Ada: Provably Reliable

Ada Europe 2013

Thomas Quinot

AdaCore

Berlin, 2013-06-12

AdaCore
The GNAT Pro Company

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

AdaCore
The GNAT Pro Company

Can we do better in Ada?

```
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;
```

AdaCore
The GNAT Pro Company

A Ring Buffer: API

```
1   function Is_Empty (R : Ring_Buffer)
2   return Boolean;
3   -- Check whether the buffer is empty
4
5   procedure Pop (R : in out Ring_Buffer;
6                  Element : out Integer);
7   -- Return the first element of the buffer,
8   -- and remove it from the buffer.
9   -- The buffer should not be empty.
10  -- The length of the buffer is decreased by one.
```

AdaCore
The GNAT Pro Company

Can we do better in Ada 2012? (1/2)

expression functions completely define simple getters in the spec

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

AdaCore
The GNAT Pro Company

Can we do better in Ada 2012? (2/2)

Contracts define the interface between a subprogram and its caller

```
1   function Head (R : in Ring_Buffer)
2   return Content
3   is (R.Data (R.First));
4
5   procedure Pop (R : in out Ring_Buffer;
6                  Element : out Integer)
7   with
8     Pre  => not Is_Empty (R),
9     Post => not Is_Full (R) and then
10            R.Length = R.Length'Old - 1 and then
11            Element = Head (R'Old);
12  -- Remove the returned element from the buffer.
```

AdaCore
The GNAT Pro Company

What if a contract is violated?

Contract = assertion

Run-time violation ⇒ run-time exception raised

Pop is passed an empty ring

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

Pop implementation is faulty

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

AdaCore
The GNAT Pro Company

What about static verification? (1/3)

Limitations of compiler

- Must run quickly → imprecise analysis
- Can detect obvious errors

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

```
postcondition refers only to pre-state
```

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

```
function postcondition does not mention result
```

AdaCore
The GNAT Pro Company

What about static verification? (2/3)

Need for a static verifier

- 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
```

AdaCore
The GNAT Pro Company

What about static verification? (3/3)

Verifier checks:

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

Verifier gives 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
```

AdaCore
The GNAT Pro Company

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

AdaCore
The GNAT Pro Company

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)))))))))
```

AdaCore
The GNAT Pro Company

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 times out
5. prover is not smart enough

SPARK 2014 methodology to investigate unproved VCs: investigate causes from easier to harder.

AdaCore
The GNAT Pro Company

Investigate incorrect code and/or assertion

Contrats expressed in Ada

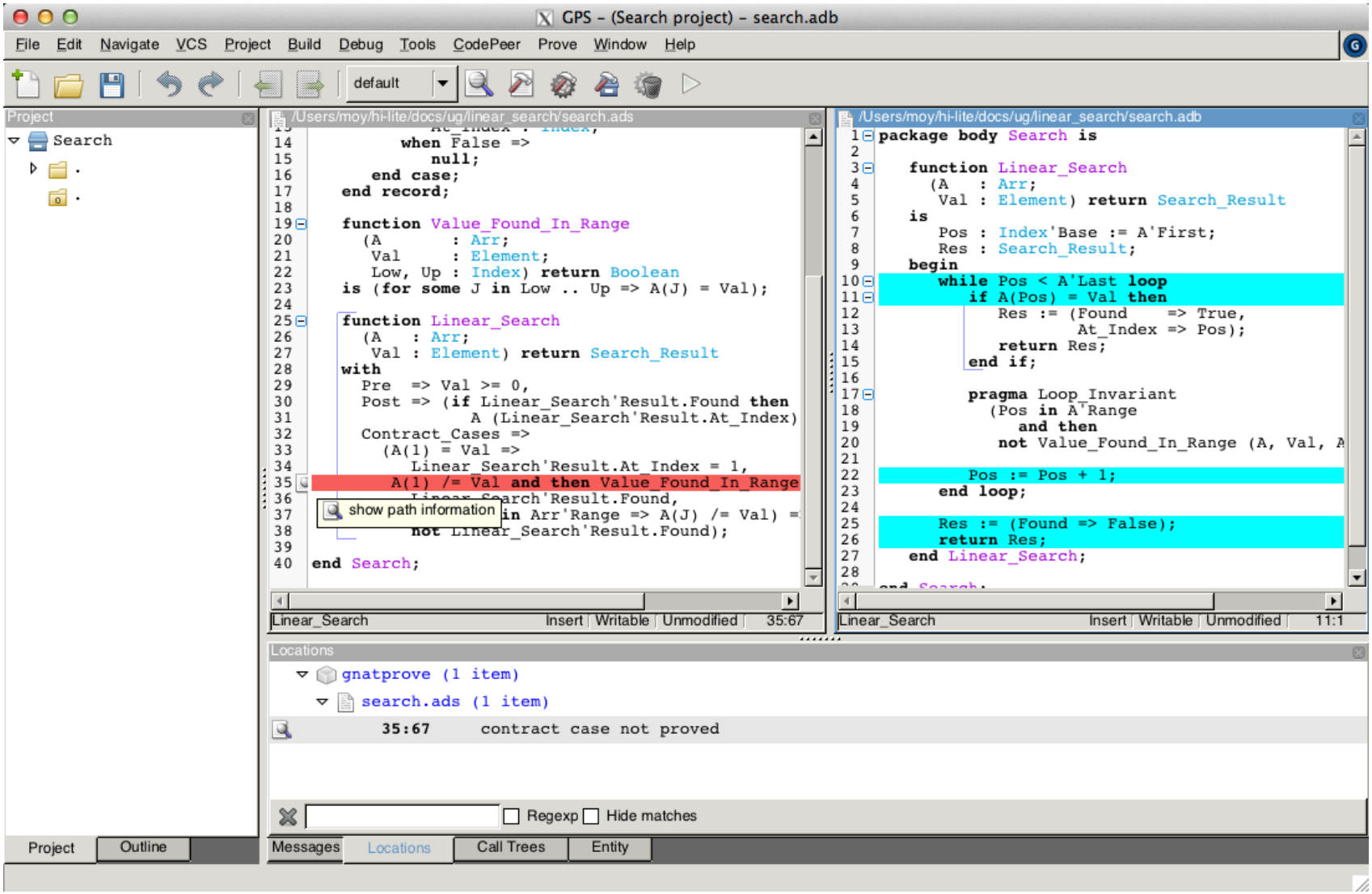
- Code **and** assertions can be executed
- Compiler and verifier fully agree on meaning of assertions
  $\Rightarrow$ code **and** assertions can be tested and debugged

Checks enabled by compiler switches:

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

AdaCore
The GNAT Pro Company

Investigate missing assertions



AdaCore
The GNAT Pro Company

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

AdaCore
The GNAT Pro Company

What to do next?

Traditional fallbacks when proof fails

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

Drawbacks:

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

$\Rightarrow$ new fallback: testing

AdaCore
The GNAT Pro Company

Testing and formal verification (1/3)

Testing is the last resort for...

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

But how to articulate it consistently with formal verification?

New combination provides results **as good as testing alone** (at a fraction of the cost!)

AdaCore
The GNAT Pro Company

Testing and formal verification (2/3)

Combined 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*

AdaCore
The GNAT Pro Company

Testing and formal verification (3/3)

Special testing mode in proof context

- precondition of proved function when called in tested
- postcondition of tested function when called in proved
- initialization of `in out` parameters
- no 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

AdaCore
The GNAT Pro Company

SPARK 2014 language (1/2)

Builds upon 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;
```

AdaCore
The GNAT Pro Company

SPARK 2014 language (2/2)

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

Aspects	`Contract_Cases`, `Global`, `Depends`
Pragmas	`Loop_Invariant`, `Loop_Variant`
Attributes	`Loop_Entry`, `Update`

AdaCore
The GNAT Pro Company

SPARK 2014 toolset

Seamlessly integrated in GNAT compiler front-end:

- produces the AST for compilation **and** verification
- analyzes all constructs (generics, contracts, etc.)
- makes all run-time checks explicit in the AST
- gnat2why back-end finally generates Why input for prover

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

AdaCore
The GNAT Pro Company

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	Normal overflow checks
Minimized	Larger base type (64bits) used when needed
Eliminated	Use bignum library in the remaining cases

- User chooses between 3 modes
- Independent choice for assertions and code
- Same choice for execution and formal verification

AdaCore
The GNAT Pro Company

Hi-lite case studies outcome

Altran Praxis confirmed no regression with respect to previous SPARK toolset

Astrium assessment on Mission and Vehicle management: 10 kSLOC, most modules 100 % automatically proved

All users consider executable contracts an invaluable help in debugging contract *and* code

More SPARK 2014 material

`http://www.spark-2014.org/`

AdaCore
The GNAT Pro Company

Latest product news

New GPL releases: 2013-06-04

GNAT GPL 2013

- Final touches on Ada 2012 support
- Automatic Endianness conversion ('Scalar_Storage_Order)
- Dimensionality checking (new aspects and packages)

SPARK Hi-Lite GPL 2013

Add-on to GNAT GPL 2013 providing toolset for SPARK 2014

- Larger supported subset of Ada (including generics, discriminants, etc.)
- Same contracts used for testing and formal verification
- Applicable to units partly in SPARK
- Improved automatic proof of complex contracts
- New integration in GPS