Integrating Formal Program Verification with Testing

Yannick Moy, AdaCore

Workshop on Theorem Proving in Certification
December 6, 2011

Outline

Project Hi-Lite

Application to Nose Gear Velocity

Tool Qualification

Unit proofs vs. unit tests

unit tests are costly to develop and maintain

use instead unit proof:

1. express LLRs as function contracts
2. interpret code+contracts in Hoare logics
3. use Dijkstra's WP calculus to generate VCs
4. prove VCs with automatic prover

unit proof used industrially:

- SPARK toolset (SPARK code): data/information flows, run-time errors
- Frama-C platform (C code): contracts and run-time errors

DO-178C supports replacing unit tests with unit proofs

Unit proofs vs. unit tests

unit tests are costly to develop and maintain

use instead unit proof:

1. express LLRs as function contracts
2. interpret code+contracts in Hoare logics
3. use Dijkstra's WP calculus to generate VCs
4. prove VCs with automatic prover

unit proof used industrially:

- SPARK toolset (SPARK code): data/information flows, run-time errors
- Frama-C platform (C code): contracts and run-time errors

DO-178C supports replacing unit tests with unit proofs

How do we define contracts?

usual approach: first-order logic + program locations

- problem! avoid inconsistencies
  solution? generation of models
- problem! detect incorrect contracts
  solution? generation of counterexamples

our approach: pure Boolean expressions (no writes)

- avoid inconsistencies? forbid axioms
- detect incorrect contracts? test/execute
- possible effects? analyze and reject
- possible run-time errors? generate VCs and prove

How do we define contracts?

usual approach: first-order logic + program locations

- problem! avoid inconsistencies
  solution? generation of models
- problem! detect incorrect contracts
  solution? generation of counterexamples

our approach: pure Boolean expressions (no writes)

- avoid inconsistencies? forbid axioms
- detect incorrect contracts? test/execute
- possible effects? analyze and reject
- possible run-time errors? generate VCs and prove

How do we deal with unprovable code?

usual approach:

1. restrict language to potentially provable subset
2. use multiple automatic provers
3. write proof script in proof assistant
4. manually inspect and validate VC

our approach:

1. limit proof to potentially provable subset
2. generate VCs targetting selected prover
3. ensure possible combination with tested code
4. test remaining functions

How do we deal with unprovable code?

usual approach:

1. restrict language to potentially provable subset
2. use multiple automatic provers
3. write proof script in proof assistant
4. manually inspect and validate VC

our approach:

1. limit proof to potentially provable subset
2. generate VCs targetting selected prover
3. ensure possible combination with tested code
4. test remaining functions

The three pillars of formal methods in DO-178C

Unambiguous formal semantics

match compiler choices of sizes & alignments for target
prevent compiler-dependent behavior:

- functions (not procedures) cannot have side-effects
- expressions cannot have side-effects
- arithmetic expressions are parenthesized if needed

Sound formal analysis

deductive verification à la Hoare

Justified assumptions for proofs

...

The three pillars of formal methods in DO-178C

Unambiguous formal semantics

match compiler choices of sizes & alignments for target
prevent compiler-dependent behavior:

- functions (not procedures) cannot have side-effects
- expressions cannot have side-effects
- arithmetic expressions are parenthesized if needed

Sound formal analysis

deductive verification à la Hoare

Justified assumptions for proofs

...

The three pillars of formal methods in DO-178C

Unambiguous formal semantics

match compiler choices of sizes & alignments for target
prevent compiler-dependent behavior:

- functions (not procedures) cannot have side-effects
- expressions cannot have side-effects
- arithmetic expressions are parenthesized if needed

Sound formal analysis

deductive verification à la Hoare

Justified assumptions for proofs

...

Justified assumptions for proofs

formal verification of P assumes:

- precondition of P
- postcondition of subprograms called
- both user-defined and implicit ones

assumptions made for proof should be verified by testing

2 cases:

- tested T calls proved P
  $\rightarrow$ check precondition of P at run-time
- proved P calls tested T
  $\rightarrow$ check postcondition of T at run-time
- ...during test of T!

Justified assumptions for proofs

formal verification of `P` assumes:

- precondition of `P`
- postcondition of subprograms called
- both user-defined and implicit ones

assumptions made for proof should be verified by testing

2 cases:

- tested `T` calls proved `P`
  $\rightarrow$ check precondition of `P` at run-time
- proved `P` calls tested `T`
  $\rightarrow$ check postcondition of `T` at run-time
- ...during test of `T`!

Justified assumptions for proofs

formal verification of `P` assumes:

- precondition of `P`
- postcondition of subprograms called
- both user-defined and implicit ones

assumptions made for proof should be verified by testing

2 cases:

- tested `T` calls proved `P`
  $\rightarrow$ check precondition of `P` at run-time
- proved `P` calls tested `T`
  $\rightarrow$ check postcondition of `T` at run-time
- ...during test of `T`!

Justified assumptions for proofs (cont'd)

1. identify implicit contracts (effects, strong typing, non-aliasing)
2. effects: generated (restrictions on complete program)
3. strong typing:
   - strongly typed language (Ada)
   - forbid unsafe language features (pointer conversion)
   - proof: generate VCs
   - test: compiler inserts checks
4. non-aliasing:
   - limit proof to subset with references (no pointers)
   - global static analysis for non-aliasing with globals
   - proof: semantic verification for parameter non-aliasing
   - test: compiler inserts checks for parameter non-aliasing

Justified assumptions for proofs (cont'd)

1. identify implicit contracts (effects, strong typing, non-aliasing)
2. effects: generated (restrictions on complete program)
3. strong typing:
   - strongly typed language (Ada)
   - forbid unsafe language features (pointer conversion)
   - proof: generate VCs
   - test: compiler inserts checks
4. non-aliasing:
   - limit proof to subset with references (no pointers)
   - global static analysis for non-aliasing with globals
   - proof: semantic verification for parameter non-aliasing
   - test: compiler inserts checks for parameter non-aliasing

Justified assumptions for proofs (cont'd)

1. identify implicit contracts (effects, strong typing, non-aliasing)
2. effects: generated (restrictions on complete program)
3. strong typing:
   - strongly typed language (Ada)
   - forbid unsafe language features (pointer conversion)
   - proof: generate VCs
   - test: compiler inserts checks
4. non-aliasing:
   - limit proof to subset with references (no pointers)
   - global static analysis for non-aliasing with globals
   - proof: semantic verification for parameter non-aliasing
   - test: compiler inserts checks for parameter non-aliasing

Justified assumptions for proofs (cont'd)

1. identify implicit contracts (effects, strong typing, non-aliasing)
2. effects: generated (restrictions on complete program)
3. strong typing:
   - strongly typed language (Ada)
   - forbid unsafe language features (pointer conversion)
   - proof: generate VCs
   - test: compiler inserts checks
4. non-aliasing:
   - limit proof to subset with references (no pointers)
   - global static analysis for non-aliasing with globals
   - proof: semantic verification for parameter non-aliasing
   - test: compiler inserts checks for parameter non-aliasing

Outline

Project Hi-Lite

Application to Nose Gear Velocity

Tool Qualification

Matching data to types

two different types of data:

- external counters with modulo semantics
- non-negative values for time/distance/velocity

coded in C example as `unsigned` causing 4 kinds of errors in code:

1. useless wraparound code
2. wrong wraparound: `65534 - (prevTime - thisTime)`
3. inconsistent pattern: `if (prevCount < thisCount)`
4. copy-paste error: `currTime` or `thisTime` for `t3`?

in Ada, `modulo` integer (semantics) $\neq$ non-negative (constraint)

Dealing with dimensions and units

two dimensions: distance and time
three combinations: velocity, acceleration, jerk
many units: distance (cm, m, inch), time (ms, s), velocity (km/h)

vulnerability in code:

```
static init whcf = .. * 254) / 7) * 22) / 100;
```

equivalent to

```
static init whcf = .. * 254) * 22) / 7) / 100;
```

only up to `WHEEL_DIAMETER` = 51

errors in code:

1. wrong conversion: missing /100 for `maxMsecs`
2. wrong conversion: *500 should be *50 for `maxClicks`

Christof Grein's SI units library

```
package SI is new Constrained_Checked_SI (Float);
package U  is new SI.Generic_Units;
use SI, U;

Pi     : constant          := 3.14;
Inch   : constant Distance := 2.14*centi*Meter;
WHEEL  : constant Distance := 26.0*Inch;
WHCF   : constant Distance := WHEEL * Pi;

PrevCount : Count := 0;
PrevTime  : Time  := 0.0*milli*Meter;
```

```
raised NG.SI.SI.UNIT_ERROR :
  unconstrained_checked_si.adb:72
    instantiated at constrained_checked_si.ads:204
      instantiated at ng.ads:9
```

```
package SI is new Constrained_Checked_SI (Float);
package U  is new SI.Generic_Units;
use SI, U;

Pi     : constant          := 3.14;
Inch   : constant Distance := 2.14*centi*Meter;
WHEEL  : constant Distance := 26.0*Inch;
WHCF   : constant Distance := WHEEL * Pi;

PrevCount : Count := 0;
PrevTime  : Time  := 0.0*milli*Meter;
```

```
raised NG.SI.SI.UNIT_ERROR :
  unconstrained_checked_si.adb:72
    instantiated at constrained_checked_si.ads:204
      instantiated at ng.ads:9
```

Compile-time dimensional analysis in GNAT

```
subtype Time is Natural with
  Dimension (second => 1, others => 0);
subtype Distance is Natural with
  Dimension (meter => 1, others => 0);
subtype Count is Natural;
subtype Velocity is Natural with
  Dimension(meter => 1, second => -1, others=>0);

WHEEL : constant Distance := 26;  -- inches
WHCF  : constant Distance :=      -- cm
          (((WHEEL * 254) / 7) * 22) / 100;

PrevCount : Count := 0;
PrevTime  : Time  := 0;  -- ms
```

Defining execution conditions

update should not occur if no new click
translated as precondition:

```
procedure ComputeNGVelocity
  (CurrTime   : in      Mod_Time;
   ThisTime   : in      Mod_Time;
   ThisCount  : in      Mod_Count;
   Success    :     out Boolean;
   Result     :     out Velocity)
with
  Pre => ThisTime /= PrevTime
           and then ThisCount /= PrevCount;
```

explicit precondition avoids error in C example:

```
if (thisTime = prevTime) return;
...
if (thisCount = prevCount) { ...
```

Checking absence of run-time errors

```
procedure ComputeNGVelocity (...) is
   T1, T2 : Time;
   D1, D2 : Distance;
begin
   if ThisCount - PrevCount < ThisTime - PrevTime
   then
      Success := False;
      return;
   end if;

   T1 := Time (ThisTime - PrevTime);
   T2 := Time (CurrTime - ThisTime);
   D1 := WHCF * Count (ThisCount - PrevCount);
   D2 := (D1 * T2) / T1;

   Success := True;
   Result  := ((D1 + D2) * 3600) / (T1 + T2);
end ComputeNGVelocity;
```

Formal verification for run-time errors

```
ng.adb:49:28: range check not proved
ng.adb:50:28: range check not proved
ng.adb:51:18: (info) overflow check proved
ng.adb:51:18: (info) range check proved
ng.adb:52:17: overflow check not proved
ng.adb:52:23: (info) division check proved
ng.adb:52:23: (info) overflow check proved
ng.adb:52:23: range check not proved
ng.adb:55:23: overflow check not proved
ng.adb:55:29: overflow check not proved
ng.adb:55:37: (info) division check proved
ng.adb:55:37: (info) overflow check proved
ng.adb:55:37: (info) range check proved
ng.adb:55:43: (info) overflow check proved
```

Formal contract

```
procedure ComputeNGVelocity (...)
with
  Pre => ThisTime /= PrevTime
           and then ThisCount /= PrevCount,
  Post =>
    (if Success then
       Result = Velocity(
       (((WHCF * Integer (ThisCount-PrevCount)) +
        ((WHCF * Integer (ThisCount-PrevCount))
          * Integer (CurrTime - ThisTime))
          / Integer (ThisTime - PrevTime))
        * 3600)
        / Integer (CurrTime - PrevTime)));
```

Testing of contract

change code from

```
        D2 := (D1 * T2) / T1;
```

to erroneous

```
        D2 := (D1 + T2) / T1;
```

leads to

```
raised SYSTEM.ASSERTIONS.ASSERT_FAILURE :
  failed postcondition from ng.adb:33
```

Formal verification of contract

```
ng.adb:24:15: postcondition not proved
ng.adb:27:27: (info) overflow check proved
ng.adb:27:62: overflow check not proved
ng.adb:28:29: (info) overflow check proved
ng.adb:29:22: (info) overflow check proved
ng.adb:30:22: (info) division check proved
ng.adb:30:22: overflow check not proved
ng.adb:31:19: overflow check not proved
ng.adb:32:19: (info) division check proved
ng.adb:32:19: overflow check not proved
```

Formal contract (cont'd)

```
procedure UpdateNGVelocity with
  Post =>
    (if EstimatedGroundVelocityIsAvailable then
       EstimatedGroundVelocity =
        (DistanceSinceLastClickBeforeLastUpdate
         * 3600)
         / TimeSinceLastClickBeforeLastUpdate);

function DistanceSince... return Distance is
   (DistanceFromLastClickToLastUpdate
    + DistanceSinceLastUpdate);

function DistanceFrom... return Distance is
   (WHCF * (ThisCount - PrevCount));
```

Outline

Appropriate TQL for DO-178C

1. used for requirement based verification (replaces unit testing)
   $\Rightarrow$ TQL-5
2. used for robustness verification (replaces robustness tests)
   $\Rightarrow$ TQL-5
3. used for both
   $\Rightarrow$ TQL-4 (levels 1 and 2) or TQL-5
4. TQL-4 & TQL-5: Tool Operational Requirements are defined
5. TQL-4: TORs are complete, accurate and consistent
6. TQL-4: tool requirements are developed and verified

Objective B: tools are qualified

- formalism: first-order Hoare logics + run-time checks
- software tools: compiler + translator + VCgen + prover
- assumptions: user-defined contracts + implicit contracts
- all the chain is FLOSS $\Rightarrow$ facilitates duplication
- soundness argument for each piece
- prover removed from qualification if produces checkable trace
- super qualification: formally proved correct (VCgen, prover)

Objective C: software requirements

1. complete
   - divide contract in cases (behaviors in JML)
   - contract cases cover the precondition
2. consistent
   - contract cases are disjoint
   - consistency can be expressed and checked:
     $\forall inputs \in Pre . \exists outputs \in Post . Pre \Rightarrow Post$
3. unambiguous
   - expression as Boolean predicates
4. verifiable
   - by testing or formal verification

Objective K: source code vs. software requirements

1. compliance
   - contract-based verification (testing or proving)
2. traceability
   - by nature, contracts are attached to function

http://www.open-do.org/projects/hi-lite/