THE UNIVERSITY OF MELBOURNE

SWEN90010 High Integrity System

Workshop 6 SPARK Tools

Chuang Wang

Contact

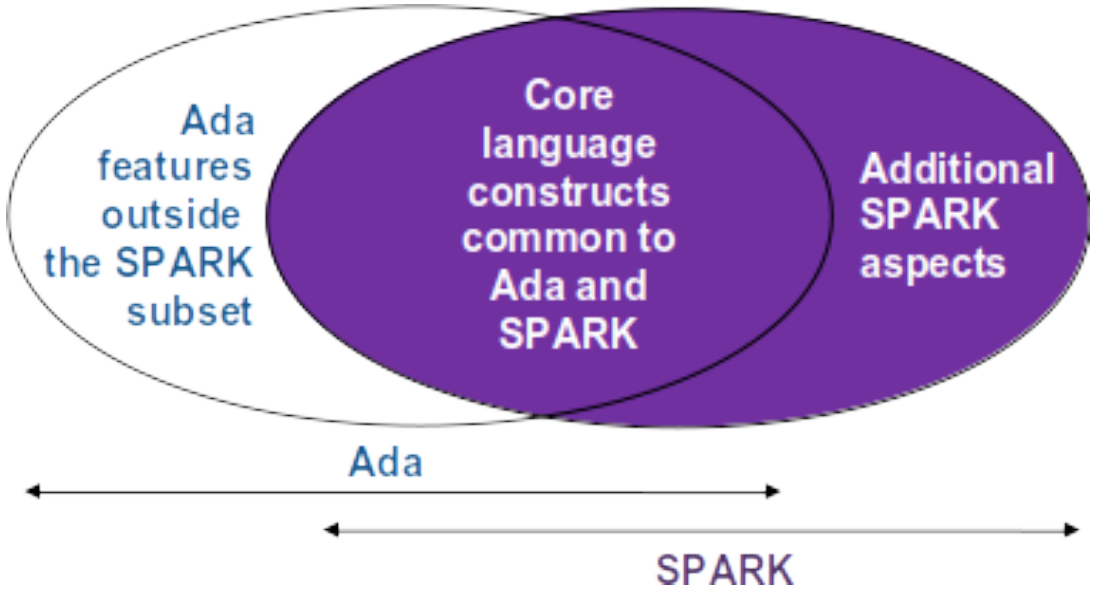
Email: chuangw1@unimelb.edu.au

LinkedIn:

THE UNIVERSITY OF MELBOURNE

1. Recap

Ada VS SPARK



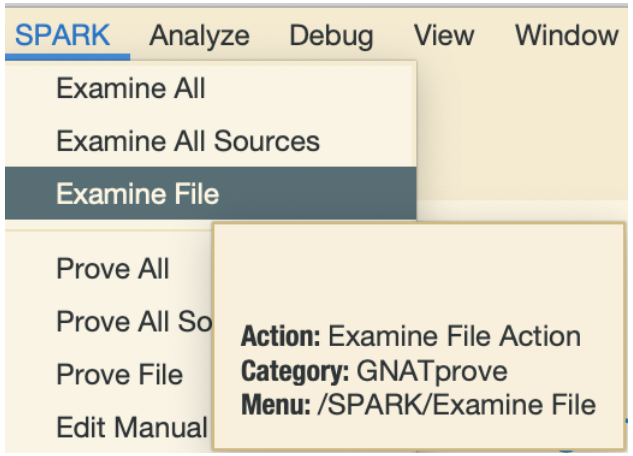
The proof stage of formal verification (which includes flow analysis)

- Flow contracts (“Depends”)
- Functional contracts (“Pre/Post” conditions)
- etc.

Referenced from https://learn.adacore.com/courses/intro-to-spark/chapters/01_Overview.html

SPARK Tools

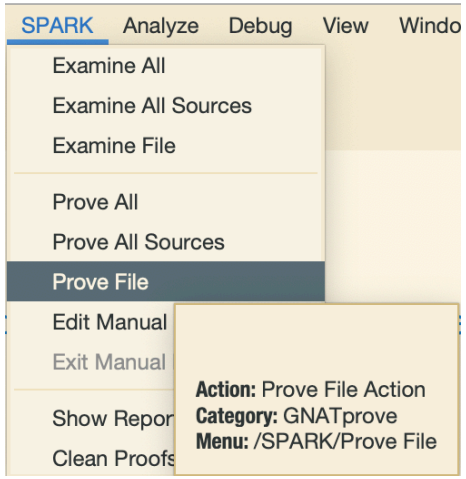
SPARK Examine




```
gnatprove -P workshop.gpr -u ./task1.adb --mode=flow
```

- Conformance to SPARK restrictions
- Unused assignments
- Uses of uninitialised data
- Missing return statements
- Violations of flow contracts (“Depends”)

SPARK Prove




```
gnatprove -P workshop.gpr -u ./task1.adb
```

- Conformance to SPARK restrictions
- Plus: uses of uninitialised data
- Possible run-time errors
- Functional contracts (“Pre/Post” conditions)

Data Anomaly

Three ways to act on a variable

1. **Define**: give it a value

   `e.g. x = 3`

2. **Reference**: read its value

   `e.g. if (x > 3)`

2. **Undefined**: give it no value

   `e.g. x = null`

Anomaly

1. u-r anomaly:
An **undefined** variable is **referenced**

2. d-u anomaly:
A **defined** variable is **not referenced** before becoming **undefined**

3. d-d anomaly:
A **defined** variable is **not referenced** before being **re-defined**

THE UNIVERSITY OF MELBOURNE

2. Exercise

Question 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.

Takeaway points

Compiler Warnings VS Errors

- **Warnings** do not stop compilation (unless the user wants them to) but inform that some code probably doesn't do what the user wants. This can be a useless with clause, or a loop where the loop condition never changes, or many more.
- **Errors** are used to indicate serious problems with the code which prevent the compiler from generating code for the program: syntax or typing errors, combinations of features which are not allowed by the language, etc.

Spark Warnings VS Errors

- **Warnings** both for **suspicious code** (e.g. dead code, unused value of a variable) and some **verification errors** (variable uninitialized on some paths, failing run-time checks or assertions)
- **Errors**, for **illegal code** and also for some **verification errors** (variable uninitialized on all paths)

Question 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!)

Question 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?

Question 3 - A

The proof is failing because the line:

```
Result := Input * 10;
```

can produce an integer overflow. That is, if `Input * 10` is greater than `Integer'Last`, then the computation will give an unexpected value.

Question 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.

Question 4 - A

4. If we uncomment the commented lines, the constraint that cannot be proved refers not to the line above, but to the conditions in the branch:

```
1     if Input * 10 <= Integer'Last and
2        Input * 10 >= Integer'First then
```

Again, the problem is that `Input * 10` could overflow. Although this is not assigned to a variable, our underlying architecture still only handles integers of a certain size. To check that the computation will not overflow, we instead have to switch around the computation to use division instead of multiplication:

```
1   procedure Task3Procedure(Input : in Integer; Result : out Integer) is
2   begin
3      if Input <= Integer'Last / 10 and
4        Input >= Integer'First / 10 then
5         Result := Input * 10;
6      else
7         Result := Input;
8      end if;
9   end Task3Procedure;
```

Question 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`.

Question 5 - A

5. For `task4.adb`, the problem again relates to an overflow: in this case, that the expression `AnArray(AnIndex) + 1` will overflow if `AnArray(AnIndex)` is equal to `Integer'Last`.

   To correct this, we insert a new guard:

```
1  procedure Task4Procedure(AnArray : in out MyArray; AnIndex : in Index) is
2  begin
3     if AnArray(AnIndex) < Integer'Last then
4        AnArray(AnIndex) := AnArray(AnIndex + 1);
5     end if;
6  end Task4Procedure;
```

What this means is that, using the SPARK tools, if all of the conjectures are successfully proved, then we have *no integer overflows in our program* — a very useful (and sometimes imperative!) property.

In the next workshop, we'll look at a different solution that involves using preconditions.

Question 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.

Question 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?

THE UNIVERSITY OF MELBOURNE

3. Open Discussion

THE UNIVERSITY OF MELBOURNE

Thank you !