SPARK is not Ada
SPARK 2014 is not Ada 2012
SPARK 2014 is contract-based practical static verification for Ada
SPARK 2014 is contract-based practical static verification for Ada

modular
scalable
user-defined properties
precise
SPARK 2014 is contract-based practical static verification for Ada
SPARK 2014 is contract-based practical static verification for Ada

- dataflow analysis
- information flow analysis
- robustness analysis
- functional analysis
SPARK 2014 is contract-based practical static verification for Ada usable on existing codebase large subset combined with testing
with Door; use Door;
with AlertTypes; use AlertTypes;

-- Tokeneer ID Station Core Software
-- Copyright (2003) United States Government, as represented
-- by the Director, National Security Agency. All rights reserved.
--
-- This material was originally developed by Praxis High Integrity
-- Systems Ltd. under contract to the National Security Agency.

-- Alarm
-- Description:
-- Provides interface to the alarm

with Auditlog;

package Alarm is

-- PROOF ANNOTATIONS FOR SECURITY PROPERTY 3
--
-- A proof function is required to model the proof
-- function InterFac.prf_isAlarming(InterFac.Output)
-- (which is not visible outside the package body).
-- InterFac.Output is a refinement of Output, so
-- need to take Output as a parameter of the
-- function.To do this, need to define an abstract
-- type for Output.
-- The InterFac.prf_isAlarming proof function is
-- effectively a refinement of this proof function.

function IsAlarming return Boolean
with Global => null,
Convention => Ghost;

-- UpdateDevice
--
-- Description:
-- Updates the physical alarm depending on the state of the
-- Door alarm and the Auditlog alarm.
Limitations of Vintage SPARK

1. Cost of adding mandatory contracts
2. Not usable on existing code
   - Constraints on visibility / program structure
   - Very restricted language subset
   - Constraints on the control flow graph
3. Limitations of proof
   - Floating-point interpreted as real
   - Very simple VC generation
   - Prover does not handle well disjunctions and quantifiers
4. Not integrated in traditional development process
   - Incompatible with testing
   - Impossible to debug contracts
   - Weak IDE support
Strengths of SPARK 2014

1. Cost of adding mandatory contracts

2. Not usable on existing code
   - Constraints on visibility / program structure
   - Very restricted language subset
   - Constraints on the control flow graph

3. Limitations of proof
   - Floating-point interpreted as real
   - Very simple VC generation
   - Prover does not handle well disjunctions and quantifiers

4. Not integrated in traditional development process
   - Incompatible with testing
   - Impossible to debug contracts
   - Weak IDE support
Strengths of SPARK 2014

1. generation of required contracts

2. not usable on existing code
   - constraints on visibility / program structure
   - very restricted language subset
   - constraints on the control flow graph

3. limitations of proof
   - floating-point interpreted as real
   - very simple VC generation
   - prover does not handle well disjunctions and quantifiers

4. not integrated in traditional development process
   - incompatible with testing
   - impossible to debug contracts
   - weak IDE support
Strengths of SPARK 2014

1. generation of required contracts
2. usable on existing code
   • use Ada rules for visibility / program structure
   • subset includes generics, discriminants, etc.
   • allow any loop exit, early return, recursion
3. limitations of proof
   • floating-point interpreted as real
   • very simple VC generation
   • prover does not handle well disjunctions and quantifiers
4. not integrated in traditional development process
   • incompatible with testing
   • impossible to debug contracts
   • weak IDE support
Strengths of SPARK 2014

1. generation of required contracts
2. usable on existing code
   - use Ada rules for visibility / program structure
   - subset includes generics, discriminants, etc.
   - allow any loop exit, early return, recursion
3. powerful automatic proof
   - support IEEE 754 floating-point semantics
   - efficient and precise VC generation
   - use state-of-the-art SMT solver
4. not integrated in traditional development process
   - incompatible with testing
   - impossible to debug contracts
   - weak IDE support
Strengths of SPARK 2014

1. generation of required contracts
2. usable on existing code
   - use Ada rules for visibility / program structure
   - subset includes generics, discriminants, etc.
   - allow any exit, early return, recursion
3. powerful automatic proof
   - support IEEE 754 floating-point semantics
   - efficient and precise VC generation
   - use state-of-the-art SMT solver
4. integrated in developer toolbox
   - combined with testing
   - contracts can be executed and debugged
   - fine-grain interactions in two IDEs
Essential Principles of the Retooling

1. **convergence with compiler technology (GNAT)**
   - allows to support a larger subset of Ada in SPARK
   - target-dependent & compiler-dependent proofs

2. **use of intermediate verification language (Why3)**
   - powerful VC generation and transformations
   - rich language features (exceptions, types)

3. **use of state-of-the-art SMT solvers (Alt-Ergo + ...)**
   - powerful automation of proofs
Tool Architecture

**note**: all components are Free / Libre / Open Source Software

- GNAT project support
- GNAT compiler frontend
- Why3 VCgen & driver
- Alt-Ergo SMT Solver
Tool Architecture

note: all components are Free / Libre / Open Source Software

Ada

Why

•

GNAT project support

GNAT compiler frontend

Why3 VCgen & driver

Alt-Ergo SMT Solver

Ada

Ada

Ada

Why

Why

flow error
initialization error

OK

runtime error
contract violation

OK

VC

VC
Tool Architecture

note: all components are Free / Libre / Open Source Software

Ada

Why

•

•

•

GNAT project support

GNAT compiler frontend

Why3 VCgen & driver

Alt-Ergo SMT Solver

incremental / parallel / distributed

flow error

initialization error

OK

runtime error

contract violation

OK
Case Studies by Airbus Defence & Space

carried by David Lesens, expert in formal methods
in *Formal Validation of Aerospace Software*, DASIA 2013

with Vintage SPARK
(2011, 182 subp, 44 Pre, 66 Post)

with SPARK 2014
(2010-2013, 1500 subp, 1400 Pre, 400 Post)
Case Studies by Airbus Defence & Space

carried by David Lesens, expert in formal methods
in *Formal Validation of Aerospace Software*, DASIA 2013

**with Vintage SPARK**

(2011, 182 subp, 44 Pre, 66 Post)

+ proved absence of RE

**with SPARK 2014**

(2010-2013, 1500 subp, 1400 Pre, 400 Post)

+ proved absence of RE (93%)
+ proved functional behavior (98%)
Case Studies by Airbus Defence & Space

carried by David Lesens, expert in formal methods
in Formal Validation of Aerospace Software, DASIA 2013

**with Vintage SPARK**

- (2011, 182 subp, 44 Pre, 66 Post)
- proved absence of RE
- language restrictions cause over-cost & limited scope

**with SPARK 2014**

- (2010-2013, 1500 subp, 1400 Pre, 400 Post)
- proved absence of RE (93%)
- proved functional behavior (98%)
- generics and discriminants support variability in versions & data
Case Studies by Airbus Defence & Space

with Vintage SPARK

(2011, 182 subp, 44 Pre, 66 Post)

➕ proved absence of RE

➖ language restrictions cause over-cost & limited scope

➖ unacceptable to engineers

with SPARK 2014

(2010-2013, 1500 subp, 1400 Pre, 400 Post)

➕ proved absence of RE (93%)

➕ proved functional behavior (98%)

➕ generics and discriminants support variability in versions & data

➕ usable by non experts

carried by David Lesens, expert in formal methods
in *Formal Validation of Aerospace Software*, DASIA 2013
# Case Studies by Airbus Defence & Space

<table>
<thead>
<tr>
<th>with Vintage SPARK</th>
<th>with SPARK 2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2011, 182 subp, 44 Pre, 66 Post)</td>
<td>(2010-2013, 1500 subp, 1400 Pre, 400 Post)</td>
</tr>
</tbody>
</table>

**+** proved absence of RE  
**+** proved functional behavior (98%)  
**+** proved absence of RE (93%)  
**+** generics and discriminants support variability in versions & data  
**+** usable by non experts  
**+** contracts used for test and proof  

**−** language restrictions cause over-cost & limited scope  
**−** unacceptable to engineers  
**−** not combined with test  

---

Case Studies by Airbus Defence & Space

carried by David Lesens, expert in formal methods
in *Formal Validation of Aerospace Software*, DASIA 2013

with Vintage SPARK
(2011, 182 subp, 44 Pre, 66 Post)

➕ proved absence of RE
➖ language restrictions cause over-cost & limited scope
➖ unacceptable to engineers
➖ not combined with test
➖ interactive proof required, too complex and expensive

with SPARK 2014
(2010-2013, 1500 subp, 1400 Pre, 400 Post)

➕ proved absence of RE (93%)
➕ proved functional behavior (98%)
➕ generics and discriminants support variability in versions & data
➕ usable by non experts
➕ contracts used for test and proof
➕ use test when not proved
EXAMPLE OF CONTRACT

-- Set the value of a variable
procedure Set_Nat32_Variable (Variable_Id : T_Variable_Id;
New_Value : T_Nat32;
Variables : in out T_Variables)
with
Pre =>
(Is_Valid (Variables) and then
Is_Nat32 (Variable_Id, Variables) and then
Get_Min_Nat32 (Variable_Id, Variables) = New_Value and then
New_Value <= Get_Max_Nat32 (Variable_Id, Variables))
Post =>
(Is_Valid (Variables) and then
Is_Nat32 (Variable_Id => Variable_Id,
Variables => Variables) and then
Get_min_Nat32 (Variable_Id => Variable_Id,
Variables => Variables) = Get_Min_Nat32 (Variable_Id => Variable_Id,
Variables => Variables) and then
Get_max_Nat32 (Variable_Id => Variable_Id,
Variables => Variables) = Get_Max_Nat32 (Variable_Id => Variable_Id,
Variables => Variables) and then
Get_Nat32 (Variable_Id, Variables) = New_Value and then
(for all Id in T_Variable_Id =>
(if Id /= Variable_Id then
Get_Variable (Id, Variables) = Get_Variable (Id, Variables'Old)))));
procedure Set_Nat32_Variable (Variable_Id : T_Variable_Id;
   New_Value : T_Nat32;
   Variables : in out T_Variables)

with
  Pre  =>
    (Is_Valid (Variables) and then
    Is_Nat32 (Variable_Id, Variables) and then
    Get_Min_Nat32 (Variable_Id, Variables) <= New_Value and then
    New_Value <= Get_Max_Nat32 (Variable_Id, Variables)),
  Post =>
    (Is_Valid (Variables) and then
    Is_Nat32 (Variable_Id => Variable_Id,
    Variables  => Variables) and then
    Get_Min_Nat32 (Variable_Id => Variable_Id,
    Variables  => Variables) = Get_Min_Nat32 (Variable_Id => Variable_Id,
    Variables  => Variables'Old) and then
    Get_Max_Nat32 (Variable_Id => Variable_Id,
    Variables  => Variables) = Get_Max_Nat32 (Variable_Id => Variable_Id,
    Variables  => Variables'Old) and then
    Get_Nat32 (Variable_Id, Variables) = New_Value and then
    (for all Id in T_Variable_Id =>
      (if Id /= Variable_Id then
        Get_Variable (Id, Variables) = Get_Variable (Id, Variables'Old)))));
procedure Set_Nat32_Variable (Variable_Id : T_Variable_Id;
                          New_Value : T_Nat32;
                          Variables : in out T_Variables)

with
  Pre =>
    (Is_Valid (Variables) and then
     Is_Nat32 (Variable_Id, Variables) and then
     Get_Min_Nat32 (Variable_Id, Variables) <= New_Value and then
     New_Value <= Get_Max_Nat32 (Variable_Id, Variables)),
Post =>
  (Is_Valid (Variables) and then
   Is_Nat32 (Variable_Id => Variable_Id,
             Variables  => Variables) and then
   Get_Min_Nat32 (Variable_Id => Variable_Id,
                      Variables  => Variables) = Get_Max_Nat32 (Variable_Id => Variable_Id,
                                                                    Variables  => Variables'Old) and then
   Get_Nat32 (Variable_Id, Variables) = New_Value and then
   (for all Id in T_Variable_Id =>
    (if Id = Variable_Id then
     Get_Variable (Id, Variables) = Get_Variable (Id, Variables'Old)))));
proof of absence of RE in contracts
(Pre should be self-guarded)

-- Set the value of a variable
procedure Set_Nat32_Variable (Variable_Id : T_Variable_Id;
                            New_Value : T_Nat32;
                            Variables : in out T_Variables)

with
Pre =>
   (Is_Valid (Variables) and then
    Is_Nat32 (Variable_Id, Variables) and then
    Get_Min_Nat32 (Variable_Id, Variables) <= New_Value and then
    New_Value <= Get_Max_Nat32 (Variable_Id, Variables)),
Post =>
   (Is_Valid (Variables) and then
    Is_Nat32 (Variable_Id => Variable_Id,
                Variables => Variables) and then
    Get_Min_Nat32 (Variable_Id => Variable_Id,
                    Variables => Variables) = Get_Min_Nat32 (Variable_Id => Variable_Id,
                    Variables => Variables'Old) and then
    Get_Max_Nat32 (Variable_Id => Variable_Id,
                    Variables => Variables) = Get_Max_Nat32 (Variable_Id => Variable_Id,
                    Variables => Variables'Old) and then
    Get_Nat32 (Variable_Id, Variables) = New_Value and then
    (for all Id in T_Variable_Id =>
     (if Id /= Variable_Id then
      Get_Variable (Id, Variables) = Get_Variable (Id, Variables'Old))));
procedure Set_Nat32_Variable (Variable_Id : T_Variable_Id;
    New_Value : T_Nat32;
    Variables : in out T_Variables)
with
    Pre =>
        (Is_Valid (Variables) and then
         Is_Nat32 (Variable_Id, Variables) and then
         Get_Min_Nat32 (Variable_Id, Variables) <= New_Value and then
         New_Value <= Get_Max_Nat32 (Variable_Id, Variables)),
    Post =>
        (Is_Valid (Variables) and then
         Is_Nat32 (Variable_Id => Variable_Id, Variables => Variables) and then
         Get_Min_Nat32 (Variable_Id => Variable_Id, Variables => Variables) = Get_Min_Nat32 (Variable_Id => Variable_Id, Variables => Variables'Old) and then
         Get_Max_Nat32 (Variable_Id => Variable_Id, Variables => Variables) = Get_Max_Nat32 (Variable_Id => Variable_Id, Variables => Variables'Old) and then
         Get_Nat32 (Variable_Id, Variables) = New_Value and then
         (for all Id in T_Variable_Id =>
            (if Id /= Variable_Id then
               Get_Variable (Id, Variables) = Get_Variable (Id, Variables'Old))));
-- Set the value of a variable

procedure Set_Nat32_Variable (Variable_Id : T_Variable_Id;
                          New_Value   : T_Nat32;
                          Variables   : in out T_Variables)

with
  Pre  =>
    (Is_Valid (Variables) and then
    Is_Nat32 (Variable_Id, Variables) and then
    Get_Min_Nat32 (Variable_Id, Variables) <= New_Value and then
    New_Value <= Get_Max_Nat32 (Variable_Id, Variables)),

Post  =>
  (Is_Valid (Variables) and then
  Is_Nat32 (Variable_Id => Variable_Id,
  Variables  => Variables) and then
  Get_Min_Nat32 (Variable_Id => Variable_Id,
        Variables  => Variables) = Get_Min_Nat32 (Variable_Id => Variable_Id,
        Variables  => Variables'Old) and then
  Get_Max_Nat32 (Variable_Id => Variable_Id,
        Variables  => Variables) = Get_Max_Nat32 (Variable_Id => Variable_Id,
        Variables  => Variables'Old) and then
  Get_Nat32 (Variable_Id, Variables) = New_Value and then
  (for all Id in T_Variable_Id =>
    (if Id /= Variable_Id then
      Get_Variable (Id, Variables) = Get_Variable (Id, Variables'Old))))
-- Set the value of a variable
procedure Set_Nat32_Variable (Variable_Id :  T_Variablename_Id;
  New_Va
  Variables : in out T_Variablename)

with
  Pre   =>
    (Is_Valid (Variables) and then
     Is_Nat32 (Variable_Id, Variables) and then
     Get_Min_Nat32 (Variable_Id, Variables) <= New_Value and then
     New_Value <= Get_Max_Nat32 (Variable_Id, Variables)),
  Post  =>
    (Is_Valid (Variables) and then
     Is_Nat32 (Variable_Id => Variable_Id,
                  Variables => Variables) and then
     Get_Min_Nat32 (Variable_Id => Variable_Id,
                     Variables => Variables) = Get_Min_Nat32 (Variable_Id => Variable_Id,
                                             Variables => Variables'Old) and then
     Get_Max_Nat32 (Variable_Id => Variable_Id,
                      Variables => Variables) = Get_Max_Nat32 (Variable_Id => Variable_Id,
                                                  Variables => Variables'Old) and then
     Get_Nat32 (Variable_Id, Variables) = New_Value and then
     (for all Id in T_Variablename_Id =>
      (if Id /= Variable_Id then
       Get_Variable (Id, Variables) = Get_Variable (Id, Variables'Old))));

need unbounded arithmetic in contract? use Overflow_Mode
function Init return _Variables is
  Result : T_Variables;
begin
  pragma Assert (Variable.Is_Valid (Var => Variable.C_Variable));
  for Variable in T_Variable_Id'loop
    Result.Variable_Id := Variable.C_Variable;
    pragma loopInvariant
    (for all I in T_Variable_Id'range => Variable_Id => Variable.Is_Valid (Result (I)));
  end loop;
  return Result;
end Init;
function Init return T_Variables is
  Result : T_Variables;
begin
  pragma Assert (Variable.Is_Valid (Var => Variable.C_Variable));
  for Variable_Id in T_Variable_Id loop
    Result (Variable_Id) := Variable.C_Variable;
    pragma Loop_Invariant
    (for all I in T_Variable_Id range T_Variable_Id'First .. Variable_Id =>
      Variable.Is_Valid (Result (I)));
  end loop;
  return Result;
end Init;
function Init return T_Variables is
    Result : T_Variables;
begin
    pragma Assert (Variable.Is_Valid (Var => Variable.C_Variable));
    for Variable_Id in T_Variable_Id loop
        Result (Variable_Id) := Variable.C_Variable;
        pragma Loop_Invariant
            (for all I in T_Variable_Id'First .. Variable_Id => Variable.Is_Valid (Result (I)));
    end loop;
    return Result;
end Init;
function Init return T_Variables is
   Result : T_Variables;
begin
   pragma Assert (Variable.Is_Valid (Var => Variable.C_Variable));
   for Variable_Id in T_Variable_Id loop
      Result (Variable_Id) := Variable.C_Variable;
      pragma Loop_Invariant
      (for all I in T_Variable_Id range T_Variable_Id'First .. Variable_Id =>
         Variable.Is_Valid (Result (I)));
   end loop;
   return Result;
end Init;
now: methodology for writing loop invariants
soon: common patterns of loop invariants
planned: generation of loop invariants based on patterns

```haskell
function Init return T_Variables is
  Result : T_Variables;
begin
  pragma Assert (Variable.Is_Valid (Var => Variable.C_Variable));
  for Variable_Id in T_Variable_Id loop
    Result (Variable_Id) := Variable.C_Variable;
    pragma Loop.Invariant
    (for all I in T_Variable_Id range T_Variable_Id'First .. Variable_Id =>
     Variable.Is_Valid (Result (I)));
  end loop;
  return Result;
end Init;
```
need to prove while-loop termination? use Loop_Variant

```ada
function Init return T_Variables is
  Result : T_Variables;
begn
  pragma Assert (Variable.Is_Valid (Var => Variable.C_Variable));
  for Variable_Id in T_Variable_Id loop
    Result (Variable_Id) := Variable.C_Variable;
    pragma Loop_Invariant
    (for all I in T_Variable_Id range T_Variable_Id'First .. Variable_Id => Variable.Is_Valid (Result (I)));
  end loop;
  return Result;
end Init;
```
Combining Test & Proof

**goal:** be at least as good as test alone

strategy presented in *Integrating formal program verification with testing*, ERTS 2012

& *Explicit assumptions - a prenup for marrying static and dynamic program verification*, Test & Proof 2014

---

**Main_Program (Ada)**

*calls*

**Core_Service (SPARK)**

*calls*

**Low_Level_Service (Ada)**

---

**verification method**

- integration tests
- formal verification
- units tests

**assumptions**

- Pre respected
  - non-aliasing of inputs
  - initialization of inputs

- Post respected
  - no unintended side-effects
  - initialization of outputs
Combining Test & Proof

**goal**: be at least as good as test alone

strategy presented in *Integrating formal program verification with testing*, ERTS 2012

& *Explicit assumptions - a prenup for marrying static and dynamic program verification*, Test & Proof 2014

---

**Main_Program** (Ada)

---

**Core_Service** (SPARK)

---

**Low_Level_Service** (Ada)

---

**verification method**

integration tests

formal verification

units tests

**assumptions**

Pre respected
- non-aliasing of inputs
- initialization of inputs

Post respected
- no unintended side-effects
- initialization of outputs

**verified during tests by executing contracts**
Combining Test & Proof

**goal:** be at least as good as test alone

strategy presented in *Integrating formal program verification with testing*, ERTS 2012

& *Explicit assumptions - a prenup for marrying static and dynamic program verification*, Test & Proof 2014

---

**Main_Program (Ada)**

**Core_Service (SPARK)**

**Low_Level_Service (Ada)**

---

**verification method**

integration tests

**assumptions**

Pre respected
- non-aliasing of inputs
- initialization of inputs

Post respected
- no unintended side effects
- initialization of outputs

verified during tests by special compiler instrumentation
Combining Test & Proof

goal: be at least as good as test alone
strategy presented in Integrating formal program verification with testing, ERTS 2012
& Explicit assumptions - a prenup for marrying static and dynamic program verification, Test & Proof 2014

**Main_Program** (Ada) -------- calls -------- **Core_Service** (SPARK) -------- calls -------- **Low_Level_Service** (Ada)

verification method
integration tests
formal verification
units tests

assumptions
Pre respected
non-aliasing of inputs
initialization of inputs

Post respected
no unintended side-effects
initialization of outputs

verified by construction or by review
Combining Test & Proof

**goal:** be at least as good as test alone

strategy presented in *Integrating formal program verification with testing*, ERTS 2012

& *Explicit assumptions - a prenup for marrying static and dynamic program verification*, Test & Proof 2014

---

**Main_Program (Ada)**

**Core_Service (SPARK)**

**Low_Level_Service (Ada)**

known objective of formal verification projects: justify assumptions

**proposal:** switch from ad-hoc to tool-assisted assumptions management

---

<table>
<thead>
<tr>
<th>verification method</th>
<th>assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>integration tests</td>
<td>Pre respected non-aliasing of inputs initialization of inputs</td>
</tr>
<tr>
<td>formal verification</td>
<td>Post respected no unintended side-effects initialization of outputs</td>
</tr>
<tr>
<td>units tests</td>
<td></td>
</tr>
</tbody>
</table>
Test & Proof in DO-178C

goal: be at least as good as test alone, for all objectives assigned to test

Testing or Formal Verification: DO-178C Alternatives and Industrial Experience, IEEE Software, June 2013
& Guidelines for the Use of Theorem Proving in the Certification of Critical Systems, workshop TPC, 2014
Test & Proof in DO-178C

**goal:** be at least as good as test alone, for all objectives assigned to test

goal: be at least as good as test alone, for all objectives assigned to test

Testing or Formal Verification: DO-178C Alternatives and Industrial Experience, IEEE Software, June 2013

& Guidelines for the Use of Theorem Proving in the Certification of Critical Systems, workshop TPC, 2014
**Test & Proof in DO-178C**

**goal:** be at least as good as test alone, for all objectives assigned to test


as alternative objectives to coverage, need to:

- justify assumptions
- specify contracts by cases
- specify intended dataflows
- review for absence of dead code
SPARK2014 is the only language and toolset providing industrial support for both dynamic and formal contract-based verification of software.

http://www.adacore.com/sparkpro

http://www.spark-2014.org