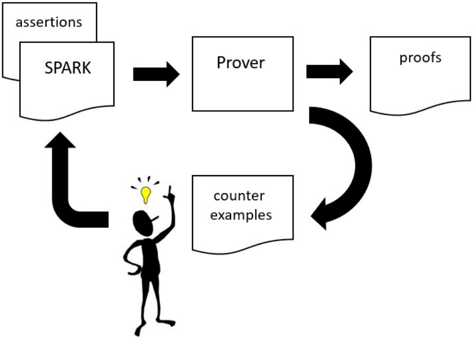
Rigorous Methods for Software Engineering (F21RS-F20RS)
Industrial Strength Program Verification

Andrew Ireland
Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Overview

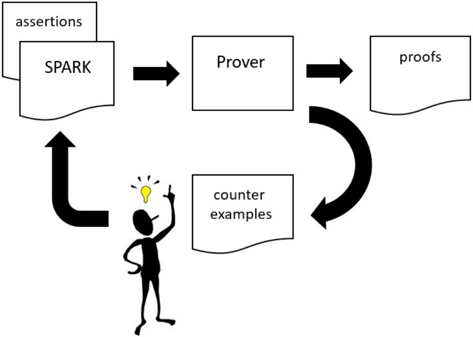
- Introduce the technique of program verification via the SPARK tool-set.
- Foucs on exception freedom and functional verification.
- The material in this lecture will directly support you with the formal verification tasks associated with the coursework.

Formal Verification of Code



- Code is verified with respect to a formal specification represented by assertions.
- An **assertion** is a logical statement which can be inserted at any point within the control flow of your program or within the contract (i.e. pre- and postconditions).
- Verification involves mathematical proof.

Formal Verification of Code



- GNATprove provides both automatic and interactive proof tools. In general program verification is undecidable hence the need for interactive proof tools. More details later in the lecture.
- When an automatic proof attempt fails, GNATprove generates a counter example, i.e. an assignment to program variables that makes an assertion false.

An Example of an Assertion

```
procedure Int_Dec(X: in out Integer)
is
begin
   if X > 0 then  X:= X-1; end if;
   pragma Assert (X >= 0);
end Int_Dec;
```

- Note that the `Assert` **pragma** allows an assertion to be inserted within the code.
- **Question:** will the above assertion always be true?

An Example of an Assertion

```
procedure Int_Dec(X: in out Integer)
is
begin
   if X > 0 then  X:= X-1; end if;
   pragma Assert (X >= 0);
end Int_Dec;
```

Using GNATprove to in **mode** `prove`:

```
...
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
... assertion might fail, cannot prove X >= 0 (e.g. when X = -1)
    [possible explanation: ... should mention X in a precondition]
```

Note that **"when X = -1"** is a counter example. Note that GNATprove also hints at how to overcome the failure, i.e. introduce a precondition *(that excludes negative integers)*. Preconditions are introduced in slide 11.

Constructing Assertions

Logical operators:

P **and** Q
P **or** Q
not P

Conditionals:

if P **then** Q
if P **then** Q **else** R

Quantification:

for all X **in** Y => P(X)
for some X **in** Y => P(X)

Note that quantification is required when verifying properties involving ranges, *e.g. for all elements of an array a given property is true.*

Exception Freedom Specification

- Specifying what must be true in order to prove that no run-time exceptions will occur.
- To achieve this, the GNATprove **automatically** inserts assertions corresponding to the places in the code where an Ada compiler inserts run-time checks, *e.g.* checking before a division that a divide-by-zero is not about to occur.
- These assertions specify what conditions must be true so that the run-time checks will not fail, i.e. thus proving exception freedom.
- By definition, SPARK eliminates many of the run-time exceptions that can be raised within Ada. However, **index**, **range**, **division** and **overflow** checks can still raise exceptions in SPARK code.
- Failed run-time checks may cause a program to crash with potential safety implications. But such failures may also be exploited by hackers, e.g. buffer overflows.

Integer Overflow: A Simple Example

```
...
package Inc_Value
is
type T is range -128 .. 128;

procedure Inc(X: in out T)
with
  Depends => (X => X);
end Inc_Value;
```

```
...
package body Inc_Value
is
procedure Inc(X: in out T)
is
 begin
   X:= X+1;
 end Inc;
end Inc_Value;
```

Calling procedure `Inc` with a value of 128 will cause an overflow, *i.e.* raise a `Constraint_Error` exception. As a consequence, exception freedom is unprovable, *i.e.* assuming that X is in the range $-128 \ldots 128$, then we can not prove $X + 1 \leq 128$.

Integer Overflow: A Simple Example

```
...
package Inc_Value
is
type T is range -128 .. 128;

procedure Inc(X: in out T)
with
  Depends => (X => X);
end Inc_Value;
```

```
...
package body Inc_Value
is
procedure Inc(X: in out T)
is
 begin
   X:= X+1;
 end Inc;
end Inc_Value;
```

Using GNATprove to prove exception freedom:

```
gnatprove -P myproject.gpr --mode=prove -u inc_value.ads inc_valve_adb

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
inc_value.adb: ... range check might fail (e.g. when X = T'Last)
[possible explanation: subprogram at inc_value.ads: 6
                       should mention X in a precondition]
```

Defence via Contract

```
...
package Inc_Value
is
type T is range -128 .. 128;

procedure Inc(X: in out T)
with
  Depends => (X => X);
  Pre => X < T'Last;
end Inc_Value;
```

```
...
package body Inc_Value
is
procedure Inc(X: in out T)
is
 begin
   X:= X+1;
 end Inc;
end Inc_Value;
```

- We can add a precondition (logical assertion) to the procedure `Inc` using the `Pre` aspect, e.g. `X < T'Last`
- GNATprove assumes that a precondition is true when checking the body of a procedure (or function).
- GNATprove checks that preconditions are true at each point in the code where the procedure (or function) is called.

Defence via Contract

```
...
package Inc_Value
is
type T is range -128 .. 128;

procedure Inc(X: in out T)
with
  Depends => (X => X);
  Pre => X < T'Last;
end Inc_Value;
```

```
...
package body Inc_Value
is
procedure Inc(X: in out T)
is
 begin
   X:= X+1;
 end Inc;
end Inc_Value;
```

Using GNATprove to prove exception freedom:

```
gnatprove -P myproject.gpr --mode=prove -u inc_value.ads inc_valve_adb

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
Summary logged in ... gnatprove/gnatprove.out
```

Defence via Code

```
...
package Inc_Value
is
type T is range -128 .. 128;

procedure Inc(X: in out T)
with
  Depends => (X => X);
end Inc_Value;
```

```
...
package body Inc_Value
is
procedure Inc(X: in out T)
is
 begin
   if X < T'Last then
      X:= X+1;
   end if;
 end Inc;
end Inc_Value;
```

Using GNATprove to prove exception freedom:

```
gnatprove -P myproject.gpr --mode=prove -u inc_value.ads inc_valve_adb

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
Summary logged in ... gnatprove/gnatprove.out
```

Code versus Contract?

- Defensive coding adds to the complexity of the software and incurs a run-time overhead.
- Contracts are a design-time defence – they are used to establish the correctness of the code by formal argument with no run-time overhead.
- Contracts promote modular verification, i.e. a divide and conquer strategy.
- Defensive code is still important, e.g. validating inputs to a system from unverified sources.

Exercise: the example given on slide 6 gave rise to a proof failure, with a counter example being offered by the proof tool. Add a precondition (proof contract) to the specification (`.ads`) of the `Int_Dec` procedure that eliminates this proof failure. (The solution is available via `https://www.macs.hw.ac.uk/~air/rmse/SPARK/code/Solutions/Dec/`

Functional Specifications

Consider the `Int_Switch` subprogram from an input-output perspective:

$$\begin{array}{c} X = 2 \\ \wedge \\ Y = 3 \\ \textbf{Input} \end{array} \quad \Rightarrow \quad \boxed{\text{Int\_Switch}} \quad \Rightarrow \quad \begin{array}{c} X = 3 \\ \wedge \\ Y = 2 \\ \textbf{Output} \end{array}$$

- A **functional specification** describes the input-output relationship of a subprogram.
- A functional specification is represented by assertions within a contract, i.e. **preconditions** and **postconditions**.
- While preconditions constrain the inputs to a subprogram, postconditions constraint the outputs.

A Functional Specification of Int_Switch in SPARK

```
procedure Int_Switch(X, Y: in out Integer)
with
  Depends  => (X => Y, Y => X),
  Pre      =>  true,
  Post     => (X = Y'Old and Y = X'Old);
...
procedure Int_Switch(X, Y: in out Integer) is
T: Integer;
begin
     T:=X; X:=Y; Y:=T;
end Int_Switch;
```

- Int_Switch is specified above by means of precondition (Pre) and postcondition (Post) aspects.
- Note that X'Old denotes the initial value of X — X'Old is known as a **ghost variable**.
- Likewise, Y'Old denotes the initial value of Y — where Y'Old is a ghost variable.

A Functional Specification of `Int_Switch` in SPARK

```
procedure Int_Switch(X, Y: in out Integer)
with
  Depends  => (X => Y, Y => X),
  Pre      =>  true,
  Post     => (X = Y'Old and Y = X'Old);
...
procedure Int_Switch(X, Y: in out Integer) is
T: Integer;
begin
     T:=X; X:=Y; Y:=T;
end Int_Switch;
```

- The specification states that whenever `Int_Switch` is executed, if it terminates then the final value of `X` will be equal to the initial value of `Y` (i.e. `Y'Old`) and that the final value of `Y` will be equal to the initial value of `X` (i.e. `X'Old`).

Functional Specification of Aggregations

- While `X = Y'Old` works for scalar parameters, a different mechanism is required to specifying properties of aggregates, i.e. array and record parameters.
- **delta aggregate** provides such a mechanism.
- To illustrate, consider an array version of the `Integer_Switch` procedure:

```
...
type Pair is array (1..2) of Integer;
...
procedure Int_Switch(P: in out Pair)
is
   T: Integer;
begin
   T:= P(1); P(1):= P(2); P(2):= T;
end Int_Switch;
```

Functional Specification of Aggregations

```
...
type Pair is array (1..2) of Integer;

procedure Int_Switch(P: in out Pair)
with
   Depends => (P => P),
   Pre     =>  true,
   Post    => (P = (P'Old with delta 1 => P'Old(2),
                                     2 => P'Old(1)));
...
procedure Int_Switch(P: in out Pair)
is
   T: Integer;
begin
   T:= P(1); P(1):= P(2); P(2):= T;
end Int_Switch;
...
```

Functional Specification of Aggregations

```
...
type Pair is array (1..2) of Integer;

procedure Int_Switch(P: in out Pair)
with
   Depends => (P => P),
   Pre     =>  true,
   Post    => (P = (P'Old with delta 1 => P'Old(2),
                                     2 => P'Old(1)));
...
```

Note that:

`P = (P'Old with delta 1 => P'Old(2), 2 => P'Old(1))`

states that the final value of P is equal to the initial value of P (`P'Old`) with the first element (1) updated with the value of the initial second element (`P'Old(2)`) and the second element (2) updated with the initial value of the first element (`P'Old(1)`).

Int_Min Revisited

```
package Min is
 function Int_Min(X, Y: in Integer) return Integer
 with
 Depends => (Int_Min'Result => (X, Y)),
 Pre     => true,
 Post    => (Int_Min'Result = (if X > Y then Y else X));
end Min;

package body Min is
 function Int_Min(X, Y: in Integer) return Integer
 is
 begin
   if X > Y then return(Y);
      else return(X);
   end if;
 end Int_Min;
end Min;
```

Int_Min Revisited

```
function Int_Min(X, Y: in Integer) return Integer
with
Depends => (Int_Min'Result => (X, Y)),
Pre     => true,
Post    => (Int_Min'Result = (if X > Y then Y else X));
```

- Note that the `<func-id>'Result` notation is used both by the `Depends` and `Post` aspects.
- The postcondition above should be read as follows:
  The function returns Y if X is strictly greater than Y, otherwise it returns X.

Integer Division (Int_Div)

- Computing 7 Int_Div 3 gives:

  Quotient $= 2$
  Remainder $= 1$

- Computation:

  $7 - 3 = 4$
  $4 - 3 = 1$
  $1 - 3 = -2$

  Quotient equals the number of subtractions.

  Remainder equals the result of the repeated subtractions.

  Stop before a subtraction gives a negative result.

Int_Div

```
package Div
is
  procedure Int_Div(X, Y: in Integer; Q, R: out Integer);

end Div;

package body Div
is
  procedure Int_Div(X, Y: in Integer; Q, R: out Integer)
  is
  begin
     R:= X; Q:= 0;
     while (Y <= R) loop
        R:= R-Y; Q:= Q+1;
     end loop;
  end Int_Div;
end Div;
```

Int_Div – **mode** check

```
gnatprove -P myproject.gpr --mode=check -u div.ads div.adb

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: fast partial checking of SPARK legality rules ...
```

Int_Div: Dependency Contract – **mode** flow

```
...
procedure Int_Div(X, Y: in Integer; Q, R: out Integer)
with
  Depends  => (R => (X, Y), Q => (X, Y));
...
```

```
gnatprove -P myproject.gpr --mode=flow -u div.ads div.adb

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ... ...
```

Int_Div: Proof Contract – **mode** prove

```
 ...
 procedure Int_Div(X, Y: in Integer; Q, R: out Integer)
  with
    Depends  => (R => (X, Y), Q => (X, Y)),
    Pre      => true;
    Post     => ((X = R + (Y * Q)) and (R < Y));
...
```

```
gnatprove -P myproject.gpr --mode=prove -u div.ads div.adb

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
... overflow check might fail (e.g. when R = Integer'Last and Y = -1)
...
```

Bug or feature of the algorithm?

Int_Div: Proof Contract – **mode** prove

```
...
procedure Int_Div(X, Y: in Integer; Q, R: out Integer)
is
begin
   R:= X; Q:= 0;
   while (Y <= R) loop
      R:= R-Y; Q:= Q+1;
   end loop;
end Int_Div;
...
```

```
gnatprove -P myproject.gpr --mode=prove -u div.ads div.adb

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
... overflow check might fail (e.g. when R = Integer'Last and Y = -1)
```

The algorithm only works for positive divisors – a stronger precondition is required, i.e. $Y > 0$

Int_Div: Strengthened Precondition – *is not Enough*

```
 ...
 procedure Int_Div(X, Y: in Integer; Q, R: out Integer)
 with
   Depends  => (R => (X, Y), Q => (X, Y)),
   Pre      => Y > 0;
   Post     => ((X = R + (Y * Q)) and (R < Y));
...
```

```
gnatprove -P myproject.gpr --mode=prove -u div.ads div.adb

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
... overflow check might fail (e.g. when Q = Integer'Last) ...
... postcondition might fail, cannot prove X = R + (Y * Q) ...
... overflow check might fail (e.g. when Q = 2 and ...
... overflow check might fail (e.g. when Q = -2 and Y = 2) ...
```

How could Q be negative?

Int_Div: Strengthened Precondition – *is not Enough*

```
 ...
 procedure Int_Div(X, Y: in Integer; Q, R: out Integer)
 with
   Depends  => (R => (X, Y), Q => (X, Y)),
   Pre      => Y > 0;
   Post     => ((X = R + (Y * Q)) and (R < Y));
...
```

```
gnatprove -P myproject.gpr --mode=prove -u div.ads div.adb

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
...
... [possible explanation: ... should mention Q in a loop invariant]
... [possible explanation: ... should mention Q in a loop invariant]
... [possible explanation: ... should mention Q in a loop invariant]
```

A **loop invariant** specifies the input-output relationship of a loop, i.e. a loop invariant must be true **before** and **after** each iteration.

Int_Div: Loop Invariant

```
...
R:= X; Q:= 0;
while (Y <= R) loop
   pragma Loop_Invariant (X = R + (Y * Q));
   R:= R-Y; Q:= Q+1;
end loop;
...
```

```
gnatprove -P myproject.gpr --mode=prove -u div.ads div.adb

Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
```

Note that a loop invariant is a special kind of assertion, hence the special **pragma**. Note that in general the creation of loop invariants cannot be automated (i.e. it is undecidable in general). As a consequence, the formal verification of programs can at best be semi-automatic.

Compositional Reasoning

```
procedure A
with
  Depends  => ...,
  Pre      => P_A,
  Post     => Q_A;
...
procedure A is
...
begin
   ... B ...;
end A;
```

```
procedure B
with
  Depends  => ...,
  Pre      => P_B,
  Post     => Q_B;
...
procedure B is
...
begin
   ...;
end B;
```

- Note that A calls B, therefore to reason about the correctness of A with respect to its functional specification we require a functional specification for B.
- The use of functional specifications (assertions) represents a divide-and-conquer verification strategy, i.e. assertions allow the overall verification task to be decomposed into a set of smaller verification tasks.

Detector.Control: Functional Specification

```
procedure Control
with
...
Pre  => True,
Post => (if Warning.Enabled then Sensor.Enabled);
...
procedure Control
is
begin
   if Sensor.Enabled then Warning.Enable;
                     else Warning.Disable;
   end if;
end Control;
```

Note that the definition of `Control` involves `Warning.Enabled`, `Warning.Enable`, `Warning.Disable` and `Sensor.Enabled`.

Sensor: Functional Specifications

```
procedure Write_Sensor(Value: in Boolean)
with
...
Pre  => True,
Post => (State = Value);

function Enabled return Boolean
with
...
Pre  => True,
Post => (Enabled'Result = State);
```

Note that `State` is of type `Boolean`.

Warning: Functional Specifications

```
procedure Enable
with
...
Pre  => True,
Post => State;

procedure Disable
with
...
Pre  => True,
Post => not(State);

function Enabled return Boolean
with
...
Pre  => True,
Post => (Enabled'Result = State);
```

Note that State is of type Boolean.

Summary

Learning outcomes:

- Understand the nature of formal program verification.
- Understand how a program can be specified via assertions, i.e. pre- and postconditions and loop invariants.
- Understand the notion of an exception freedom specification (and verification), and why it is of importance to industry.
- Understand the notion of a functional specification (and verification).

Remaining two lectures:

- How code and its specification are translated into a mathematical problem, i.e. logical conjectures.
- How such mathematical problems are solved using formal proof – includes a closer look at the role that loop invariants play within program verification.

Summary

Recommended reading:

- C. Jones, P. O'Hearn and J. Woodcock, "Verified Software: A Grand Challenge", IEEE Computer, 39(4), pp. 93-95, 2006.
- J-C. Filliatre and A. Paskevich, "Why3 – Where Programs Meet Provers", In proceedings of *Programming Languages and Systems* – (22nd ESOP'13 & 16th ETAPS'13), Lecture Notes in Computer Science (LNCS), vol 7792, 2013.
- R. Chapman and P. Amey, "Industrial Strength Exception Freedom", Proceedings of ACM SigAda, 2002.
- A. Ireland and B.J. Ellis and A. Cook and R. Chapman and J. Barnes, "An Integrated Approach to High Integrity Software Verification" Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning, Kluwer, Vol 36(4), 2006. (see also MACS Technical Report HW-MACS-TR-0027: