Hi-Lite
Combining Formal Program Verification and Testing

Valentine Reboul

2013-Feb-4

A Ring Buffer: Data

```
type Buf_Array is array (0 .. Buf_Size - 1)
of Integer;
-- The array which stores the buffer

type Ring_Buffer is record
   Data   : Buf_Array;
   First  : Integer := 0;
   Length : Integer := 0;
end record;
-- The record representing the buffer.
-- First is the first cell containing valid data.
-- Length is the number of stored items.
-- Wrapping around the array borders is possible.

-- The field Length is between 0 and Buf_Size.
-- The field First is always a valid array index,
-- hence between 0 and Buf_Size - 1.
```

Can we do better in Ada?

```ada
type Length_Type is new Integer
  range 0 .. Buf_Size;
-- The integer type of buffer length

subtype Index_Type is Length_Type
  range 0 .. Length_Type'Last - 1;
-- The integer type for valid array indices.

type Buf_Array is array (Index_Type) of Integer;

type Ring_Buffer is record
   Data   : Buf_Array;
   First  : Index_Type := 0;
   Length : Length_Type := 0;
end record;
```

A Ring Buffer: API

```
function Is_Empty (R : Ring_Buffer)
return Boolean;
-- Check whether the buffer is empty

procedure Pop (R : in out Ring_Buffer;
               Element : out Integer);
-- Return the first element of the buffer,
-- and remove it from the buffer.
-- The buffer should not be empty.
-- The length of the buffer is decreased by one.
```

Can we do better in Ada 2012?

expression functions completely define simple getters in the spec

```
function Is_Empty (R : Ring_Buffer)
return Boolean
is (R.Length = 0);
-- Check whether the buffer is empty
```

Can we do better in Ada 2012? (Cont'd)

contracts define the interface between a subprogram and its caller

```ada
function Head (R : in Ring_Buffer)
return Content
is (R.Data (R.First));

procedure Pop (R : in out Ring_Buffer;
               Element : out Integer)
with
  Pre  => not Is_Empty (R),
  Post => not Is_Full (R) and then
          R.Length = R.Length'Old - 1 and then
          Element = Head (R'Old);
-- Remove the returned element from the buffer.
```

What if a contract is violated?

contract = assertion
run-time violation = run-time exception raised

ex: Pop is passed an empty ring

```
raised SYSTEM.ASSERTIONS.ASSERT_FAILURE :
failed precondition from ring_buf.ads:53
```

ex: Pop implementation is faulty

```
raised SYSTEM.ASSERTIONS.ASSERT_FAILURE :
failed postcondition from ring_buf.ads:55
```

What about static verification? (1/3)

compiler is limited:

- must run quickly $\rightarrow$ imprecise analysis
- can detect obvious errors

```
procedure P (X : in Integer) with
  Post => X > 0;
```

`postcondition refers only to pre-state`

```
function F return Boolean with
  Post => X > 0;
```

`function postcondition does not mention result`

What about static verification? (2/3)

need for verifier:

- precise analysis $\rightarrow$ longer than compilation
- scalable analysis $\rightarrow$ modular, based on contracts
- can detect subtle errors

```
ring_buf.adb:19:26: range check not proved
ring_buf.ads:56:21: postcondition not proved
```

What about static verification? (3/3)

verifier checks:

- all possible run-time errors
- all user properties (assertions, contracts, invariants)

verifier can give complete guarantee:

```
ring_buf.ads:37:18: info: postcondition proved
ring_buf.adb:11:36: info: division check proved
ring_buf.adb:12:28: info: range check proved
ring_buf.ads:48:48: info: postcondition proved
ring_buf.adb:19:32: info: division check proved
ring_buf.adb:20:28: info: range check proved
ring_buf.ads:56:21: info: postcondition proved
ring_buf.ads:56:23: info: precondition proved
```

How does it work?

a VC (Verification Condition) is generated for every check:

- based on Hoare logics (1969) - $\{P\}C\{Q\}$
- automated by Dijkstra's calculus (1975)
- further automated by Filliâtre's effect computation (1996)
- made more efficient by Leino's calculus (2005)

each VC is proved separately by calling an SMT prover:

```
> alt-ergo ring_buf.ads_56_21_postcondition.why
< Valid
```

SMT = Satisfiability Modulo Theories

An example of VC

```
[...]

type length_type

logic to_int1 : length_type -> int

axiom range_axiom1 : (forall x:length_type. in_range1(to_int1(x)))

goal WP_parameter_def :
  (forall r:content map. forall r1:int. forall r2:index_type.
  forall r3:length_type. forall element:content. forall r4:content map.
  forall r5:int. forall r6:index_type. forall r7:length_type.
  forall r8:content map. forall r9:int. forall r10:index_type.
  forall r11:length_type. ((not (is_empty(mk_ring_buffer(mk_buf_array(r, r1),
  r2, r3)) = true)) -> (((((((r4 = r8) and (r5 = r9)) and (r6 = r10)) and
  (of_int1((to_int1(r7) - 1)) = r11)) and ((((r = r4) and (r1 = r5)) and
  (of_int2(((to_int2(r2) + 1) % 10000)) = r6)) and (r3 = r7))) and
  (element = get(r, ((to_int2(r2) + r1) - 0)))) ->
  ((not (is_full(mk_ring_buffer(mk_buf_array(r8, r9), r10, r11)) = true)) and
  ((to_int1(r11) = (to_int1(r3) - 1)) and
  (to_int(element) = to_int(head(mk_ring_buffer(mk_buf_array(r, r1), r2,
  r3)))))))))
```

What if a VC is not proved?

various possible causes:

1. code is incorrect
2. assertion is incorrect
3. missing assertions about program behavior
4. prover timeouts
5. prover is not smart enough

methodology to investigate unproved VCs

investigate causes from easier to harder

Investigate incorrect code and/or assertion

code **and** assertions can be executed

compiler and verifier fully agree on meaning of assertions

$\rightarrow$ code **and** assertions can be tested and debugged

checks enabled by compiler switches:

- `-gnata`: run-time checking of assertions
- `-gnato`: run-time checking of intermediate overflows

Investigate missing assertions

/work/tns/path/p.adb

```
package body P is

   procedure Swap (X : in out Arr) is
      Diff : Boolean := False;
   begin
      for J in X'Range loop
         if X(J) then
            if J = X'First or else not X(J-1) then
               Diff := True;
            end if;
         elsif X(J-1) then
            Diff := True;
         end if;
         if Diff then
            X(J-1) := not X(J-1);
            X(J)   := not X(J);
         end if;
      end loop;
   end Swap;

end P;
```

Investigate prover shortcomings

verifier switches:

- `-timeout`: increase prover timeout
- `-prover`: use alternative SMT prover

verification can be focused:

- on an individual subprogram or line of code
- both on command-line and inside IDE

What to do next?

traditional fallbacks when automatic proof fails:

- manual review
- hand-written proof (automatically assisted)

drawbacks:

- require proof & tool expertise
- time consuming, costly
- maintenance problems

$\rightarrow$ new fallback: testing

Testing and formal verification (1/3)

testing has always been the fallback:

- parts of the code that cannot be formally analyzed
- properties that cannot be formalized
- assumptions needed by formal verification

but no methodology for the combination

new combination provides results **as good as testing alone**

Testing and formal verification (2/3)

new combination with a precise methodology:

- subprogram contract captures complete property to verify
- each subprogram is either tested or proved
- testing is done in special mode with additional run-time checks

case 1: *when proved subprogram P calls tested subprogram T, proof depends on correct call result*

case 2: *when tested subprogram T calls proved subprogram P, proof depends on correct calling context*

Testing and formal verification (3/3)

special mode of testing needed to check assumptions for proof:

- precondition of proved function when called in tested
- postcondition of tested function when called in proved
- also, initialization of `in out` parameters
- also, non-aliasing of parameters

checks enabled by compiler switches:

- `-gnata`: run-time checking of contracts
- `-gnateV`: run-time checking of parameter initialization
- `-gnateA`: run-time checking of parameter non-aliasing

Current practice of formal program verification

small number of industries using:

- B method (railway)
- CAVEAT for C programs (Airbus)
- SPARK 2005 subset of Ada (avionics, defense, security)

new tools combine static and dynamic analyses:

- Frama-C (successor of CAVEAT)
- SPARK 2014 (subset of Ada 2012)

SPARK 2014 language

completely based off Ada 2012:

- new specification aspects: contracts, invariants
- new expressions: if-expression, case-expression, quantified expression (`for all`, `for some`)
- new attributes: `Result`, `Old`

examples:

```
(if Condition then Expr else Expr)

(for all Index in Range => Boolean_Expression)

subtype Multiple is Natural
  with Dynamic_Predicate => Multiple mod 3 = 0;
```

SPARK 2014 language (Cont'd)

main restrictions w.r.t. Ada:

- functions cannot have side-effects
- no pointers (= access types)
- no aliasing (between references)
- no exceptions
- no tasking

additional constructs specific to SPARK 2014:

- new aspects: `Contract_Cases`, `Global`, `Depends`
- new pragmas: `Loop_Invariant`, `Loop_Variant`
- new attributes: `Loop_Entry`, `Update`

SPARK 2014 toolset

completely based off the compiler frontend:

- produces the AST for compilation **and** verification
- analyzes all constructs (generics, contracts, etc.)
- puts all the checks in the AST

notable compiler extensions:

- support for new aspects/pragmas/attributes in SPARK 2014
- 3 overflow checking modes $\rightarrow$ mathematical contracts
- target parametrization $\rightarrow$ correct proofs for target

The overflow problem

example of problem:

- user wants to add two numbers: `X + Y`
- user wants to assert that addition cannot overflow:
  `with Pre => X + Y in Integer`
- but this expression may overflow itself!

3 overflow checking modes:

- strict mode: normal overflow checks
- minimized mode: larger base type (64bits) used when needed
- eliminated mode: use bignum library in the remaining cases

flexible solution:

- user chooses between 3 modes
- independent choice for assertions and code
- same choice for execution and formal verification

More information (and code, binaries) on...

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