AdaCore
The GNAT Pro Company

Ada 2012, SPARK 2014, and Combining Proof and Test

Tucker Taft
AdaCore Inc

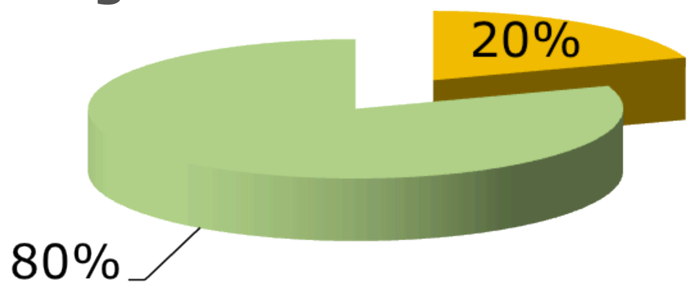
Languages and Tools for High Integrity

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:**



← 80% of effort!

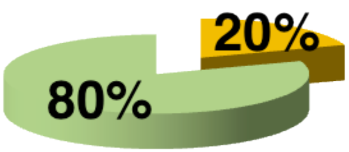
- **Uneven quality: 80% of errors traced to 20% of code**
  (NASA Software Safety Guidebook)
- **Need to reduce and focus the cost of testing**

DO-178C: formal methods can replace testing

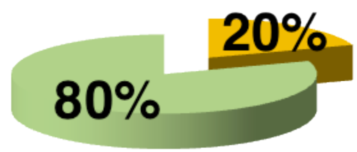
Formal methods […] might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification.

2011: Formal Methods Supplement (DO-333)

Cost of verification

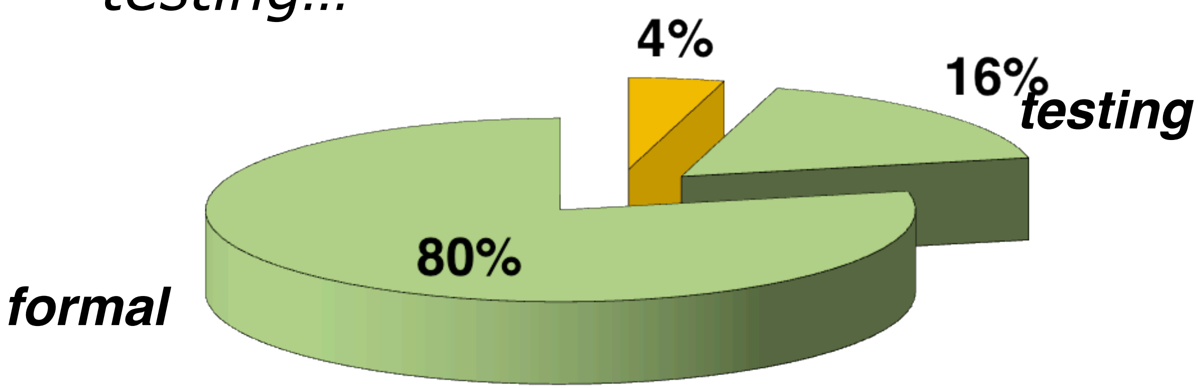


← *80% of*
***testing** effort*



← *80% of*
***formal** effort*

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



… to reduce and focus the cost of verification

AdaCore
The GNAT Pro Company

SPARK 2014 + Ada 2012

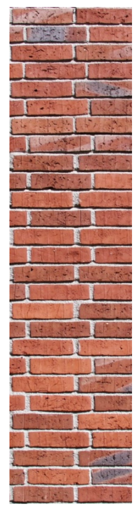
Proof + Test

Programming Contracts

{P}C{Q} **Hoare logic (1969)**

logic contracts
for proofs

SPARK (1987)

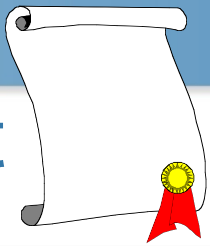


executable contracts
for tests

Eiffel DbC (1986)

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

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

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

function Max (X : in Int_Array) return Int with
   Post => (for all J in X'Range =>
              Max'Result >= X(J)) and then
           (for some J in X'Range =>
              Max'Result = X(J));
```

Ada 2012 Pre- and Postconditions

```
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;
```

Ada 2012 Type invariants

```
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;
```

AdaCore
The GNAT Pro Company

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**

AdaCore
The GNAT Pro Company

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

```
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:

```
procedure Push(…) with Pre => Count(S) < Size(S);
```

- Now we have the following code:

```
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:

```
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:

```
procedure Push(…) with Pre => Count(S) < Size(S),
                       Post => Count(S) = Count(S)'Old + 1;
```

Bounded Stack with Pre/Postconditions

```
generic
package Stack_Interfaces.Bounded is
   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 …
```

Now suppose we use the abstract stack…

- **Imagine we have a class-wide operation:**

```
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:**

```
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:**

```
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**

Abstract Stack with Pre/Postconditions

```
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:

    Count(S) /= Size(S) and Count(S) <= Size(S) ➔ Count(S) < Size(S)

    Count(S) = Count(S)'Old+1 and Count(S)'Old >= 0 ➔ Count(S) /= 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.

Ada 2012 Type invariants

```
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.

Pointer Stack Type Invariant

```
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;
```

Verify Pointer Stack Type Invariant

```
   ...
   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 =>
    Operator_Node.Kind in Unary_Kind | Binary_Kind;

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

Ada 2012 Container/Array Iterators

- **Allows indexing over containers, with and without cursors:**

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

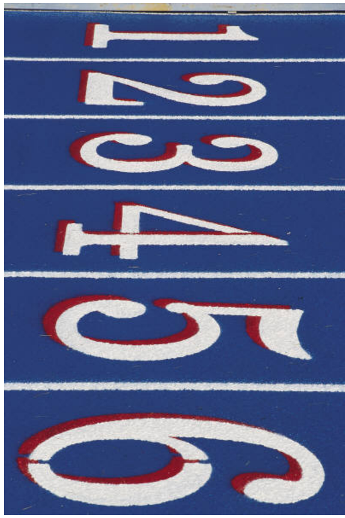
```
for Thing of Box loop
    Modify (Thing);
end loop;
```

Both forms apply to arrays and containers.

Ada 2012 Quantified expressions

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**

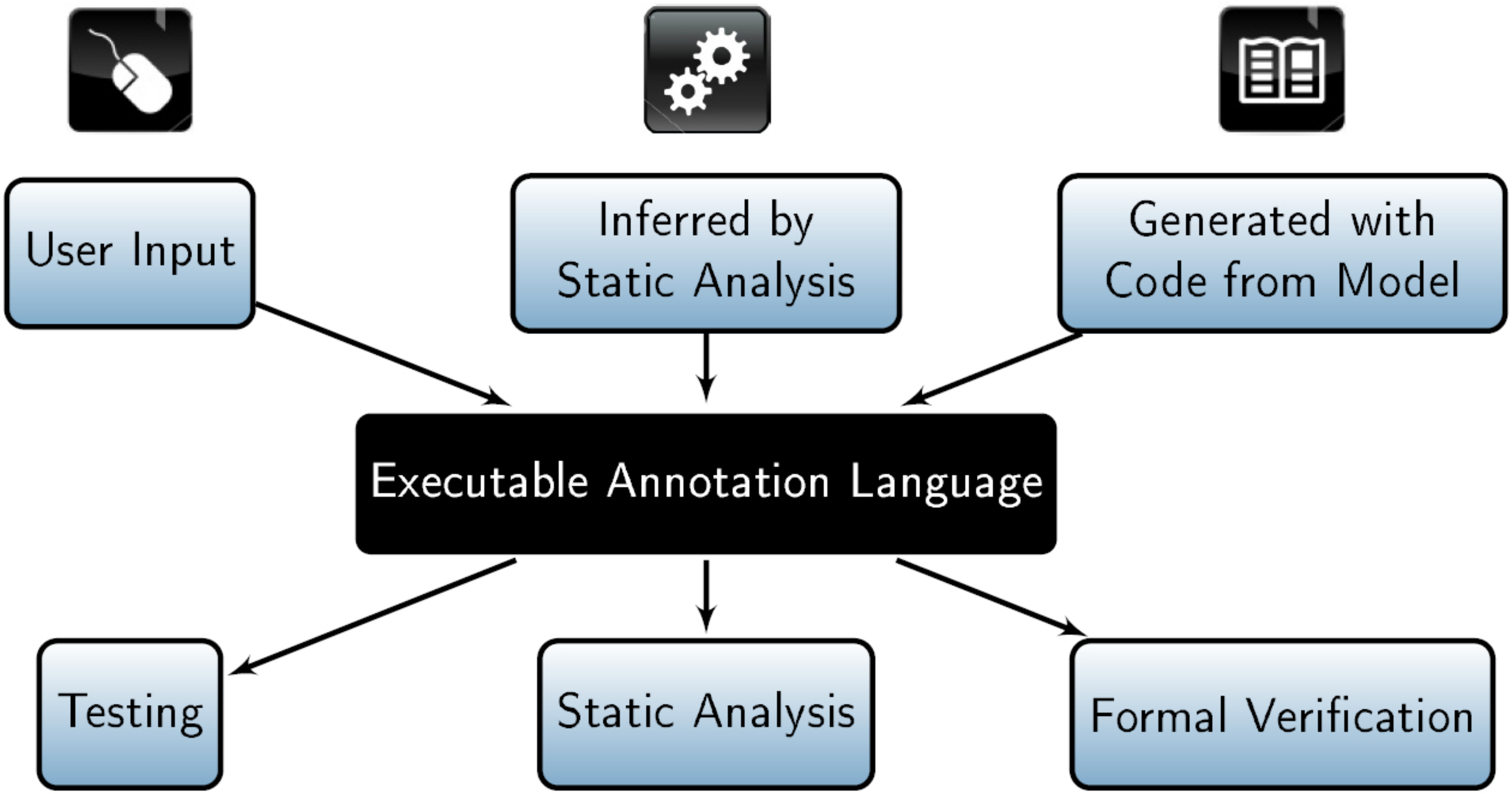
AdaCore
The GNAT Pro Company

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

```
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;
```

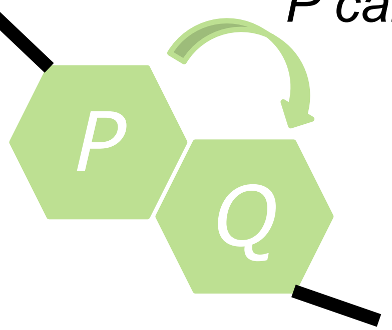
SPARK 2014 toolset
based on open-source “Hi-Lite”Project



Testing vs. Formal Verification

use Q code
cover P constructs

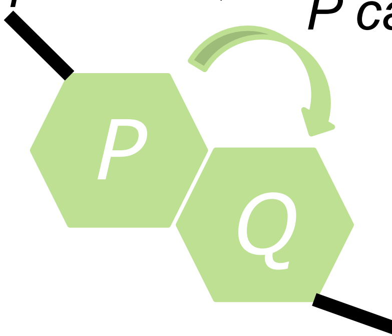
P calls Q



actual body of Q
or stub…

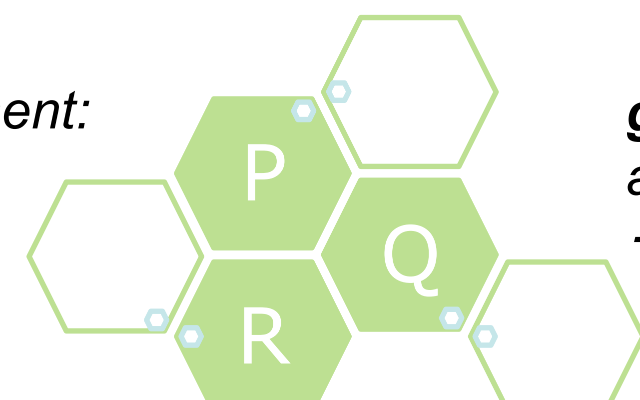
prove pre of Q
assume post of Q

P calls Q



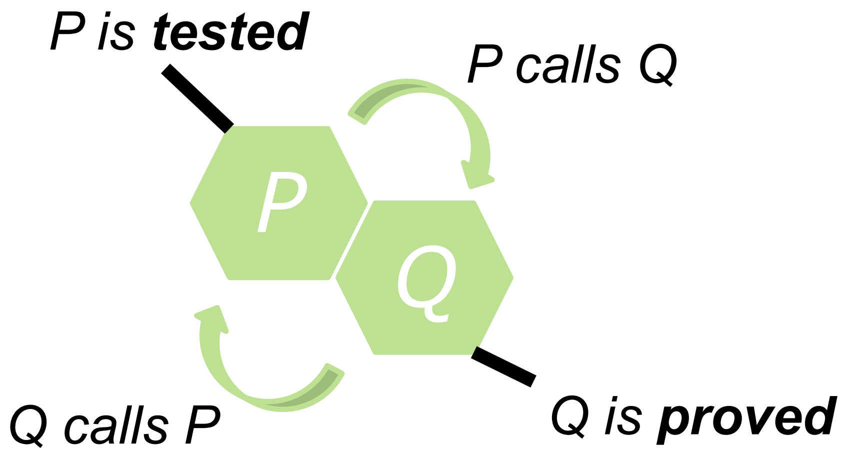
assume pre of Q
prove post of Q

***local** exhaustivity argument:*
each function covered
→ *enough behaviors explored*



***global** soundness argument:*
all functions proved
→ *all assumptions justified*

Combining tests and proofs



How do we justify assumptions made during proof?

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

Caution: contracts are not only pre/post!

strong typing

parameters not aliased

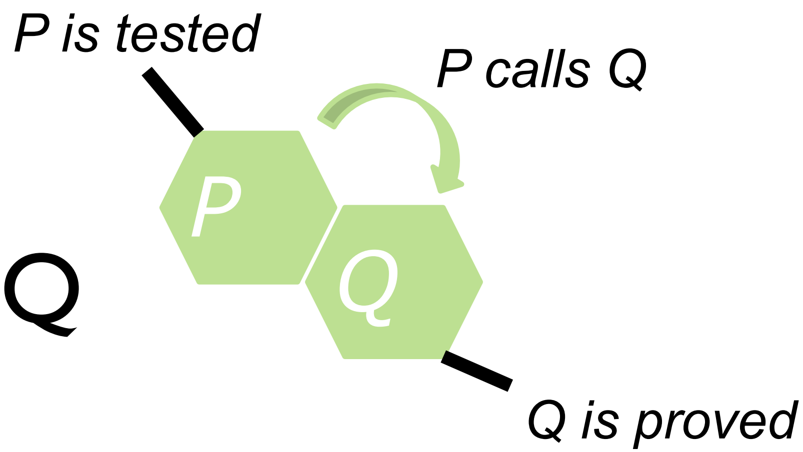
```
procedure Open
   (Customer : in      Identity.Name;
    Id       : in      Identity.Id;
    Cur      : in      Money.CUR;
    Account  :     out Account_Num)
with
   Pre  => not Max_Account_Reached,
   Post => Existing (Account) ...
```

parameters initialized

data dependences

Combination 1: tested calls proved

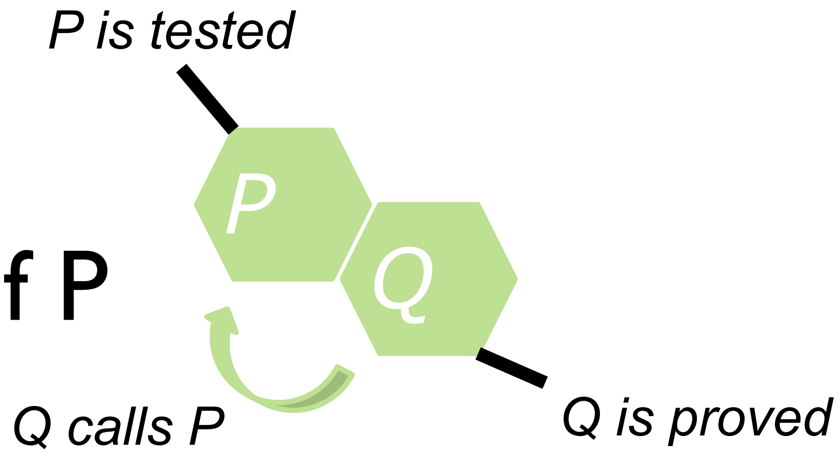
during testing:
check that
*p*re-condition of Q
is respected



assumption for proof:
*p*re-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

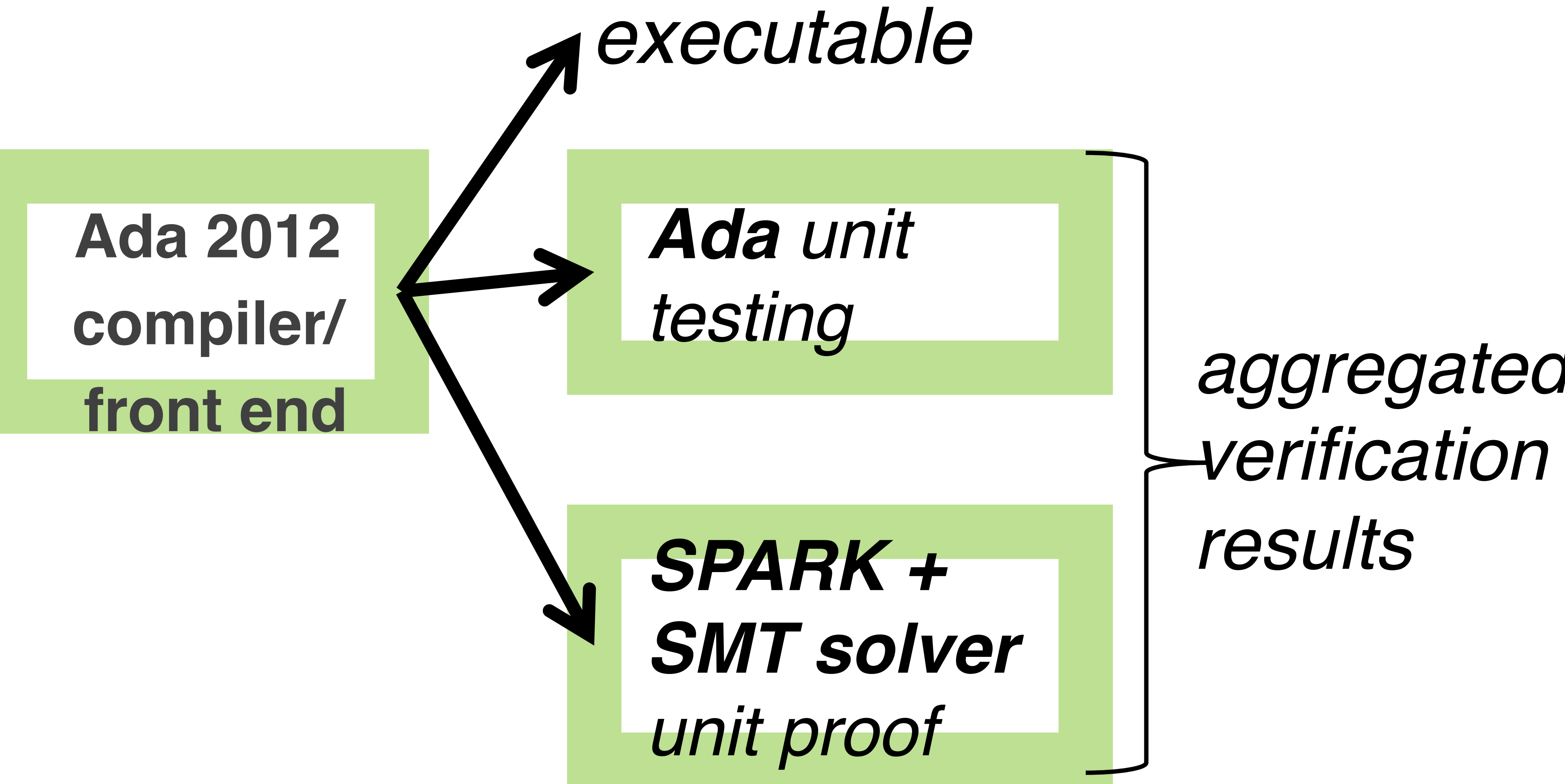
Testing + Formal Verification

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



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

SPARK 2014

Ada 2012 and SPARK 2014

programming language for long-lived embedded critical software

Ada 2012

programming by contract

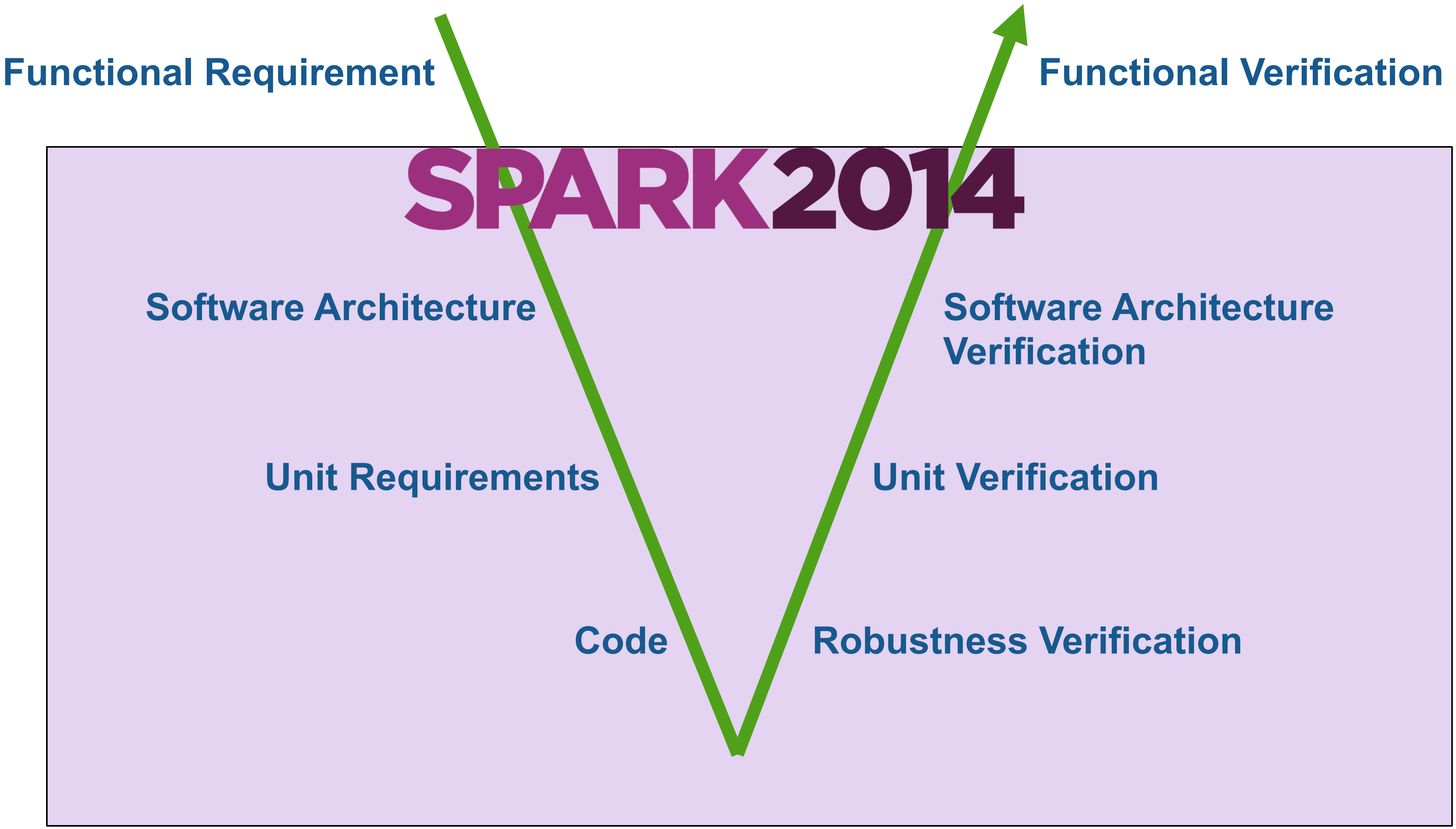
SPARK Ada

Ada subset for formal verification

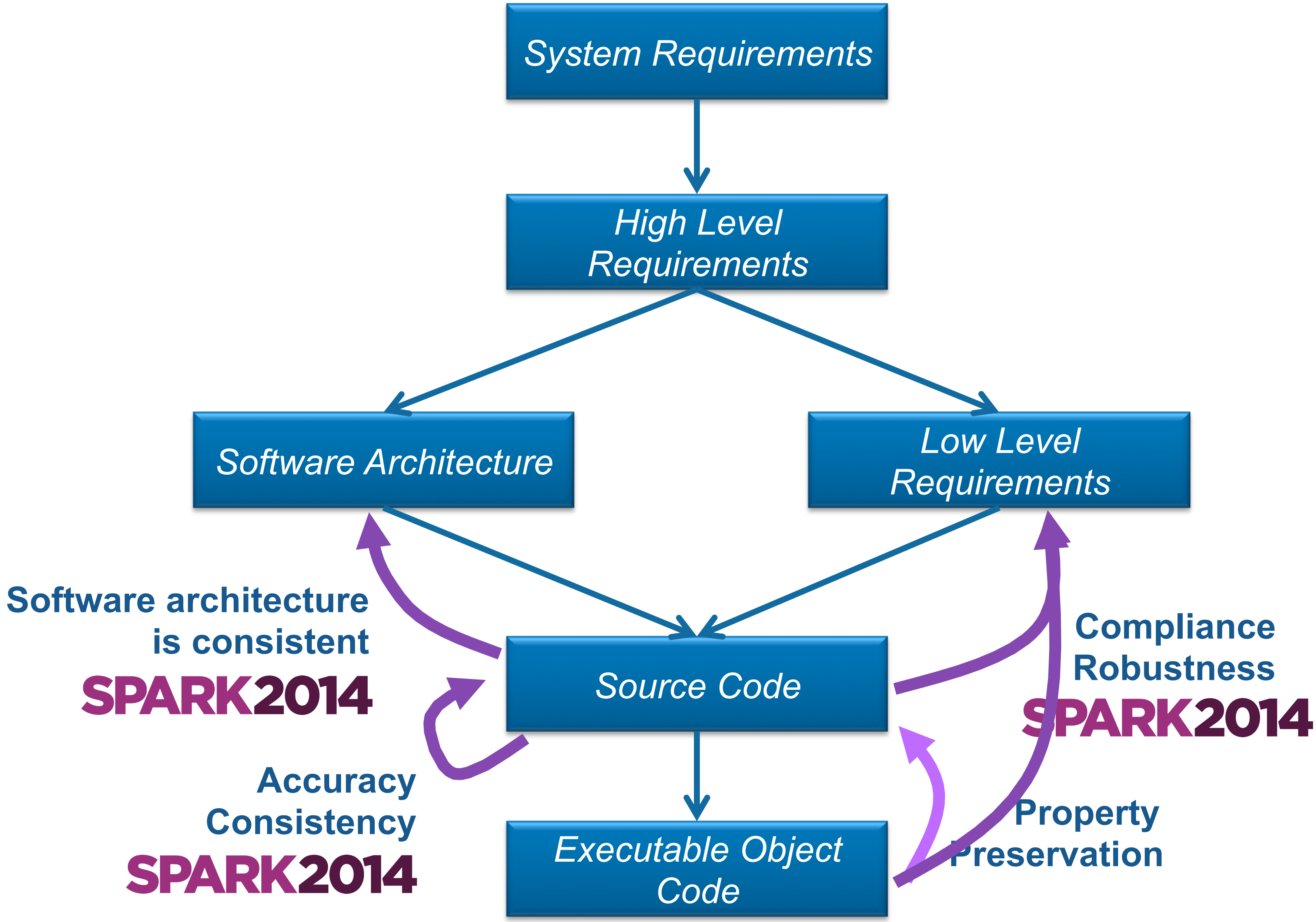
SPARK 2014

practical formal verification

SPARK 2014 Value Proposition



SPARK 2014 Value Proposition (DO-178C Version)



Program

Contract = agreement between ~~client & supplier~~

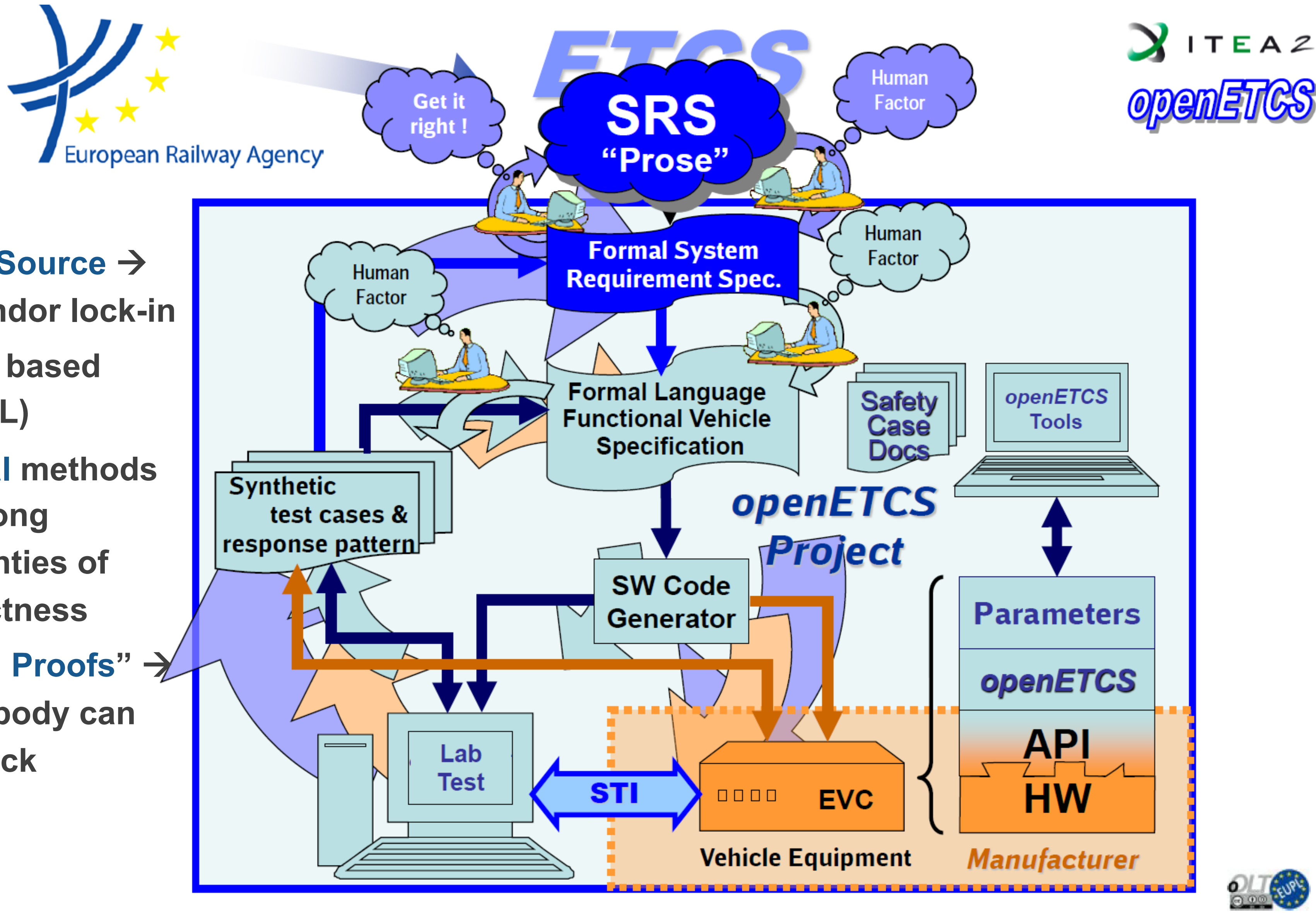
caller & callee

Case Studies

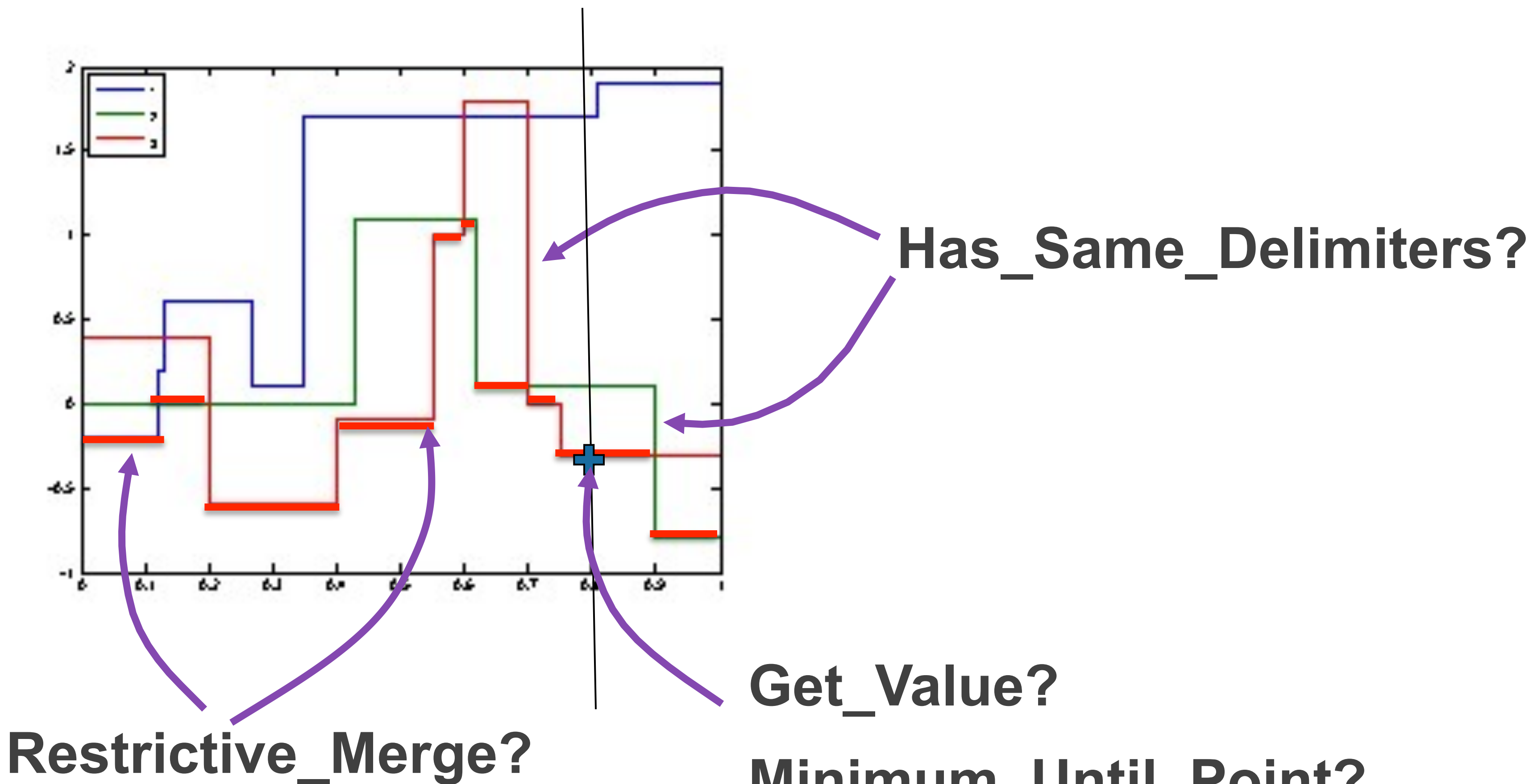
Case study 1: Train Control Systems

David Mentré

- **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



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

ECSS-E-ST-70-01C
16 April 2010

Space engineering

Spacecraft on-board control procedures

EUROPEAN COOPERATION
ECSS
FOR SPACE STANDARDIZATION

- **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*

```
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

Part	# subprograms	# checks	% proved
Math library	15	27	92
Numerical algorithms	30	265	98

Mission and vehicle management

Part	# subprograms	# checks	% proved
Single variable	85	268	100
List of variables	140	252	100
Events	24	213	100
Expressions	331	1670	100
Automated proc	192	284	74
On board control proc	547	2454	95

Formal Verification of Aerospace Software, DASIA 2013,
http://www.open-do.org/wp-content/uploads/2013/05/DASIA_2013.pdf

Results

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**

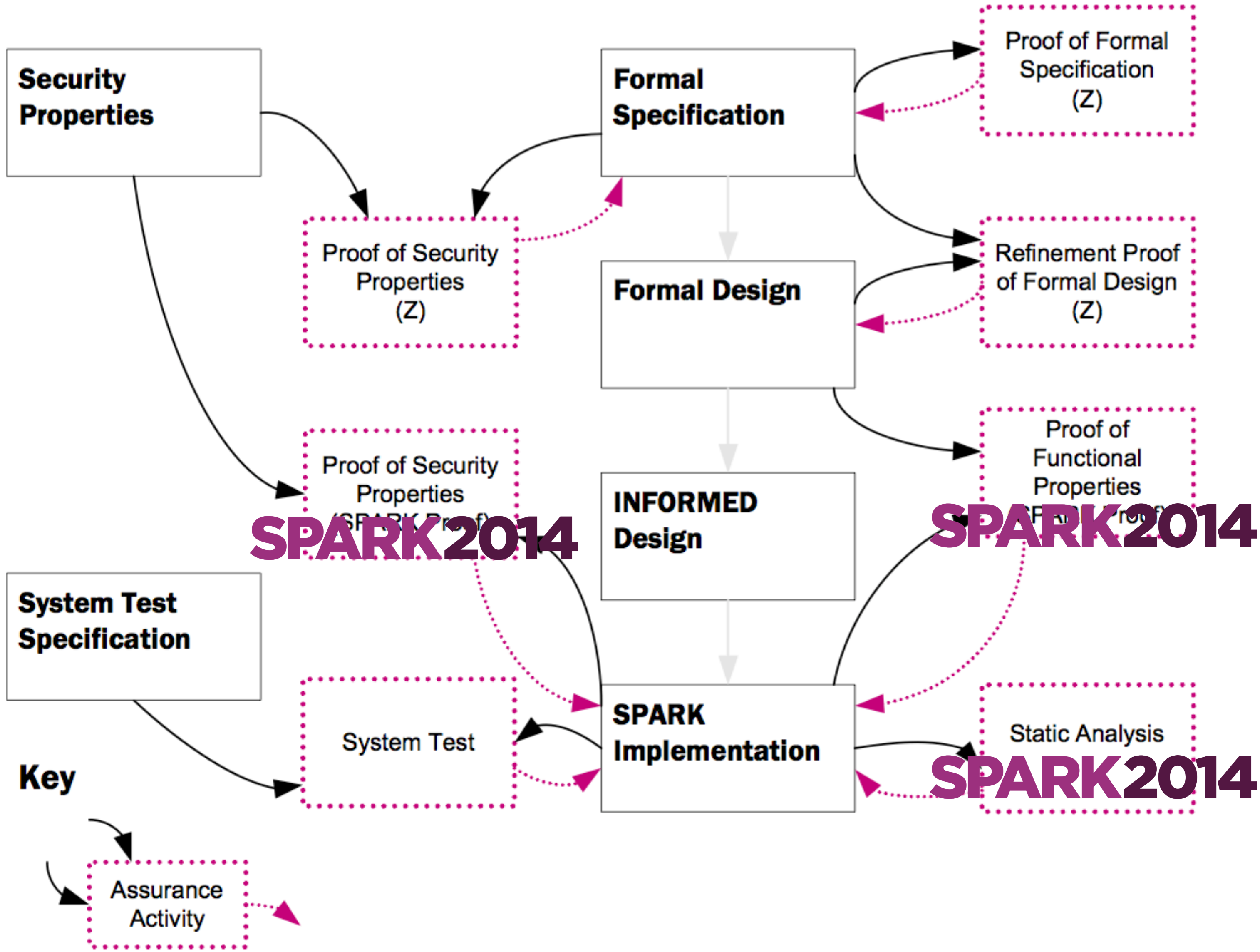
Areas requiring improvements:

- 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

Tokeneer



Security Properties
Formal Specification
Proof of Formal Specification (Z)
Proof of Security Properties (Z)
Formal Design
Refinement Proof of Formal Design (Z)
Proof of Security Properties
SPARK2014
INFORMED Design
Proof of Functional Properties
SPARK2014
System Test Specification
System Test
SPARK Implementation
Static Analysis
SPARK2014
Key
Assurance Activity

Aspect / Pragma	Num. of occurrences
Global	197
Refined_Global	71
Refined_Depends	40
Depends	202
Pre	28
Post	41
Assume	3
Loop_Invariant	10

Dataflow

Refinement

Information flow

Functional contracts

User guidance

Assumptions

Results

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)

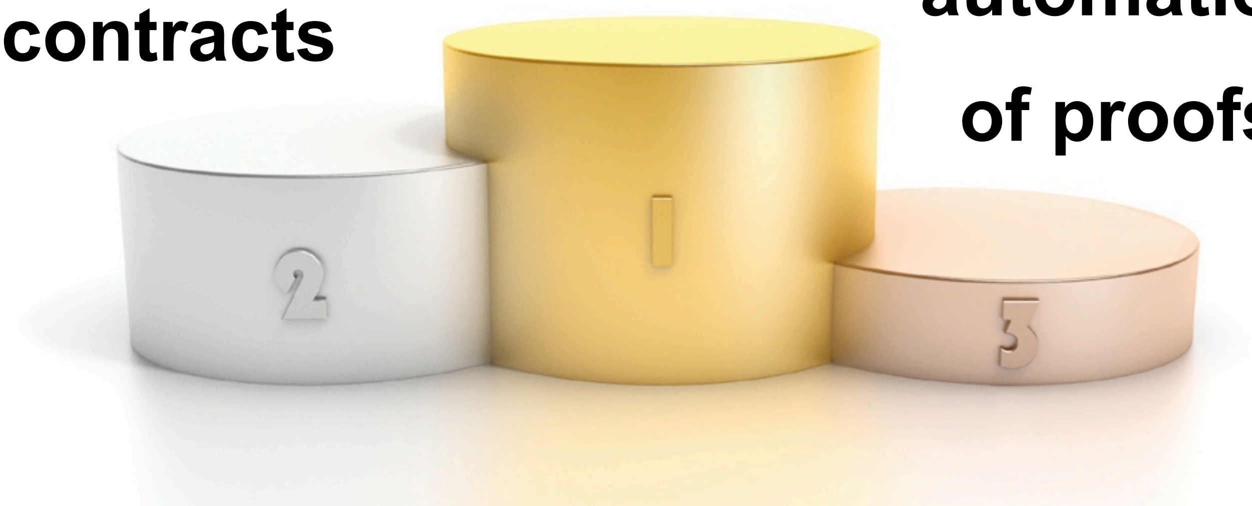
Lessons Learned

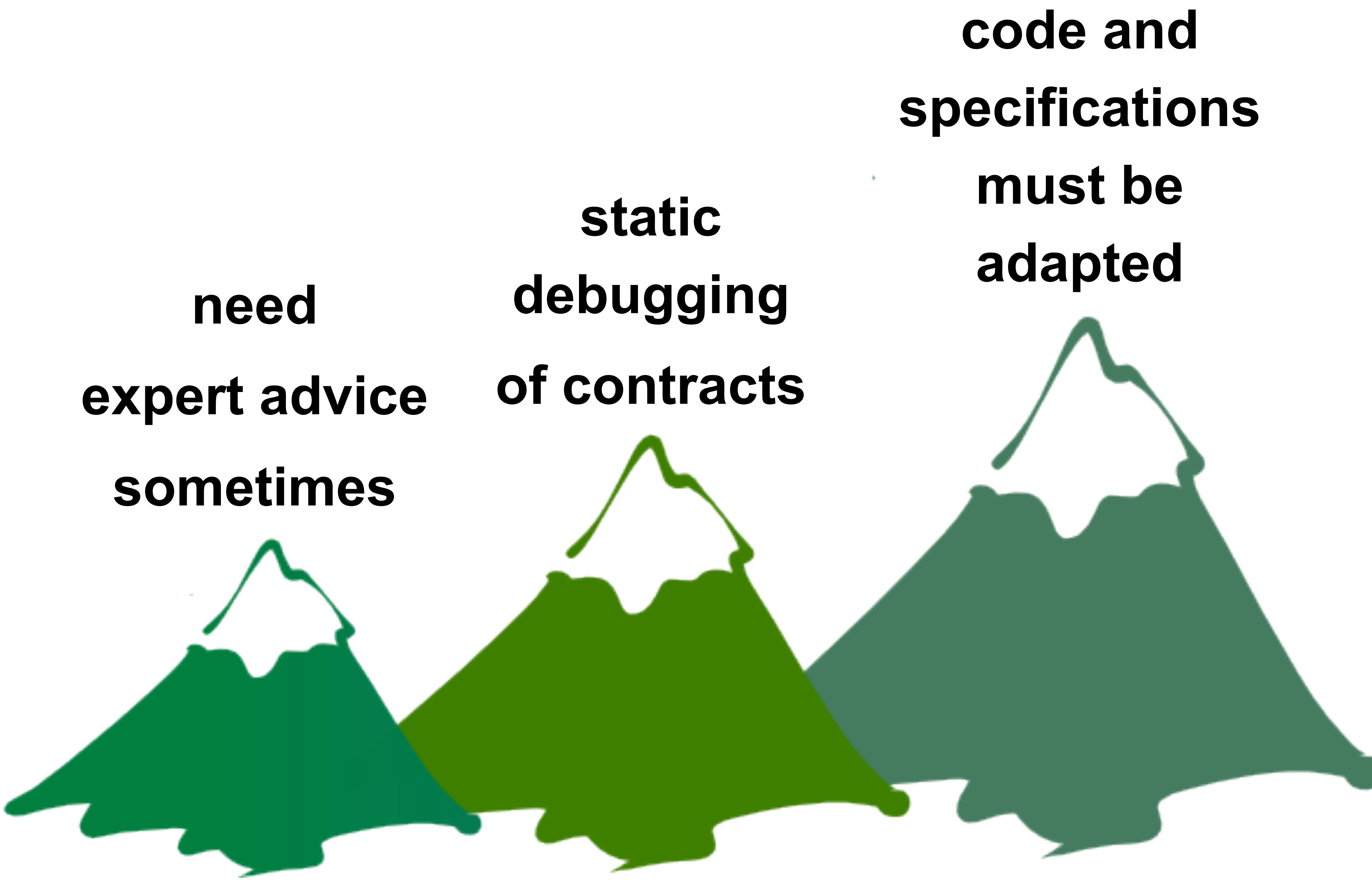
SPARK 2014 Strengths

**expressive
yet analyzable
language**

**executable
contracts**

**better
automation
of proofs**




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.spark-2014.org
- 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