SPARK 2014: Hybrid Verification using Proofs and Tests

José F. Ruiz
Senior Software Engineer

FOSDEM 2014 – February 1st, 2014

Outline

- **Language evolution**
  - From a safety point of view
- **SPARK 2014**
  - Why and what
- **Modular verification**
  - Dynamic validation
  - Formal proofs
- **Conclusions**
  - Including some results from case studies

Languages for Critical Software

"Ada, why not?"

Ada favors clarity over conciseness, reader over writer

C
116 characters

```c
char *strcpy(char *ret, const char *s2) {
  char *s1 = ret;
  while (*s1++ = *s2++)
    /*EMPTY*/ ;
  return ret;
}
```

Ada
208 characters

```ada
procedure Strcpy (Dest : out String; Src : in String) is
begin
   for J in 1 .. Integer'Min (Src'Length, Dest'Length) loop
      Dest (Dest'First + J - 1) := Src (Src'First + J - 1);
   end loop;
end Strcpy;
```

Ada favors clarity over conciseness, reader over writer

Means that “ret” is an output

Means that “s2” is an input

C

```c
char *strcpy(char *ret, const char *s2) {
  char *s1 = ret;
  while (*s1++ = *s2++)
    /*EMPTY*/ ;
  return ret;
}
```

At the same time an assignment and a test that the end of the string has been reached

Means that “Dest” is an output

Means that “Src” is an input

Ada

```ada
procedure Strcpy (Dest : out String; Src : in String) is
begin
   for J in 1 .. Integer'Min (Src'Length, Dest'Length) loop
      Dest (Dest'First + J - 1) := Src (Src'First + J - 1);
   end loop;
end Strcpy;
```

Loop

Assignment

“Ada, why not?”

Ada allows run-time detection of typing and memory errors

C

```c
char *strcpy(char *ret, const char *s2) {
  char *s1 = ret;
  while (*s1++ = *s2++)
    /*EMPTY*/ ;
  return ret;
}
```

Possible buffer overflow

Ada

```ada
procedure Strcpy (Dest : out String; Src : in String) is
begin
   for J in 1 .. Integer'Min (Src'Length, Dest'Length) loop
      Dest (Dest'First + J - 1) := Src (Src'First + J - 1);
   end loop;
end Strcpy;
```

Overflow detected at run-time

"SPARK, why not?"

SPARK allows static detection of typing and memory errors

C

```c
char *strcpy(char *ret, const char *s2) {
  char *s1 = ret;
  while (*s1++ = *s2++)
    /*EMPTY*/ ;
  return ret;
}
```

Possible buffer overflow

SPARK

```ada
procedure Strcpy (Dest : out String; Src : in String) is
begin
   for J in 1 .. Integer'Min (Src'Length, Dest'Length) loop
      Dest (Dest'First + J - 1) := Src (Src'First + J - 1);
   end loop;
end Strcpy;
```

Overflow detected statically

Advantages of Ada and SPARK over C

Buffer overflows in Ada?

- Easily avoided by programmers (array types carry their bounds)
- Automatically caught at run-time

Integer overflows in Ada?

- Easily avoided by programmers (using bounded integer types)
- Automatically caught at run-time

Buffer and integer overflows in SPARK?

- If present, automatically caught by analysis
- If absent, automatic proof that no such error can occur

Buffer overflows and integer overflows are still major sources of pain in C

Ada: programming language for long-lived embedded critical software

Ada version	Main features
Ada 83	modularity + genericity + type safety + tasking
Ada 95	+ object orientation + protected objects
Ada 2005	+ containers + interfaces
Ada 2012	+ contracts

SPARK: Ada subset for formal validation

SPARK version	Main features
SPARK 83/95/2005	contracts in comments mathematical semantics of contracts many language restrictions
SPARK 2014	contracts in the language executable semantics of contracts few language restrictions

Ada and SPARK Contracts

Contract: agreement between the client and the supplier of a service

Program contract: agreement between the caller and the callee subprograms

```
-- Set a variable with a given value
-- If the value is out of range, Is_Error is set to true
procedure Set_Protected_Nat32_Variable (Var       : in out T_Variable;
                                         New_Value :        T_Nat32;
                                         Is_Error  :    out Boolean)
with
  Global  => null,
  Depends => ((Var, Is_Error) => (Var, New_Value)),
  Pre     =>
    (Is_Valid (Var) and then
     Var.Data_Type = Nat32_Type),
  Post    =>
    (Is_Valid (Var) and then
     Var.Data_Type = Nat32_Type and then
     Get_Min_Nat32 (Var) = Get_Min_Nat32 (Var'Old) and then
     Get_Max_Nat32 (Var) = Get_Max_Nat32 (Var'Old));
```

Ada Type Invariants and Predicates

Type predicate: permanent property of objects

```
type T_Day is new Day with Static_Predicate => T_Day in Tuesday | Thursday;

type Message is record
   Sent     : Day;
   Received : Day;
end record with
  Dynamic_Predicate => Message.Sent <= Message.Received;
```

Type invariant: property of objects at public boundary

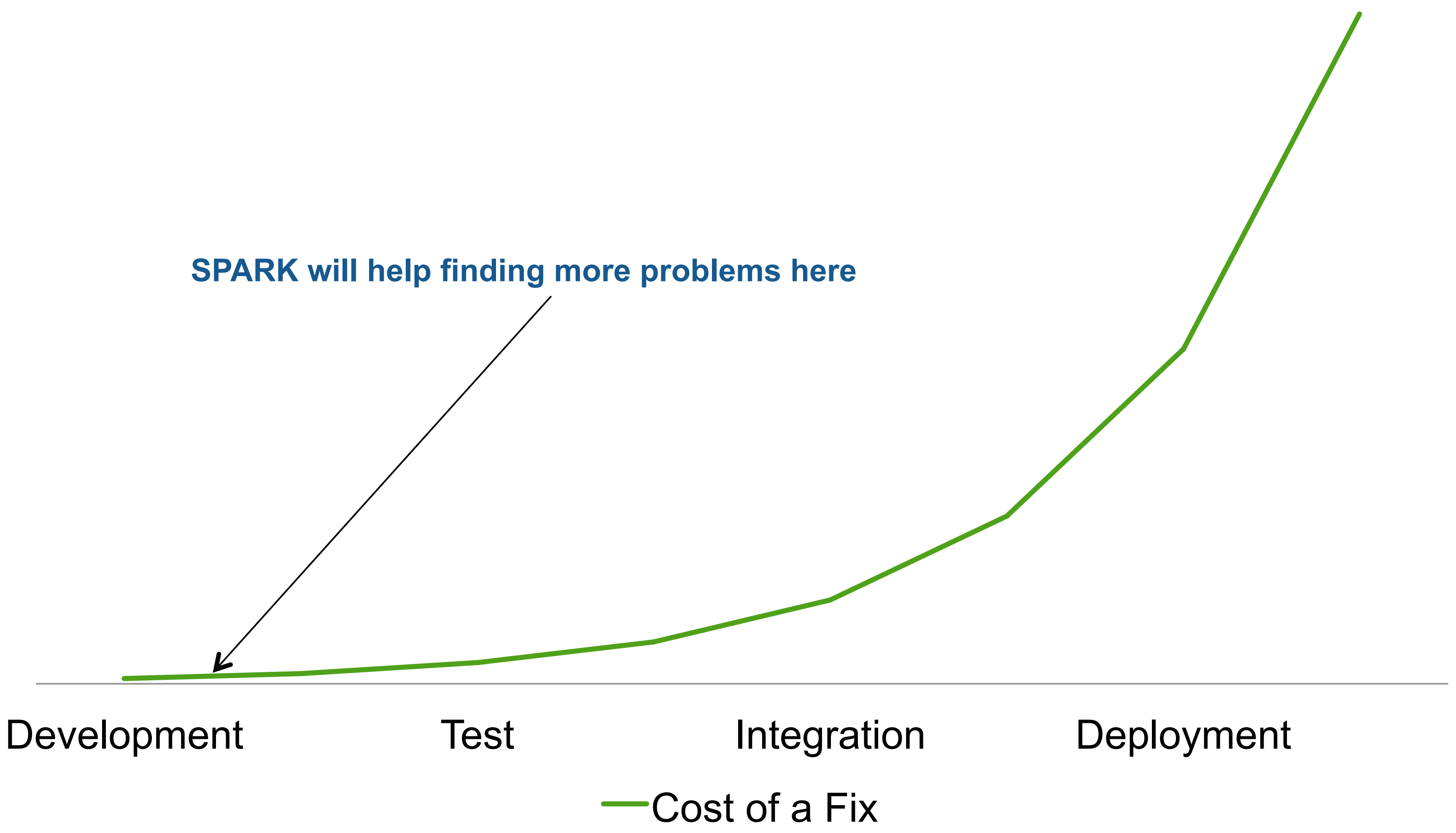
```
type Communication (Num : Positive) is record
   Msgs : Message_Arr (1 .. Num);
end record with
  Type_Invariant =>
    (for all Idx in 1 .. Communication.Num-1 =>
      Communication.Msgs(Idx).Received <= Communication.Msgs(Idx+1).Received);
```

SPARK 2014

What is SPARK 2014?

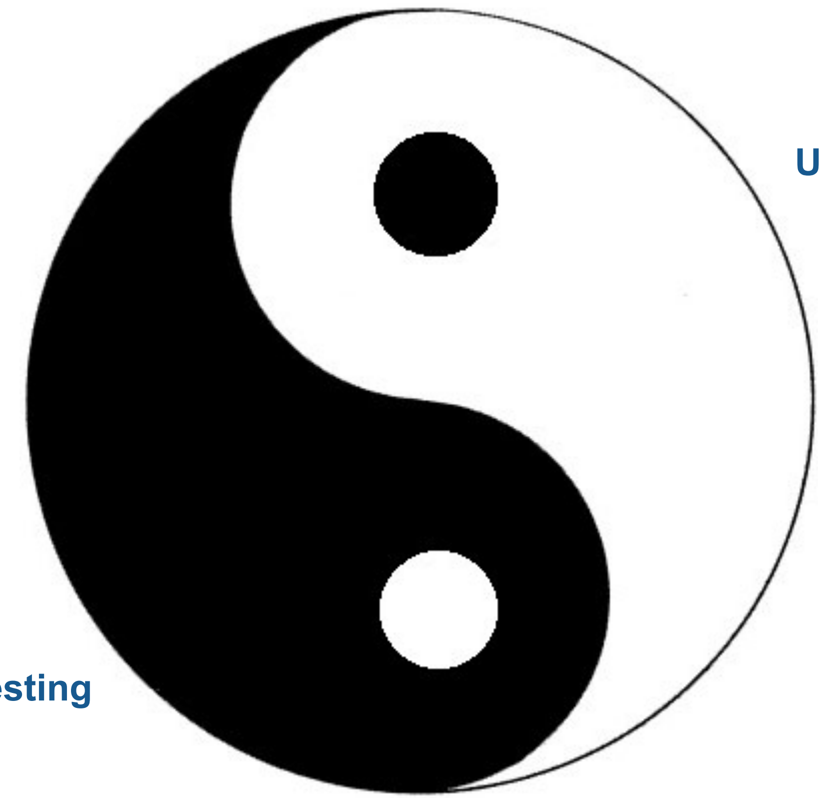
- **SPARK (Vintage) is a Ada-based language and toolset allowing to perform formal analysis**
  - Absence of run-time errors
  - Data Flow and Information Flow analysis
  - Correctness with regards to contracts
- **SPARK (Vintage) is based on a provable Ada 2005 subset and additional annotations**
- **The Ada 2012 language extends the Ada 95 / 2005 definition, including requirements from SPARK language**
- **SPARK 2014 is a new language and toolset based on Ada 2012 including**
  - A much wider support for the Ada constructions
  - A format for contract associated with formal and executable semantics
  - An environment ready for local type substitutability proof
  - An environment ready for low-level requirement compliance and robustness verification
  - An integration environment between unit testing and unit proof

Why SPARK 2014: … the usual story of finding problems earlier …



Why SPARK 2014: … but we can do better …

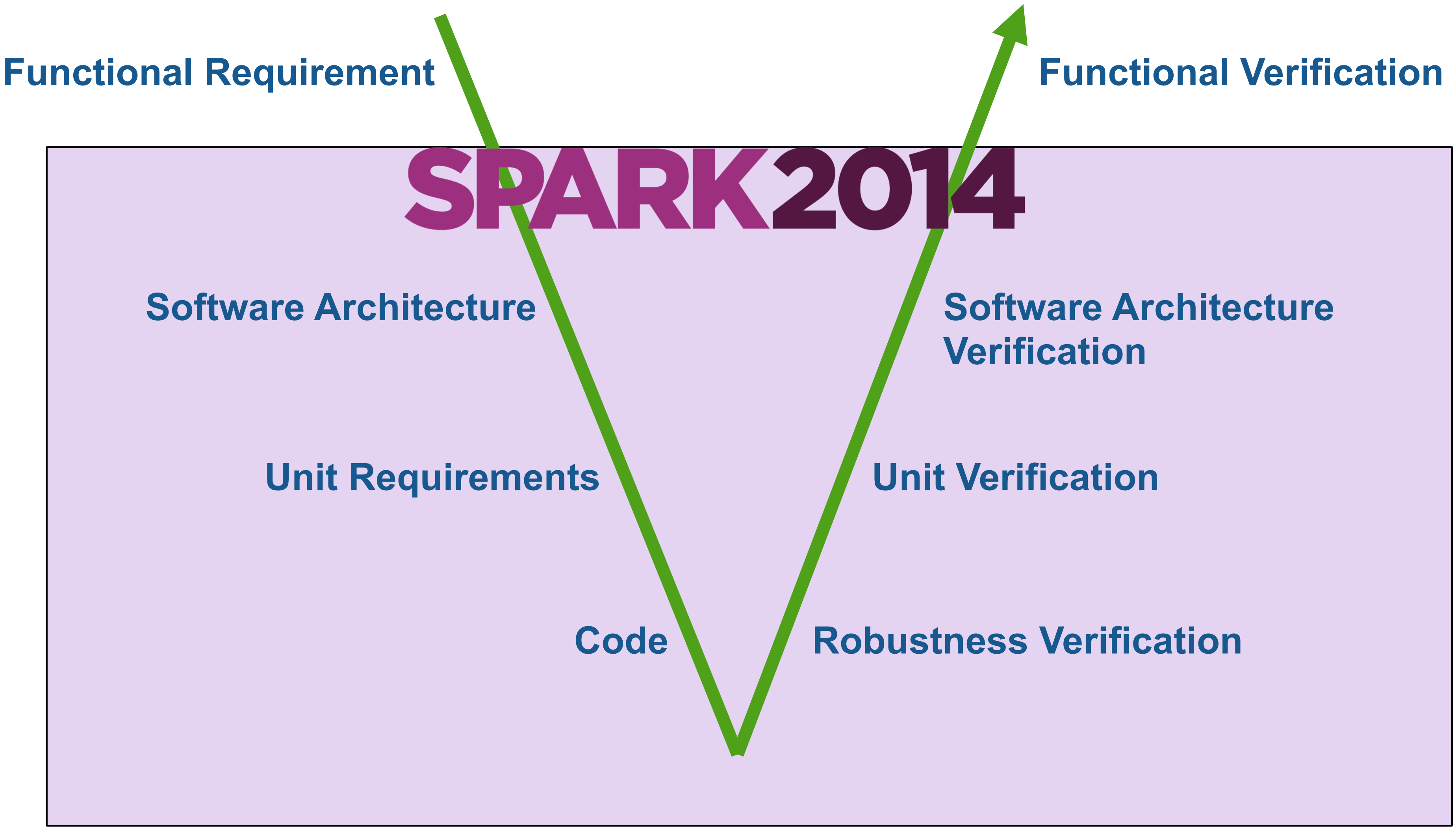
Unit Proof



Unit Testing

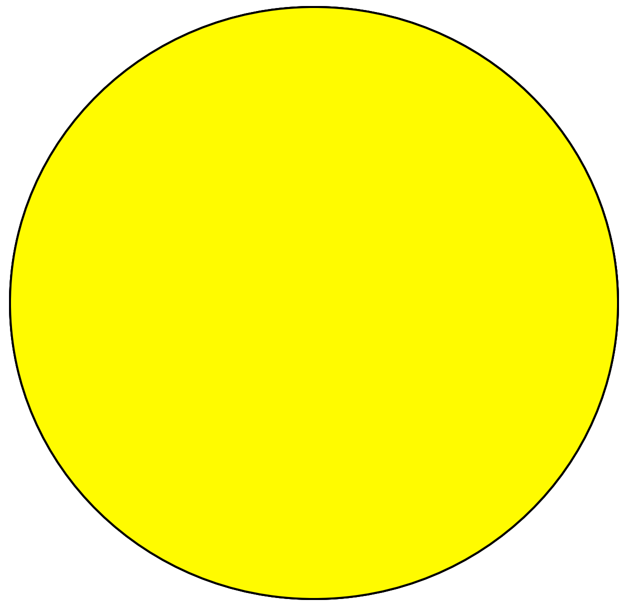
Unit Verification
(Modular Verification)

Why SPARK 2014: Value Proposition



Mixing Proof and Test

- Testing is expensive and inaccurate
- Proving is more accurate, but proving 100% is even more expensive…



- … especially the last 20%
- How about proving what’s easy to prove and test the rest?

Ada 2012 Additions for Contract Specification

- **Contracts on subprograms**

```
G : Integer;

procedure P (X, Y : Integer)
   with Pre => X + Y > 0,
        Post => G = G'Old + 1;
```

- **Contracts on types**

```
type Even is new Integer with
   Dynamic_Predicate => Even mod 2 = 0;
```

- **New expressions such as quantifiers**

```
type Sorted_Array is array (Integer range <>) of Integer
   with Dynamic_Predicate =>
      Sorted_Array'Size <= 1
      or else (for all I in Sorted_Array'First .. Sorted_Array'Last - 1 =>
                 Sorted_Array (I) <= Sorted_Array (I + 1));
```

- **All of these construction are provided with dynamic semantics by Ada 2012**

- **All of these construction are provided with proof semantics by SPARK 2014**

Extensions to Architecture Definition

- **Clarify access to global variables**
  - Aspects for subprograms

```
with Global => null;         -- Not reference to global items
with Global => V;            -- V is an input of the subprogram
with Global => (X, Y, Z);    -- X, Y and Z are inputs of the subprogram

with Global => (Input  => V); -- V is an input of the subprogram.
with Global => (Input  => (X, Y, Z)); -- X, Y and Z are inputs of the subprogram
with Global => (Output => (A, B, C)); -- A, B and C are outputs of the subprogram
with Global => (In_Out => (D, E, F)); -- D, E and F are both inputs and outputs of
                                      -- the subprogram

with Global => (Proof_In => (G, H)); -- G and H are only used in assertion
                                     -- expressions within the subprogram

with Global => (Input    => (X, Y, Z),
                Output   => (A, B, C),
                In_Out   => (P, Q, R),
                Proof_In => (T, U));
                -- A global aspect with all types of global specification
```

Extensions to Architecture Definition (II)

- **Clarify information flow**
  - Aspects for subprograms

```
procedure P (X, Y, Z : in Integer; A, B, C : in out Integer; D, E out Integer)
  with Depends => ((A, B) =>+ (A, X, Y),
                   C      =>+ null,
                   D      =>  Z,
                   E      =>  null);
-- The "+" sign attached to the arrow indicates self-dependency
-- The exit value of A depends on the entry value of A as well as the entry
-- values of X and Y.
-- Similarly, the exit value of B depends on the entry value of B as well as
--  the entry values of A, X and Y.
-- The exit value of C depends only on the entry value of C.
-- The exit value of D depends on the entry value of Z.
-- The exit value of E does not depend on any input value.
```

G is a global variable written
No other global variable is accessed or modified

```
G : Integer;

procedure P (X, Y : Integer)
   with Global  => (Out => G),
        Depends => ((G) => (X, Y));
```

The value of G is computed from the value of X and Y

Extensions to Language Definition

- **Excluded features not amenable to sound static verification**
  - Access types
  - Function side effects
  - Aliasing
    - Renaming is allowed
  - goto statements
  - Tagged types not yet supported
  - Exception handlers
    - Limited raise statements are allowed
  - Tasking not yet supported

- **Additions with respect to previous versions of SPARK**
  - Generic subprograms and packages
  - Discriminated types
  - Types with dynamic bounds
  - Array slicing
  - Array concatenation
  - Recursion
  - Early exit and return statements
  - Computed constants
  - A limited form of raise statements

Extensions to Language Definition (II)

- **Formal Container Library**
  - SPARK 2014 excludes data structures based on pointers
  - Vectors, Lists, Maps, Sets are being defined
    - Specifically designed to facilitate the proof of client units

Extensions to Contract Definition

- **Contract cases**
  - Enhance clarity and readability
  - Structured way of defining a subprogram contract using mutually exclusive subcontract cases

```
function Compute_Action (S : State) return Action
  with Post => (if S = Off then Compute_Action'Result = Activate)
                and then (if S = Normal then Compute_Action'Result = Nothing)
                and then (if S = Alarm then Compute_Action'Result = Evacuate);
```

Better this way

```
function Compute_Action (S : State) return Action
  with Contract_Cases => (S = Off    => Compute_Action'Result = Activate,
                          S = Normal => Compute_Action'Result = Nothing,
                          S = Alarm  => Compute_Action'Result = Evacuate);
```

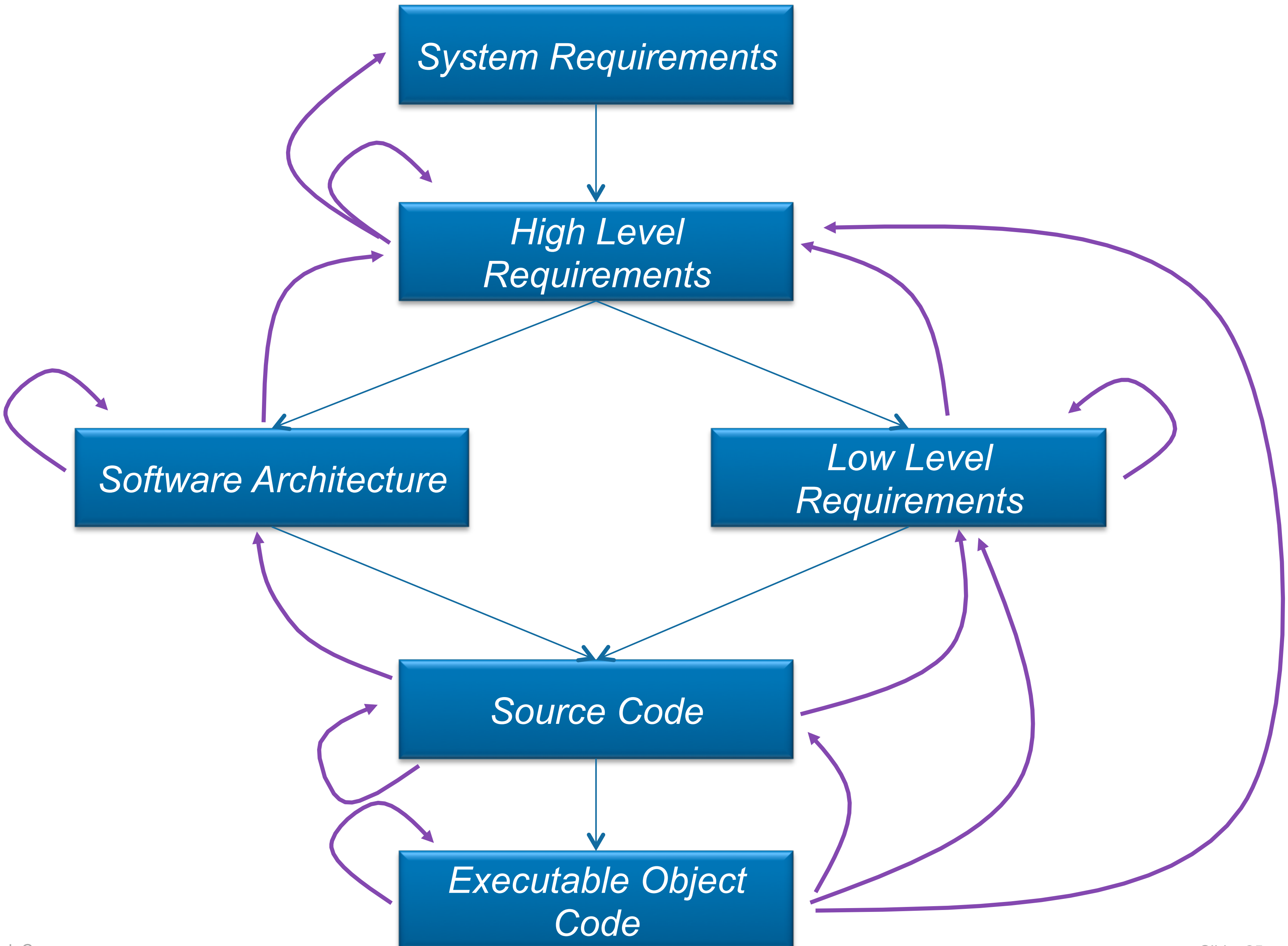
Extensions to Behavior Definition

Non-varying properties

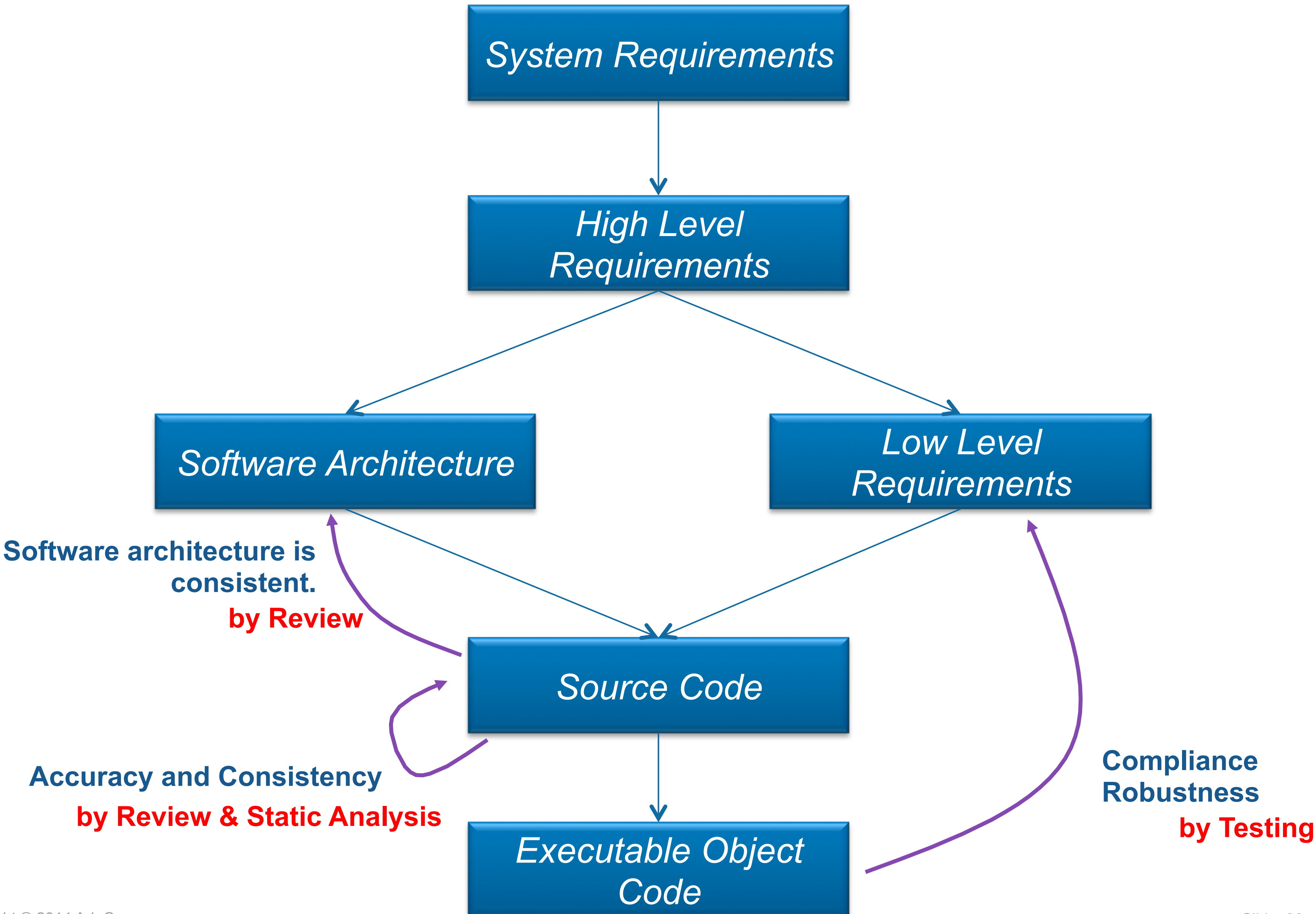
```
procedure Loop_Var_Loop_Invar is
   type Total is range 1 .. 100;
   subtype T is Total range 1 .. 10;
   I : T := 1;
   R : Total := 100;
begin
   while I < 10 loop
      pragma Loop_Invariant (R >= 100 - 10 * I);
      pragma Loop_Variant (Increases => I,
                           Decreases => R);
      R := R - I;
      I := I + 1;
   end loop;
end Loop_Var_Loop_Invar;
```

Used to demonstrate that a loop will terminate by specifying expressions that will increase or decrease as the loop is executed

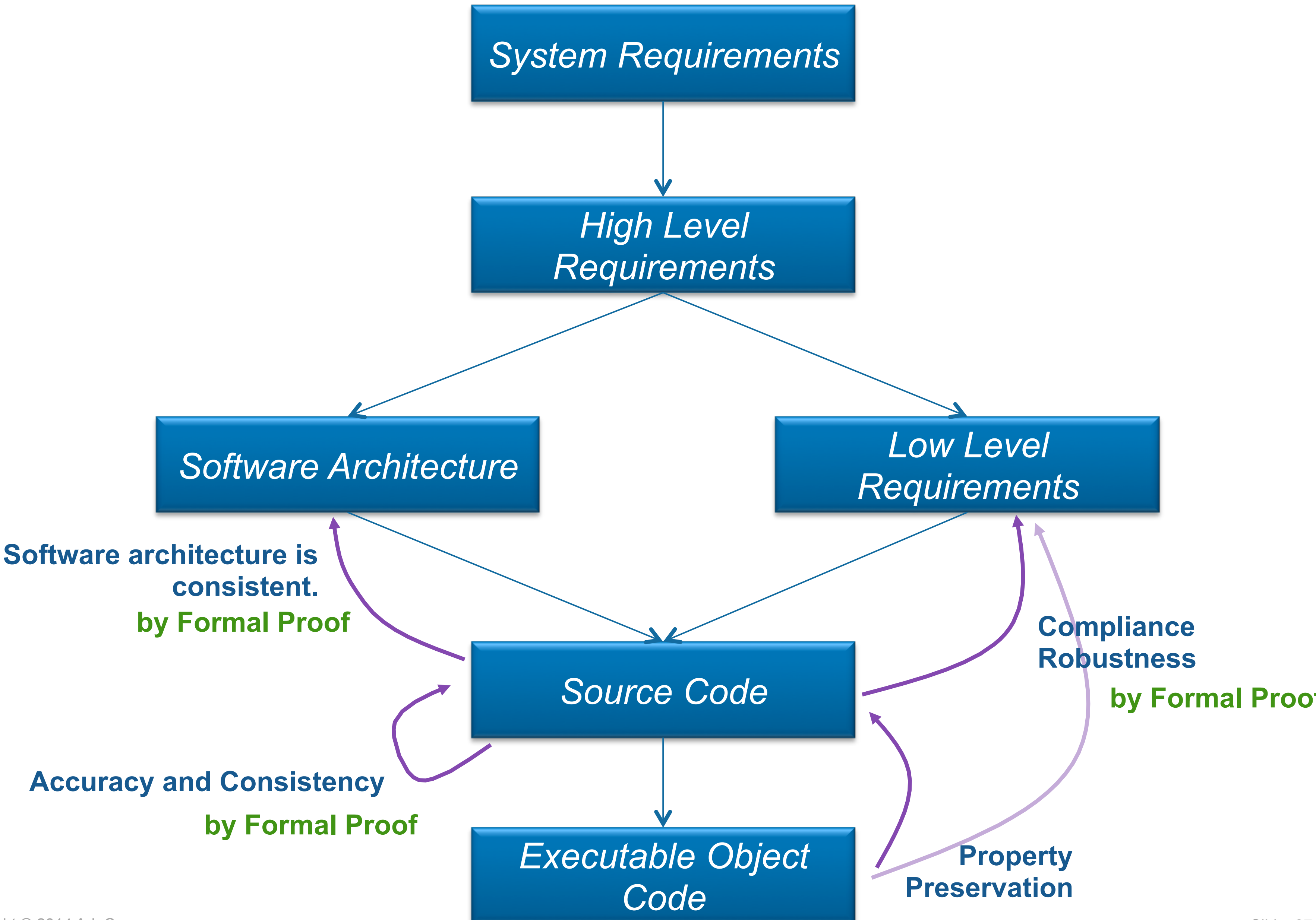
DO-178C Verification Activities



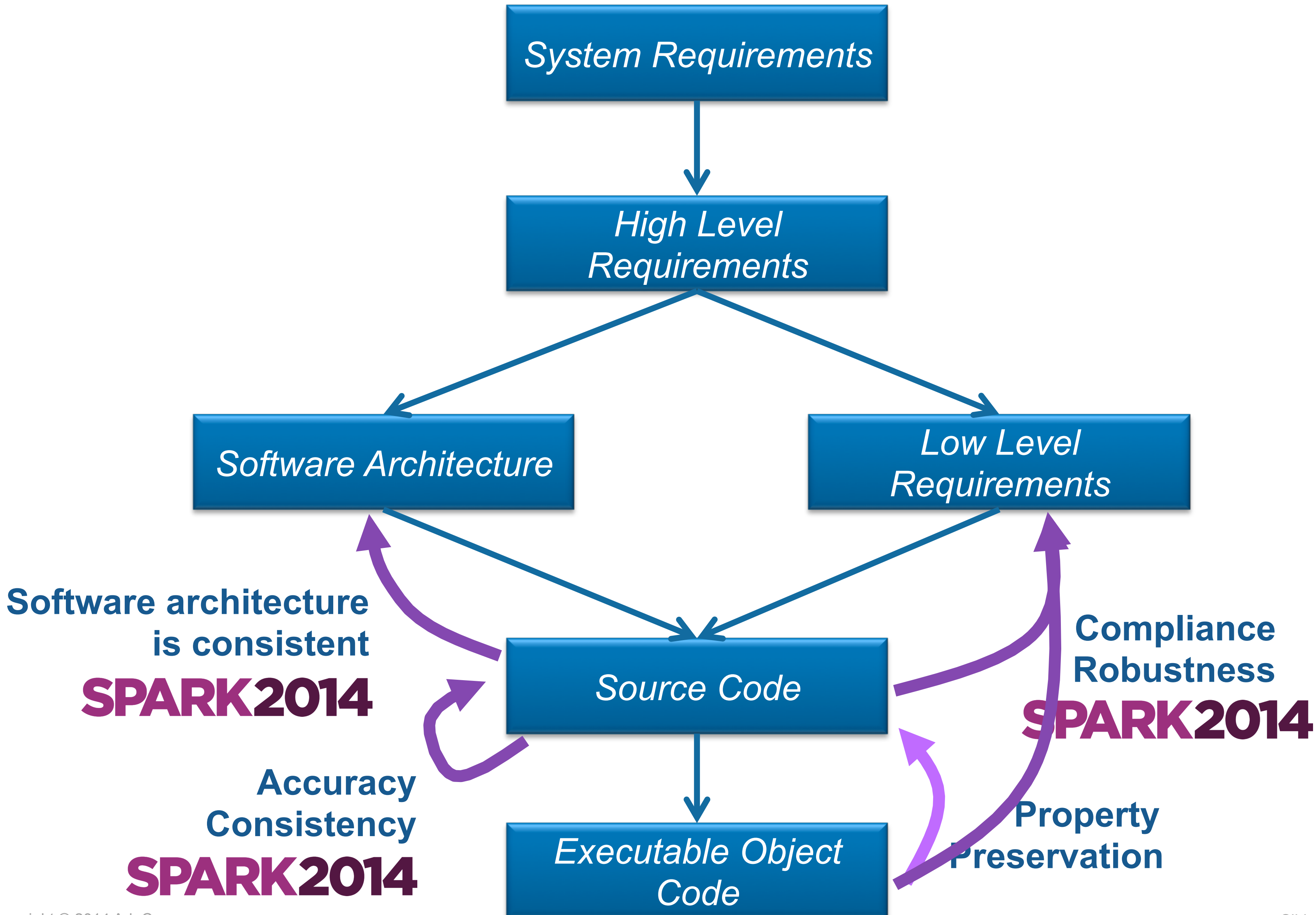
What we want to address



How we want to do it



SPARK 2014 Value Proposition (DO-178C Version)

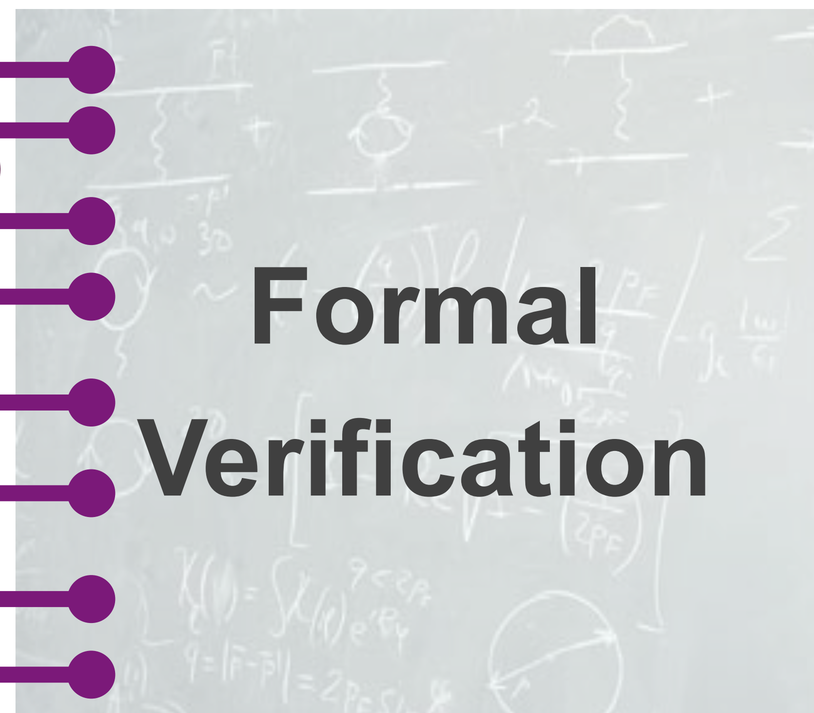


Program

Contract = agreement between ~~client & supplier~~
caller & callee

Dynamic Verification

SPARK 2014



Formal Verification

Dynamic Validation

Contracts = Assertions

Precondition: assertion checked at subprogram entry

- Replaces defensive coding

Postcondition: assertion checked at subprogram return

- Replaces equivalent assertion at all “return” points
- Makes it easy to mention pre-call values X’Old
- In a function, makes it easy to mention function result F’Result

At run time:

- Assertion failure → exception Assertion_Error is raised
- X’Old → copy of X made at subprogram entry

Fine-grain control of enabled assertions:

- Compiler switch
- Pragma Assertion_Policy in code, for each kind of assertion

Contracts by Cases

Many subprograms are more easily specified by cases that are:

- Disjoint: two different cases should not be enabled at the same time
- Complete: cases should cover all possibilities

→ Additional properties also checked at run time

```
-- Update the status of all events following time passing
procedure Check_Monitoring (Event        : in out T_Event;
                            Events       :        T_Events;
                            Current_Time :        Mvm.Obit.T_Obit)
with
  Pre  => Is_Valid (Event => Event),
  Post => Is_Valid (Event => Event),
  Contract_Cases =>
    ((Event.Event_Status in Inactive | Detected | Failed_To_Be_Detected) =>
       (Event = Event'Old),
     (Event.Event_Status in Not_Yet_Detected)                            =>
       (Event.Event_Status = Not_Yet_Detected or else
        Event.Event_Status = Detected or else
        Event.Event_Status = Failed_To_Be_Detected));
```

Use Cases for Checking Contracts at Run Time

1. **During design and specification**
   – Express dependencies and constraints on unit specs

2. **During development**
   – More precise documentation of intent than comments
   – Provides quick initial feedback
   – Assertion failure can be analyzed in debugger

3. **During testing**
   – Provides oracles for unit testing
   – Can be reused for integration testing
   – Can sometimes achieve exhaustive verification (using “for all”)

4. **In production code**
   – Preconditions can replace defensive coding

Formal Validation

What can you verify formally?

- **Data-flow analysis**
  - Initialization of variables
  - Data dependencies of subprograms
    - Parameters and variables read or written
- **Information-flow analysis**
  - Coupling between the inputs and outputs of a subprogram
    - Which input values of parameters and variables influence which output values
- **Robustness analysis**
  - Predefined checks will never fail at run time
- **Functional analysis**
  - Contracts expressed as preconditions, postconditions, type invariants, …

Flow analysis: Correct Access to Data

Correct access to global variables

- Abstract_State contract on packages specs specifies hidden state
- Global contract on subprograms specifies modes of global variables accessed
- Depends contract on subprograms specifies flow of information

Correct access to initialized data

- no reads of uninitialized data
- in parameters and Input globals must be fully initialized on subprogram entry
- out parameters and Output globals must be fully initialized on subprogram exit

Fast static analysis (≈ compilation time) checks correct access

Also detects unused parameters, globals, assignments, statements

Flow analysis: Correct Access to Global Variables

Global state of a program is made up of:

- Visible global variables
- Hidden global variables (in private parts and bodies)

Abstract_State contract on packages specs specifies hidden state

Global contract on subprograms specifies modes of global variables accessed

Depends contract on subprograms specifies flow of information

Fast static analysis (≈ compilation time) checks correct access

Typical errors / warnings:

```
file.adb:20:09: "G" must be a global output of ”T" [illegal_update]
f.adb:18:3: warning: missing dependency "Y => Pa" [depends_missing]
```

Flow analysis: Correct Access to Initialized Data

Modes of parameters and globals have a strong semantics in SPARK:

- in parameters and Input globals must be fully initialized on subprogram entry
- out parameters and Output globals must be fully initialized on subprogram exit

Packages declare initialized state in Initializes contract

Fast static analysis (≈ compilation time) checks correct initialization

Typical errors / warnings:

```
file.adb:103:13: ”Rec.Arr" is not initialized [uninitialized]
file.adb:8:4: warning: "X" might not be initialized [uninitialized]
```

Also detects unused parameters, globals, assignments, statements

Proof of Properties

Properties of interest:

- Absence of run-time errors
- Compliance with Low-Level Requirements formalized as subprogram contracts

(See Airbus use of Unit Proof:
http://www.open-do.org/wp-content/uploads/2013/04/IEEE_Software_Formal_Or_Testing.pdf)

Property = mathematical formula

Proof is done subprogram by subprogram

Run-time checks in contracts are also proved

Robustness analysis: Absence of Run-Time Errors

Run-time check = mathematical formula

Ex: absence of division by zero in expression “X / Y” if formula “Y /= 0” holds

Checks in SPARK:

- Range check (bounds of scalar types)
- Index check (bounds of arrays)
- Overflow check (bounds of machine scalar types)
- Division by zero check
- Length check (array operations)
- Discriminant check (access to discriminant record)

Proof is done subprogram by subprogram

Run-time checks in contracts are also proved

Functional analysis: Compliance with Low-Level Requirements

Low-level requirement can be formalized as subprogram contract

(See Airbus use of Unit Proof:
http://www.open-do.org/wp-content/uploads/2013/04/IEEE_Software_Formal_Or_Testing.pdf)

Assertion = mathematical formula

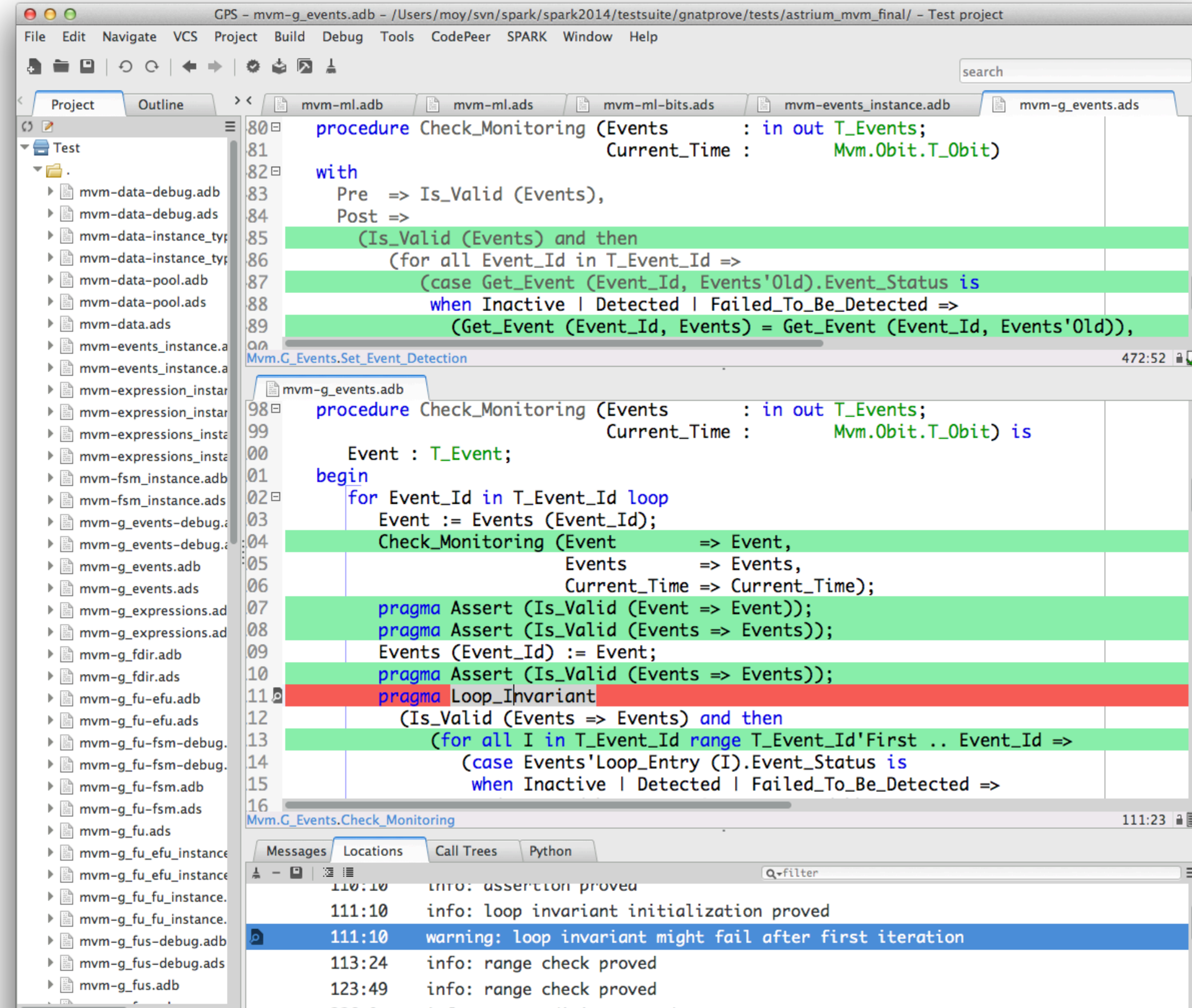
Proof is done subprogram by subprogram:

- Precondition is assumed
- Precondition of each call is proved
- Postcondition of each call is assumed
- Postcondition is proved

Additional assertions may be needed for proof:

- Pragma Loop_Invariant for summarizing loop effects
- Pragma Loop_Variant to prove loop termination
- Pragma Assert to guide automatic prover

Integration of Static Validation in Developer Workflow



Space Case Study

Case Study by Astrium Space Transportation (David Lesens)

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

Conclusions

Main Strengths

- **SPARK 2014 provides one of the highest level of safety/security/reliability on the market**
  - Implementing strategies used by industrials
  - Inheriting from 20+ years of industrialization and usage of SPARK
  - Fit for safety and security standard

- **SPARK 2014 allows to progressively introduce higher reliability standards**
  - Offers a unique formalism for executable and formal notations
  - Can be deployed at the subprogram level

- **SPARK 2014 reduces unit testing costs as well as integration work and deployment hazards**
  - Unit test are replaced by unit proof verifying 100% of the cases

Case Study Conclusions

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 the development roadmap for 2014)
- Helping user with unproved checks (for example counter-examples, in roadmap)

SPARK 2014 is the only language and toolset providing industrial support for both dynamic and formal contract-based validation of software.

Now available as beta

First release April 2014

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