Hi-Lite - Verification by Contract

Jérôme Guitton, Johannes Kanig,
Yannick Moy (AdaCore)

Ada Deutschland - June 30th, 2011

Overview

AdaCore The GNAT Pro Company

Outline

Formal Methods - I

Apply mathematical techniques to programs

- Find potential bugs (static analysis)
- Prove absence of bugs (program verification)
- Strong guarantees

Examples

- CodePeer, Polyspace (bugfinding)
- SPARK (program verification)

AdaCore
The GNAT Pro Company

Formal Methods - II

are often presented as an *alternative* to testing, inherently *superior*, covering all possible cases

In reality ...

Program verification also has drawbacks:

- Complex code (pointers, concurrency) intractable by (automated) formal methods in the current state of the art
- Many proposed methods require expert knowledge
- Specifications can contain errors, cannot be tested
- Guarantees only as good as the specifications
- Proving *termination* is often omitted (partial correctness)
- Non-functional properties (timing, memory) not considered

As a consequence, applying program verification to an entire nontrivial program is unrealistic

Current practice: Testing

Testing

- Current practice in verification / validation (DO-178B)
- Some form of completeness usually desired (MC/DC)
- Unit testing: from agile development to mainstream
- Simple to setup

Disadvantages

- Cost: initial setup, maintenance, availability of benchmarks, ...
- Impossibility to cover all cases

DO-178C will allow formal methods to partially replace or complement testing

Unit Proof - I

Concept

- Apply formal methods and tests on a per-subprogram basis
- If formal methods fail (VC too complex for automated tools), one can still test the subprogram
- Has been applied at Airbus to avionics software Level A

AdaCore
The GNAT Pro Company

Unit Proof - II

Problems

- Expertise: required for writing contracts and carrying proof
- Duplication: contract not shared between testing and proof
- Isolation: unit test and unit proof cannot be combined
- Confusion: not the same semantics for testing and proof
- Debugging: contracts and proof cannot be executed

Outline

Hi-Lite Partners

Hi-Lite - I

Main objective

- Combination of testing and proof to increase confidence in software
- Avoid many problems of traditional unit proof by using a common specification language of tests and proofs

Lightweight approach to formal methods

- Automated proofs
- Ease transition from all-testing
- Application to existing projects possible
- Contrast with SPARK: stronger guarantees, but more restrictive

Hi-Lite - II

The Specification language

- Subprograms have pre- and postconditions (contracts)
- Ada Boolean expressions
- New forms of expressions in Ada 2012

Unit testing

- Possibility to specify testcases next to the subprogram
- A tool GNATtest that generates test stubs that correspond to test cases

Program proof

- A tool GNATprove that generates *verification conditions* and attempts to prove them

New forms of expressions in Ada 2012

- if-expressions:

```
(if X = 0 then 0 else 1 / X)
```

- case-expressions:

```
type Week_Day is
  (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
...
(case X is
   when Mon .. Fri => True
   when others     => False)
```

- quantified expressions:

```
(for all I in X'Range => X (I) > 0)
(for some I in X'Range => X (I) > 0)
```

An Example

A function with pre- and postcondition

```
function Search (S : String; C : Character)
   return Natural
   with
      Pre  => S /= "",
      Post =>
          (if Search'Result /= 0 then
              S (Search'Result) = C
           and
             (for all X in
                S'First .. Search'Result - 1 =>
                S (X) /= C));
```

Test cases

```ada
function Sqrt (X : Integer) return Integer
with
   Test_Case =>
      (Name      => "nominal test case",
       Mode      => Nominal,
       Requires  => X < 100,
       Ensures   =>
          Sqrt'Result >= 0 and
          Sqrt'Result < 10),
   Test_Case =>
      (Name      => "robustness test case",
       Mode      => Robustness,
       Requires  => X = -1,
       Ensures   => Sqrt'Result = 0);
```

The Alfa subset of Ada

Definition

- Includes all features suitable for program verification
- Excludes pointers, concurrency
- Close to the SPARK language, but more permissive

Classification of each subprogram

- Non-Alfa: no restrictions
- Partially in Alfa: specification and contract of the subprogram are in Alfa, no restriction on the body
- (Entirely) in Alfa: specification, contract and body of the subprogram are in Alfa, only subprograms at least partially in Alfa are called

Proofs

Procedure

- For subprograms that are (partially) in Alfa, translate all
  - Contracts
  - Assertions
  - Checks

  to *verification conditions* (VCs)
- Try to prove each VC automatically
- Unproved VCs are reported to the user

Underlying technology

Procedure

- Specs and subprograms in Alfa are translated to an intermediate language
- A VC generator called Why generates VCs
- VCs are discharged using the Alt-Ergo theorem prover

$$
\begin{tikzcd}
\text{Ada} \arrow[r, "\text{gnat}"'] \arrow[rr, bend left] & \text{ALI} \arrow[r, "\text{gnat2why}"'] & \text{Why Prog} \arrow[r, "\text{why}"'] & \text{Why VC} \arrow[r, "\text{alt-ergo}"'] & \text{Yes/No}
\end{tikzcd}
$$

Outline

Introduction and Motivation

Presentation of the Hi-Lite Project

Nonstandard Verification

Addressing Shortcomings of Formal Methods

Assertions can contain run-time errors themselves

A question ...

What is the meaning of an assertion that raises a run-time error?

Our answer

It's the wrong question: assertions should never do that.

One goal of GNATprove

Prove the absence of run-time errors in programs *and* assertions

Assertions generate additional checks

Given the type definitions:

```
type Array_Range is range 1 .. 10;
type IntArray is array (Array_Range) of Integer;
```

The following assertion will require an additional check:

```
for Index in Table'Range loop
   --  This will generate a (provable) check:
   --     J in Table'Range
   pragma Assert
      (for all J in Table'First .. Index - 1 =>
         Table (J) /= Value);
   ...
end loop;
```

Preconditions must be self-guarded

Preconditions

- Are treated as any other assertion;
- But cannot use any context

Wrong:

```
procedure P (X : IntArray; I : Integer)
   with Pre => (X (I) > 0);
```

Correct:

```
procedure P (X : IntArray; I : Integer)
   with Pre => (I in X'Range and then X (I) > 0);
```

A precondition must always contain all guards that guarantee run-time error free execution

Incomplete postconditions

Goal: improve postconditions

Detect situations where the postcondition is correct, but:

- The postcondition is trivial
- Some code does not contribute to the postcondition;

A trivial postcondition

```
function Max (X, Y : Integer) return Integer
  with Post => ((if X < Y then Max'Result = Y)
               or (if X >= Y then Max'Result = X));

function Max (X, Y : Integer) return Integer is
begin
   if X < Y then
      return Y;
   else
      return X;
   end if;
end Max;
```

- The postcondition is trivial (always true)
- The programmer wanted to join the conditions with "and"

An incomplete contract

```
procedure Set_Zero (X, Y : out Integer)
   with Post => (X = 0);

procedure Set_Zero (X, Y : out Integer) is
begin
   X := 0;
   Y := 0;
end Set_Zero;
```

- The postcondition does not mention all effects;
- The assignment to `Y` is not used to establish the postcondition.

Detecting inconsistent and redundant preconditions

```
procedure P (X, Y : in out Integer)
   with Pre  => (X <= 0 and X > 0),
   with Post => (...);

procedure Q (X, Y : in out Integer)
   with Pre  => (X > 0 and X > 0),
   with Post => (...);
```

- In both examples, the programmer made a mistake and wrote `X` instead `Y` in the precondition;
- The precondition of `P` is *inconsistent*, it can never be true; without any special mechanism, this subprogram will be proved correct, regardless of the postcondition;
- The precondition of `Q` contains a *redundant* part;
- We propose to detect such situations in GNATprove.

Unimplementable contracts

```
procedure Compute
            (X : in Integer;
             Y : out Integer) with
    Post =>
      ((if X >= 0 then Y = 1) and
      (if X <= 0 then Y = -1));
```

A (terminating) subprogram with this contract is impossible to implement

Outline

- Introduction and Motivation
- Presentation of the Hi-Lite Project
- Nonstandard Verification
- Addressing Shortcomings of Formal Methods

Addressing Shortcomings of Formal Methods

Writing contracts is for experts only?

Ada 2012 expressions:

- common to programs and specifications
- no new language to learn
- no more complicated than programming

Errors in specs

can be found by testing, because contracts are executable

Termination

termination problems can be detected by testing

Hi-Lite and Open-DO

Open-DO

- Address the "Big Freeze" problem
- Open-source tools for safety-critical software development
- Decrease the barrier of entry for the development of safety-critical software
- Research in the area of safety-critical software development

Hi-Lite is part of the Open-DO Initiative

- Entirely Open Source
- Lower the barrier of application of program verification
- Online resources: `www.open-do.org/projects/hi-lite`

Conclusion

We have presented Hi-Lite

- a lightweight approach to formal methods
- support for test cases to improve unit testing experience
- gradual replacement or complement of testing by proofs
- application to a legacy code base is possible

Work in progress ...

- Unit testing and test cases are well supported
- GNATprove still in an early prototype phase
- Now starting experiments at EADS Astrium and Thales Communications

You can participate ...

- in Open-DO
- in Hi-Lite: open source