AdaCore

A Practical Introduction to Formal Development and Verification of High-Assurance Software with SPARK

Ben Brosgol
brosgol@adacore.com
Senior Technical Staff

IEEE Secure Development Conferenc
Hilton McLean Tysons Corner
McLean, VA

September 25, 2019, 3:30 - 5:00pm

SPARK Demo tomorrow-
Pat Rogers
rogers@adacore.com
Senior Technical Staff

Outline

- **Part 1: SPARK Technology**
  - Formal methods
  - SPARK language summary
  - Examples
  - Restrictions for deterministic behavior
  - Flow analysis
  - Behavioral analysis
  - Other features

Quiz 1

Quiz 2

- **Part 2: SPARK Pragmatics**
  - Incremental adoption: levels of assurance
  - Hybrid verification: combining proofs with testing
  - Industrial experience
  - SPARK and Frama-C
  - Conclusions

Formal Methods

- **What is a formal method?**
  - A technology for statically analyzing a program's source code and verifying specified dynamic properties with mathematics-based rigor
- **Not new**
  - Subject of research since early days of programming
- **Many benefits compared with traditional techniques (testing)**
- **But largely confined to academia/research, because of practical challenges**
- **Times are changing; formal methods are being adopted in critical industries**
- **This tutorial uses SPARK as example of a formal methods technology that addresses the challenges**
  - In principle, other approaches (e.g., Frama-C) could be used
  - SPARK has the benefit of a more secure foundation

Brief History

- **Originated at Southampton University (UK) in 1970s**
  - Based on work at Royal Signal and Radar Establishment
  - **SPADE** = **S**outhampton **P**rogram **A**nalysis **D**evelopment **E**nvironment
  - Pascal subset
- **Alternative foundation languages considered**
  - Modula-2
  - Ada 83
  - C

  } **Ada was selected**
- **SPARK = SPADE Ada Kernel (1988)**
  - Possible interpretation of "R" ⇨ "Ratiocinate" (to reason)
  - Toolset acquired/commercialized by Praxis Critical Systems (now Altran)
  - Language and toolset have undergone several revisions
- **SPARK technology currently maintained by AdaCore and Altran**

SPARK Language Summary

- **Design principles**
  - Eliminate sources of ambiguity / implementation dependence
  - Exclude language features difficult to verify
  - Add aspects that facilitate formal analysis
  - Allow "mix and match" of SPARK and non-SPARK code in same program
- **Included Ada features**
  - Program structure (packages, subprograms, generic templates)
  - Most data types, including “safe pointers”
  - Object-Oriented Programming
  - Contract-based programming
    - Pre- and postconditions
    - Scalar ranges, type invariants, type/subtype predicates
  - Ravenscar tasking profile
- **Excluded Ada features**
  - Access types (pointers) in general
  - Side effects in functions / expressions
  - Problematic “aliasing” of names
  - **goto** statement
  - Exception handling
  - Most tasking features

Digression: Arrays in Ada (and SPARK)

- **Ada array types may be "unconstrained"**
  - Different objects of the same type may have different bounds
  - Lower and upper bounds may be run-time computed
  - For any one-dimensional array object X:
    - X'First is the lower bound
    - X'Last is the upper bound
    - X'Length is the number of elements in the array
    - X'Range means X'First .. X'Last
  - Index check on X(J) requires J to be in X'Range
- **Example: the predefined type String is a one-dimensional array of Character values**
  - No concept of nul termination

```
S : String(11..15) := "ABCDE";
-- S'First is 11, S'Last is 15, S'Length is 5
…
for I in S'Range loop
   S(I) := 'Z';
end loop;
-- Now S = "ZZZZZ"
```

SPARK Example (1)

- **A variety of program properties can be specified via *contracts***

```
Char : Character := … ;

procedure Replace( S     : in out String;
                   Index : in Integer )
   with
        Global  => (Input => Char),
        Depends => (S => (S, Char, Index)),
        Pre     => S'Length > 0 and then
                   Index in S'Range,
        Post    => S(Index) = Char
   is
begin
   S(Index) := Char;
end Replace;
```

data dependencies ✓
flow dependencies ✓
functionality ✓
absence of run-time errors ✓

  - *Global:* the only access to non-local data is to read the value of Char
  - *Depends:* the value of S on return depends on its value on entry and the values of Char and Index, and S depends only on these values
  - *Pre:* Boolean condition that subprogram assumes on entry
  - *Post:* Boolean condition that is guaranteed at subprogram return

SPARK Example (2)

- **SPARK* can prove all the contracts in the body of Replace**
  - Global and Depends contracts are complete and correct
  - No run-time errors during execution of Replace body
  - If precondition is met, then the postcondition is met when procedure returns
- **SPARK will attempt to prove precondition at each point of call**

```
procedure Example is
   Char : Character := 'Z' ;
   procedure Replace( S     : in out String;
                      Index : in Integer ) with … is …;
   My_String : String(1..5) := "ABCDE";
   N         : Integer := 2;
begin
   Replace (My_String, N);  -- Precondition proved
   N := 10;
   Replace (My_String, N);  -- Problem detected
end Example;
```

  - Can prove precondition met at 1st invocation of Replace but not at 2nd

```
medium: precondition might fail, cannot prove Index in S'range (e.g. when N = 10)
```

*** In these slides the reference to SPARK as the proof tool means the *GNATprove* tool, which is available in AdaCore's GNAT Community Edition of the Ada technology and in the commercial SPARK Pro product**

SPARK Example (3)

- **If the postcondition is proved, SPARK assumes it holds after the procedure returns**

```
procedure Example is
   Char : Character := 'Z' ;
   procedure Replace( S     : in out String;
                      Index : in Integer ) with … is …;
   My_String : String(1..5) := "ABCDE";
   N         : Integer := 2;
begin
   Replace (My_String, N);                -- Precondition proved
   pragma Assert (My_String(2) = 'Z');    -- Proved
   pragma Assert (My_String = "AZCDE"); -- Not proved
end Example;
```

- **SPARK can prove 1st Assert but not the 2nd**

```
medium: assertion might fail, cannot prove My_String = "AZCDE"
(e.g. when My_String = (1 => '@', 2 => 'Z', others => 'A'))
```

  - SPARK only assumes what is stated in the postcondition
  - Strengthen the postcondition if needed (and if the stronger postcondition can be proved)

SPARK Example (4)

- **A stronger postcondition more completely expresses functionality**

```
procedure Example is
  Char : Character := 'Z' ;
  procedure Replace( S : in out String; Index : in Integer )
     with Pre     => S'Length > 0 and then Index in S'Range,
          Post    => S = S'Old'Update(Index => Char)
     is
  begin
   S(Index) := Char;
  end Replace;
  My_String : String(1..5) := "ABCDE";
  N         : Integer := 2;
begin
   Replace (My_String, N);                -- Precondition proved
   pragma Assert (My_String(2) = 'Z');    -- Proved
   pragma Assert (My_String = "AZCDE");   -- Proved
end Example;
```

SPARK can prove the postcondition based on the precondition and the procedure body

  - `S'Old` is a copy of the String's value at procedure entry
  - `S'Old'Update( N => Value)` is the old value of S except that the element at position N is now Value
  - Global and Depends contracts omitted for simplicity (discussed below)

SPARK Example (5)

- In practice Replace would be declared in a package and thus have a specification (the "what") separate from its implementation (the "how")

Specification

```
package String_Utilities is
   Char : Character := 'Z';

   procedure Replace( S     : in out String;
                      Index : in Integer )
     with Pre     => S'Length > 0 and then
                     Index in S'Range,
          Post    => S = S'Old'Update(Index => Char);
   …
end String_Utilities;
```

```
with String_Utilities;
procedure Example is
   My_String : String(1..5) := "ABCDE";
begin
   String_Utilities.Replace(My_String, 2);

   pragma Assert (My_String = "AZCDE");
end Example;
```

Implementation

```
package body String_Utilities is
   procedure Replace( S     : in out String;
                      Index : in Integer ) is
   begin
      S(Index) := Char;
   end Replace;

   …
end String_Utilities;
```

The package facility is fundamental in Ada and SPARK

- **Modularization**
- **Encapsulation**

- But what about encapsulation?
  - Char should not be visible to "clients" of the package
  - SPARK approach: "state abstraction"

SPARK Example (6): What Was Illustrated

- **SPARK allows you to define invariant program properties, which it will attempt to verify statically**
  - Several are part of a subprogram's specification
    - Flow contracts
      - Global
      - Depends
    - Proof contracts
      - Pre
      - Post
  - Other behavioral claims are part of a subprogram's implementation
    - pragma Assert (*Boolean-expression*)
    - Others to be shown later
- **SPARK attempts to prove:**
  - Flow contracts, if present, are correct and complete
  - No possibility of run-time errors (e.g., index out of range)
  - When a subprogram is called, its precondition is met
  - If a subprogram's precondition is met, then the postcondition is met when the subprogram returns
- **SPARK assumes a subprogram's postcondition holds after the subprogram returns**

Another SPARK Example

```
package Array_Handling is
   type Row is array (1..100) of Float;
   function Max (R : Row) return Float
     with Global  => null,
          Depends => (Max'Result => R),
          Post    => (for all  X of R => Max'Result >= X) and
                     (for some X of R => Max'Result =  X);
end Array_Handling;
```

Quantified expressions are an Ada 2012 feature, not unique to SPARK

```
package body Array_Handling is
  function Max (R : Row) return Float is
    Current_Max : Float := R(1);
  begin
    for I in 2..100 loop
      pragma Loop_Invariant( for all J in 1 .. I-1 =>
                               Current_Max >= R(J) );
      pragma Loop_Invariant( for some J in 1 .. I-1 =>
                               Current_Max = R(J) );
      if Current_Max < R(I) then
         Current_Max := R(I);
      end if;
    end loop;
    return Current_Max;
   end Max;
end Array_Handling;
```

SPARK can prove the Global and Depends contracts, but needs additional guidance to prove the postcondition

- Loop invariant

SPARK checks pragma Loop_Invariant

- True at the 1st iteration
- If true on the (K-1)st iteration then also true on the Kth iteration

SPARK can prove absence of run-time errors and, given the pragmas, can prove the postcondition

- In a postcondition for function F, F'Result is the value being returned

Note: in the body of Max the indexing uses uses specific values (1, 2, 100) for simplicity. In practice these would be expressed as R'First, R'First+1, R'Last for robustness.

SPARK Language Pragmatics

- **Behavioral claims ("assertions") can be checked dynamically (Ada semantics) or statically (SPARK proof tool)**
  - Assertion_Policy pragma dictates whether code generated
    - `pragma Assertion_Policy (Check);`
    - `pragma Assertion_Policy (Ignore);`
  - If no Assertion_Policy pragma, the policy is implementation defined
    - For GNAT (AdaCore) the default is Ignore
  - Affects Pre and Post contracts, pragmas Assert and Loop_Invariant, and some other constructs
  - Assertion_Policy can be applied selectively to specific kinds of checks
- **Workflow**
    - Do initial development with checks enabled
    - Analyze the code with SPARK to prove the assertions, and disable checks for proved assertions (or unproved assertions otherwise verified)
- **Parts of the program can be subject to SPARK analysis while other parts are ignored by the SPARK tools**
  - Apply pragma SPARK_Mode to those units / code regions you want to analyze

SPARK Restriction: No Side Effects in Functions

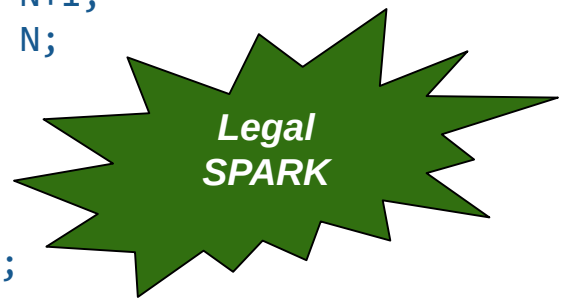
- **Ada example**
  - Ada does not define the order of evaluation of operands in an arithmetic expression
  - Result depends on whether the left or the right operand is evaluated first
    - May be either -1 or 1
- **SPARK prevents this implementation dependence**
  - Functions are not allowed to assign to non-local variables (directly or indirectly), or to take **out** or **in out** parameters
  - Express instead as a procedure with an **out** parameter
  - Compiler's choice of evaluation order doesn't matter

```
procedure Proc is
   N : Integer := 0;
   M : Integer;

   function Glorp return Integer is
   begin
      N := N+1;
      return N;
   end Glorp;
begin
   M := Glorp - Glorp; -- ???
end Proc;
```


```
procedure Proc is
   N : Integer := 0;
   M1, M2 : Integer;

   procedure Glorp (Result : out Integer) is
   begin
      N      := N+1;
      Result := N;
   end Glorp;
begin
   Glorp(M1);
   Glorp(M2);
   M := M1 - M2;
end Proc;
```



SPARK Restriction: No “Bad Aliasing”

- **Aliasing: referring to same object via different names**
- **Ada example**
  - Suppose a variable is passed as a parameter and referenced as a global
  - Effect may depend on whether the parameter is passed by copy or by reference
    - Displays (1.0, -1.0) if by reference, (0.0, 0.0) if by copy

```
procedure Bad_Alias is
   ZG : Complex := (0.0, 0.0);
   procedure Glorp (ZF : out Complex) is
   begin
      ZF := (1.0, -1.0);
      Display(ZG);  --???
   end Glorp;
begin
  Glorp(ZG);
end Bad_Alias;
```

Illegal SPARK

- **SPARK prevents this implementation dependence**
  - A global variable cannot be passed as an **out** or **in out** parameter to a procedure that accesses the variable
  - Other types of "bad aliasing" are likewise prohibited
  - Rewrite to avoid the aliasing
  - Compiler’s choice of by copy or by reference doesn’t matter

```
procedure No_Alias is
   ZG   : Complex := (0.0, 0.0);
   Temp : Complex;
   procedure Glorp (ZF : out Complex) is
   begin
      ZF := (1.0, -1.0);
      Display(ZG);  -- (0.0, 0.0)
   end Glorp;
begin
  Glorp(Temp);
  ZG := Temp;
end No_Alias;
```

Legal SPARK

Flow Analysis

- **Functionality**
  - Data-flow analysis
    - Keep track of the variables used by a subprogram
  - Information-flow analysis
    - Analyze coupling between a subprogram's inputs and its outputs
- **Errors detected ("check messages")**
  - Attempts to read an uninitialized variable
  - Aliasing violations
  - Data races
- **Suspicious constructs detected ("warnings")**
  - Incorrect parameter mode
  - Ineffective statements
  - Ineffective initialization
  - Unused variables

Flow Contracts

- **Global**
  - Identifies the non-local variables that are referenced by a subprogram
    - Access modes are Input, Output, and In_Out
    - If no use of globals, specify **null**
    - If present, the contract must be correct and complete
- **Depends**
  - Identifies relationship between outputs and inputs of a subprogram
  - Considers parameters and global variables
  - If specified, a Depends contract is checked for consistency with the subprogram body
- **These contracts are optional**
  - Synthesized automatically if absent
  - Generally omit, unless required for a specific assurance reason

```
X, Y, Z : Integer;
…
procedure Q (S : in Boolean)
  with Global => (Input  => X,
                  Output => Y,
                  In_Out => Z);

procedure R (B : in out String)
  with Global => null;
```

```
X, Y, Z : Integer;
…
procedure Q (S : in Boolean)
  with Depends => ((Y  => X, S),
                   (Z => (Z, S, X)));
                -- (Z =>+ (S, X))

procedure R (B : in out String)
  with Depends =>  (B => B);
                -- (B =>+ null)
```

Quiz Time!

Preconditions

- **Precondition can reference formal parameters and global variables**
  - Beware of raising an exception during evaluation

```
N : Integer range 0 .. 1_000 := … ;
procedure Increment( X : in out Integer )
   with Pre     => X + N <= Integer'Last,
        Post    => X = X'Old + N
   is
begin
   X := X + N;
end Increment;
```

  - SPARK diagnostic message

medium: overflow check might fail (e.g. when N = 1 and X = Integer'Last)

- **Techniques for avoiding the possibility of an exception**
  - Express the precondition differently

```
with Pre => X <= Integer'Last - N
```

```
with Pre => Long_Integer(X) + Long_Integer(N) <=
            Long_Integer(Integer'Last)
```

  - Use `pragma Overflow_Mode` (or a compiler switch) to evaluate the expression either in a type with a wider range if possible, or as a "mathematical integer" (no overflow)
- **If precondition is omitted, it is assumed to be True**

Postconditions (1)

- **Postcondition can reference formal parameters, global variables, and function result**
  - Formal parameters and global variables can be referenced to reflect their values on subprogram return or (via `'Old`) at subprogram invocation
  - *Target*'Old implies evaluating *Target* and making a copy of tits value on entry to the procedure
    - May be inefficient if *Target* is large and assertions are enabled at run-time
  - Rules prevent anomalies (e.g., where the target might not be evaluated during the postcondition)

```
procedure Extract (S : in out String; I : in Integer; Result : out Character)
  with Post => (if I in S'Range then Result = S(I)'Old); --Illegal
```

```
procedure Extract (S : in out String; I : in Integer; Result : out Character)
  with Post => (if I in S'Range then Result = S'Old(I)); --Legal
```

- **If omitted, postcondition is assumed to be True**

Postconditions (2)

- **The postcondition is the only information SPARK will use concerning the effect of a subprogram**
  - SPARK diagnostic on the 2nd call of Increment:

medium: precondition might fail, cannot prove N < Integer'last (e.g. when K = Integer'Last)

```
function Incr (N : Integer) return Integer
   with Pre => N < Integer'Last is
begin
   return N+1;
end Incr;

K : Integer := 0;
   …
K := Incr (K); -- OK
K := Incr (K); -- Check message
```

- **For an expression function, SPARK treats the expression as implicit postcondition**
  - `Post => Incr'Result = N+1`
  - SPARK can prove absence of integer overflow

```
function Incr (N : Integer) return Integer
   is   (N+1)
   with Pre => N < Integer'Last;

K : Integer := 0;
   …
K := Incr (K); -- OK
K := Incr (K); -- OK
```

- **Pre- and postconditions have a subtle interaction with OOP inheritance**
  - "Liskov Substitution Principle"

Postcondition Example: Sorting an Array

```
package Array_Handling is
   type Row is array (1 .. 100) of Integer;
   procedure Sort (R : in out Row)
      with Post => (for all I in R'First .. R'Last-1 => R(I) <= R(I+1));
end Array_Handling;
```

- **The postcondition shown is necessary but not sufficient**
- **A stronger postcondition would express additional requirements**
  - The sorted array is a permutation of the original array

```
function Is_Permutation(R1, R2 : Row) return Boolean is …;
procedure Sort (R : in out Row)
   with Post => (for all  I in R'First .. R'Last-1 => R(I) <= R(I+1)) and
                Is_Permutation(R, R'Old);
```

- **Whether to specify a weak or strong postcondition depends on the properties you are trying to prove**
  - Might not be practical to try to prove strong postcondition, but may be useful during testing

Loop Invariants (1)

- **Syntax:** `pragma Loop_Invariant (Boolean-expression);`
- **SPARK will try to prove**
  - *Initialization:* the Boolean expression is True during the 1st iteration
  - *Preservation:* the Boolean expression is True during an arbitrary iteration if it was also True during the previous one
- **Usage**
  - Generally appears at the start of the loop but may be anywhere (except nested in inner control structures)
  - May have several loop invariants (ease of reading)
  - Loop invariants need to be precise enough ("inductive') to allow the proof of the preservation property
    - For arrays and records, the proof tool needs to know which components have not been modified (*frame condition*)
    - In simple cases SPARK can generate the frame condition automatically
  - May reference 'Loop_Entry for value of variable at start of loop (analogous to 'Old)

Loop Invariants (2)

- **Properties of a good loop invariant**
  - Initially: provable in the 1st iteration
  - Inside: allow proving local assertions and absence of run-time errors in the loop
  - After: allow proving local assertions, absence of run-time errors, and subprogram postcondition after the loop
  - Preservation: provable after the first iteration
- **Example**

```
procedure Replace(S : in out String; Char, By : in Character)
   with
      Post => (for all I in S'Range =>
                  S(I) = (if S'Old(I) = Char then By else S'Old(I)))
   is
begin
   for I in S'Range loop
      pragma Loop_Invariant( for all J in S'First .. I-1 =>
                S(J) = (if S'Loop_Entry(J) = Char then By else S'Loop_Entry(J) ));
      pragma Loop_Invariant( for all J in I .. S'Last =>
                S(J) = S'Loop_Entry(J) );
      if S(I) = Char then
         S(I) := By;
      end if;
   end loop;
end Replace;
```

SPARK can prove the loop invariants and the postcondition

The 2nd loop invariant is not needed (SPARK generates an equivalent frame condition)

Other SPARK Features

- **Initialization policy**
  - Strong initialization required
  - Enforced by Flow analysis
- **State abstraction**
  - Allow specifying Global and Depends contracts without breaking encapsulation
- **Loop variants**
  - Prove termination for plain loops and while loops
- **Ghost code**
  - Code that is needed for proofs but is not part of the program logic
- **Interfacing with the external world**
  - Support for "volatile" data
- **Concurrency**
  - "Ravenscar" profile (subset of Ada tasking)
- **Contract cases**
  - Convenient syntax for expressing complex postconditions
- **Libraries**
  - Containers
  - Lemma library
- **Safe Pointers**
  - Rust-like "ownership" facility

Current Limitations

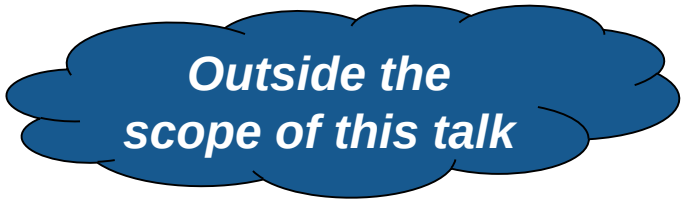
- **Non-linear arithmetic**
- **Floating-point**
- **Recursion**
- **Pointers**

Quiz Time!

Levels of Assurance

Summary of Levels

- **Stone**
  - Restrict code to SPARK subset
- **Bronze**
  - Stone + Verify data flow
- **Silver**
  - Bronze + Prove absence of run-time errors
- **Gold**
  - Silver + Prove integrity properties
- **Platinum**
  - Gold + Prove functional correctness



Higher level requires increased effort, brings increased assurance
Choice depends on degree of confidence that is required

Stone Level: Valid SPARK

- **Approach**
  - Implement as much of the code as possible in the SPARK subset
  - Run SPARK analyzer on codebase (or new code) and look at violations
  - For each violation, choose an approach
    - Convert to valid SPARK
    - Exclude from analysis
- **Benefits**
  - Portability: SPARK code is free of side effects in functions
  - Prelude to higher assurance levels
- **Costs and limitations**
  - Problems if code makes heavy use of pointers
  - Initial pass may entail large (but tractable) rewrites to transform the code

Stone Level: Example

- **Issue: function with side effect**

```
Last : Integer := 0;

procedure Log (J : Integer) is
begin
   Last := J;
end Log;

function Incr_And_Log (K : Integer)
   return Integer is
begin
   Log (K);    --<<-- VIOLATION
   return (K+1) mod 100;
end Incr_And_Log;
```

- **Solution 1: procedure with out parameter**

```
procedure Incr_And_Log
   (K      : Integer;
    Result : out Integer) is
begin
   Log (K);    --OK
   Result := (K+1) mod 100;
end Incr_And_Log;
```

- **Solution 2: function where side effect is masked**
  - Use if side effect does not affect properties to be verified

```
procedure Log (J : Integer)
   with Global => null;

Last : Integer := 0;

procedure Log (J : Integer)
   with SPARK_Mode => Off is
begin
   Last := J;
end Log;

function Incr_And_Log (K : Integer)
   return Integer is
begin
   Log (K);    --OK
   return (K+1) mod 100;
end Incr_And_Log;
```

Bronze Level: Initialization and Correct Data Flow

- **Approach**
  - Perform flow analysis of SPARK code to verify intended data usage
  - Review each check failure and choose an approach
    - Correct the code
    - Justify why it is a false alarm (pragma Annotate)
  - Review each warning and suppress when spurious
- **Benefits**
  - Absence of check failures guarantees a variety of program properties
    - No reads of uninitialized variables
    - No interference between parameters and global objects
    - No unintended access to global variables
    - No race conditions on accesses to shared data
  - Global and Depends contracts clearly document the programmer's intent
- **Costs and limitations**
  - Array analysis is conservative since element accesses are dynamic
  - Flow analysis does not track specific data values and may generate warnings for situations that can never occur

Silver Level: Absence of Run-Time Errors (AoRTE)

- **Approach**
  - Run SPARK tool in “proof” mode to locate all potential check failures
    - Attempts to prove that they will not occur
  - Investigate unproved run-time checks
    - If error in code or assertion, fix it
    - Add information to try to help the prover
      - Precise (sub)types, Assert pragmas, loop invariant, …
    - Use pragma Annotate or Assume if prover not powerful enough
  - Storage error (stack overflow) handled separately
    - Recursion allowed (programmer needs to ensure enough stack space)
- **Benefits**
  - Bronze + absence of run-time vulnerabilities
  - Efficiency: no need for run-time checks
- **Costs and limitations**
  - Initial pass may require some effort
  - Proof analysis can be time consuming
  - Analysis might not be powerful enough to prove some properties

Silver Level: Example

- **Initial version**

```
I, J : Integer := …;
X    : array (1..10) of Natural := …;
…
X(I+J) := I/J;  -- Possible check failures
```

Potential run-time errors:

- **I + J could overflow**
- **I + J could be outside the range 1..10**
- **I/J could overflow**
- **I/J could be negative**
- **J could be 0**

- **Code allowing proof of AoRTE**


```
subtype Int5 is Integer range 1..5;

I, J : Int5 := …;
X    : array (1..10) of Natural := …;
…
pragma Assert (I+J in X'Range and then J /= 0 and then I/J >= 0); -- Can be proved
X(I+J) := I/J;                                                    -- No run-time errors
```

  - The Assert pragma summarizes the conditions that need to be met in order to avoid all run-time errors in the assignment statement
  - Constraining the range of I and J allows the Assert pragma to be proved
  - In this example the prover can guarantee AoRTE without the Assert pragma

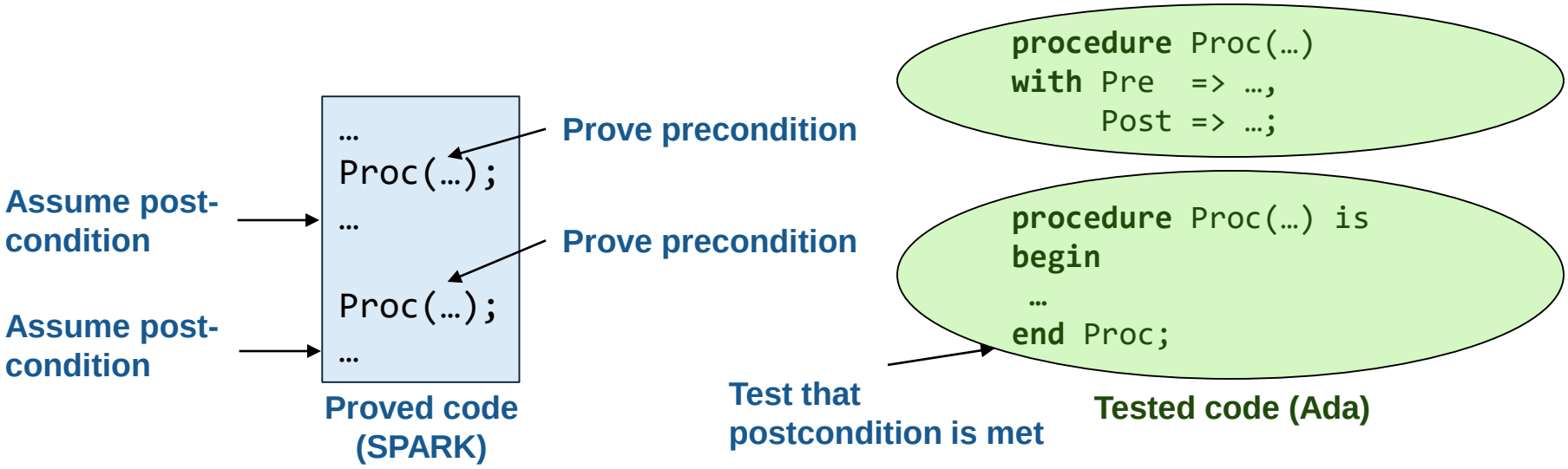
Gold Level: Proof of Key Integrity Properties (1)

- **Sample properties to be ensured**
  - Critical data invariants maintained
  - Safe transitions between program states
- **Approach**
  - Similar to process for Silver Level
  - SPARK proof tool either confirms that property holds or emits a “check” message about a possible violation
- **Benefits**
  - Similar to Silver
  - Proofs can be used for safety case rationale, to replace certain kinds of testing
- **Costs and limitations**
  - Similar to Silver
  - Some properties might not be easily expressible as data invariants or subprogram contracts
- **Useful features**
  - (Sub)type predicates
  - Ghost code

Hybrid Verification

Hybrid Verification (1)

- **You can combine tested code with proved (formally verified) SPARK code**
  - Proved code invokes tested subprogram, or tested code invokes proved subprogram
  - Tested code may be in SPARK, Ada, C, or other languages
  - Verification is performed at the call interfaces between tested and proved code
- **Scenario : Proved code calls tested subprogram**
  - SPARK tries to verify that the subprogram's precondition is met at each invocation
  - SPARK assumes the postcondition is met on return from the invocation
  - Testing must provide confidence that, if the invoked subprogram's precondition is met then its postcondition will be met when the subprogram returns



Hybrid Verification (2)

- **Example: SPARK code invoking a tested C function**

```
unsigned_char getascii(int portid);
// portid must be in 0..5, getascii return value is in 0..127
```

```
with Interfaces.C; use Interfaces.C;
package Terminal_Input is
   MAXPORTID : constant int := 5;

   function getascii (portid : int) return unsigned_char
      with Pre  => portid in 0 .. MAXPORTID,
           Post => getascii'Result in 0..127;
   pragma Import (C, getascii);
   -- Interfaces.C.unsigned_char is a modular (unsigned) integer type,
   -- ranging from 0 through 255
end Terminal_Input;
```

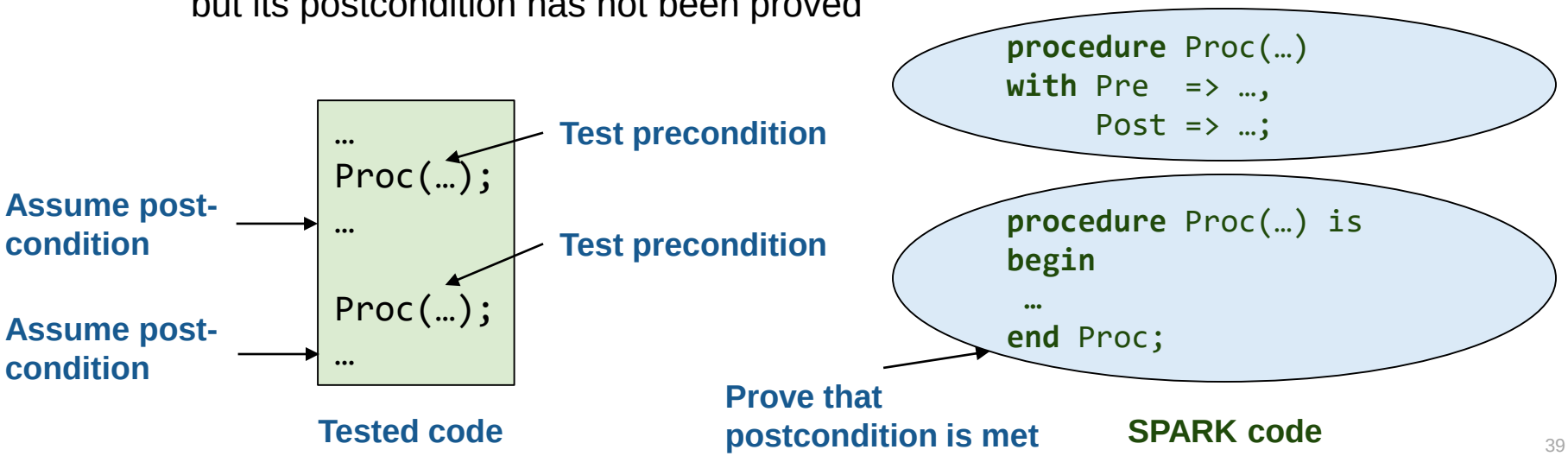
```
with Terminal_Input, Interfaces.C;
use Interfaces.C;
procedure Example is
   N : unsigned_char range 0..127;
begin
   N := Terminal_Input.getascii(portid => 2);
end Example;
```

SPARK can prove that no range check is needed for N (i.e., it assumes the postcondition)

SPARK can prove that the precondition is met

Hybrid Verification (3)

- **Scenario: Tested code calls proved subprogram**
  - Approach depends on which properties have been proved
  - "Best case": SPARK proves that the subprogram terminates and that, if the subprogram's precondition is met then the postcondition is met
    - Testing must ensure that the subprogram's precondition is met at each invocation
      - If SPARK subprogram can be called with precondition violated, enable precondition checks
    - Testing can assume that the subprogram's postcondition is met on return
  - Other cases require additional testing/analysis of the SPARK code
    - Example: the SPARK subprogram has been proved to be free of run-time errors but its postcondition has not been proved



Hybrid Verification (4)

- **Within an Ada application, some code may be in SPARK and other code in full Ada**
  - For example, a package or subprogram specification in SPARK but the body in full Ada
  - SPARK functional contracts can be checked dynamically under Ada semantics
- **Example**

```
type Vector is array (Low .. High) of Float;
procedure Sort (Vec : in out Vector) with
   SPARK_Mode => On,
   Post => (for all J in Low .. High-1 => Vec(J) <= Vec(J+1));
```

```
procedure Sort (Vec : in out Vector) is
   -- Inherit SPARK_Mode setting from spec
begin
   … -- Implementation in SPARK subset
end Sort;
```

- Body is analyzed by SPARK tools
- May be possible to prove postcondition, absence of run-time errors

```
procedure Sort (Vec : in out Vector) with
   SPARK_Mode => Off is
begin
   … -- Implementation in full Ada
end Sort;
```

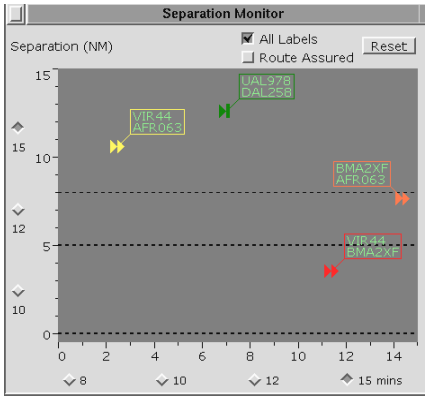
- Body is not analyzed by SPARK tools
- Use other means (testing, manual review) to verify postcondition, absence of run-time errors

Industrial Experience

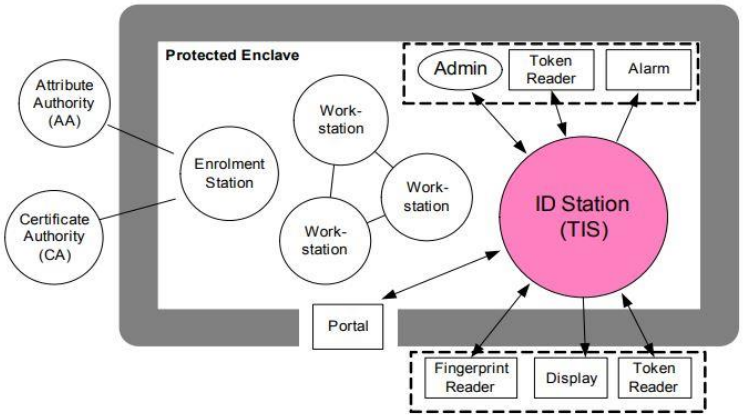
Past SPARK Projects at Altran UK

***SHOLIS**: 1995*
DEFSTAN 00-55 SIL4
*First **Gold***

***C130J**: 1996*
*Initially **Bronze***
(Lockheed Martin),
*now **Gold***
(UK RAF and BAESystems)



***iFACTS**: 2006*
*Now **Silver** (NATS)*



***Tokeneer**: 2008*
EAL 5+
Gold
(NSA)

Adoption Experience (Pilot Projects) at Thales / France

Use Case 1: Port to new platform

Context: Radar software / 300 kloc
Target: Stone level

- Significant manual refactoring (e.g., pointers)
- Large amount done in only several days
- In progress / on the way to completion

Use Case 2: Demonstrate compliance with Low-Level Requirements

Context: Small numerical function
Target: Gold level

- Difficulties in writing suitable contracts
- Property was not proved automatically
- Gold level may be too ambitious for numeric computation

Use Case 3: Identify and fix weakness

Context: Code generator
Several hundred loc
Target: Gold level

- Half a day to reach Silver
- Property related to buffer overflow
- Two days to reach Gold

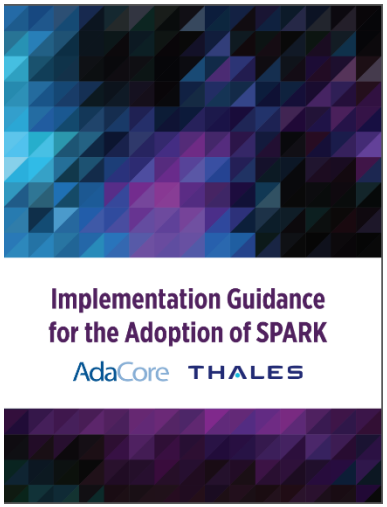
Use Case 4: Guarantee safety properties

Context: Command & Control
7 kloc
Target: Gold level

- One day to reach Silver
- Property expressed as automaton
- Four days to reach Gold

Adoption Guidelines with Thales

- **Booklet based on experience with SPARK**
  - Usage by Thales on pilot projects
  - Internal usage on SPARK tool itself
- **For each level, the booklet explains**
  - Benefits, impact on process, costs and limitations
  - Setup and tool usage
  - Violation messages issued by the tool
  - Remediation solutions
- **Thales and others are using / adapting the guidance on specific projects**



Other Formal Methods Approaches: SPARK *versus* Frama-C (1)

- **What Is Frama-C?**
  - A platform for static analysis, with plug-ins for various purposes
  - Eva plug-in: sound tool that can prove absence of run-time errors
  - WP plug-in: sound tool that can prove functional correctness
- **Similarities between SPARK and Frama-C**
  - Basic approach: source code with annotations
  - Assertions

```
x++
/*@ assert x > 0 ; */
```

```
X := X+1;
pragma Assert (x > 0);
```

  - Contracts
    - Both allow pre- and postconditions with comparable functionality
  - User specification of a module's side effects
  - Formal verification of contracts
  - Ghost code

Reference: J. Kanig, *A Comparison of SPARK with MISRA C and Frama-C*
`www.adacore.com/papers/compare-spark-misra-c-frama-c`

Other Formal Methods Approaches: SPARK *versus* Frama-C (2)

- **Differences between SPARK and Frama-C**
  - Precise subtypes for integers and floating-point types
  - Pointers and the memory model
  - Annotation language
    - Standard Ada syntax for SPARK contracts, versus stylized comments in Frama-C
    - Specialized language (ACSL) for Frama-C assertions, versus standard Ada expressions in SPARK
    - Executable contracts fully supported in SPARK, partially supported (E-ACSL) in Frama-C
    - Support for mathematical concepts (unbounded integers etc) better integrated in Frama-C
    - SPARK includes useful Ada features (subtype predicates, type invariants)
  - Library support
    - Part of C standard library available for Frama-C, lemma library for SPARK

Other Formal Methods Approaches: SPARK *versus* Frama-C (3)

- **Differences between SPARK and Frama-C (cont'd.)**
  - Toolsets
    - SPARK includes functionalities not supported (at present) in Frama-C
    - Frama-C needs to be compiled and installed with supporting tools, SPARK proof tool is self-contained as binary package
- **Summary of main benefits of SPARK**
  - Contracts are standard Ada syntax, not a separate language
  - Contracts for functional correctness are executable, can be used by testing tools and not just static analysis
  - Simpler to express contracts (no aliasing)
  - Different Frama-C tools may support different sets of features
- **Summary of main benefits of Frama-C**
  - Large potential user base (C programmers)
  - Can prove properties of pointer manipulation
  - Support for unbounded integers, reals

Other Formal Methods Approaches: SPARK *versus* Frama-C (4)

	Frama-C	SPARK
Pre/Postconditions?	Yes	Yes
Global annotations?	Yes	Yes
Pointers?	Yes	Restricted
Aliasing excluded?	No	Yes
Precise subtypes?	No	Yes
Executable annotations?	Via E-ACSL	Yes
Unbounded integers, reals?	Yes	No
Specification language same as programming language?	No	Yes
Ghost code?	Yes	Yes

Conclusions

Summary

- **SPARK technology addresses needs of high-security community**
  - Language is expressive but formally analyzable
  - Can be used to provide high confidence in
    - Correctness of security functions
    - Absence of vulnerabilities in application code outside of security functions
  - Toolset is sound without generating an abundance of false alarms
- **Assurance levels range from strong semantic coding standard to full functional correctness**
  - Every level implicitly builds on the lower levels
  - Lower levels entail lower cost/effort
  - Good match from DAL/SIL to Bronze-Silver-Gold-Platinum
- **Industrial experience**
  - Formal methods can reduce life cycle costs while increasing assurance
  - Users do not need to be experienced with formal methods
  - "Formal verification via SPARK allowed flexibility to fine-tune its gradual insertion into existing processes while mitigating its associated risks and costs"

Acknowledgments

THANK YOU

To Yannick Moy,
Claire Dross,
Pat Rogers
And the rest of the SPARK team
at AdaCore and Altran

References

- **Interactive training**
  - learn.adacore.com
- **SPARK toolset**
  - Commercially supported *SPARK Pro* version: `www.adacore.com/sparkpro`
  - Freely Licensed Open Source Software version: available in GNAT Community Edition
    - Downloadable from `www.adacore.com/community`
- **Textbook for students and software professionals**
  - J. McCormick and P. Chapin, *Building High Integrity Applications with SPARK*; Cambridge University Press; 2015
- **Booklet: *Implementation Guidance for the Adoption of SPARK***
  - `www.adacore.com/knowledge/technical-papers/implementation-guidance-spark`
  - Paper copies available on request: email to `info@adacore.com`
- **General information and other links**
  - `www.adacore.com/about-spark`

AdaCore Contact Information

North American Headquarters:

150 W. 30th St., 16th Floor
New York, NY 10001
USA

+1-212-620-7300 (voice)
+1-212-807-0162 (FAX)

European Headquarters:

46 rue d'Amsterdam
75009 Paris
France

+33-1-4970-6716 (voice)
+33-1-4970-0552 (FAX)

info@adacore.com
www.adacore.com