

Hi-Lite

Combining Formal Program Verification and Testing

Valentine Reboul

2013-Feb-4

"High Integrity Lint Integrated with Testing and Execution"

Research project :

- ▶ based on OpenSource software
- ▶ involving various industrial and academic partners:
Altran, Astrium Space Transportation, CEA-LIST, the ProVal team of INRIA, Thales Communications and AdaCore
- ▶ aiming at popularizing formal methods and providing modular and lite-to-use tools

On a simple example:
a Ring Buffer

A Ring Buffer: Data

```
1  type Buf_Array is array (0 .. Buf_Size - 1)
2  of Integer;
3  — The array which stores the buffer
4
5  type Ring_Buffer is record
6      Data      : Buf_Array;
7      First     : Integer := 0;
8      Length    : Integer := 0;
9  end record;
10 — The record representing the buffer.
11 — First is the first cell containing valid data.
12 — Length is the number of stored items.
13 — Wrapping around the array borders is possible.
14 — The field Length is between 0 and Buf_Size.
15 — The field First is always a valid array index,
16 — hence between 0 and Buf_Size - 1.
```

A Ring Buffer: API

```
18 function Is_Empty (R : Ring_Buffer)
19 return Boolean;
20 — Check whether the buffer is empty
21
22 function Head (R : in Ring_Buffer)
23 return Integer;
24 — Return the first element of the buffer
25
26 procedure Pop (R : in out Ring_Buffer;
27               Element : out Integer);
28 — Return the first element of the buffer ,
29 — and remove it from the buffer .
30 — The buffer should not be empty .
31 — The length of the buffer is decreased by one .
```

Can we do better in Ada
?

Strong typing

```
1  type Length_Type is new Integer
2    range 0 .. Buf_Size;
3  — The integer type of buffer length
4
5  subtype Index_Type is Length_Type
6    range 0 .. Length_Type'Last - 1;
7  — The integer type for valid array indices.
8
9  type Buf_Array is array (Index_Type) of Integer;
10
11 type Ring_Buffer is record
12     Data      : Buf_Array;
13     First     : Index_Type := 0;
14     Length    : Length_Type := 0;
15 end record;
```

Can we do better in Ada 2012
?

Expression Functions

Expression functions completely define simple getters in the specifications

```
17 function Is_Empty (R : Ring_Buffer)
18 return Boolean
19 is (R.Length = 0);
20 — Check whether the buffer is empty
21
22 function Head (R : in Ring_Buffer)
23 return Integer
24 is (R.Data (R.First));
25 — Return the first element of the buffer
```

Introducing contracts

Contracts define the interface between a subprogram and its caller

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

What if a contract is violated?

Contracts (Ada 2012) behave like Assertions (Ada 2005)

A violation at Run-Time raises a Run-Time exception.

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
?

The Compiler

The Compiler is limited in its checks:

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

```
1 procedure P (X : in Integer)
2 with
3   Post  $\Rightarrow$  X  $\geq$  0;
```

The post-condition refers only to pre-state

```
1 function F (X : in Integer) return Boolean
2 with
3   Post  $\Rightarrow$  X  $\geq$  0;
```

The post-condition of a function does not mention result (F'Result)

The Verifier (1/2)

A new tool is required: The **Verifier**

It will perform:

- ▶ precise analysis → longer than compilation
- ▶ scalable analysis → 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
```

The Verifier (2/2)

the Verifier checks:

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

The Verifier can give complete guarantees:

```
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 automatic proof
work
?

Mechanism:

The Verifier generates specific logical formulas thanks to a generation tool and uses a prover to verify them

Those logical formulas are called: **Verification Conditions**

They have strong mathematical origins:

- ▶ 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)

An example of generated VC

ring_buf.ads_56_21_postcondition.why generated with WHY:

```
[...]
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))))))))))
```

Each VC is proved separately by calling an SMT prover
(**alt-ergo**):

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

SMT = Satisfiability Modulo Theories

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

Both code **and** assertions can be executed

The Compiler and the Verifier fully agree on meaning of assertions

→ 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
1 package body P is
2
3   procedure Swap (X : in out Arr) is
4     Diff : Boolean := False;
5     begin
6       for J in X'Range loop
7         if X(J) then
8           if J = X'First or else not X(J-1) then
9             Diff := True;
10          end if;
11         elsif X(J-1) then
12           Diff := True;
13         end if;
14         if Diff then
15           X(J-1) := not X(J-1);
16           X(J) := not X(J);
17         end if;
18       end loop;
19     end Swap;
20
21 end P;
```

Investigate prover shortcomings

Verifier switches:

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

The 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 and really costly
- ▶ maintenance issues

→ New fallback: ... Testing !

Testing and Formal Proof Combination ?

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

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

Special mode of testing needed to check assumptions for proof:

- ▶ postcondition of tested function when called in proved
- ▶ precondition of proved function when called in tested
- ▶ 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

Some help for tests ?

Pragma Test_Case

```
1 pragma Test_Case (  
2     [Name      =>] static_stringh_Expression  
3     ,[Mode      =>] (Nominal | Robustness)  
4     [, Requires => Boolean_Expression]  
5     [, Ensures  => Boolean_Expression]);
```

```
1 package Math_Functions is  
2     function Sqrt (Arg : Float) return Float;  
3     pragma Test_Case (Name      => "Test 1",  
4                       Mode      => Nominal,  
5                       Requires => Arg < 10000,  
6                       Ensures  => Sqrt'Result < ←  
7                               10);  
7 end Math_Functions;
```

Back to the Ring Buffer example:

```
36 procedure Push (B : in out Ring_Buffer;  
37                E : Element)  
38 with Test_Case =>  
39   (Name      => "Test with empty buffer",  
40    Mode      => Nominal,  
41    Requires  => Is_Empty (B),  
42    Ensures   => True),  
43   Test_Case =>  
44   (Name      => "Test with not empty buffer",  
45    Mode      => Nominal,  
46    Requires  => not Is_Empty (B),  
47    Ensures   => True);
```


Tools and Technologies

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)

The Hi-Lite Project at AdaCore

Technologies and tools developed at AdaCore in Hi-Lite scope.

- ▶ the compiler GNAT Pro
- ▶ the analyzer CodePeer
- ▶ the verifier GNATProve
using WHY as VC generator and alt-ergo as SMT prover (but complete modularity !)
- ▶ the test harness creator GNATTest

SPARK 2014 Platform

SPARK 2014 language (1/2)

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;
```

Main restrictions with respect to 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`

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 → mathematical contracts
- ▶ target parametrization → 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 $\text{Pre} \Rightarrow X + Y \text{ in Integer}$
- ▶ 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/`