THE UNIVERSITY OF MELBOURNE

SWEN90010 — High Integrity Systems Engineering

Lecture 13 - Safe Language Subsets, SPARK Ada

Hira Syeda
hira.syeda@unimelb.edu.au
Level 2, Melbourne Connect

16th April, 2025

In the quest of programming high-integrity systems

THE UNIVERSITY OF MELBOURNE

Today, we continue our exploration of building systems with high assurance, integrity, and security

Last time, we looked at Ada—a language designed to make code more predictable and reliable

This lecture takes us a step further into Spark, a safer subset of Ada

Spark allows us to write bulletproof code that can be formally proven to be correct and secure

What's wrong with this code?

THE UNIVERSITY OF MELBOURNE

```
int binarySearch(int [] list, int target) {
    int low = 0;
    int high = len(list) - 1;
    int mid;
    while(low <= high) {
        mid = (low + high)/2;
        if (list[mid] < target) {
            low = mid + 1;
        } else if (list[mid] > target) {
            high = mid - 1;
        } else {
            return mid;
        }
    }
    return -1;
}
```

What's wrong with this code?

THE UNIVERSITY OF MELBOURNE

```
int binarySearch(int [] list, int target) {
    int low = 0;
    int high = len(list) - 1;
    int mid;
    while(low <= high) {
        mid = (low + high)/2;
        if (list[mid] < target) {
            low = mid + 1;
        } else if (list[mid] > target) {
            high = mid - 1;
        } else {
            return mid;
        }
    }
    return -1;
}
```

THE UNIVERSITY OF MELBOURNE

SAFE LANGUAGE SUBSETS

what do we mean by “safe” in “safe language”

THE UNIVERSITY OF MELBOURNE

when we say “safe language”, we mean a programming language that is **designed** to prevent or eliminate classes of common programming errors that could lead to u**npredictable, incorrect, or insecure behaviour** at runtime

what do we mean by “safe” in “safe language”

THE UNIVERSITY OF MELBOURNE

a safe language:

- disallows undefined behaviour
- performs strong compile-time checks
- has clear, well-defined semantics
- restricts or controls low-level operations
- supports formal verification or runtime checks

Safe Languages

Which is safer? Ada C

Safe Languages

Which is safer? Ada C

What might make a programming language “safe”?

Predictability: when programs behave as the programmer intended

Safe Languages

Which is safer? Ada C

What might make a programming language “safe”?

Predictability: when programs behave as the programmer intended

Implies **Verifiability**

Doesn’t always mean that the program won’t crash.

Does mean that safe languages are simpler than unsafe ones.

Design Space

Clean Slate: Design a new, safe programming language

Safe Subset: use an existing language,
without all of its unsafe bits.

Design Space

Clean Slate: Design a new, safe programming language

e.g. Rust

Safe Subset: use an existing language,
without all of its unsafe bits.

Design Space

Clean Slate: Design a new, safe programming language

e.g. Rust

Safe Subset: use an existing language,
without all of its unsafe bits.

e.g. SPARK Ada, MISRA C

Safe Subsets

THE UNIVERSITY OF MELBOURNE

Removes unsafe parts from a language.

Philosophy: Make programmer work harder
in exchange for much simpler testing and verification.

Safe Subsets

THE UNIVERSITY OF MELBOURNE

Removes unsafe parts from a language.

Philosophy: Make programmer work harder
in exchange for much simpler testing and verification.

Advantages

Safe Subsets

Removes unsafe parts from a language.

Philosophy: Make programmer work harder
in exchange for much simpler testing and verification.

Advantages

Programs in a safe subset can be compiled
and debugged using existing toolchains.

Safe Subsets

Removes unsafe parts from a language.

Philosophy: Make programmer work harder
in exchange for much simpler testing and verification.

Advantages

Programs in a safe subset can be compiled
and debugged using existing toolchains.

Programmers don’t need to learn a new
programming language.

Safe Subsets

Removes unsafe parts from a language.

Philosophy: Make programmer work harder
in exchange for much simpler testing and verification.

Advantages

Programs in a safe subset can be compiled and debugged using existing toolchains.

Programmers don't need to learn a new programming language.

Resulting programs far more tractable to **automatically** analyse and verify.

NASA's Subset of C (non-examinable)

A subset of MISRA C (itself a C subset)

1. Language Compliance

(a) **Do not stray outside the language definition.**
(b) Compile with all warnings enabled; use static source code analysers.

2. Predictable Execution

(a) Use verifiable loop bounds for all loops meant to be terminating.
(b) **Do not use** direct or indirect **recursion**.
(c) **Do not use dynamic memory allocation** after task initialisation.
(d) **Use IPC messages** for task communication.
(e) **Do not use task delays** for task synchronisation.
(f) Explicitly transfer write-permission (ownership) for shared data objects.
(g) **Place restrictions on the use of semaphores and locks.**
(h) Use memory protection, safety margins, barrier patterns.
(i) **Do not use goto**, setjmp or longjmp.
(j) Do not use selective value assignments to elements of an enum list.

NASA's Subset of C (non-examinable)

A subset of MISRA C (itself a C subset)

THE UNIVERSITY OF MELBOURNE

3. Defensive Coding

(a) Declare data objects at smallest possible level of scope.
(b) Check the return value of non-void functions, or explicitly cast to (void).
(c) Check the validity of values passed to functions.
(d) Use static and dynamic assertions as sanity checks.
(e) **Use U32, I16, etc instead of predefined C data types such as int, short, etc.**
(f) Make the order of evaluation in compound expressions explicit.
(g) **Do not use expressions with side effects.**

NASA's Subset of C (non-examinable)

A subset of MISRA C (itself a C subset)

4. Code Clarity

(a) Make only very limited use of the C pre-processor.
(b) Do not define macros within a function or a block.
(c) Do not undefine or redefine macros.
(d) Place #else, #elif, and #endif in the same file as the matching #if or #ifdef.
(e) Place no more than one statement or declaration per line of text.
(f) Use short functions with a limited number of parameters.
(g) Use no more than two levels of indirection per declaration.
(h) Use no more than two levels of dereferencing per object reference.
(i) Do not hide dereference operations inside macros or typedefs.
(j) **Do not use non-constant function pointers.**
(k) **Do not cast function pointers into other types.**
(l) Do not place code or declarations before an #include directive.

NASA's Subset of C (non-examinable)

A subset of MISRA C (itself a C subset)

4. Code Clarity

(a) Make only very limited use of the C pre-processor.
(b) Do not define macros within a function or a block.
(c) Do not undefine or redefine macros.
(d) Place #else, #elif, and #endif in the same file as the matching #if or #ifdef.
(e) Place no more than one statement or declaration per line of text.
(f) Use short functions with a limited number of parameters.
(g) Use no more than two levels of indirection per declaration.
(h) Use no more than two levels of dereferencing per object reference.
(i) Do not hide dereference operations inside macros or typedefs.
(j) **Do not use non-constant function pointers.**
(k) **Do not cast function pointers into other types.**
(l) Do not place code or declarations before an #include directive.

Maximise: portability, predictability, simplicity.

On Simplicity

THE UNIVERSITY OF MELBOURNE

"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other is to make it so complicated that there are no obvious deficiencies."

On Simplicity

THE UNIVERSITY OF MELBOURNE

"*There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other is to make it so complicated that there are no obvious deficiencies.*"

Tony Hoare (ACM Turing Award Lecture, 1980)

SPARK Ada

Start with Ada (already pretty safe language)

Remove:

Dynamic Memory Allocation

Tasks (but see Ravenscar Profile)

Goto

Unrestricted Access Types and Aliasing (pointers)

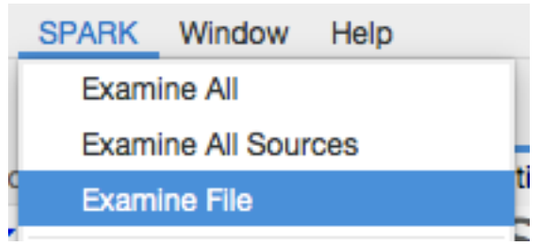
Side-effectful expressions and functions

Exception Handling

Add: Annotations

SPARK Examiner

THE UNIVERSITY OF MELBOURNE



Checks for

- conformance to SPARK restrictions
- unused assignments,
- uses of uninitialised data,
- missing return statements
- violations of flow contracts (“Depends”)

```
package Swapping with
      SPARK_Mode => On
is

   procedure Swap (X, Y : in out Float) with
     Depends => (X => Y,
                 Y => X);

end Swapping;
```

SPARK Prover

Checks for

- conformance to SPARK restrictions
- Plus: uses of uninitialised data,
- possible run-time errors, and
- functional contracts (“Pre/Post” conditions)

```
procedure Swap (X, Y : in out Integer)
  with Post => (X = Y'Old and Y = X'Old);
```

```
function Divide (X, Y : Integer) return Integer
  with Pre  => Y /= 0 and (if X = Integer'First then Y /= -1),
       Post => Divide'Result = X / Y;
```

SPARK Annotations

Mostly By Demonstration