Main Criteria for Choice of Programming Languages for Critical Systems

(a) Overview

Main Criteria for Choice of Programming Languages for Critical Systems

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

- Complexity of definition:
  - Are there simple, formal definitions of the language features?

- Bounded space and time requirements:
  - Can it be shown that time and memory constraints are not exceeded?

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

- Expressive power:
  - 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.

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

(b) SPARK Ada

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

(c) Overview

(d) Overview
Common Reasons for Program Errors

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

- **Aliasing.**
  - Two or more variables share the same storage location.

  - Changing one variable changes a seemingly different one.

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

- **Wild jumps.**
  - Can a language override an arbitrary memory location?

  - Can a language guarantee that a program cannot jump to an arbitrary memory location?

Comparison of Programming Languages

- **Differing behavior of compilers of the same language in case of arithmetic errors.**

- **Expression evaluation errors.**

- **Operational arithmetic.**

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

- **Model of mathematics.**

Comparison of Programming Languages (Cont.)

- **Wild jumps.**
  - Variables in the calling environment are unexpectedly changed.

- **Arithmetic errors.**
  - Out-of-range array subscript, division by zero, arithmetic overflow.

- **Different behavior of compilers of the same language in case of arithmetic errors.**

- **Expression evaluation errors.**

  - Can a language guarantee that a program cannot jump to an arbitrary memory location?
## 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?

---

### Legend for next slide:
- + means protection available
- - means no protection
- ? means partial protection
- + means protection available

---

### 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?
CORAL66 = compiled structured programming language related to Algol. Developed at the Royal Radar Establishment, Malvern, UK. Used for real-time systems. Allowed inline assembly code. No free CORAL 66 compilers seem to be available today.

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

Analysis

* Same problem as for Modula-2: Limited industrial use.
  - Development of new languages for high integrity software.
  - Problem: Limited industrial use.
  - No industrial use contributes to reliability of compilers.
  - Therefore lack of tools, compilers.

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

C most unsuitable language.

Modula-2 most suitable.


Rely on standard compilers and support tools.

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

Case study reveals:

- Problem: Are compilers safe?
- Additional checks required to the language.
- Only additional checks which verify that the program is in the subset.


Examples (cont.)

<table>
<thead>
<tr>
<th>Safe Subsets</th>
<th>CORAL S, SPAD, Modula 2, Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Subject</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Analysis (Cont.)

- Problem: Are compilers safe?
  - Casestudy revealed:
    - Compiler faultsare equivalent to one undetected fault in 50,000 lines of code.
    - Case study reveals:
      - Problem: Are compilers safe?
      - Additional checks required to the language.
      - Only additional checks which verify that the program is in the subset.


Examples (cont.)

<table>
<thead>
<tr>
<th>Safe Subsets</th>
<th>CORAL S, SPAD, Modula 2, Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Subject</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Remarks on CORAL 66

No free CORAL 66 compilers seem to be available today.

Allowed inline assembly code.

Used for real-time systems.

Developed at the Royal Radar Establishment, Malvern, UK.

CORAL 66 = Compiled structured programming language related to Algol.
**Programming Languages Used**

**Aerospace.**
- Trend towards Ada.
- Types of languages like FORTRAN, JOVIAL, C, C++.
  - 140 languages used in development of the Boeing 757/767.
  - 75 languages used in development of the Boeing 747-400.
  - E.g. C++ for the seatback entertainment system of the Boeing 777.
  - Northrop B2 bomber control system: C++
  - E. C++ for the seatback entertainment system of Boeing 777.
  - 75 languages used in development of the Boeing 757/767.
  - Use of languages like FORTRAN, JOVIAL, C++.

**Railway industry:**
- Ada as de-facto standard.
- Denver Airport baggage system written in C++, but initial problems probably
- Much assembler, also C, C++, Modula-2

**Automotive systems:**
- Most money spent on maintaining software, not developing it.
- Not directly related to the use of C++.
- Problems particular to embedded systems.
- Hard to reuse.
- No modularity.
- Existing languages too primitive.

**Motivation for Developing Ada**

- Many problems and difficulties in software development.
- Ada was developed for military applications.
- Name Ada = name of Ada Lovelace.
- First release: Ada 83.
- First release: Ada 83.
- Name Ada = name of Ada Lovelace (1815-1852).
- Ada was developed for military applications.
- Decision by DOD: Development of new standard programming language for military applications.
- Decision by DOD: Development of new standard programming language for military applications.
- Trend towards Ada for the high-integrity parts of the software.
- Use of Ada in the development of the Boeing 757/767.
- Use of Ada in the development of the Boeing 757/767.
- Trend towards Ada.
- Trend towards Ada.

**Specialty:**
- Air traffic control systems in U.S. Canada, France: Ada.
- European space agency: use of Ada in mission-critical systems.

**Critical Systems, CS 411, Lent tterm 2003, Sec. A3"**
Reasons for the use of Ada in Critical Systems

Ada is the often required standard for military applications in the US, and has therefore adopted by the Western military industry.

Software for military applications forms a large portion of critical systems.

SPARK Ada code compiles with standard Ada compilers.

Factors for Programming Languages

Addressed by SPARK Ada

- Simplicity of language definition
- Logical soundness
- No unusual types
- No errors (continuation)
- No undefined types, but no complete compiler time checking
- Omission of too complex principles

Factors for Programming Languages (Cont.)

Addressed by SPARK Ada

- Omission of language definition
- 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: derive order of evaluation.
  * $x := 0
  * if $x$ then if $Z = 0$ then
  * $E := F(x)$ and $G(x)$ have side-effects.
  * Order of execution not specified.
  * Problem: statement like $x := 0$ if $F(x)$ and $G(x)$ have side-effects.

Factors for Programming Languages

Addressed by SPARK Ada

- Omission of language definition
- Logical soundness
- No unusual types
- No errors (continuation)
- No undefined types, but no complete compiler time checking
- Omission of too complex principles

Factors for Programming Languages (Cont.)

Addressed by SPARK Ada

- Omission of language definition
- 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: derive order of evaluation.
  * $x := 0
  * if $x$ then if $Z = 0$ then
  * $E := F(x)$ and $G(x)$ have side-effects.
  * Order of execution not specified.
  * Problem: statement like $x := 0$ if $F(x)$ and $G(x)$ have side-effects.

Factors for Programming Languages

Addressed by SPARK Ada

- Omission of language definition
- 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: derive order of evaluation.
  * $x := 0
  * if $x$ then if $Z = 0$ then
  * $E := F(x)$ and $G(x)$ have side-effects.
  * Order of execution not specified.
  * Problem: statement like $x := 0$ if $F(x)$ and $G(x)$ have side-effects.
Factors for Programming Languages

Addressed by SPARK Ada

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

- **Security.**
  - Features already supported by Ada.
  - Allows to specify strong assertions about variables.
  - Hiding of variables allowed.

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

- **Bounded space and time requirements.**
  - Programs do not stray outside the computational model (one cannot jump or write to an arbitrary memory location).
  - Constraints (array bounds) have to be static (determined at compile-time).
  - In order to be verifiable at compile-time, programs do not stray outside the computational model (one cannot jump or write to an arbitrary memory location).

- **Complex time.**
  - In order to be verifiable at compile-time, programs do not stray outside the computational model (one cannot jump or write to an arbitrary memory location).
  - Programs do not stray outside the computational model (one cannot jump or write to an arbitrary memory location).

Languages should be as explicit as possible.

- No polymorphism (ie. that an operation is defined for different types).
- No dynamic dispatch.
- No overloading of functions.
- No array shifting.

- No object creation.
- No default parameters.

- No overloading.
- No arrays let alone.
- No strings, only kind of operations are allowed.
- No overloading of arrays.
- No overloading of variables.

- No recursion.
- No arrays.
- No anonymous subtypes.

Typedef x of a package x\#package\# has to be referenced as:
* x\#package\# x = ...

Languages should be as explicit as possible.

- No anonymous ranges.
- No polymorphism (ie. that an operation is defined for different types).
- No array shifting.
- No overloading of functions.
- No default parameters.
- No overloading.
- No arrays.
Factors for Programming Languages Addressed by SPARK Ada

- No structures which are too complex to grasp.

Programs:
- Generation of proof conditions which allow to prove correctness of

Verification conditions:
- Verifies interdependencies between variables.

Information flow analysis:
- Checks that changed and imported variables are used later (possibility of output variables).
- Checks initialization of variables.
- Checks input/output behavior of parameters and variables.

Data flow analysis:

Three Levels of Analysis

Data flow analysis:
- Checks input/output behavior of parameters and variables.

- Restriction on return statements and exits.
  - However, subclasses (restriction of the range of a type) allowed.
  - Properties of inheritance: properties are inherited remotely.
  - Therefore no inheritance.

- No derived classes (essentially a copy of a type).
- No class-wide types (classes which have subclasses).
- No type extension (subtypes in object-oriented programs).

- No inheritance.

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

- No exit out of if condition (since there the innermost loop is the if-statement).
  - No exit out of if condition (since there the innermost loop is the if-statement).

SPADE-examiner:
- Verifies the following:
  - Correct Ada syntax.
  - SPAK-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.
  - SPARK subset of Ada chosen, as described above.

SPADE-proof-checker:
- Prooftool for interactively proving verification conditions.

Architecture of the SPARK Ada

Architecture of the SPARK Ada (cont.)
Three Levels of Analysis (Cont.)

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.)

Data Flow Analysis {Parameters (Cont.)}

Annotations

Example:

Data Flow Analysis

Annotations

Data Flow Analysis (Cont.)
Data Flow Analysis

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 inout C
```

Data now analysis carries out the same analysis as for parameters.

• Variables by annotations.

```
-- #global in our C
-- #global our D
-- #global in A
-- #global inout B
```

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

Data Flow Analysis - Packages (Cont.)

Package variables = variables global to a package.

If a package is used it has to be declared:

```
#package MyPackage;
```

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

```
#own X, Y;
```

Package variable must be declared by annotations.

```
package MyPackage = variables global to a package;
```

Neither parameters nor global variables can be changed.

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

Functions have only read access to global parameters.

Critical Systems, CS411, Lentterm 2003, Sec.A3
Example (Data Flow Analysis)

Consider the following wrong program, which should exchange X and Y:

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

Mistake: Body should be:
```
T := X; X := Y; Y := T
```

The following statements result 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.)

Consider the following wrong program, which should exchange X and Y:

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

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

Example (Data Flow Analysis)

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

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

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

Information flow verifies that these dependencies are fulfilled.
Example (Information Flow Analysis; Cont.)

Data analysis succeeds without problems.

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

Proof Conditions

1. Procedures without Loops
   - Pre-conditions, e.g.:
     \[ M \geq 0 \land N < 0 \]

2. Post-conditions, e.g.:
   \[ F \]

For procedures without loops, two kinds of annotations are relevant:

- Claims for all
- Boolean connectives and operators

Formulas are built using:

- Pre-condition holds after executing the procedure is the value of \( X \) before executing the procedure + 1.
- \( X \) stands for the value of \( X \) before executing the procedure.
- \( X \) stands for the value of \( X \) after executing the procedure.
- \( E \) expresses:
  \[ X = X \land T \]

\[ \text{if the pre-condition holds, and the procedure is executed, afterwards the post-condition holds.} \]

\[ \text{If there are no pre-conditions, then the formula expresses that the post-condition always holds.} \]

\[ \text{If the pre-condition holds and the procedure is executed, afterwards the post-condition holds.} \]

\[ \text{The examiner enters the formulas which express:} \]

- \[ \text{post} F = \text{post} M = M + 1 \]

- \[ \text{pre} F = \text{pre} M \]

- \[ \text{post} F = \text{pre} M + 0 \]

- \[ \text{post} F = \text{pre} M + 0 \]

Procedure ExchangeAndCount:

\[ \text{begin} \]

\[ T \leftarrow X ; Y \leftarrow T \land T \]

\[ Z = \text{ExchangeAndCount} (X, Y, Z) \]

\[ \text{in out integer} \]

\[ \text{count the number of exchanges in } Z \]

\[ \text{Consider the following wrong program, which should exchange } X \text{ and } Y \text{ and } \]

\[ \text{count the number of exchanges in } Z \]

\[ \text{The error is that } Z = Z + 1 \text{ should be replaced by } Z := Z + 1 \]

\[ \text{end} \]
Assume the following wrong program:

```pascal
procedure Exchange(X, Y: inout 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 postcondition is not fulfilled in general.

Example (Proof Conditions, Cont.)

The examiner generates the formula:

- **H1**: `x > = 0.0`.
- **H2**: `true`.
- **H3**: `true`.
- **C1**: `x = y`.
- **C2**: `y = x`.

which is not provable.

(Thedata and information now check is so good, that it is difficult to find simple but wrong programs, which pass it.)

Assume the correct program:

```pascal
procedure Exchange(X, Y: inout Float);
-- #derives Y from X
-- #X from Y
-- #post X = Y and Y = X;

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

The postcondition is fulfilled in general.

Example (Proof Conditions, Cont.)

The examiner generates the formula:

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

The simplifiers show 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:

```
-- # check T > 0.0;
```

Now the formulas express:

1. From the pre-condition follows the post-condition.
2. The post-condition follows the check-condition.
3. If one exits the loop, the condition.

Example (Proof Conditions with Loop)

```
procedure test(X,Y:inout Float)
-- derives X from X, Y;
-- returns X+Y ;
-- pre X > 0.0;
-- assert X+Y=X+Y;
-- loop begin
-- if X < 0.0;
-- exit;
-- endloop;
endtest;
```

Procedure with Loops

If one has a loop, a loop invariant is required.

```
assert X+Y=X+Y;
```

The syntax is for instance:

```
-- # assert X+Y=X+Y;
```

If one has a loop, a loop invariant is required.

```
-- assert X+Y=X+Y;
```

The syntax is for instance:

```
-- # assert X+Y=X+Y;
```

If one has a loop, a loop invariant is required.

```
-- assert X+Y=X+Y;
```

The syntax is for instance:

```
-- # assert X+Y=X+Y;
```

If one has a loop, a loop invariant is required.

```
-- assert X+Y=X+Y;
```

The syntax is for instance:

```
-- # assert X+Y=X+Y;
```

If one has a loop, a loop invariant is required.
All proof conditions are verified by the simplifier.

\[
\begin{align*}
0 > x & > 0. \\
C_1 & : x + y = x + y. \\
C_2 & : x > 1. \\
C_1 & : x + y = x + y. \\
\end{align*}
\]

The main program is declared by:

```
#main-program
```

\[
\begin{align*}
0 > x & > 0. \\
C_1 & : x + y = x + y. \\
C_2 & : x > 1. \\
\end{align*}
\]

The examiners generate in the last example the following proof conditions:

\[
\begin{align*}
\neg y + \neg & x = \neg y + \neg x. \\
C_1 & : x + y = x + y. \\
C_2 & : x > 1. \\
\end{align*}
\]