A3. Programming Languages for Writing Safety-Critical Software
(a) Overview.
(b) SPARK Ada.
Main Criteria for Choice of Programming Languages for Critical Systems

- Logical soundness.
- Complexity of definition.

Complexity and support tools:
- Too high complexity results in high complexity and therefore in errors in compilers and support tools.
- Are there simple, formal definitions of the language features?
- Is there a sound, unambiguous definition of the language?
Main Criteria for Choice of Programming Languages for Critical Systems

- Can violations of the language definitions be detected before execution?
  - Security
- The easier the program one has written, the easier it is to verify it.
  - Expressive power
- Can program features be expressed easily and efficiently?
Main Criteria for Choice of Programming Languages for Critical Systems

- Verifiability
- Bounded space and time requirements

- Can it be shown that time and memory constraints are not exceeded?
- Is there support for verifying that program code meets the specification?
Common Reasons for Program Errors

- Variable is used before it is initialized.
- Failure to initialize.

  Changing one variable changes a seemingly different one.

  - Two or more distinct names refer to the same storage location.

- Aliasing.

- Variables in the calling environment are unexpectedly changed.

- Subprogram side-effects.
Common Reasons for Program Errors (Cont.)

- Expression evaluation errors.
- Different behavior of compilers of the same language in case of arithmetic errors.
- E.g.: out-of-range array subscript, division by zero, arithmetic overflow.
Comparison of Programming Languages

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?

Critical Systems, CS-411, Lentterm 2003, Sec. A3-7
model of arithmetic when running on the target processor?
- Are there procedures for checking that the operational program obeys the
operational arithmetic?

- Is there a rigorous definition of integer and floating point arithmetic?

- Model of mathematics.

- Operational arithmetic.

Overflow, errors?

Programming Languages (Cont.)

Comparison of
Comparison of Programming Languages (Cont.)

- Are there facilities to guard against running out of memory?
  - **Exhaustion of memory.**

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

  - Are there means of data typing that prevent misuse of variables?
    - **Data typing.**
Comparison of Programming Languages (Cont.)

- Will designers and programmers understand the language sufficiently to write safety critical software?
  - Will they understand the language more adequately than the full language?
  - 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?

- Safe subsets?
  - Is there a safe subset of the language that satisfies requirements more adequately than the full language?
Legend for next slide:
- means no protection,
? means partial protection,
+ means protection available,

• Legend for next slide.

Comparison of Programming Languages (Cont.)
<table>
<thead>
<tr>
<th>Structured assembler</th>
<th>C CORAL 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>
</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 mathematics</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>+</td>
</tr>
<tr>
<td>Operational arithmetic</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>+</td>
</tr>
<tr>
<td>Data 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</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>+</td>
</tr>
<tr>
<td>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>

Critical Systems, CS 411, Lentterm 2003, Sec. A3
CORAL66 = compiled structured programming language related to Algol.

Developed at the Royal Radar Establishment RE, Malvern, UK.

Allowed inline assembly code.

Used for real-time systems.

No free CORAL 66 compilers seem to be available today.

Remarks on CORAL 66
Analysis

- Same problem as for Modula-2: Limited industrial use.
- Problem: Limited industrial use.
- Therefore lack of tools, compilers.
- Problem: Development of new languages for high integrity software.
- Solution: Development of new languages for high integrity software.
- Modula-2 most suitable.
- C most unsuitable language.
Especially problem of optimization.

Compiler faults are equivalent to one undetected fault in 50 000 lines of code.

Case study revealed:
- Problem: Are compilers safe?
- Case study revealed:
  - Only additional checker, which verifies that the program is in the subset.
  - Rely on standard compilers and support tools.

Better solution: introduction of safe subsets.

Analysis (cont.)
Safe Subsets

Well understood
Separate compilation
Exhaustion of mem.
Safe subsets
Exception handling
Data typing
Operational arithmetic
Model of mathematics
Semantics
Overwrites
Wild jumps

Coral, Spade, Modula 2, Ada

(A) Anton Setzer 2003 (except for pictures)
- Air traffic control systems in US, Canada: Ada.
- Space shuttle: Hal/s and Ada plus other languages.
- European Space Agency: use of Ada in mission-critical systems.

• Aerospace

- Northrup B2 bomber control system: C++

  • E.g. C++ for the seat back entertainment system of Boeing 777.
  • 75 languages used in development of the Boeing 747-400.
  • 140 languages used in the development of the Boeing 757/767.

  • Use of languages like FORTRAN, Jovial, C, C++.

  • Trend towards Ada.

Programming Languages Used

(c) Anton Setzer 2003 (except for pictures)
In general:
- Use of assembler where necessary.
- Trend towards Ada for the high-integrity parts of the software.

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

Railway Industry:
- Ada as de-facto standard.

Automotive Systems:

Programming Languages Used

(A3) Anton Setzer 2003 (except for pictures)
Motivation for Developing Ada

(b) SPARK Ada

Original problem of Department of Defense in USA (DOD):

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

C) Anton Setzer 2003 (except for pictures)

(1983 - same year C++ was released).

First release: Ada 83

Therefore called "the first computer programmer".

Wrote programs for Babbage's computer.

- Name Ada = name of Ada Lovelace (1815-1852).

Military applications.

Decision by DOD: Development of new standard programming language for military applications.
Ada is the often required standard for military applications in the US, and has therefore adopted by the Western military industry. Therefore lots of tools for critical systems support Ada.

Ada was developed taking into account its use in real-time critical systems:

- It is possible to write efficient code and code close to assembler code, while still using high level abstractions.
- Features were built in which prevent errors (e.g., array bound checks, no writing to arbitrary memory locations).

Software for military applications forms a large portion of critical systems.
SPARK Ada is a subset of Ada compiled with standard Ada compilers.

It is as well possible to integrate unchecked or even unimplemented Ada code.

- Others allow to generate and prove verification conditions.
- Some required for data and information flow analysis.
- Annotations to Ada.
- Commercial tools available from Praxis Critical Systems.
- Adapted to Ada95.
- Several revisions carried out by Praxis Critical Systems Ltd.
- Original definition by B. Carre and T. Jennings, Univ. of Southampton.
- Subset of Ada.

SPARK Ada
Factors for Programming Languages

Addressed by SPARK Ada

Logical Soundness.

Functions are not allowed to have side-effects.

Solution in SPARK Ada:

Not possible, if SPARK Ada should compile on standard compilers.

Solution in many languages: define order of evaluation.

- Problem: statement like $Y := F(X) + G(X)$

- Order of evaluation not specified.

- Problem: $F(X)$ and $G(X)$ have side effects.

- $F(X)$ has effect $Z := 1$.

- $G(X)$ has effect $Z := 0$.

- E.g. $F(X)$ has effect $Z := 0$.

- Problem if $F(X)$ and $G(X)$ have side effects.

- Problem statement like $Y := F(X) + G(X)$.

Critical Systems, CS411, Lentterm 2003, Sec. A3
Factors for Programming Languages Addressed by SPARK Ada (Cont.)

- 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.**
- **Security.**

### Expressive power.
- "Hiding of variables allowed."
- "Allowing strong assertions about variables."

### Security.
- "Array bounds are checked."
- "Programs does not stray outside the computational model (one cannot jump or write to an arbitrary memory location)."
- "Feature enforced by SPARK Ada: In order to be verifiable at compile-time:"
  - Constraints (array bounds, ranges) have to be static (determined at compile time)."

(C) Anton Setzer 2003 (except for pictures)
Factors for Programming Languages Addressed by SPARK Ada

- Verifiability
  - Extra annotations
  - Control of data flow
  - Control of information flow
  - Proof annotations
- Bounded space and time requirements
  - Every fragment of code has a single entry point and limited exit points.
  - Recursion disallowed.
  - No arrays without bounds.
  - No pointers (called access types in Ada).
- Bounded time difficult to guarantee.
  - The above guarantees bounded space.
  - Can be declared, but only subtypes of it can be used.

(C) Anton Setzer 2003 (except for picture)
Languages should be as explicit as possible.

- No polymorphism (ie. that an operation is defined for different types);
- No overloading of functions;
- No array sliding:
- No concatenation of arrays allowed.
- No behaviors in an unexpected way in this case.
- No array index sets.
- No default parameters, default record components.
- No polymorphism.
- +, - are overloaded.
- However, standard +, - are overloaded.

Factors for Programming Languages Addressed by SPARK Ada

(© Anton Setzer 2003 (except for pictures))
No anonymous subtypes.

Instead of:

```
package Mypackage.

type Vector is array (0..100) of Integer;
```

One has to write:

```
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:

A variable X of a package Mypackage has to be referenced as

```
Mypackage.x
```

Package variables have to be used explicitly:

```
A package variables have to used explicitly:
```

Languages should be as explicit as possible, cont.:
No structures which are too complex to grasp.

Factors for Programming Languages Addressed by SPARK Ada

- No structures which are too complex to grasp.
- No type hierarchies.
- No type extension (subtypes in object-oriented programs).
- No derived types (essentially a copy of a type).
- No type extension (subtypes in object-oriented programs).
- 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.
- No return statements in functions.
- Exactly one return statement in functions.
- Exit from loops only possible to innermost loop.
- No exit out of if conditional statement.
- Since there the innermost loop is the if-statement.
- Critical Systems, CS 411, Lent Term 2003, Sec. A3
Veriﬁest the following:

- Correct Ada syntax.
- SPARK subset of Ada, chosen, as described above.
- Generation of veriﬁcation conditions.
- Information flow analysis.
- Data ﬂow analysis.
- Carries out three levels of analysis:
  - Trivial ones are already proved.
  - Simpliﬁes veriﬁcation conditions of the examiner.
  - SPADEx-simpliﬁer

↑

Archiecture of the SPARK Ada

(© Anton Setzer 2003 (except for pictures)}
SPADE-proof-checker

– Prooftool for interactively proving verification conditions.

Architecture of the SPARK Ada (Cont.)
Three levels of analysis:

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

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

**Information flow analysis.**
- Verifies that changed and imported variables are used later (possibly as output variables).
- Checks initialization of variables.
- Checks input/output behavior of parameters and variables.
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).
Certain annotations are added to the programs.

### Annotations

- **Global in out #**
  - Syntax: start with `--`, e.g.
    ```plaintext
    --global in out
    MyGlobalVariable;
    ```

- Written as Ada comments:

- Specific to the 3 levels of analysis.

- Certain annotations are added to the programs.
In Ada all parameters have to be labeled as input\, output\, or \(\text{in out}\). Example:

\[
\text{procedure \(\text{ABC}(A:\text{in float}, B:\text{out integer}, C:\text{in out colour})\)}
\]
Data Flow Analysis – Parameters (Cont.)

Examiner verifies that

• Input/Output parameters are not read before being initialized,'  
• Output parameters are not read at least once,  
• Input/Output parameters are not modified,  
• Input parameters are not modified.  

Read, *  
Initialized, *  
Initialized.
Data Flow analysis carries out the same analysis as for parameters.

- Global variables must be given status as input or output or input/output.

Syntax examples:

```c
-- # global in A;
-- # global out B;
-- # global out C;
```

Global variables are annotated by

- "global in A" for input
- "global out B" for output
- "global out C" for input/output

Data Flow Analysis - Global Variables
Therefore functions don't have side effects.

- Neither parameters nor global variables can be changed.

\[ \text{-- # Global A,B,C,} \]

\[ \text{or} \]

\[ \text{-- # Global A,} \]

Therefore the syntax for global parameters is simply:

- Functions have only read access to global parameters.

- Functions can have only input parameters (no keyword required).
Data Flow Analysis - Packages

Procedure:

- However, even an uninitialized package variable is allowed to be used by a
  procedure.

  --own X,Y

  Syntax example:

  If a variable is initialized it has to be declared, whether it is initialized will be
  verified.

  --#initializes X

  Syntax example:

  Package variables must be declared by annotations.

  # Package variables = variables global to a package.

(C) Anton Setzer 2003 (except for pictures)
Data Flow Analysis – Packages (Cont.)

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

\[
\text{procedure Exchange}(X, Y; \text{in out Float})
\]

\[
T := X; X := Y; Y := T
\]

\text{Mistake: body should be:}

\[
\text{end Exchange;}
\]

\[
T := X; X := Y; Y := X
\]

\text{begin}

\[
T :\text{Float}
\]

\text{is}

\end{Exchange}(X, Y; \text{in out Float})

\text{(Example (Data Flow Analysis))}
Data flow analysis results in 3 error messages:

- \( T \) is neither referenced nor used.
- Import of initial value of \( X \) is ineffective.
- \( T := X \) is ineffective statement.

**Example (Data Flow Analysis: Cont.)**
Here is another wrong program, which should exchange X and Y:

\begin{verbatim}
procedure Exchange(X,Y: in out Float)
begin
X:=Y; Y:=X
end Exchange;
\end{verbatim}

Data flow analysis results in error message:

- Importation of initial value of X is ineffective.

Example 2 (Data Flow Analysis)
Additional annotations on how variables depend on each other.

Syntax examples:

- `#derives X from Y` or `#derives Y from X` if nothing is used.

Information Flow Analysis

Information Flow verifies that these dependencies are fulfilled.
Consider the following wrong program, which should exchange X and Y and count the number of exchanges in Z:

```
procedure ExchangeAndCount(X,Y,Z: inout 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;

```
procedure ExchangeAndCount(X,Y,Z: inout Integer)
    begin
        T:=X; Y:=X; Y:=T; Z:=Z+1;
    end ExchangeAndCount;
```

Critical Systems, CS-411, Lentterm 2003, Sec. A3

A3-44
Data flow analysis succeeds without problems.

Information flow analysis gives warnings, since $Z$ depends on $Z$ and $X$.

Example (Information Flow Analysis; Cont.)
If there are no pre-conditions, then the formula expresses that the post-
condition holds always.

- If the pre-conditions hold, and the procedure is executed, afterwards the
  post-condition holds.

Then examiner generates formulas which express:

- For procedures without loops, two kinds of annotations are relevant:
In the post-conditions,

\[ \text{The value of } X \text{ after executing the procedure is the value of } X \text{ before executing it.} \]

- \( X \) stands for the value after executing the procedure.
- \( X \) stands for the value before executing the procedure.

Formulas are built using:

- \( \text{forall} \quad \text{for some} \)
- Boolean connectives \( \text{and}, \quad \text{or}, \quad \text{not}, \quad \text{xor} \)
- relational operators \( \text{<}, \quad \text{>}, \quad \text{<=>}, \quad \text{->} \)
- \( \text{forall} \quad \text{for some} \)

- \( \text{forall} \quad \text{for some} \)

Procedures without Loops (Cont.)
Example (Proof Conditions)

Assume the following wrong program:

```plaintext
procedure Exchange(X,Y: in out Float);

-- #derives X from X &
-- # Y from Y;

-- #pre X > 0.0;
-- # post X = Y & Y = X;

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

The post condition is not fulfilled in general.
The examiner generates the formula:

\begin{align*}
H_1 & : x > 0.0. \\
H_2 & : \text{true.} \\
H_3 & : \text{true.} \\
C_1 & : x = y. \\
C_2 & : y = x. \\
\end{align*}

\textit{The data and information flow check is so good, that it is difficult to find simple but wrong programs, which pass it.}
Assume the correct program:

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

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

Example 2 (Proof Conditions)
The examiner generates the formula:

\[ H_1: \text{true} \]
\[ H_2: \text{true} \]
\[ H_3: \text{true} \]

The simplifier shows that this is provable.

\[ C_1: x=x \]
\[ C_2: y=y \]
\[ C_3: x=x, y=y \]
\[ C: x=x \]

Example (Proof Conditions, cont.)
One can insert in between the procedures a check condition.

---

Check Conditions serve therefore as intermediate proof-goals.

- The post-condition.
- From the pre-condition and the check-condition at that position follows.
- From the pre-condition follows at that position the check-condition.

Now the formulas express:

```
-- check \( T < 0.0 \).
```

Example in the previous example, insert between \( T = X \) and \( X = Y \):
If one has functions, one can either state the result of the function:

E.g.

```cpp
## return X + 1;
```

expresses: the result is `X + 1`.

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

```cpp
## return X => Y;
```

if `Y` is a parameter or a global parameter.

The example expresses:

the returned value is `> Y`.
If one has a loop, a loop invariant is required.

-- assert \( X + Y = Y \) --

The syntax is for instance:

- 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.
- From assert follows, if exit conditions are true, the post condition is true, the post condition.

Proof Conditions

- Procedures with Loops

(C) Anton Setzer 2003 (except for pictures)
Example (Proof Conditions with Loop)

```pascal
procedure test(X,Y: inout Float);
begin
  loop
    assert X + Y = X + Y;
    X := X - 1.0;
    Y := Y + 1.0;
    exit when X < 0.0;
  end loop;
end test;
```

Critical Systems, CS 441, Lent term 2003, Sec. A3
The examiner generates in the last example the following proof conditions:

\[ C1: x - 1 + (y + 1) = x + y + 1 \]
\[ < \]
\[ H2: \text{not} \ (x - 1 > 0) \]
\[ \sim y + \sim x = y + x \]

\[ C1: x + y = y + x \]
\[ < \]
\[ H3: \text{true} \]
\[ H2: \text{true} \]
\[ H1: x < 0 \]

**Generated Proof Conditions**
All proof conditions are verified by the simplifier.

\[ 0 > x - 1 \]
\[ \sim y + \sim x = x + (y + 1) \]
\[ \sim y + \sim x = x + y \]

\[ C2: x-1 \]
\[ \sim 0 > 0 \]
\[ \sim y + \sim x = x + (y + 1) \]
\[ \sim y + \sim x = x + y \]

\[ C1: x-1 < 0 \]
\[ \sim \sim y + \sim x = x + y \]

\[ H2: x-1 \]
\[ 0 > 0 \]
\[ \sim y + \sim x = x + y \]

\[ H1: x-1 \]
\[ \bullet \]
Other Annotations

The main program is declared by:

```
--#main-program:

The main program is declared by:
```

Parts which shouldn't be examined can be declared by:

```
--#hide:

Parts which shouldn't be examined can be declared by:
```

---

Allows especially direct interaction with non-critical and therefore non-
verified Ada programs.

---

Allows as well to integrate not yet implemented code.