Boundaries of Formal Program Verification

Yannick Moy – AdaCore

AdaCore ALTRAN PARTNERSHIP

SPARK – the language

strong typing

low level programming

generics

object orientation

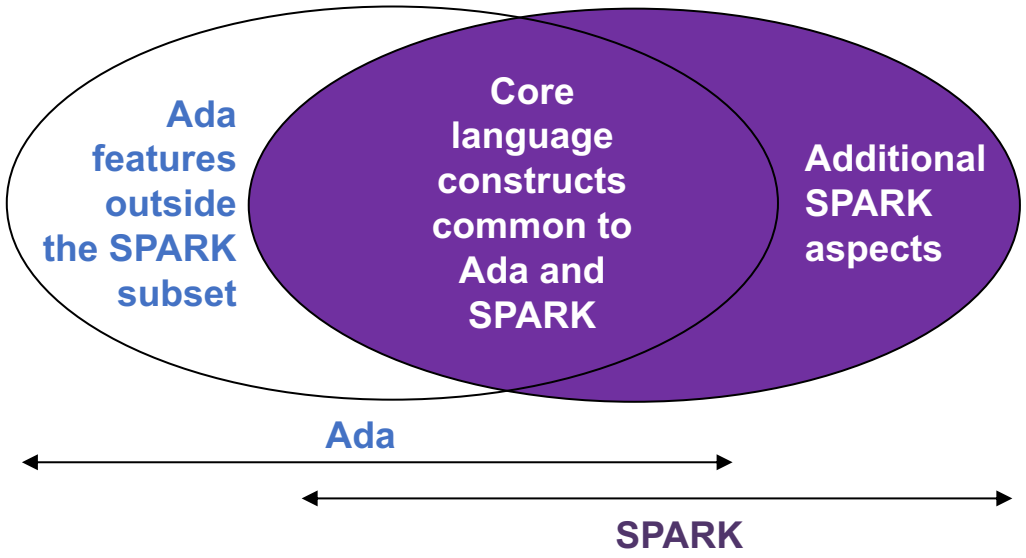
concurrency

pointers

exception handlers

controlled types

function with effects



Abstract_State

Initializes

Initial_Condition

Contract_Cases

Global

Depends

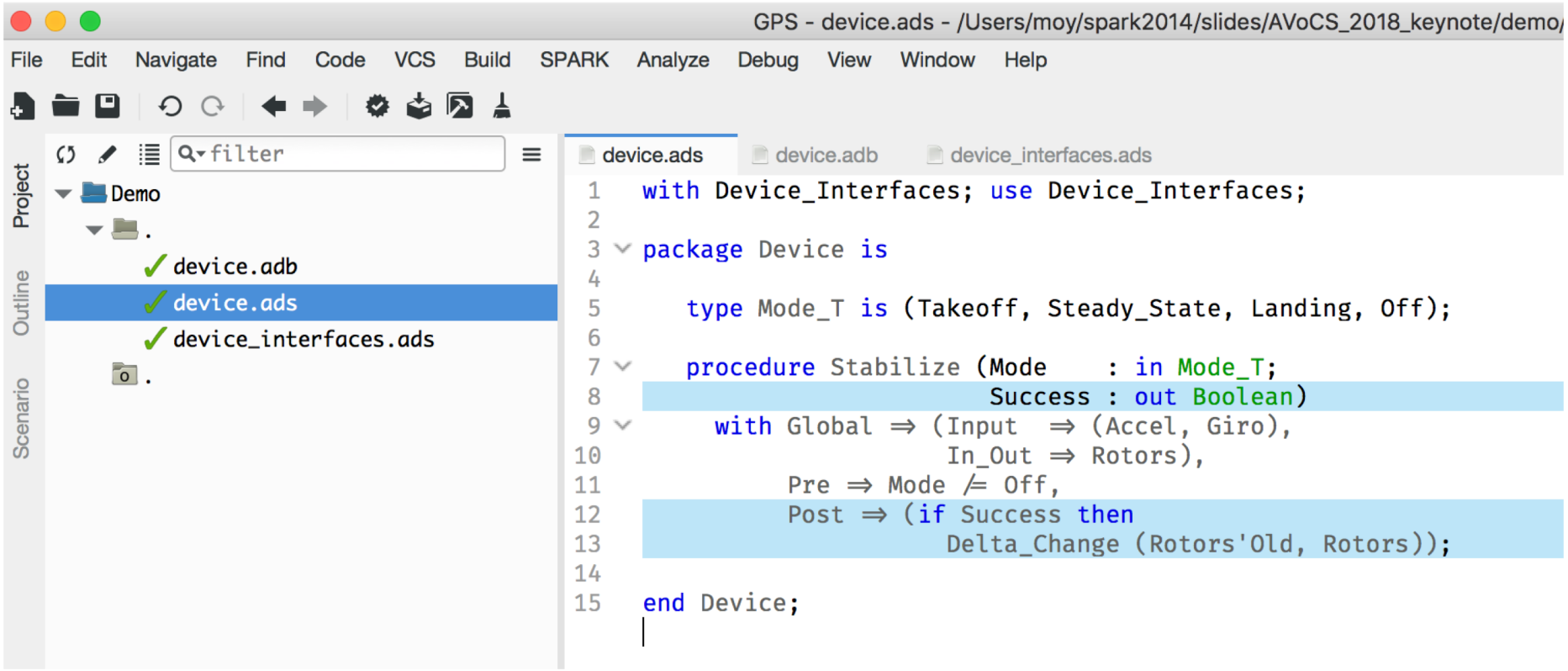
SPARK – flow analysis

```
procedure Stabilize (Mode    :  in Mode_T;
                     Success : out Boolean)
  with Global => (Input  => (Accel, Giro),
                  In_Out => Rotors);
```

SPARK – proof

```
procedure Stabilize (Mode    :  in Mode_T;
                     Success : out Boolean)
  with Pre  => Mode /= Off,
       Post => (if Success then
                   Delta_Change (Rotors'Old, Rotors));
```

SPARK – demo


GPS - device.ads - /Users/moy/spark2014/slides/AVoCS_2018_keynote/demo/
File Edit Navigate Find Code VCS Build SPARK Analyze Debug View Window Help
Project
Outline
Scenario
filter
Demo
.
device.adb
device.ads
device_interfaces.ads
.
device.ads
device.adb
device_interfaces.ads
1 with Device_Interfaces; use Device_Interfaces;
2
3 package Device is
4
5 type Mode_T is (Takeoff, Steady_State, Landing, Off);
6
7 procedure Stabilize (Mode : in Mode_T;
8 Success : out Boolean)
9 with Global ⇒ (Input ⇒ (Accel, Giro),
10 In_Out ⇒ Rotors),
11 Pre ⇒ Mode /= Off,
12 Post ⇒ (if Success then
13 Delta_Change (Rotors'Old, Rotors));
14
15 end Device;

Bounding the language

Previous SPARK based on its own grammar subset of Ada

→ many restrictions on program structure, control flow, language features

→ very hard to show value to new users before commitment!

David A. Wheeler: *“Be a good date; commitment happens later”*

Now only exclude features that make formal analysis impossible: catching exceptions, using pointers (but ownership pointers on the way)

Bounding the program

Previous SPARK based on opt-out only (with annotation #hide)

→ Need for shadow (spec) files at boundaries (libraries, hardware, OS)

→ Not adapted to retrospective analysis

Now a mix of opt-in, opt-out and opt-auto (for included specs)

Choice of boundary at top-level for mixing unit-level test & proof

→ Choice is questioned by some users requiring more flexibility

Code-level specifications and beyond

Previous SPARK based on logical specifications only (#pre, #post)

Now based on executable specifications by default (with escape hatch):

- preconditions, postconditions on subprograms
- predicates, invariants on types

Looking at expanding the specification towards design models:

- data-flow programs in Simulink
- design models in VDM, AADL+AGREE, SysML+SpeAR

Analysis at function level and beyond

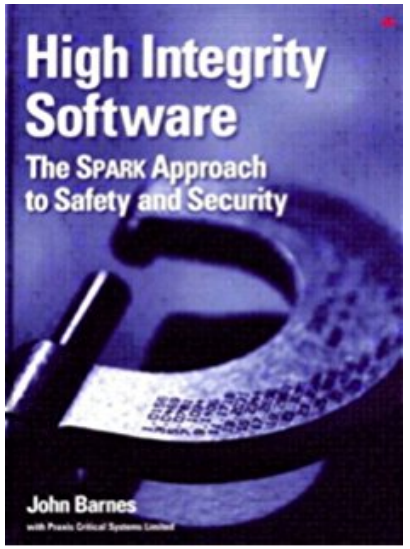
Previous SPARK: only function-level analysis (dataflow analysis or proof)

→ Requires too much specification effort

Current SPARK: mostly function-level analysis, but…

- Read/write effects are generated if needed
- Instances of generics (templates) are separately analyzed
- Read/write concurrent accesses are analyzed globally
- Subprograms may be inlined, loops may be unrolled…

Bounding the expertise



From the start, SPARK aimed at “good engineering”

Peter Amey, foreword of “High Integrity Software – the SPARK Approach to Safety and Security”, 2002:

“The migration of static analysis from a painful, post-hoc verification exercise to an integral part of a sound development process is now well-established.”

Most companies still found the expertise required too high

Example of required expertise: manual proof

```
procedure_push_5.
H1:    stack__notfull(state~) .
H2:    ptr~ < maxstacksize .
H3:    vector~ = fld_vector(state~) .
H4:    vector = fld_vector(state) .
H5:    ptr~ = fld_ptr(state~) .
H6:    ptr = fld_ptr(state) .
H7:    x >= integer__first .
H8:    x <= integer__last .
H9:    for_all(i___1: ptrs, ((i___1 >= indexes__first) and (
           i___1 <= indexes__last)) -> ((element(vector~, [
           i___1]) >= integer__first) and (element(vector~, [
           i___1]) <= integer__last))) .
H10:   ptr~ >= ptrs__first .
H11:   ptr~ <= ptrs__last .
H12:   for_all(i___1: ptrs, ((i___1 >= indexes__first) and (
           i___1 <= indexes__last)) -> ((element(vector, [
           i___1]) >= integer__first) and (element(vector, [
           i___1]) <= integer__last))) .
H13:   ptr >= ptrs__first .
H14:   ptr <= ptrs__last .
H15:   ptr = ptr~ + 1 .
H16:   vector = update(vector~, [ptr], x) .
        ->
C1:    state = append(state~, x) .
```

Verification Condition in SPARK 2005

```
1.
prove c#1 by induction.
i.
1.
unwrap h#2.
inst 1.
forw h#4.
replace h#4: sigma(1-1) by 0 using eq.
y
infer sigma(1)=1 using eq.
infer c#1 using inequals.
prove c#2 by implication.
unwrap h#2.
inst int_i_1+1.
forw h#9.
replace c#1: A by B using eq.
4.
y
y
stand c#1.
y
unwrap h#8.
inst int_i_1.
forw h#10.
replace c#1: A by B using eq.
6.
y
y
stand c#1.
y
done
exit
```

Manual Proof in SPARK 2005

Bounding the expertise

Critical change in new SPARK: specification is code

- same semantics in code and specification
- same tools to operate on specification: IDE, compiler, debugger, test
- users never look at Verification Conditions

Tool support is most needed to help users with:

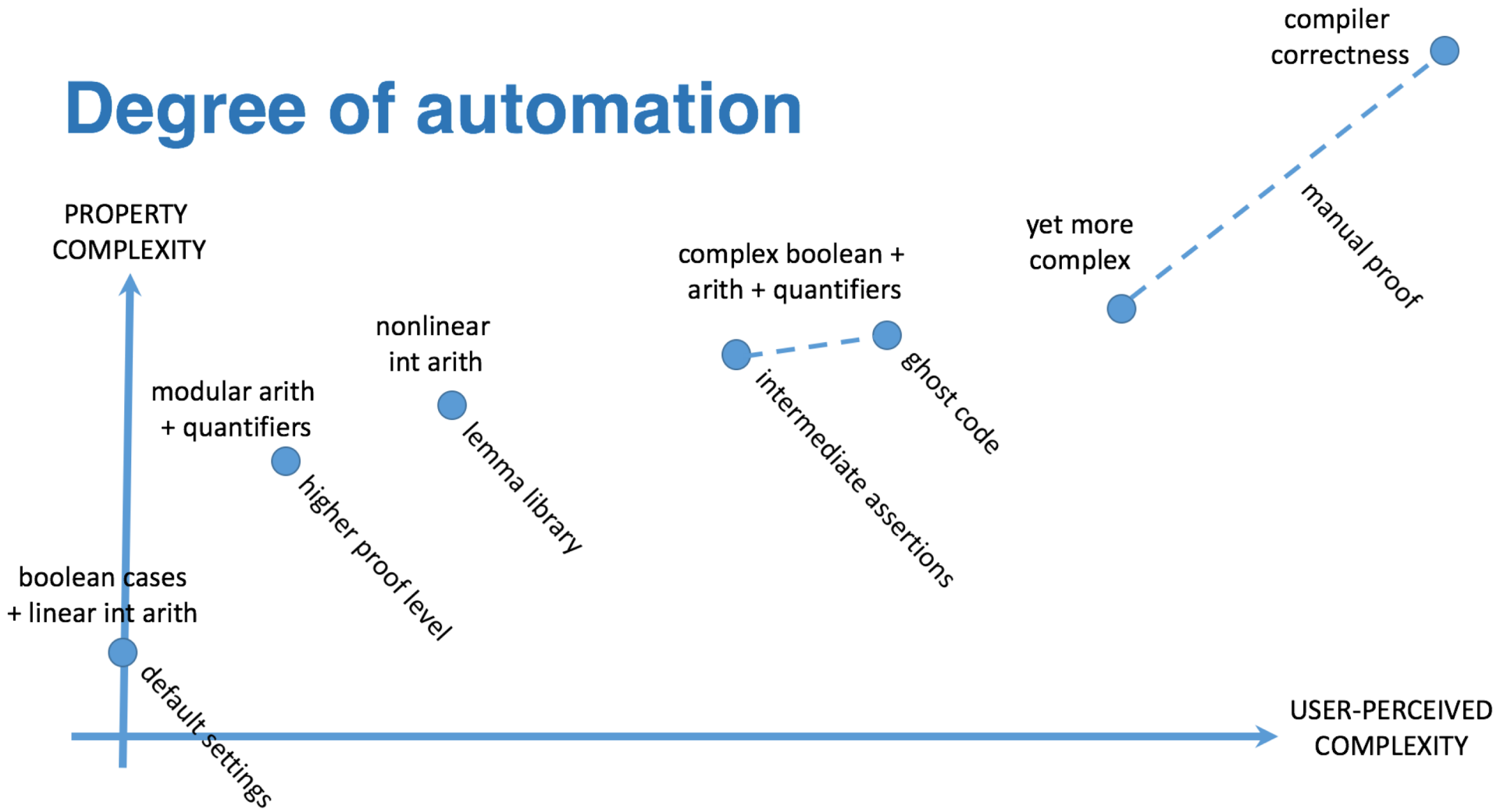
- modularity – counterexamples, safety guards, smoke detectors
- induction – loop invariant generation, loop unrolling, loop patterns
- undecidability – guidance on how to address unproved properties

From tour-de-force to run-of-the-mill

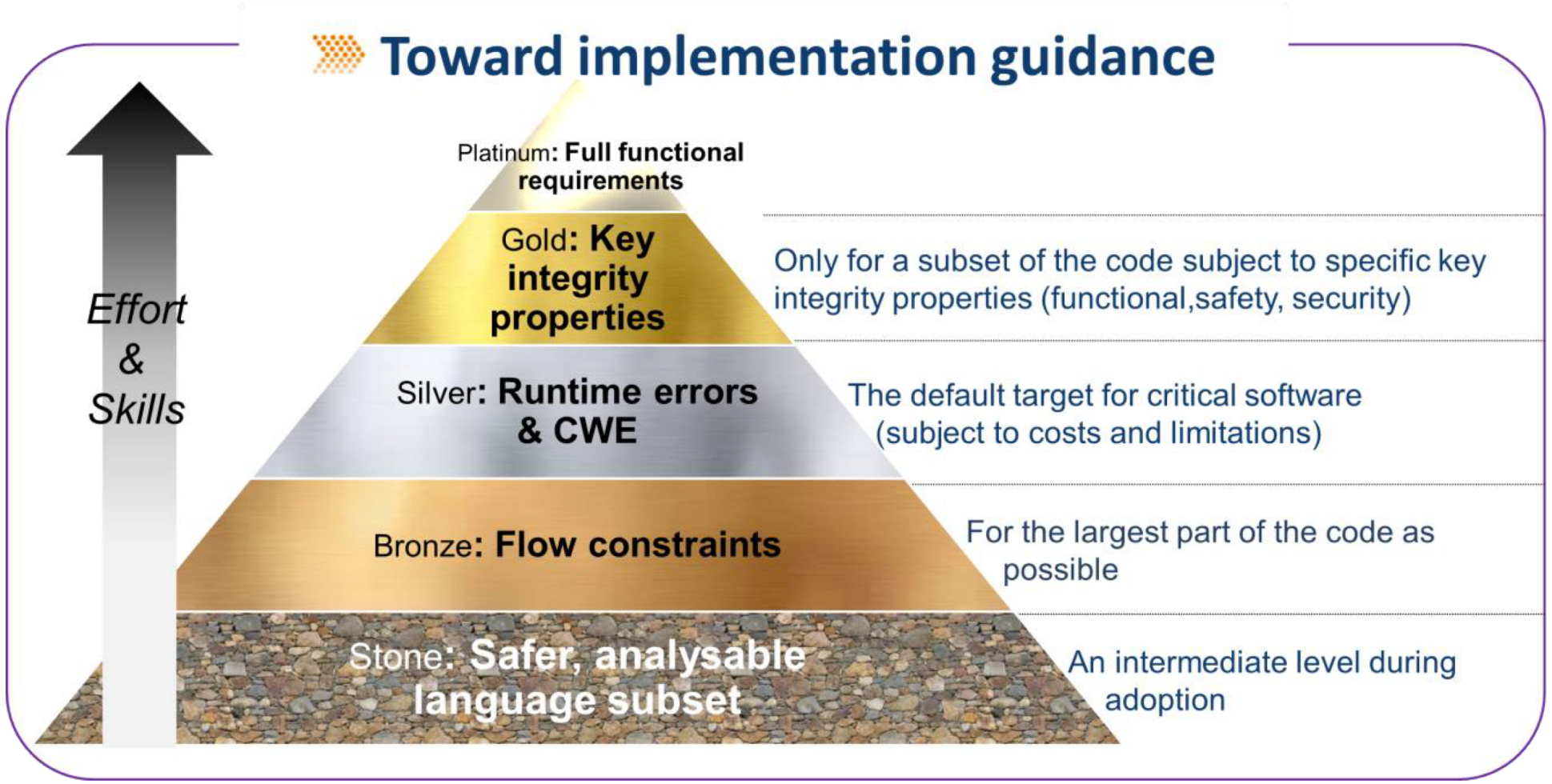
Example: Skein cryptographic hash algorithm in SPARK (Chapman, 2011)

initial version (SPARK 2005)	current version (SPARK 2014)
41 non-trivial contracts for effects and dependencies	1 – effects and dependencies are generated
31 conditions in preconditions and postconditions on internal subprograms	0 – internal subprograms are inlined
43 conditions in loop invariants	1 – loop frame conditions are generated
24 cuts to avoid combinatorial explosion	0 – no combinatorial explosion
22 hint assertions to drive proof	0 – no need
23 manual proofs	0 – no need

Building the expertise



Bounding the effort



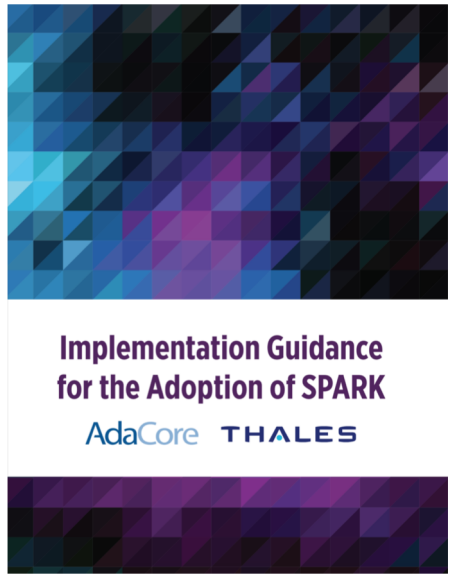
Expanding to application on legacy software

Traditional SPARK development known as “Correct-by-Construction”

→ Not possible to “sparkify” existing codebases

→ Not applicable to legacy codebases

David A. Wheeler: *“If a system works, it’s a legacy system”*



Moving towards application to legacy codebases

→ Levels of assurance are critical to support progressive adoption

Expanding the user base

Traditional SPARK customers: military, avionics, space, security

More recent applications to medical device, automotive, autonomous vehicles

→ All in the context of industrial R&D projects / POC

→ Still need for general awareness, education, case studies, etc.

Example of successful spreading: seL4 highly visible success ➔ Muen separation kernel in SPARK ➔ SPARK kernels at ANSSI, ETH Zurich, etc.