Ada 2012, SPARK 2014, and Combining Proof and Test

Tucker Taft
AdaCore Inc

Languages and Tools for High Integrity
Bergen, Norway
February, 2014
Cost of testing

- Cost of testing greater than cost of development
- 10% increase each year for avionics software (*Boeing META Project*)
- Uneven partitioning:
  - Uneven quality: 80% of errors traced to 20% of code (*NASA Software Safety Guidebook*)
  - Need to reduce and focus the cost of testing
Formal methods […] might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification.

2011: Formal Methods Supplement (DO-333)
Cost of verification

80% of testing effort

80% of formal effort

Proof+test goal: using formal verification first, then testing...

... to reduce and focus the cost of verification
Programming Contracts

Hoare logic (1969)

{P}C{Q}

logic contracts
for proofs

SPARK (1987)

Eiffel DbC (1986)

executable contracts
for tests

SPARK 2014: Executable Annotation Language
based on Ada 2012
Ada 2012 Programming by contract

• **Pre- and post-conditions for subprograms:**
  – Call is legal if initial conditions satisfy precondition predicate
  – Subprogram works properly if result satisfies postcondition predicate

• **Type invariants for an abstraction:**
  – Every externally accessible value of the type must satisfy a consistency condition
  – For private types and type extensions: specify a consistency condition that objects of the type must obey (e.g. the entries in a bar chart must add up to 100%)
  – Interacts well with OOP

• **Subtype predicates to define applicability:**
  – Only a subset of the values of the type satisfy a named predicate
**Ada 2012 has built-in support for run-time contract checking**

```ada
1 function One_Of (V, X, Y : in Int) return Boolean is (V = X or else V = Y);

4 function Max (X, Y : in Int) return Int with
5   Pre => X /= Y,
6   Post => Max'Result >= X and then
7       Max'Result >= Y and then
8       One_Of (Max'Result, X, Y);

10 function Max (X : in Int_Array) return Int with
11   Post => (for all J in X'Range =>
12       Max'Result >= X(J)) and then
13       (for some J in X'Range =>
14           Max'Result = X(J));
```
generic
  type Item is private;
package Stack_Interfaces is
  type Stack is interface;
  function Is_Empty (S : Stack) return Boolean is abstract;
  function Is_Full (S : Stack) return Boolean is abstract;

  procedure Push (S : in out Stack; I : in Item) is abstract
    with Pre'Class => not Is_Full (S),
         Post'Class => not Is_Empty (S);

private
  ...

end Stack_Interfaces;
package Bars is
  type Bar_Chart is private
    with Type.Invariant => Is_Complete(Bar_Chart);
  function Is_Complete (X : Bar_Chart) return Boolean;
private
  type Bar_Chart is array (1 .. 10) of Integer;
end Bars;

package body Bars is
  function Is_Complete (X : Bar_Chart) is
    -- verify that component values add up to 100
  end;
Contracts and Program Correctness

• Contracts help the programmer (force the programmer?) to make his intention more explicit (strong typing is an earlier step in the same direction).

• Checking of contract may be
  – static (compiler)
  – dynamic (run-time assertions)

• Contracts help develop testing protocols

• Contracts complement and assist static analysis tools

• Ada 2012 is one of the first mainstream language to incorporate contracts as a general programming tool
Abstract Stack Interface

generic
  type Item is private;
package Stack_Interfaces is
  type Stack is interface;
  function Is_Empty (S : Stack) return Boolean is abstract;
  function Is_Full (S : Stack) return Boolean is abstract;

  procedure Push (S : in out Stack; I : in Item) is abstract;

  function Pop (S : in out Stack) return Item is abstract;

end Stack_Interfaces;
Bounded Stack implements Stack Interface

```ada
generic
package Stack_Interfaces.Bounded is
type Bounded_Stack(<>) is new Stack with private;
function Create(Size: Natural) return Bounded_Stack;

function Size(S : Bounded_Stack) return Natural;
function Count(S : Bounded_Stack) return Natural;

function Is_Empty (S : Bounded_Stack) return Boolean
  is (Count(S) = 0); -- expression functions
function Is_Full (S : Stack) return Boolean
  is (Count(S) = Size(S)); -- expression functions

procedure Push (S : in out Bounded_Stack; I : in Item);
function Pop(S : in out Bounded_Stack) return Item;

private ...
```
Bounded Stack Internals

generic
package Stack_Interfaces.Bounded is

... private

  type Item_Array is array(Positive range <> ) of Item;
  type Bounded_Stack( Size : Natural ) is new Stack with record
    Count : Natural := 0;
    Data : Item_Array( 1..Size );
  end record;
end Stack_Interfaces.Bounded;

package body Stack_Interfaces.Bounded is

... procedure Push ( S : in out Bounded_Stack ; I : in Item ) is
begin
  S.Count := S.Count + 1;
  S.Data( S.Count ) := I;
end Push;
end Stack_Interfaces.Bounded;
What sort of Pre- and Postconditions are appropriate here?

• **Preconditions prevent failures; Postconditions define effects**
• Push will get an index out of bounds if $S\.Count = S\.Size$ on entry
• Create precondition to prevent that:
  
  ```ada
  procedure Push(...) with Pre => Count(S) < Size(S);
  ```
• Now we have the following code:
  ```ada
  Stk : BI_Inst.Bounded_Stack := BI_Inst.Create(10);
  ...
  BI_Inst.Push(Stk, X);  -- Can we be sure this will satisfy the Pre?
  ```
• We need a **Post** on Create to know initial Size and Count:
  ```ada
  function Create(...) return Bounded_Stack
    with Post => Bounded.Size(Create'Result) = Size
    and Count(Create'Result) = 0;
  ```
• We also need a **Post** on Push itself so 10 Pushes are known safe:
  ```ada
  procedure Push(...) with Pre => Count(S) < Size(S),
      Post => Count(S) = Count(S)’Old + 1;
  ```
package Stack_Interfaces.Bounded is

generic

type Bounded_Stack<> is new Stack with private;

function Create(Size : Natural) return Bounded_Stack
    with Post => Bounded.Size(Create’Result) = Size
    and Count(Create’Result) = 0;

function Size(S : Bounded_Stack) return Natural;

function Count(S : Bounded_Stack) return Natural
    with Post => (Count(S) <= Size(S));

function Is_Empty (S : Bounded_Stack) return Boolean
    is (Count(S) = 0);

function Is_Full (S : Stack) return Boolean
    is (Count(S) = Size(S));

procedure Push (S : in out Bounded_Stack; I : in Item)
    with Pre => Count(S) < Size(S),
    Post => Count(S) = Count(S)’Old + 1;

function Pop(S : in out Bounded_Stack) return Item
    with Pre => Count(S) > 0,
    Post => Count(S) = Count(S)’Old - 1;

private ...

Bounded Stack with Pre/Postconditions
Now suppose we use the abstract stack...

- **Imagine we have a class-wide operation:**
  
  ```ada
  procedure Replace_Top(S : in out Stack'Class; I : Item) is
    Discard : constant Item := Pop(S);
  begin
    Push(S, I);
    end Replace_Top;
  ```

- **Need a classwide precondition on Pop, and a normal precondition on Replace_Top to make things safe:**
  
  ```ada
  function Pop(…) with Pre'Class => not Is_Empty(S)
  procedure Replace_Top(…) with Pre => not Is_Empty(S);
  ```

- **Need a classwide postcondition on Push and a normal postcondition on Replace_Top to safely do it twice:**
  
  ```ada
  procedure Push(…) with Post'Class => not Is_Empty(S)
  procedure Replace_Top(…) with Post => not Is_Empty(S)
  ```

- **Classwide pre/postconds must be checked on overridings**
generic
type Item is private;
package Stack_Interfaces is
  type Stack is interface;
  function Is_Empty (S : Stack) return Boolean is abstract;
  function Is_Full (S : Stack) return Boolean is abstract;

procedure Push (S : in out Stack; I : in Item) is abstract
  with Pre'Class => not Is_Full (S),
       Post'Class => not Is_Empty (S);
function Pop (S : in out Stack) return Item is abstract
  with Pre'Class => not Is_Empty (S),
       Post'Class => not Is_Full (S);
end Stack_Interfaces;
Now should verify that Bounded_Stack will abide by ancestor’s Pre’Class and Post’Class

• **Ancestor type Stack specifies:**
  procedure Push (S : in out Bounded_Stack; I : in Item)
    with Pre'Class => not Is_Full (S),
    Post'Class => not Is_Empty (S);

• **Bounded_Stack explicitly specifies:**
  function Is_Empty (S : Bounded_Stack) return Boolean
    is (Count(S) = 0); -- not Is_Empty == Count(S) /= 0
  function Is_Full (S : Stack) return Boolean
    is (Count(S) = Size(S)); -- not Is_Full == Count(S) /= Size(S)
  procedure Push (S : in out Bounded_Stack; I : in Item)
    with Pre => Count(S) < Size(S),
    Post => Count(S) = Count(S)'Old + 1;

• **Liskov Substitution Principle (LSP) says:**
  – Caller sees ancestor precondition, so must *imply* descendant precondition
  – Caller sees ancestor postcondition, so must *be implied by* descendant postcondition
  – Verified:
    \[ \text{Count}(S) \neq \text{Size}(S) \text{ and } \text{Count}(S) \leq \text{Size}(S) \Rightarrow \text{Count}(S) < \text{Size}(S) \]
    \[ \text{Count}(S) = \text{Count}(S)’\text{Old} + 1 \text{ and } \text{Count}(S)’\text{Old} \geq 0 \Rightarrow \text{Count}(S) \neq 0 \]
Ada 2012 and Liskov Substitution Principle

- Ada 2012 compiler is *not* required to statically check that Pre’Class implies Pre nor that Post implies Post’Class
  - Ada 2012 compiler is only required to do run-time checks
  - Other tools can attempt proofs that the run-time checks will not fail

- Ada 2012 language ensures implications by *effectively*:
  - “or”ing Pre’Class of ancestors with Pre’Class of descendant, and
  - “and”ing Post’Class of ancestors with Post’Class of descendant

- The Pre’Class “or”ing is done “implicitly”:
  - In a “dispatching” call, caller only checks the Pre’Class annotations that they can “see”;
  - Pre’Class of descendants of T where controlling operand is of type T’Class are *not* even checked.

- The Post’Class “and”ing is done by checking all of them.
package Bars is
  type Bar_Chart is private
    with Type_Invariant => Is_Complete(Bar_Chart);
  function Is_Complete (X : Bar_Chart) return Boolean;
private
  type Bar_Chart is array (1 .. 10) of Integer;
end Bars;

package body Bars is
  function Is_Complete (X : Bar_Chart) is
    -- verify that component values add up to 100
  end;
The Role of Type Invariants

- Type invariants are used to encode some property that is preserved by all operations on a type.
  - Becomes implicit Pre and Post condition for every operation
- Type invariants are generally introduced when attempts to prove that a given postcondition is satisfied requires that all operations guarantee certain minimum requirements.
  
  **Example:**
  - Imagine a stack of pointers, and we ensure that Push is only passed `not null` pointers.
    - Can we ensure that Pop returns only `not null` values back?
  - Solution is to come up with a Type_Invariant that says:
    - All elements at or “below” the stack pointer are /= null
    - Then show that Push (and other ops) preserve it.
  - Note that type invariants are often representation specific
    - In Ada 2012, they can be given in the private part.
generic
  type T(<>) is limited private;
  type T_Ptr is access T;
package Pointer_Stacks is
  type Pointer_STACK is private;
  procedure Push(PS : in out Pointer_STACK; Ptr : not null T_Ptr);
  function Pop(PS : in out Pointer_STACK) return not null T_Ptr;
private
  type Ptr_Array is array(Positive range <>) of T_Ptr;
  type Pointer_STACK(Size : Natural) is record
    Count : Natural := 0;
    Data : Ptr_Array(1..Size) := (others => null);
  end record
  with Type.Invariant =>
    (for all I in 1..Pointer_STACK.Count => Pointer_STACK.Data(I) /= null);
end Pointer_Stacks;
type Pointer_Stack(Size : Natural) is record
  Count : Natural := 0;
  Data : Ptr_Array(1..Size) := (others => null);
end record
with Type.Invariant =>
  (for all I in 1..Pointer_Stack.Count =>
     Pointer_Stack.Data(I) /= null);
end Pointer_Stacks;
package body Pointer_Stacks is
  procedure Push(PS : in out Pointer_Stack; Ptr : not null T_Ptr) is
  begin
    PS.Count := PS.Count + 1;  PS.Data(PS.Count) := Ptr;
  end Push;
  function Pop(PS : in out Pointer_Stack) return not null T_Ptr is
  begin
    PS.Count := PS.Count – 1;  return PS.Data(PS.Count + 1);
  end Pop;
end Pointer_Stacks;
Subtype Predicates
Static_PREDICATE and Dynamic_PREDICATE

• A subtype “predicate” is a generalization of the notion of a “constraint”
  – It identifies a *subset* of the values of a type or subtype

• Examples of constraints:
  – subtype Digit is Integer range 0..9
    – “range 0..9” is a range constraint
  – Data : Ptr_Array(1..Size)
    – “(1..Size)” is an index constraint

• Examples of predicates:
  – subtype Long_Weekend is Weekday
    with Static_PREDICATE =>
    Long_Weekend in Friday | Saturday | Sunday | Monday;
  – subtype Operator_Node is Node
    with Dynamic_PREDICATE =>
Static vs. Dynamic Predicates

**Static_Predicate:**
- Must apply to a scalar or string type and may involve one or more comparisons between the value being tested and static values
- All possible values can be determined statically
- Subtypes with such a predicate can be used as the choice in a case statement or the bounds of a loop iteration
- Initialized objects to which such a predicate applies always satisfy the predicate

**Dynamic_Predicate:**
- Defined by an arbitrary boolean expression involving the value being tested
- All possible values need not be determinable statically
- Subtypes with such a predicate can be used to declare an object and in a membership test, but may not be used for looping or as choices in a case statement
- Some violations of the predicate might not be immediately detected
  - Only checked on certain “whole object” operations
• Allows indexing over containers, with and without cursors:

```ada
for Cursor in Iterate (Container) loop
    Container (Cursor) := Container (Cursor) + 1;
end loop;
```

```ada
for Thing of Box loop
    Modify (Thing);
end loop;
```

Both forms apply to arrays and containers.
State that A is sorted:

(for all J in A'First .. T'Pred (A'Last) =>
  A (J) <= A (T'Succ (J)))

State that N is *not* a prime number:

(for some X in 2 .. N / 2 =>
  N mod X = 0)

*some* is a new reserved word
**SPARK 2014 Builds on Ada 2012**

- **Remove features that can create aliasing**
  - No access types
  - No parameter aliasing
  - No undeclared use of global variables

- **Add annotations to specify information flow**
  - Global variable usage
  - Information flow dependence
  - Named abstract state variables to represent package state
    - Refined in package body

```ada
package Random with Abstract_State => Seed is
    function Next_Rand return Float
      with Global => (In_Out => Seed),
      Depends => (Seed => Seed,
                    Next_Rand'Result => Seed),
      Post => Next_Rand'Result in 0.0 .. 1.0;
```
SPARK 2014 toolset based on open-source “Hi-Lite” Project
Testing vs. Formal Verification

local exhaustivity argument:
- each function covered
  - enough behaviors explored

global soundness argument:
- all functions proved
  - all assumptions justified

use Q code
cover P constructs

prove pre of Q
assume post of Q

P calls Q

actual body of Q
or stub…

assume pre of Q
prove post of Q

P calls Q
Combining tests and proofs

Verification combining tests and proofs should be AT LEAST AS GOOD AS verification based on tests only.

How do we justify assumptions made during proof?

P is tested

P calls Q

Q calls P

Q is proved

Ada 2012, SPARK 2014, Proof + Test 32
Caution: contracts are not only pre/post!

```ada
procedure Open
  (Customer : in Identity.Name;
   Id       : in Identity.Id;
   Cur      : in Money.CUR;
   Account  : out Account_Num)
with
  Pre  => not Max_Account_Reached,
  Post => Existing (Account)...
```

- strong typing
- parameters not aliased
- data dependences
- parameters initialized
Combination 1: tested calls proved

during testing:
check that pre-condition of Q is respected

assumption for proof: pre-condition of Q is respected
**Combination 2: proved calls tested**

during testing: check that post-condition of P is respected

assumption for proof: post-condition of P is respected
Testing + Formal Verification

**local** exhaustivity argument:
- test: function covered
- proof: by nature of proof

**global** soundness argument:
- proof: assumptions proved
- test: assumptions tested

Testing must check additional properties
Done by compiler instrumentation
Proof + Test toolsuite

Ada 2012 compiler/front end

Ada unit testing

SPARK + SMT solver unit proof

Executable

Aggregated verification results
Rail, Space, Security: Three Case Studies for SPARK 2014

Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré and Yannick Moy

Embedded Real Time Software and Systems – February 5th, 2014
programming language for long-lived embedded critical software

programming by contract

Ada subset for formal verification

practical formal verification
SPARK 2014 Value Proposition

- Functional Requirement
- Functional Verification
- Software Architecture Verification
- Unit Requirements
- Unit Verification
- Code
- Robustness Verification
SPARK 2014 Value Proposition (DO-178C Version)

System Requirements

High Level Requirements

Software Architecture

Low Level Requirements

Source Code

Executable Object Code

- Software architecture is consistent
- Accuracy
- Consistency

- Compliance
- Robustness
- Property Preservation
Program

Contract = agreement between client & supplier

caller & callee
Case Studies
Case study 1: Train Control Systems

David Mentré
openETCS

- **Open Source** → no vendor lock-in
- **Model** based (SysML)
- **Formal** methods → Strong guaranties of correctness
- “**Open Proofs**” → Everybody can re-check
Formalization of the Correctness of Step Functions

Restrictive_Merge?

Has_Same_Delimiters?

Get_Value?

Minimum_Until_Point?
Results

SPARK 2014 very good for:
• Capturing objects in the requirements
• Readability of the specifications (= contracts)
• Automatic proof of absence of run-time errors
• Automatic proof of simple functional contracts
• Dynamic verification of contracts and assertions

SPARK 2014 is not good for:
• Proving existing code without any modifications
• Proving automatically complex functional contracts

Areas requiring improvements:
• Possibility to prove some properties interactively (in 2014 roadmap)
• Better diagnostic for incomplete loop invariants (in 2014 roadmap)
• Training for developers to use proof tools (available in SPARK Pro subscription)
• Workflow to make efficient use of developers’ time (in progress)
Case study 2: Flight Control and Vehicle Management in Space

David Lesens
On-board Control Procedure

- **On-board control procedure**
  - Software program designed to be executed by an OBCP engine, which can easily be loaded, executed, and also replaced, on-board the spacecraft

- **OBCP code**
  - Complete representation of an OBCP, in a form that can be loaded on-board for subsequent execution

- **OBCP engine**
  - Application of the on-board software handling the execution of OBCPs

- **OBCP language**
  - Programming language in which OBCP source code is expressed by human programmers
Example:

- A list of event detection statuses
- Request to reset the detection status for Event

```ada
procedure Reset_Event_Status (Event : in T_Event) with
Post =>
not Event_Status (Event).Detection and
(for all Other_Event in T_Event =>
(if Other_Event /= Event then
   Event_Status (Other_Event) = Event_Status'Old (Other_Event)));
```

**Post-condition**

- The detection of event is reset
- For all other events

**The detection status is unchanged**

Event1 | Event2 | Event3
---|---|---
Not detected | Not detected | Detected
### Automatic Proof Results

#### Numerical control/command algorithms

<table>
<thead>
<tr>
<th>Part</th>
<th># subprograms</th>
<th># checks</th>
<th>% proved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math library</td>
<td>15</td>
<td>27</td>
<td>92</td>
</tr>
<tr>
<td>Numerical algorithms</td>
<td>30</td>
<td>265</td>
<td>98</td>
</tr>
</tbody>
</table>

#### Mission and vehicle management

<table>
<thead>
<tr>
<th>Part</th>
<th># subprograms</th>
<th># checks</th>
<th>% proved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single variable</td>
<td>85</td>
<td>268</td>
<td>100</td>
</tr>
<tr>
<td>List of variables</td>
<td>140</td>
<td>252</td>
<td>100</td>
</tr>
<tr>
<td>Events</td>
<td>24</td>
<td>213</td>
<td>100</td>
</tr>
<tr>
<td>Expressions</td>
<td>331</td>
<td>1670</td>
<td>100</td>
</tr>
<tr>
<td>Automated proc</td>
<td>192</td>
<td>284</td>
<td>74</td>
</tr>
<tr>
<td>On board control proc</td>
<td>547</td>
<td>2454</td>
<td>95</td>
</tr>
</tbody>
</table>

*Formal Verification of Aerospace Software, DASIA 2013,*

SPARK 2014 very good for:

- Proof of absence of run-time errors
- Correct access to all global variables
- Absence of out-of-range values
- Internal consistency of software unit
- Correct numerical protection
- Correctness of a generic code in a specific context

SPARK 2014 is good for:

- Proof of functional properties

Areas requiring improvements:

- Sound treatment of floating-points (done)
- Support of tagged types (in 2014 roadmap)
- Helping user with unproved checks (in 2014 roadmap)
Case study 3: Biometric Access to a Secure Enclave

Pavlos Efstathopoulos
### Formalization of the “Admin” Package

<table>
<thead>
<tr>
<th>Aspect / Pragma</th>
<th>Num. of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>197</td>
</tr>
<tr>
<td>Refined_Global</td>
<td>71</td>
</tr>
<tr>
<td>Refined_Depends</td>
<td>40</td>
</tr>
<tr>
<td>Depends</td>
<td>202</td>
</tr>
<tr>
<td>Pre</td>
<td>28</td>
</tr>
<tr>
<td>Post</td>
<td>41</td>
</tr>
<tr>
<td>Assume</td>
<td>3</td>
</tr>
<tr>
<td>Loop_Invariant</td>
<td>10</td>
</tr>
</tbody>
</table>

**Dataflow**

**Refinement**

**Information flow**

**Functional contracts**

**User guidance**

**Assumptions**
SPARK 2014 very good for:

- Expressing **specification-only** code
- Analysis of code that was **not analyzable with SPARK 2005**
- Automating proofs with **less user efforts**
- Expressing **complete functional behavior** of functions
- **Readability** of the formal specifications
- Uncovering **corner cases** related to run-time checks

Areas requiring improvements:

- **Summary** of proof results (done)
Lessons Learned
SPARK 2014 Strengths

- Expressive yet analyzable language
- Executable contracts
- Better automation of proofs
SPARK 2014 Challenges

- need expert advice sometimes
- static debugging of contracts
- code and specifications must be adapted
SPARK in 2014
Now available as beta
First release April 2014

See http://www.adacore.com/sparkpro
and http://www.spark-2014.org

New LabCom ProofInUse between AdaCore and INRIA
(hiring 2 R&D software engineer postdocs)
SPARK 2014 proof + test

Conclusion
Conclusions

- **Ada 2012 supports contract-based programming**
  - *Pre, Post, Type_Invariant, *__Predicate* annotations
  - Executable semantics

- **SPARK 2014 builds on Ada 2012**
  - Provides formal static verification of contract annotations
  - Adds annotations for global variable usage and information flow
  - Supported by new open-source toolset based on Why3 and SMT

- **Proof + Test approach supports real-world applications**
  - Get best of static and dynamic verification
  - Reduces overall cost while increasing confidence
Airbus “must-have”s for formal methods

- Soundness
- Applicability to the code
- Usability by normal engineers on normal computers
- Improve on classical methods
- Certifiability

ongoing research
How to learn more

- http://www.ada2012.org
- http://www.adacore.com

S. Tucker Taft, VP & Director of Language Research
AdaCore
24 Muzzey Street 3rd Floor
Lexington, MA 02421 USA

taft@adacore.com