SWEN90010: HIGH INTEGRITY SYSTEMS ENGINEERING
DEPARTMENT OF COMPUTING AND INFORMATION SYSTEMS
THE UNIVERSITY OF MELBOURNE

Workshop 9

Introduction

In the lectures, we discussed at the SPARK language, and discussed some static analysis that is possible using the SPARK examiner. In this workshop, we will use the SPARK examiner and simplifier to prove some important properties about our programs. If these proofs pass, we can guarantee that our source code is free from some fairly serious problems that are permitted in non-safe languages.

This workshop will look at just some of the most important properties.

The SPARK tools

In this subject, we will be looking mainly at the SPARK examiner and the GNATProve. These tools prove certain properties about some source code using static analysis. The properties are intended to find errors that will or could possibly result in run-time errors, but find them at design time.

In the first workshop, you would have installed GNAT GPL and SPARK GPL 2014 (unless working on the lab machines where they are already installed). If you did not install the SPARK tools, go back to workshop 1, find the directions, and install them.

Exercises

1. The source code accompanying the workshop contains a set of SPARK files for use. Create a new project using the source (including the `workshop.gpr` as a properties file). Compile the source code in `task1.adb` in the GPS environment.

   Next, run SPARK examiner over the source code in `task1.adb` by going to *Spark* → *Examine File* from the GPS menu.

   This reports that the variable `Ok` is used without being initialised, and the variables `I` and `J` are only initialised if `Ok` is true. Note the difference between a variable being *never* initialised (`Ok`), and a variable *possibly* being uninitialised (`I` and `J`).

   Note the differences between the compiler errors/warnings, and the SPARK examiner errors/warnings.

2. Modify the source code from task 1 to include an *ineffective statement*, which is a statement that can be removed from a program without changing the behaviour of that program. Run the examiner over the modification, and analyse the results.

   Do these types of problems look familiar? (HINT: Think back to data-flow analysis in SWEN90006!)

3. Now, open and compile the source code in `task3.adb`, and then run the SPARK examiner and GNATProve over the file. To run GNATProve, select *Spark* → *Prove File*. Click *Execute*.

   Why do you think the proof from `task3.adb` could not be proved?

4. Go to the source file `task3.adb` and uncomment the commented lines. Re-run the the SPARK examiner and GNATProve, and see what changes.

   Is the program still not proved? If not, try to change the program to correct this problem.

HINT: Remember `Integer'First` `Integer'Last` return the lowest and highest integers respectively.

If you are struggling with this task, complete the rest of the workshop and come back to it.

5. Run the SPARK examiner and GNATProve over the code `task4.adb`.

   The inability to prove this is actually an indication that the program is not a valid SPARK program (although this is not always the case – sometimes the tools are just not powerful enough to prove some properties).

   What do you think this failed proof relates to? How could you re-write this to arrive at a correct SPARK program? The relevant types are declared in `task4.ads`.

6. Modify line 5 of `task4.adb` from:

   ```
   AnArray(AnIndex) := AnArray(AnIndex) + 1;
   ```

   to:

   ```
   AnArray(AnIndex) := AnArray(0);
   ```

   Run the SPARK examiner over note the error message. This error will also be generated by the GNAT compiler.

7. Modify line 5 of `task4.adb` from:

   ```
   AnArray(AnIndex) := AnArray(AnIndex) + 1;
   ```

   to:

   ```
   AnArray(AnIndex) := AnArray(AnIndex + 1);
   ```

   Run the SPARK examiner over this and see what has failed to prove in `task4/task4procedure.siv`. What do you think this failed proof relates to?