Lecture aims

1. Introduce "Correctness by construction"
2. Motivate the use of safe programming language subsets.
3. Present the SPARK language.
4. Show examples of annotations in SPARK.

Lecture plan

Correctness by construction — introduction

1. Show the quote from Tony Hoare below.
2. Introduce the idea of *Correctness by construction* (from Altran Praxis): keeping things simple to make them easier to verify.
3. Principles of correctness by construction:
   (a) Expect requirements to change.
   (b) Know why you're testing: finding faults, not showing correctness.
   (c) Eliminate errors before testing. Testing is the second most expensive way of finding errors. The most expensive is to let your customers find them for you.
   (d) Write software that is easy to verify. Testing takes up to 60% of total effort. Implementation typically takes 10%. Doubling implementation effort is cost-effective if it reduces verification cost by as little as 20%.
   (e) Develop incrementally. Make small changes, incrementally. After each change, verify that the updated system behaves according to its updated specification.
   (f) Some aspects of software development are just plain hard. The best tools and methods take care of the easy problems, allowing you to focus on the difficult problems.
   (g) Software is not useful by itself. The executable software is only part of the picture: user manuals, business processes, design documentation, well-commented source code and test cases should not just be added at the end.
4. Today: introduce safe programming language subsets, which are simple parts of a language.

Safe programming language subsets

1. Define safe programming language subset: use structured programming to provide a small but powerful programming language that is easier to verify than non-safe languages.
2. Programming languages are inherently evolutionary. New features added without thought of impact on safety. Most features are intended to be programmer friendly, but not machine/-maintainer/reader friendly.

3. Examples: SafeD (safe subset of D — itself related to C++), MISRA C (safe subset of C), Joe-E (*secure* subset of Java), and SPARK.

4. Show coding standard for NASA's Curiosity Mars Rover (below).

SPARK

1. $SPARK = safe(Ada) + annotations.$

2. SPARK: Take Ada, remove unsafe constructs to leave a simple structured programming language, and add new annotations to support static checking.

3. Annotations are extensions to Ada, so need to switch to "SPARK mode". However, all new versions of Ada compilers will compile the code — extra tools process annotations.

4. Features left out of Ada:

   - Dynamic memory allocation
   - Tasks
   - Gotos
   - Exceptions
   - Generics
   - Access types (similar to references in other languages)
   - Recursion

   Ada favours reader over programmer. SPARK carries this principle to further extremes. Why? Because the reader may be a software tool used to verify the program.

5. Q: Doesn't this make programmer harder?

   A: Yes, but.... $\approx 50\%$ of effort is in verification and testing (more for certifiable systems) — and 10% in programming. So, if we make programming a bit more long winded, we only need minor improvements in verification to get that effort back again.

6. Note symmetry between NASA JPL's coding standard and SPARK.

7. SPARK annotations: formal comments. Improve documentation and allow consistency checks. Contain information about a program.

8. Two annotations today: "global" and "derives".

9. SPARK Examiner: checks conformance with SPARK (syntax, annotation consistency), and provides debugging information and warnings.

10. Go through examples from subject notes using SPARK Examiner.

11. Note that the last three examples are directly from Software Engineering Methods:

    (a) Ineffective statement: du anomaly.
    (b) Ineffective initialisation: dd anomaly.
    (c) Reference to undefined value: ur anomaly.

Coding rules for NASA's Mars Rover *Curiosity*

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.
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.
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.

5. MISRA *shall* compliance

   (a) All MISRA shall rules not already covered at Levels 1-4 (73 rules in total).

"*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.*" — Professor Tony Hoare, 1980 ACM Turing Award Lecture

```
package body IncorrectReassignment  with
     SPARK_Mode => On
is

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

```
gnatprove -P/home/tmill/subjects/swen90010/repository/trunk/course_notes/spark/code/de
Phase 1 of 2: frame condition computation ...
Phase 2 of 2: analysis of data and information flow ...
incorrectreassignment.ads:5:20: warning: unused initial value of "X"
incorrectreassignment.adb:8:9: warning: unused assignment
gprbuild: *** compilation phase failed
gnatprove: error during analysis of data and information flow, aborting.
```

```
package body WeirdSwap  with
     SPARK_Mode => On
is

   procedure Swap (X, Y : in out Float) is
      T : Float;
   begin
      if X < Y then
         T := X;
      end if;
      X := Y;
      Y := T;
   end Swap;
end WeirdSwap;
```

```
gnatprove -P/home/tmill/subjects/swen90010/repository/trunk/course_notes/spark/code/de
Phase 1 of 2: frame condition computation ...
Phase 2 of 2: analysis of data and information flow ...
weirdswap.adb:12:9: warning: "T" might not be initialized
gprbuild: *** compilation phase failed
gnatprove: error during analysis of data and information flow, aborting.
```

```
package body InitialiseLocal  with
     SPARK_Mode => On
is

   procedure Swap (X, Y : in out Float) is
      T : Float := 0.0;
   begin
      T := X;
      X := Y;
      Y := T;
   end Swap;
end InitialiseLocal;
```

```
gnatprove -P/home/tmill/subjects/swen90010/repository/trunk/course_notes/spark/code/de
Phase 1 of 2: frame condition computation ...
Phase 2 of 2: analysis of data and information flow ...
initialiselocal.adb:6:7: warning: initialization has no effect
gprbuild: *** compilation phase failed
gnatprove: error during analysis of data and information flow, aborting.
```