



SPARK – The Libre Language and Toolset for High-Assurance Software

Rod Chapman

Contents

- Correctness by Construction
- The Catch...
- Languages
- SPARK
 - Design goals and features
 - Security
 - Projects & Theorem proving performance
- What's next
- Conclusions

High-Assurance Software...

- Observation:
- We can't rely on testing alone as the primary verification activity - much too expensive and risk prone.
- Also, for the most critical systems, testing can **never** generate sufficient evidence.
 - Some high-integrity standards call for probability of failure of 10^{-9} per hour.
 - 10^9 hours is...?

High-Assurance Software

- Observation 2:
- We normally have to produce evidence of fitness-for-purpose *before* any in-service experience.
 - For the NSA, FAA, MoD for example.
- “Patch it later” is *not* possible!
- We cannot depend on the evolution of ultra-reliability over many years and releases.

Correctness by Construction (2)

- A design approach characterized by:
 - Use of **static** verification to **prevent** defects at all stages.
 - Small, verifiable design steps.
 - Appropriate use of formality.
 - “Right tools and notations for the job” approach.
 - Generation of certification/evaluation evidence as a side-effect of the development process. E.g. for a safety- or security-case.

Two independent views...

*“Those who want really reliable software will find that they must find means of **avoiding the majority of bugs to start with**, and as a result the programming process will become cheaper”*

E. Dijkstra, 1972 Turing Award Lecture

Two independent views...

“Some people argue that the easy defects are found by inspections and the difficult ones are left for testing, but I have seen no data to support this. The PSP data show that, for every defect type and for every language measured, **defect-repair costs are highest in testing and during customer use**. Anyone who seeks to reduce development cost or time must focus on **preventing or removing every possible defect before they start testing.**”

Watts Humphrey, in “PSP – A Self-Improvement Process for Software Engineers”, Addison Wesley, March 2005, page 141.

(nb...nothing has changed since 1972)

Two independent views...

- Hang on...how can you prevent and detect defects *before* testing or “observation-based” (dynamic) verification?
- By *static* methods – analysis of design artefacts *prior* to deployment or testing.
- This is *identical* to everyday practice in all mature engineering disciplines.

Two independent views...

- Data from the SEI indicates that project cost-overrun and schedule-overrun both correlate strongly with *pre-test defect rate*.
- A massive incentive to reduce defects early, *regardless* of assurance level.
 - At high-assurance levels the situation is more pronounced, since “late” defects are more expensive and time-consuming to fix.

Static Verification

- Static Verification (SV)...
 - Verification of system properties based on analysis of design artefacts (e.g. source code), *without* observation or “testing” of the running system.
 - Also known as “static analysis”
 - The opposite of “dynamic analysis” – e.g. testing
- Prevent mistakes
- Discover mistakes *sooner* rather than later

Static Verification

- SV covers a broad range of activities.
 - Manual or tool-supported?
 - Depth?
- Examples:
 - Enforcing your coding standard
 - Fagan inspections
 - “Code review”
 - Automated detection of “runtime errors”
 - “Proof” of code properties
 - Compliance with security policy

Static Verification Goals

- Ideally, we would like SV to deliver analyses which are:
 - Deep
(tells you something useful...)
 - Sound
(with no false-negatives...)
 - Fast
(tells you it now...)
 - Complete
(with as few false-positives as possible...)
 - Modular and Constructive
(and works on incomplete programs.)

The Catch...

- Our ability to perform static verification critically depends on the language or notation under analysis.
- In particular, **ambiguity** in the definition of the language severely limits what is achievable.
- Ideally, languages and notations should be as unambiguous as possible.

Ambiguity in Computing Languages

- This idea is not new...
*“... one could communicate with these machines in any language provided it was an **exact** language ...”*
*“... the system should resemble normal mathematical procedure closely, but at the same time should be as **unambiguous** as possible.”*

Ambiguity in Computing Languages

- This idea is not new...

*“... one could communicate with these machines in any language provided it was an **exact** language ...”*

*“... the system should resemble normal mathematical procedure closely, but at the same time should be as **unambiguous** as possible.”*

Alan Turing (1947)

Ambiguity in Software Engineering

- Unfortunately, ambiguity plagues us at every turn:
 - English requirements
 - UML and other “OO” notations
 - Programming languages
 - Does anyone understand C++ Templates?!?
- Machine code is often the first unambiguous representation we get, which can be **tested** but not much else...oh dear...

Programming Languages...

- Standard languages? C, C++, Java?
 - All fall down on ambiguity and therefore verifiability.
 - "Modern" language design is going the **wrong way!** E.g. OO polymorphism, exceptions etc.
- Special purpose languages?
 - Ever heard of "NewSpeak"? Nope...

Programming Languages...

- High-Integrity Language subsets?
 - Potentially combine the best of both worlds: desirable properties for H-I, using standard compilers, tools, staff etc.
 - Integrity achievable critically depends on selection of base language.
 - For the highest integrity levels, subsetting alone may not be enough. Addition of **annotations** to strengthen the language ("design by contract"TM) may be required.

So...What is SPARK?

- The “SPADE Ada Kernel”
 - What does the “R” stand for?
- A sub-language of Ada95 with particular properties that make it ideally suited to the most critical of applications:
 - Completely unambiguous
 - All rule violations are detectable
 - Formally defined
 - Tool supported
- SPARK facilitates **Correctness by Construction**

SPARK Design Goals

“Design goals....hmm...yes...
...you should definitely have some...”

Guy L Steele Jr, ACM PLDI 1994

SPARK Design Goals

- Logical Soundness
- Simplicity of Language Definition
- Expressive Power
- Security and Integrity
- Formal definition
- Verifiability
- Bounded Space and Time
- Verifiability of Compiled Code
- Minimal Runtime Library

SPARK Features

- Base language: ISO-8652:1995 Ada95
- Removes: Tasking, Generics, lots of tricky stuff...
- Limits: Some control flow structures, visibility rules etc.
- Adds: a language of annotations to allow efficient and deep static analysis, including information-flow analysis, and mathematical proof of program properties.
- Tool support: The SPARK Examiner, Simplifier and Checker

SPARK Features (2)

- SPARK is **statically free** from all
 - Aliasing
 - Function side-effects
 - Erroneous behaviour (e.g. uninitialized variables)
 - Implementation-dependent behaviour
- These analyses are all sound in polynomial time. i.e. tool is very fast!
This enables **constructive** use.

Static Analysis of SPARK

- The Examiner tool implements a number of analyses, again all in P-Time:
 - Subset checking and static semantics
 - Information flow analysis
 - Verification Condition Generation - allows proof of properties such as exception freedom, partial correctness, and safety properties.
- Theorem prover tool (the Simplifier) does a good job of proving VCs.

Exception freedom

- Exception freedom proof - why is it important?
 - Can be attempted without a formal spec., or explicit pre- and post-conditions, so is approachable.
 - Provides **evidence** that compiler-generated checks can be turned off with justification, or left on for "belt and braces."
 - Forces you to really **think** about your code. Correctness emerges.
- You mainly need CPU cycles for theorem proving - and these are cheap.

SPARK and Secure Systems

- SPARK has many properties that make it ideal for the implementation of secure, embedded systems:
 - No data-flow errors. A subtle and possibly covert source of information flow.
 - Verification of required information flow. Very useful to support system and software partitioning.
 - Proof of the absence of exceptions. Virtually free given theorem proving, and very worthwhile.
 - SPARK can be compiled with absolutely no COTS run-time library or operating system. No acquisition or evaluation problem!

SPARK and Secure Systems

- SPARK is NOT a “security vulnerability scanner” kind of tool...
- Asking “What security bugs does SPARK find?” is a dumb question...
- The best answer is “Anything that can be specified as an assertion” (including stuff to do with inputs, outputs, and the application’s domain...)

SPARK Projects

- Military Aerospace:
 - EuroFighter Typhoon - nearly all critical systems are SPARK - about 5 Million lines of code.
 - Harrier II SMS. Partly specified in Z and 100% implemented in SPARK. Approx 5000 VCs discharged in proof work.
 - SHOLIS - First Def Stan 00-55 SIL4 project. 9000 VCs proved, including top-level safety-properties, partial correctness, and exception freedom. 200 pages Z spec.
 - Aermacchi M346 – “Fly by wire” fast jet trainer

SPARK Projects (2)

- Commercial Aerospace:
 - Lockheed Martin C130J Mission Computers and Bus-Interface units.
 - Rolls-Royce – Trent 1000 FADEC and EMU, Trent 900 EMU.
 - ARINC ACAMS (aircraft health monitoring)

SPARK Projects (3)

- Security:
 - The MULTOS CA.
 - All Praxis-generated deliverables to ITSEC E6.
 - Formal Security Policy in Z
 - Functional spec in Z (500 pages)
 - Concurrency design in CSP + Model Checking
 - 100,000 lines of code (mixed-language), 3500 person-days, 27 loc per day.
 - Only 4 defects 1 year after delivery, corrected under our warranty of course!

So What's Wrong with SPARK?

- It's unfashionable, and British...
- "But we can't hire Ada programmers..."
- Selling an approach that slows coding (but speeds up whole-lifecycle) is very hard.
- Fear of formality. (Don't mention the "P" word!)
- Adopting CbyC and/or SPARK is a major life-style change for most organizations and engineers.

SPARK at Universities

- So why should you/could you use SPARK in a University degree programme?
- How about...
 - High-Integrity Software Engineering (Safety, Security...)
 - Design-by-Contact
 - Embedded/Real-Time Software
 - "Formal methods"
 - Static Analysis
- Tools are "Libre" (GPL3 licence), so full sources available.
- There's a really good book, and Tokeneer project archive case-study.

SPARK at Universities

- Current research using SPARK
 - Proof Planning (Heriot-Watt)
 - Use of SMT decision procedures with SPARK VCs (Edinburgh)
 - Proof of floating point VCs (Aston)
 - Specification refinement (Virginia)
- “Interesting” research ideas
 - Counter-example finding
 - Auto test case generation
 - Language expansion: generics, interface types, polymorphism, ???

SPARK at Universities (2)

- But no-one teaches Ada, right?
- Well...errr...the following *are* teaching SPARK (but don't tell 'em it's Ada!)
 - Manchester, (Old) York, (New) York, Virginia, Northern Iowa, Oakland, James Madison, Idaho, Roger Williams...and many more...

Final Quote

"There is still no silver bullet, but dramatic improvements in software quality can be achieved through the rigorous and systematic application of *what we already know...*"

Martyn Thomas - the founder of Praxis.

Resources

- Book: "High-Integrity Software: The SPARK Approach to Safety and Security" by John Barnes. ISBN 0-321-13616-0
- www.altran-praxis.com
 - Information
 - White papers and publications
- libre.adacore.com
 - Libre tools and Tokeneer archive.



Altran Praxis Limited

20 Manvers Street

Bath BA1 1PX

United Kingdom

Telephone: +44 (0) 1225 466991

Facsimilie: +44 (0) 1225 469006

Website: www.altran-praxis.com

Email: sparkinfo@altran-praxis.com