A3. Programming Languages for Writing Safety-Critical Software

(a) Overview.

(b) SPARK Ada.
Main Criteria for Choice of Programming Languages for Critical Systems

(a) Overview

1. Logical soundness.
   - Are there simple, unambiguous definitions of the language?

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

- Expressive power.
- Security.

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

- Verifiability:
  - Can it be shown that time and memory constraints are not exceeded?

- Bounded space and time requirements:
  - Can it be shown that time and memory constraints are not exceeded?
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 over/flow.
  - E.g. out-of-range array subscript, division by zero, arithmetic over/flow.

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

(©) Anton Setzer 2003 (except for pictures)
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?
model of arithmetic when running on the target processor?

- Are there procedures for checking that the operational program obeys the
  operational arithmetic.

- (overflow, errors)

- Is there a rigorous definition of integer and floating point arithmetic
  model of mathematics.

- Programming Languages (Cont.)

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

(C) Anton Setzer 2003 (except for pictures)
Datatyping.

- 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?
Comparison of Programming Languages (Cont.)

- Write safety critical software?
- Will designers and programmers understand the language sufficiently to well-understood?
- Separate compilation?
- Is it possible to compile modules separately, with type checking against module boundaries?
- Is there a safe subset of the language that satisfies requirements more adequately than the full language?
- Are there safe subsets?
Legend for next slide:

- means no protection,

? means partial protection,

+ means protection available,

Legend for next slide: •

Programming Languages (Cont.)

Comparison of

Critical Systems, CS 411, Lenterm 2003, Sec. A3

Copyright Anton Setzer 2003 (except for pictures)
Remark on CORAL 66

- CORAL 66 = compiled structured programming language related to Algol.
- Developed at the Royal Radar Establishment RRE, Malvern, UK.
- No free CORAL 66 compilers seem to be available today.
- Allowed inline assembly code.
- Used for real-time systems.
Analysis

- Same problem as for Modula-2: Limited industrial use.
- Therefore lack of tools, compilers.
- Problem: Limited industrial use.
- Module-2 most suitable.
- C most unsuitable language.

One solution: Development of new languages for high integrity software.

Module-2 most suitable.

Critical Systems, CS-441, LentTerm 2003, Sec. A3

A3-14

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?
- Added annotations to the language.
- Only additional checker, which verifies that the program is in the subset.
- Rely on standard compilers and support tools.

---

Rely on standard compilers and support tools.
<table>
<thead>
<tr>
<th>Safe Subsets</th>
<th>Well understood</th>
<th>Separate compilation</th>
<th>Exhaustion of mem.</th>
<th>Exception handling</th>
<th>Data typing</th>
<th>Operational arithmetic</th>
<th>Model of mathematics</th>
<th>Semantics</th>
<th>Overwrites</th>
<th>Wild jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>i</td>
<td>+</td>
<td>+</td>
<td>i</td>
<td>i</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>i</td>
<td>i</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>i</td>
<td>i</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>i</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>i</td>
<td>i</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>i</td>
<td>i</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>i</td>
<td>+</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>+</td>
<td>i</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>i</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>i</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Critical Systems, CS-411, Lenterm 2003, Sec. A3. A3-16

(C) Anton Setzer 2003 (except for pictures)
Aerospace.

- Space shuttle: HALS and Ada plus other languages.
- European Space Agency: use of Ada in mission-critical systems.

- Northrop B2 bomber control system: 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++, Ada.
- Trend towards 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 C++.

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

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

(c) 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).
- 56% of the software cost of DOD in 1973 for embedded systems.
- Competition restricted, since only one vendor.
- Often minimal software available.
- Languages largely incompatible and not portable.

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

(c) Anton Setzer 2003 (except for pictures)
Decision by DOD: Development of new standard programming language for military applications.

  (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).
SPARK Ada code compiles 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. Jenning, Univ. of Southampton.

Subset of 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.
  
  \[
  \begin{align*}
  g(x) \text{ has effect } z &= 1, \\
  e.g. f(x) \text{ has effect } z &= 0.
  \end{align*}
  \]

* Problem if \( f(x) \) and \( g(x) \) have side effects.

Order of evaluation not specified.

Problem: statement like \( y := f(x) + g(x) \):

\[
\text{Critical Systems, CS411, Lenterm 2003, Sec. A3}
\]

A3-22
Factors for Programming Languages

Addressed by SPARK Ada (Cont.)

- Simplicity of language definition.
- Omission of too complex principles.
- No variant records.
- No tasks (concurrency).
- No dependent types, but no complete compile time checking.
- No generic types.

Critical Systems, CS-411, Lenterm 2003, Sec. A3
Factors for Programming Languages Addressed by SPARK Ada

- Expressed power.
- Security.
- Hiding of variables allowed.
- Array bound checks.
- Programs do 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).

Critical Systems, CS-411, Lenterm 2003, Sec. A3
Factors for Programming Languages Addressed by SPARK Ada

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

- No polymorphism (i.e., that an operation is defined for different types):
- No array shifting:
- No overloading of functions:
- However, for strings allowed.
- As well no concatenation of arrays:
- Assignment, comparison, operations on arrays only allowed on arrays with same array index sets.
- No default parameters, default record components.
- However standard +, are overloaded.

Addressed by SPARK Ada
Factors for Programming Languages Addressed by SPARK Ada

Languages should be as explicit as possible, cont.

- No anonymous subtypes.
- Unique names of entities at a given place.
- Package variables have to be used explicitly.
- Instead of:

\[
\text{type Vector is array (0..100) of Integer;}
\]

- Exception: Loop variables can be elements of an anonymous range.

\[
\text{type Vector index is range 0..100;}
\]

\[
\text{type Vector is array (Vector index) of Integer;}
\]

- A variable \( X \) of a package \( \text{MyPackage} \) has to be referenced as \( \text{MyPackage.X} \).

- Instead of:

\[
\text{type Vector is array (0..100) of Integer;}
\]
Factors for Programming Languages Addressed by SPARK Ada

No structures which are too complex to grasp.

- Restriction on return statements and exits.
  - No type hierarchies.
  - No derived types (essentially a copy of a type).
  - No type extension (subtypes in object-oriented progr.).
  - No type hierarchy.
  - No class-wide types (classes which have subclasses).
  - No tyoe extension in object-oriented progr.

- Problem of inheritance: properties are inherited remotely.
  - Therefore no inheritance.

- No type hierarchy.
  - No derived types (essentially a copy of a type).

- No structures which are too complex to grasp.
  - 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).

Critical Systems, CS 411, Lentterm 2003, Sec. A3
Architecture of the SPARK Ada

- 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.

⇒

- SPADEx-simplifier
  - Simplifies verification conditions of the examiner.
  - Trivial ones are already proved.

(C) Anton Setzer 2003 (except for pictures)
Architecture of the SPARK Ada (Cont.)

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

Data Flow Analysis:
- Checks input/output behavior 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 analyses are applied dependent on the criticality of the program. Some parts might not be checked at all.
Certain annotations are added to the programs. Written as Ada comments:

- Specific to the 3 levels of analysis.
- Ignored by Ada compilers.
- Used by the SPARK Ada tools.
- Syntax: start with --#', e.g.:

```ada
--# global in out
MyGlobalVariable;
```

- Ignored by Ada compilers.
Data Flow Analysis – Parameters

- In Ada all parameters have to be labeled as:
  - input parameters; symbol: in
  - output parameters; symbol: out
  - input/output parameters; symbol: in out

Example:

procedure ABC(A: in Float;
B: out Integer;
C: in out Colour)
Examiner verifies that

- * input/output parameters are initialized.
- * input/output parameters are not read before being initialized.
- * output parameters are not read before being initialized.
- * output parameters are but used at least once.
- * input parameters are not modified.
- * input parameters are read, and modified.

Data Flow Analysis – Parameters (Cont.)
Data Flow Analysis carries out the same analysis as for parameters.

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

Syntax examples by annotations:

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

(C) Anton Setzer 2003 (except for pictures)
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.

• Package variables = variables global to a package.

-- #initializes x;
Syntax example:

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

-- own x, y;
Syntax example:

However, even an uninitialized package variable is allowed to be used by a
procedure.
If a package is used it has to be declared:

```
# inherits MyPackage;
```

Syntax example:

- If a package is used it has to be declared:
Consider the following wrong program, which should exchange \( X \) and \( Y \):

```plaintext
procedure Exchange(X: in out Float)

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

Mistake: body should be:

```plaintext
T := X
X := Y
Y := T
begin
Float T := X := Y := X := T
end
```

Example (Data Flow Analysis)
Dataflow 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.)
Example 2 (Data Flow Analysis)

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

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

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

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

\[ \text{Importation of initial value of X is ineffective.} \]
Information Flow Analysis

- Information flow verifies that these dependences are fulfilled.

```
# derives X from Y
```

or, if nothing is used

```
# derives X from Y &
```

or

```
# derives X from Y
```

or

```
# derives X from Y &
```

Syntax examples:

Additional annotations on how variables depend on each other.
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);
begin
  T:=X; Y:=X; Y:=T; Z:=Z+1;
end ExchangeAndCount;
```

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

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

The error is that Z:=Z+1; should be replaced by Z:=Z+T;.
Example (Information Flow Analysis: Cont.)

- Data flow analysis succeeds without problems.
- Information flow analysis gives warning, since \( Z \) depends on \( Z \) and \( X \).
For procedures without loops, two kinds of annotations are relevant:

- 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.

When examining generator 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.

\begin{align*}
\text{pre} \: M &> 0 \quad \text{and} \quad M < 0; \\
\text{post} \: M &> 0.
\end{align*}
In the post-conditions:

- quantifiers for all, for some.
- Boolean connectives and, or, not, xor, 
  \(<\rightarrow\), \(<\triangleleft\) ,

Formulas are built using:

- Procedures without Loops (Cont.)

execute \( X + 1 \).

The value of \( X \) after executing the procedure is the value of \( X \) before executing it.

E.g. \( X = X + 1 \) if \( X \) stands for the value after executing it.

In the post-conditions:
Example (Proof Conditions)

Assume the following wrong program:

```pascal
procedure Exchange(X,Y: in out Float);
-- #derives X from X & -- # Y from Y;
begin
  T := X; X := T; T := Y; Y := T;
end Exchange;

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

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

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

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

This 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.

(C) Anton Setzer 2003 (except for pictures)
Example 2 (Proof Conditions)

Assume the correct program:

```
procedure Exchange(X: in out Float; 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;
```

Critical Systems, CS 411, Lent Term 2003, Sec. A3

A3-50
The examiner generates the formula:

Example 2 (Proof Conditions, cont.)
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 \( T < 0.0 \)):

E.g., in the previous example, insert between \( T = X \) and \( X = Y \):

One can insert in between the procedures a check condition.

Check Conditions – Proof Conditions –
The returned value is $\lt Y$.

The example expresses:
If $Y$ is a parameter or a global parameter:

```
-- # return X <= X < Y;
write:
```

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

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

expresses: the result is $X + 1$.

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

E.g.

If one has functions, one can either state the result of the function:
Proof Conditions

Procedures with Loops

- From \texttt{assert} follows, if one exit condition is true, the \textit{post condition}.
- From \texttt{assert} follows, if exit conditions are false, the condition of \texttt{assert}.
- From the pre-condition follows, when first entering the loop, the condition.

\begin{itemize}
\item \texttt{assert} \texttt{X Y = X + Y}.
\item The syntax is for instance:
\end{itemize}

**Critical Systems, CS 411, Lenterm 2003, Sec. A3**
Example (Proof Conditions with Loop)

procedure test(X,Y: inout Float)

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

\[ C1: x + (y + 1) = x + y - x + y = y + x = x + y. \]

\[ H1: x > 0. \]

\[ H2: \neg (x - 1 > 0). \]

\[ H3: \text{true}. \]

\[ \neg y + \neg x = x + y. \]
All proof conditions are verified by the simplifier.

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

\[\text{H1: } x + y = x + y\]
\[\text{H2: } x - 1 \underbrace{\sim y + \sim x = x + y}_{\text{C1: } x - 1 + (y + 1)}\]
\[\text{C2: } x - 1 \underbrace{> 0}_{\text{C2: } x - 1} \sim y \sim y + \sim x = x + y\]
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.

Other Annotations