited remotely.
    * However subtypes (restriction of the range of a type) allowed.
  - **Restriction on return statements and exits.**
    * No return statements in procedures.
    * Exactly one return statement in functions.
    * Exit from loops only possible to innermost loop.
    * No exit out of if condition (since there the innermost loop is the if-statement).
Architecture of the SPARK Ada

• SPARK-examiner.
  – Verifies the following:
    ∗ Correct Ada syntax.
    ∗ SPARK-subset of Ada chosen, as described above.
  – Carries out three levels of analysis:
    ∗ Data flow analysis.
    ∗ Information flow analysis.
    ∗ Generation of verification conditions.
  ⇓

• SPADE-simplifier
  – Simplifies verification conditions of the examiner. Trivial ones are already proved.
  ⇓

• SPADE-proof-checker
  – Proof tool for interactively proving verification conditions.
Three Levels of Analysis

- **Data flow analysis.**
  - Checks input/output behaviour of parameters and variables.
  - Checks initialization of variables.
  - Checks that changed and imported variables are used later (possibly as output variables).

- **Information flow analysis.**
  - Verifies interdependencies between variables.

- **Verification conditions.**
  - Generation of proof conditions, which allow to prove correctness of programs.

Idea is that the 3 different levels of analysis are applied dependent on the criticality of the program. (Some parts might not be checked at all).
Annotations

- Certain annotations are added to the programs.
  - Specific to the 3 levels of analysis.

- Written as **Ada comments**:
  - Ignored by Ada compilers.
  - Used by the SPARK Ada tools.
  - Syntax: start with `--#`, eg.
    ```
    * --# global in out MyGlobalVariable;
    ```
Data Flow Analysis – Parameters

• In Ada all parameters have to be labeled as
  – input parameters; symbol: \texttt{in},
  – output parameters; symbol: \texttt{out},
  – input/output parameters; symbol: \texttt{in out}.

• Example:

  \begin{verbatim}
  procedure ABC(A: \texttt{in} Float;
                B: \texttt{out} Integer;
                C: \texttt{in out} Colour)
  \end{verbatim}

• Examiner verifies that
  – Input parameters are
    * not modified,
    * but used at least once,
  – output parameters are
    * not read before being initialized,
    * initialized.
  – input/output parameters are
    * read,
    * and modified.
• Global variables must be given status as input or output or input/output variables by annotations.

  – Syntax examples:
    ∗ --# global in A;
    ∗ --# global out B;
    ∗ --# global in out C;
  – Data flow analysis carries out the same analysis as for parameters.
Data Flow Analysis – Functions

- Functions can have only input parameters (no keyword required).

- Functions have only read access to global parameters. Therefore the syntax for global parameters is simply
  \[ \# \text{global A;} \]
  or
  \[ \# \text{global A,B,C;} \]

- Neither parameters nor global variables can be changed. Therefore functions don’t have side effects.
Data Flow Analysis – Packages

• Package variables = variables global to a package.

• Package variables must be declared by annotations. Syntax example:
  --# own X,Y;

• If a variable is initialized it has to be declared; whether it is initialized will be verified.
  Syntax example:
  --# initializes X;
  – However, even an uninitialized package variable is allowed to be used by a procedure.

• If a package is used it has to be declared:
  Syntax example:
  --#inherits Mypackage;
Consider the following wrong program, which should exchange X and Y:

```pascal
procedure Exchange(X,Y: in out Float)
  is
    T:Float
  begin
    T:= X; X:= Y; Y:= X
  end Exchange;
```

Mistake: body should be:

```pascal
T:= X; X:= Y; Y:= T
```

Data flow analysis results in 3 error messages:

- T:= X is ineffective statement.
- Import of initial value of X is ineffective.
- T is neither referenced nor used.
Example2 (Data Flow Analysis)

- Here is another wrong program, which should exchange X and Y:

```plaintext
procedure Exchange(X,Y: in out Float)
is
begin
  X:= Y; Y:= X
end Exchange;
```

- Data flow analysis results in error message:
  - Importation of initial value of X is ineffective.
Information Flow Analysis

- Additional annotations on how variables depend on each other.
  Syntax examples:
  --#derives X from Y;
  or
  --#derives X from Y &
  --# Y from X;
  or, if nothing is used
  --#derives X from ;

- Information flow verifies that these dependencies are fulfilled
Example (Information Flow Analysis)

- Consider the following wrong program, which should exchange X and Y and count the number of exchanges in Z:

```pascal
procedure ExchangeAndCount(X,Y,Z: in out Integer)
  --# derives X from Y &
  --#   Y from X &
  --#   Z from Z;
is
  T: Integer; begin
    T:= X; Y:= X; Y:= T; Z:= Z + T;
  end ExchangeAndCount;
```

- The error is that Z:= Z + T; should be replaced by Z:= Z + 1;

- Data flow analysis succeeds without problems.

- Information flow analysis gives warning, since Z depends on Z and X.
Proof Conditions – Procedures without Loops

- For procedures without loops, two kinds of annotations are relevant:
  - Pre-conditions, eg.:
    \[ \#\text{pre} \ M \geq 0 \text{ and } M > 0; \]
  - Post-conditions, eg.:
    \[ \#\text{post} \ M = M+1; \]

- Then examiner generates formulas which express:
  - If the pre-conditions hold, and the procedure is executed, afterwards the post-condition holds.

- If there are no pre-conditions, then the formula expresses that the post-condition holds always.
Proof Conditions – Procedures without Loops (Cont.)

• In the post-conditions,
  – $X\sim$ stands for the value of $X$ before executing the procedure,
  – $X$ stands for the value after executing it,
  – e.g. $X=X\sim+1$; expresses: The value of $X$ after executing the procedure is the value of $X$ before executing it + 1.

• Formulas are built using:
  – Boolean connectives and, or, not, xor, $\rightarrow$, $\leftarrow$
  – quantifiers for all, for some.
Example (Proof Conditions)

- Assume the following wrong program:

```
procedure Exchange(X,Y: in out Float);
--# derives X from X &
--# Y from Y;
--# pre X >= 0.0;
--# post X = Y~ and Y=X~;

is
T: Float
begin
  T:= X; X:=T; T:=Y; Y:=T;
end Exchange;
```

- The post condition is not fulfilled in general.
Example (Proof Conditions, Cont.)

- The examiner generates the formula:

  \[ H_1: x \geq 0.0. \]
  \[ H_2: \text{true}. \]
  \[ H_3: \text{true}. \]
  \[ \rightarrow \]
  \[ C_1: x = y. \]
  \[ C_2: y = x. \]

  which is not provable.

- (The data and information flow check is so good, that it is difficult to find simple but wrong programs, which pass it).
Example2 (Proof Conditions)

- Assume the correct program:

```plaintext
procedure Exchange(X,Y: in out Float);
--# derives X from Y
--#       Y from X;
--# post X = Y~ and Y=X~;
    is
  T: Float
begin
  T:= X; X:=Y; Y:=T;
end Exchange;
```

- The examiner generates the formula:

  H1: true.
  H2: true.
  H3: true.
  ->
  C1: x=x.
  C2: y=y.

- The simplifier shows that this is provable
Proof Conditions – Check Conditions

- One can insert in between the procedures a check condition. E.g., in the previous example, insert between \( T := X \) and \( X := Y \):
  \[
  --\# \textbf{check} \ T > 0.0;
  \]

- Now the formulas express:
  - From the pre-condition follows at that position the check-condition.
  - From the pre-condition and the check-condition at that position follows the post-condition.
  - Check conditions serve therefore as intermediate proof-goals.
If one has functions, one can either state the result of the function:

Eg.

```c
--# return X + 1
```

expresses: the result is $X+1$.

or one can associate with the result a variable and a condition. Eg. one can write:

```c
--# return X => X > Y;
```

if $Y$ is a parameter or a global parameter. The example expresses:

the returned value is $> Y$. 
Proof Conditions – Procedures with Loops

• If one has a loop, a loop invariant is required. The syntax is for instance:
  
  --# assert X +Y = X~+Y~;

• If one has one pre condition, one loop and one post condition, the examiner generates proof conditions expressing:
  
  – From the pre-condition follows, when first entering the loop, the condition of assert.
  – From assert follows, if exit conditions are false, the condition of assert after one step.
  – From assert follows, if one exit condition is true, the post condition.
Example (Proof Conditions with Loop)

procedure test(X,Y: in out Float)
--# derives X from X &
--# Y from X,Y;
--# pre X > 0.0;
--# post X+Y=X~+Y~ and X<0.0;
is
begin
  loop
    --# assert X +Y = X~+Y~;
    X := X - 1.0;
    Y := Y + 1.0;
    exit when X < 0.0;
  end loop;
end test;
The examiner generates in the last example the following proof conditions:

- **H1**: $x > 0$.
  - **H2**: true.
  - **H3**: true.
  - =>
  - **C1**: $x + y = x + y$.

- **H1**: $x + y = x \sim + y \sim$
  - **H2**: not ($x-1 < 0$)
  - =>
  - **C1**: $x-1+(y+1) = x \sim + y \sim$.

- **H1**: $x + y = x \sim + y \sim$
  - **H2**: $x-1 < 0$
  - =>
  - **C1**: $x-1+(y+1) = x \sim + y \sim$.
  - **C2**: $x-1 < 0$

All proof conditions are verified by the simplifier. n
Other Annotations

• The main program is declared by:
  -- #main_program;

• Parts which shouldn’t be examined can be declared by:
  -- #hide;
  – Allows especially direct interaction with non-critical and therefore non-verified Ada programs.
  – Allows as well to integrate not yet implemented code.