Safe and Secure Software Using SPARK

Angela Wallenburg

1 September 2017

altran

Altran – Industrial Users of Formal Methods

- Altran has around 25000 consultants
- In the UK we focus on the development of high-integrity software
- ... and we also co-develop SPARK!

1.

Problem

Why We Do It

No bugs please!

Motivating Example

Consider the following few lines of code from the original release of the Tokeneer code:

```
if Success and then
   (RawDuration * 10 <= Integer(DurationT'Last) and
    RawDuration * 10 >= Integer(DurationT'First)) then
   Value := DurationT(RawDuration * 10);
else
```

Can you see the problem? This error escaped lots of testing!

Tokeneer

- NSA-funded demonstrator of high-security software engineering
- biometric system for user verification and access control
- formal methods used: system specification and security properties in Z, implementation in SPARK
- small system (budget), about 10 kloc logical (2623 VCs)
- 2513 VCs were proven automatically (95.8 %), with 43 left to the an interactive prover and 67 discharged by review
- `http://www.adacore.com/sparkpro/tokeneer/`
- Open source (code, formal spec, project docs). Go and download!

Static Verification Goals

Ideally we would like static verification to deliver analyses which are:

- Deep (tells you something useful...)
- Sound (with no false negatives...)
- Fast (tells you *now*...)
- Complete (with as few false alarms/positives as possible...)
- Modular and Constructive (and works on incomplete programs.)

SPARK is designed with these goals in mind. Since the 80ies!

2.

What is SPARK?

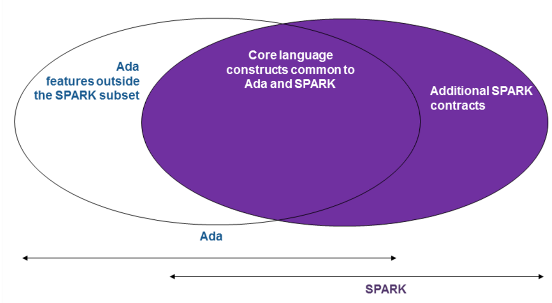
What is SPARK?

SPARK is...

- A programming language...
- A set of program verification tools...
- A design approach for high-integrity software...

All of the above!

SPARK - Analysable Subset of Ada


Ada features outside the SPARK subset
Core language constructs common to Ada and SPARK
Additional SPARK contracts
Ada
SPARK

- Exclude language features difficult to specify/verify
  - Pointers and aliasing
  - Exceptions
- Eliminate sources of ambiguity
  - Functions (not procedures) cannot have side-effects
  - Expressions cannot have side-effects

SPARK Application Domains

- Designed for embedded and real-time systems.
- Typical systems:
  - Hard real-time requirements
  - Little or no Operating System on target (no disk or VM...)
  - Fixed, known amount of storage
- Application domains:
  - The size of the problem is known in advance i.e. how many wings, engines, targets, tracks, etc.
- SPARK was not designed for building GUIs, database applications, web-servers and so on.
- Recently used for large, server-side, safety-critical system using tasking and richer data types (iFACTS).

Contracts

- *Contract*: agreement between the client and the supplier of a service
- *Program contract*: agreement between the caller and the callee subprograms

- Assigns responsibilities
- A way to organise your code
- Not a new idea (Floyd, Hoare, Dijkstra, Meyer)

Example Contract

Contracts are about *what* your code does rather than *how* it does it. Example:

```
procedure Sqrt (Input : in Integer; Res: out Natural)
with
   pre  => Input >= 0,
   post => (Res * Res) <= Input and
           (Res + 1) * (Res + 1) > Input;
```

Question: What difference do types make?

Types and Contracts

```
procedure Sqrt (Input : in Integer; Res: out Natural)
with
   pre  => Input >= 0,
   post => (Res * Res) <= Input and
           (Res + 1) * (Res + 1) > Input;
```

With the help of types:

```
procedure Sqrt (Input : in Natural; Res: out Natural)
with
   post => (Res * Res) <= Input and
           (Res + 1) * (Res + 1) > Input;
```

Less to write!

Observation: Good Fit!

SPARK offers a wide range of "built-in" contracts:

- Type ranges
- Interfaces
- Privacy
- Parameter Modes
- Generic Parameters
- Parameters not aliased
- Parameters initialised
- Strong typing ...

Strong Typing (SPARK vs C)

Example in C:

```
int A = 10 * 0.9;
```

in Ada:

```
A : Integer := 10 * Integer (0.9);
A : Integer := Integer (Float (10) * 0.9);
```

- Types are at the base of the SPARK (Ada) model
- Semantic is different from representation
- Associated with properties (ranges, attributes) and operators

Data Flow Analysis

What Is It and Why Do We Care?

Data Flow Analysis

Static analysis performed by SPARK tools, that detects *all* occurrences of:

- Use of uninitialized variables
- Ineffective statements
    - statements which update variables
    - but which have no effect on the final value of any variable
- Unused variables
- Aliasing of output parameters

Data Flow Analysis Example

```
procedure P (X, Y : in  Integer;
             Z    : out Integer)
is
   T : Integer;
begin
   T := X + 1;
   T := T + Y;
   Z := 3;
end P;
```

Which are the data flow errors here?

Data Flow Analysis Example

```
procedure P (X, Y : in  Integer;
             Z    : out Integer)
is
   T : Integer;
begin
   T := X + 1;
   T := T + Y;
   Z := 3;
end P;
```

```
warning: unused initial value of “X”
warning: unused initial value of “Y”
```

Run-Time Errors

A simple assignment statement

```
A (I + J) := P / Q;
```

Which are the possible run-time errors for this example?

Run-Time Errors

A simple assignment statement

```
A (I + J) := P / Q;
```

The following errors could occur:

1. `I + J` might overflow the base-type of the index range's subtype (arithmetic overflow)
2. `I + J` might be outside the index range's subtype
3. `P/Q` might overflow the base-type of the element type (arithmetic overflow)
4. `P/Q` might be outside the element subtype
5. `Q` might be zero

Verification Condition Generation

- Type safety (aka No run-time errors)
  - Arithmetic overflow
  - Division by zero
  - Array index range error (buffer overflow)
  - And many more...
  - ...for every statement in your program...
- Partial correctness with respect to pre- and post-conditions

Motivating Example Revisited

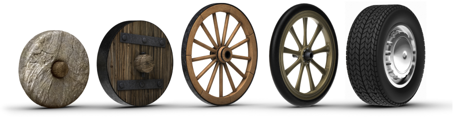
```
if Success and then
   (RawDuration * 10 <= Integer(DurationT'Last) and
    RawDuration * 10 >= Integer(DurationT'First)) then
   Value := DurationT(RawDuration * 10);
else
```

Failed VC:

```
procedure_readdduration_4.
H1:     rawduration__1 >= - 2147483648 .
H2:     rawduration__1 <= 2147483647 .
 ...
        ->
C1:     success__1 -> rawduration__1 * 10 >= - 2147483648 and
           rawduration__1 * 10 <= 2147483647 .
```

Scaling Up

SPARK has solutions for scaling up.



- Interfacing to...
  - other languages
  - other systems
  - volatiles
- Concurrency support
- Design
  - abstraction
  - refinement
  - INFORMED design approach

3.

Combining Test and Proof

SPARK 2014 Rationale...

Problem: Testing approach flawed... Proving approach flawed...

Two hurdles in the take-up of verifying compiler technology:

1. the lack of a convincing cost-benefit argument
2. the difficulty of reaching non-expert users

Solution?

Mixing Test and Proof

Executable Contracts

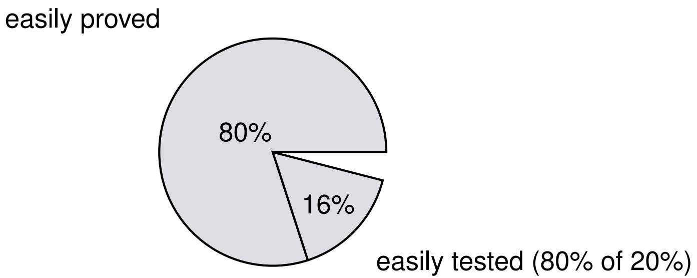
- Executable contract vs formal contract?
- The same contract interpreted in two different worlds [Cha10]:
  1. Executable Boolean expression
  2. First-order logic formula
- Ada 2012 has executable and formal contracts as part of the language

Test your contracts... or prove your contracts ... or do both!

Mixing Test and Proof

- Modular verification
- Low-level requirements expressed as contracts
- Successful execution of postcondition → test successful
- Successful proof of postcondition → low-level requirement verified for all input
- Some low-level requirements are tested, some are proved
- Is the combination as “strong” as all low level requirements tested?

Benefits of Hybrid Verification



- Helps with gradual introduction to formal proof
- The traditional 80/20% rule holds for both formal verification and testing
- More about this approach in [CKM12]

SPARK 2014 Architecture

- Joint development between Altran and AdaCore
- Built using the GNAT compiler front-end
- Why3 [BFPM11] is the intermediate proof language
- Modern implementation of data and flow analysis
- GNATprove, the end user tool, can be run from GPS IDE
- Under the hood: gnat2why translation
- Tools ship with Alt-Ergo and CVC4
- More on SPARK 2014 architecture: the convergence of compiler technology and program verification [KSD12]

4.

Sample Projects

Project: SHOLIS

1995

- Assists naval crew with the safe operation of helicopters at sea
- Shows safety limits on wind vectors, ship's roll and pitch, etc.

Project: SHOLIS

1995

- No operating system and no COTS libraries of any kind
- 27 kloc (logical) of SPARK code, 54 kloc of information-flow contracts, and 29 kloc of proof contracts
- 9000 VCs
- 75.5% proven automatically by the Simplifier
- 2200 remaining VCs proved manually using the Checker

Project: C130J
1995

- Lockheed-Martin C130J is the latest generation of the “Hercules” transport aircraft
- Mission Computer implemented in SPARK, and was subject to a large verification effort in the UK

Project: C130J

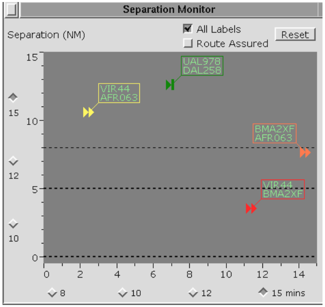
1995

- Originally started as Ada code, but was converted to SPARK
- Only flow analysis and testing to meet DO-178B Level A
- This was already very successful (used only 20% of testing budget)
- Later, UK MoD demanded proof to meet DEFSTAN 00-55
- Original spec (in Parnas-Tables) converted to pre- and post-conditions
- Proof effort was completed successfully (sorry – no stats available!)

Project: iFACTS

2006 → today

- iFACTS augments tools for en-route air-traffic controllers in the UK
- Provides electronic flight-strip management, trajectory prediction and medium-term conflict detection



Project: iFACTS

2006 $\rightarrow$ today

- In full operational service since 2011
- Formal specification in Z
- Written in SPARK – 250 kloc
- 153,000 VCs of which 98.76% are discharged automatically (user rules and review for the rest)

Project: SPARKSkein

2010

- Common misconception “Ada is slower than C because of all this safety stuff...”
- Implementation of Skein (a contender for SHA3, sadly not the winner) in SPARK
- Clean implementation (for example instead of macros, we just use normal procedures)
- After some improvements in the gcc backend, the C and SPARK implementations are equally fast

Project: SPARKSkein

2010

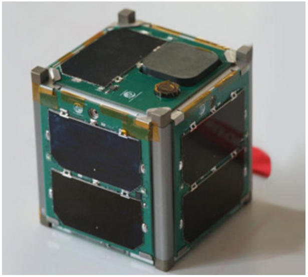
- Absence of RTE proved: originally 23 of 367 VCs proved via Checker, now 100% is proved automatically using Victor or Riposte
- Proof was difficult: non-linear arithmetic and modular types
- We found an arithmetic-overflow bug in the C reference implementation (since the SPARK implementation closely mirrored it)
- Released as free software (GPLv3)

Project: Muen Separation Kernel

2013

- Reto Buerki and Adrian-Ken Rueegsegger (both HSR – University of Applied Sciences Rapperswil)
- Separation kernel for Intel x86/64 platform
- Written in SPARK (2463 logical), and assembly (256 lines)
- Proof of absence of RTE, all 666 VCs are discharged automatically
- Again, free software (GPLv3)

Vermont Tech CubeSat



- 14 mini satellites launched in November 2013
  - NASA ELaNa IV (Educational Launch of Nanosatellites)
- the only one that remained operational until reentry
- programmed mostly by undergraduate students
- several students with little or no overlap in time
- Prof. Carl Brandon attributes success of project SPARK
- slides about project: `http://www.cubesatlab.org/`
- Next mission Lunar IceCube is a 6-Unit CubeSat mission sponsored by NASA to prospect for lunar volatiles

5.

Summary

Summary Key SPARK Benefits

1. Right First Time
2. Regulatory compliance
3. Reduced cost of testing
4. Increased trustworthiness

Regulatory Compliance Using SPARK

SPARK can be used as part of a rigorous development method for regulated industries that require certification against specific standards.
Standards come in different flavours, but SPARK has been used (at the highest levels) in all these contexts:

- Prescriptive/process-based standards *Eg. DO-178B/C (ED-12B/C); CENELEC 50128; IEC 61508; ISO 26262; BS EN 60880; Common Criteria; ITSEC; 00-55;*
- Argument-based (eg. Safety Case-based) standards *Eg. 00-56; CAP 670/SW 01*
- Standards where formal methods are mandated *Eg. Common Criteria @ EAL 7; DO-333*

Objectives of Using SPARK

- Safe Coding Standard for Critical Software
- Prove Absence of Run-Time Errors (AoRTE)
- Prove Correct Integration Between Components
- Prove Functional Correctness
- Ensure Correct Behaviour of Parameterized Software
- Safe Optimization of Run-Time Checks
- Address Data and Control Coupling
- Ensure Portability of Programs

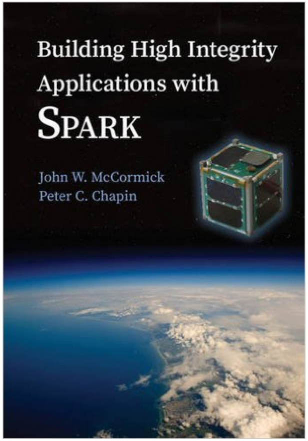
6.

Resources

SPARK - Teaching

Consider teaching SPARK:

- formal and sound
- contracts, programming language based program verification
- industrially used
- open source
- mature tools
- support for academic faculty
- code examples, problems and sample answers
- excellent books; new book 2015 (Chapin, McCormick), (Barnes' book 3rd edition)



Resources & Getting Started

- `http://www.spark-2014.org/`
- SPARK Community Page: `http://libre.adacore.com/tools/spark-gpl-edition/community/`
- GAP - GNAT Academic Program
  - Open-source, GPL release of SPARK tools
  - `http://libre.adacore.com/home/academia/`
  - Support from SPARK team for faculty
- Getting Started
  - Download the tools: `http://libre.adacore.com/download/`
  - User Guide: `http://docs.adacore.com/spark2014-docs/html/ug/`, chapter 5, SPARK tutorial, is a good start
  - SPARK 2014 Reference Manual: `http://docs.adacore.com/spark2014-docs/html/lrm/`
  - New to Ada? See `http://university.adacore.com/`

End of Talk

Thank you for your attention.

References I

[BFPM11] François Bobot, Jean-Christophe Filliâtre, Andrei Paskevich, and Claude Marché.
Why3: Shepherd your herd of provers.
In *Proceedings of the First International Workshop on Intermediate Verification Languages, Boogie*, 2011.

[Cha10] Patrice Chalin.
Engineering a sound assertion semantics for the verifying compiler.
IEEE Trans. Software Eng., 36(2):275–287, 2010.

[CKM12] Cyrille Comar, Johannes Kanig, and Yannick Moy.
Integrating formal program verification with testing.
In *Proc. Embedded Real Time Software and Systems*, Toulouse, February 2012.

References II

[Hoa03] Tony Hoare.
The verifying compiler: A grand challenge for computing research.
Journal of the ACM, 50:2003, 2003.

[KSD12] Johannes Kanig, Edmond Schonberg, and Claire Dross.
Hi-Lite: the convergence of compiler technology and program verification.
In *Proceedings of the 2012 ACM conference on High integrity language technology*, HILT '12, pages 27–34, New York, NY, USA, 2012. ACM.

altran