SPARK Language and Toolset: an intensive overview

Rod Chapman
Praxis High Integrity Systems

Agenda

- SPARK – Rationale, Goals and Language
- Coffee
- SPARK Design, Verification and Security Topics

Agenda

- SPARK – Rationale, Goals and Language
- Coffee
- SPARK Design, Verification and Security Topics

SPARK Rationale, Goals and Language

- Mini-Agenda
  - High-Integrity Software and Correctness by Construction
  - Static Verification – Goals
  - The catch…
  - SPARK is…SPARK isn't…
  - A little history…
  - SPARK language – subset and contracts

High-Integrity Software

- Characteristics:
  - Zero tolerance of defects in-the-field
    - Potential for catastrophic loss
  - Presence of a regulator and/or legal liability
  - Need to generate evidence of fitness-for-purpose *before* first deployment
    - “Patch it later” is *not* possible!
    - This is totally different from systems which can *evolve* ultra-reliability over many years and upgrades.

So what is Correctness-by-Construction (CbyC)?

- Two central principles.
- *Prevent* defect introduction throughout the lifecycle.
- Detect and remove defects *as soon as possible* after their introduction.
- Easy huh?!?

CbyC Characteristics

- A development approach characterized by:
  - Use of static verification (SV) 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 security evalution.

A Note on Testing…

- So why not just "test it to death…"?
- Program state space is *vast*. Testing only ever touches a *tiny fraction* of the paths and inputs.
- Statistics: to claim a reliability of N, how much testing to you need to do?
- Quiz: commercial aircraft aim for 1 failure in 10^9 flying hours. 10^9 hours is…?

- How much testing are you gonna do?!?
- Are you willing to stand up in court and say this?

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.
- Prevent mistakes
- Discover mistakes *sooner* rather than later (e.g. in testing!)

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.)

Static Verification – the Catch…

- There's a *big* catch…
- Our ability to deliver SV *critically* depends on the language that is being analysed.
- Most languages were *not* designed with static verification as a primary design goal. It shows!

Static Verification – the Catch…

- With contemporary unsubsetted languages, you just can't deliver all 5 goals…
  - Some interesting problems are NP-Hard or just undecideable…
  - *Ambiguity* in language definition hinders out ability to reason, or just leads to unsoundness.

Aside: The irony of language subsets and their analysis

- To gain market share, most tools have to analyse the “whole language”, or (worse…) a set of *dialects* of a language.
  - e.g. ISO 1990 C, or C “as implemented” by compilers X, Y and Z…
- But *everyone* uses a subset!
- Has your project got a coding standard?
- Does it say “you must use every language feature” ?!?

Ambiguity?

- SV is kind of like asking questions:
  - "What does this program mean?"

  Or…more specifically…
  - "Does my program have property X?" (e.g. "no buffer overflows…")
- We want *only one answer!*
- A tool that response "Don't know…" isn't much good.
- A tool that just silently gives you the *wrong answer* is dangerous!

Why (lack of) ambiguity is crucial

- The Standard definitions of all common unsubsetted programming languages are *ambiguous.*
  - E.g. unspecified and undefined behaviours in C
  - E.g. implementation-dependent and implementation-defined behaviours in Ada.
- The Standards are important, because *that's what the compilers implement*.
- Ambiguity is terrible curse from the point of view of a verification tool, since it impacts soundness and completeness.
- Here is a small example:

#include “nasty test case”

```
#include "stdio.h"
static int d;

int f(int x)
{
  d = 5;
  return (x + 1);
}

int main (int argc, char **argv)
{
  int y;
  int a[4] = {1, 2, 3, 4};

  d = 2;
  y = a[d] + f (5);
  printf ("Value of y is %d\n", y);
  return 0;
}
```

Here are a couple of clues…

```c
#include "stdio.h"
static int d;

int f(int x)
{
  d = 5;  /* Side effect */
  return (x + 1);
}

int main (int argc, char **argv)
{
  int y;
  int a[4] = {1, 2, 3, 4};

  d = 2;
  y = a[d] + f (5); /* Evaluation order dependency! */
  printf ("Value of y is %d\n", y);
  return 0;
}
```

#include "nasty test case"

- What does this program mean?
- If left-to-right evaluation order, then

```
Value of y is 9
```

- If right-to-left, then there's a buffer overflow, so behaviour is undefined.
  - GNAT Pro 3.16a (gcc 2.8.1):

    ```
    Value of y is 4198647
    ```

  - Microsoft Visual C 6.0:

    ```
    Value of y is 4198748
    ```

- Even knowing which compiler you are using doesn't help!
- What should a static analysis tool do?

SPARK…

- SPARK is…
  - A programming language designed to deliver SV that really is deep, sound, as complete as possible, fast, constructive, and modular.
  - A programming language with an *unambiguous semantics*.
  - A design philosophy for high-integrity software.

SPARK…

- SPARK is…
  - A subset of Ada…
  - A superset of Ada…
  - A totally distinct language in its own right…
  - “Eiffel on steroids…”

- All of the above!

What SPARK is *NOT*

- SPARK is *not*…
  - "just a subset" of Ada…
  - A "code scanning" or "bug finding" style static analysis tool…
  - Suitable for retrospective use on existing code…

Aside: some history

- There has been a significant growth in interest in static verification recently…
- This leads people to think that SPARK is “new”…
- Far from it…we’ve been in this game for a long time…

History

- Mid-1980s
  - UK Military starts using retrospective SV to assess aircraft software.
    - Rapid discovery that retrospective analysis is limited.
  - Program Validation Limited (PVL) founded
    - SPADE Pascal language and tools

History

- Late-1980s
  - SPARK83 designed. Based on Ada83. (Modula-2 was only other candidate base language…)
- 1990 – first big industrial project – EuroFighter. Still going!
- Early 1990s – attempt to design "SPADE C", based on ISO C90. Failed!

History

- 1995 – Praxis acquires PVL.
- 1997 – SPARK95 – based on Ada95 – a much bigger language.
- 2002 – More language growth, e.g OO stuff from Ada95.
- 2003 – RavenSPARK – “SPARK with tasking” based on Ada95 Ravenscar Profile

SPARK Design Goals…

- "Design goals…hmmm…yes….you should definitely have some!"
  - Guy L Steele Jr (ACM SIGPLAN PLDI 1994)

- The design goals of SPARK were initially laid down in the mid-1980s.

- The language and tools have grown significantly since then, but the goals have remained the same.

SPARK Design Goals…

- Logical soundness
  - The language “makes sense” as a whole, distinct language.
- Simplicity of formal language definition
  - It’s possible to write a formal semantics…
  - We did it in 1994/5 for SPARK83.
- Expressive power
  - Expressive enough to construct real-world industrial applications. Not a toy!
  - Main application domain: embedded, critical systems.
- Security
  - All language rules are statically checkable using sound algorithms.

SPARK Design Goals…

- Verifiability
  - Provision of a working Hoare-logic verification system and theorem-proving framework.
- Bounded space and time requirements
  - Programs should be amenable to the static verification of worst-case memory usage and execution time.
- Correspondence with Ada
  - So useful with standard compiler and other tools.
- Verifiability of compiled code
  - Sometimes a very difficult problem, so let's simplify it!
- Minimal run-time system requirements
  - Run-time library? What run-time library?!?
  - No requirement for *any* operating system at all…

SPARK Language

- Principal features
- Type system
- Statements
- Subprograms
- Packages
- Annotations and Contracts

Principal language features

- “Keeps the good stuff” from Ada:
  - The type system
    - Especially scalar subtypes.
  - Strict separation of specification and body for all units.
  - Packages
  - Private types
  - “Readable by default”

The SPARK Type System

- A significant simplification of Ada…
- All types are *named* (no anonymous)
- Constraints are *static*.
  - "How big" a type is (in bits) is a compile-time known value.
  - No (implicit) allocation or deallocation on a heap at all…

The SPARK Type System

- Arrays and records are first-class
  - Can be passed as parameters and returned from functions.
- Records can be “tagged” – these have OO properties of inheritance, extendability, and overriding of inherited operations.
- BUT…no polymorphism or “dynamic dispatch” of method calls…
  - How do you *statically* analyse a *dynamically* dispatched call (without looking at the whole program)?!?! Err…

The SPARK Type System

- No explicit declaration or use of access types (aka “pointers”)
  - (C programmers usually choke and fall off their chair at this point…)
- Why?
  - Permits *sound* and *efficient* aliasing analysis – a pre-requisite for Hoare-logic to work at all.
  - Ada gives us high-level parameter passing semantics – no need for pointers here!
  - Ada’s “chapter 13” allows for low-level programming (e.g. device drivers) – no pointers!
  - Array types are first class – no pointers!
  - Building linked data structures – we use arrays as “heaps” and array indexes as “references”.

The SPARK Type System

- Expressions…

- Functions may not have any side-effects, so expressions are *pure.*
- Expressions are *neutral* to evaluation order
  - it doesn't matter what order a compiler chooses, you always get the same result.
- The SPARK Examiner strictly enforces these rules.
  - Another pre-requisite for efficient flow-analysis and Hoare logic.

Statements

- Statements *may* have a side-effect.
- Simple statements:
  - Null
  - Assignment
  - Procedure call
  - Return

Statements

- Compound statements
  - Pretty much as you'd expect
  - If, case, while-loop, for-loop, general-loop
- Control-flow graphs are restricted to be *reducible* and *semi-structured* for analysis purposes:
  - Some restriction on the placement of the *return* statement.
  - No multi-level loop exits.
  - No goto statement.
- Acceptable expressive power once you get used to it!

Subprograms

- *Functions* are an abstraction of an expression – no side-effects.
- *Procedures* are an abstraction of a sequence of statements – almost always have a side-effect.

Packages

- Packages are used to group related entities together – e.g. a type and subprograms that operate on objects of that type.
- Nesting of packages and subprograms is natural and encouraged.
  - Like in Pascal!

Packages

- Child packages are allowed.
- Public child packages allow "programming by extension"
  - Very useful in combination with OO tagged types.
- Private child packages allow nested abstractions to be constructed and enforced.

Annotations and Contracts

- Subsetting is OK so far, but it's not enough to hit the "big five" goals for SV.
- We need more…
- This brings us to "annotations" – also known as "contracts" in the terminology introduced by Eiffel.

Annotations and Contracts

- Some of the annotations in SPARK are *mandatory*.
  - Without them, your program isn't SPARK at all.
- Annotations are syntactically and semantically equal in status and importance to all other language constructs.
- Saying "SPARK without the annotations" is like saying "C without the assignment statement"
  - Total nonsense!

Annotations and Contracts

- Important note: when using SPARK, there is no “Add the annotations later” phase. This doesn’t work!

- Do NOT attempt to “SPARKify” existing Ada code – not a good idea!

- Hint: require your supplier to deliver SPARK, not Ada!

Why Annotations?

- Annotations provide
  - Specification and design information about *what* your program is supposed to do.
  - *Redundant* information that can be cross-checked by tools.
  - Information *where it's needed* to enable efficient and modular analysis.

The need for annotations – an example

- Consider the following *Ada* procedure specification:

```
procedure Inc (X : in out Integer);
```

- What does this procedure do?

- What *doesn't* it do?!?

The need for annotations – an example

- Consider the following *Ada* procedure specification:

```
procedure Inc (X : in out Integer);
```

- According to the semantics of *Ada*, this procedure:
  - Has a single parameter X of type Integer, which *may* be read and/or updated.
  - *If it terminates,* then the final value of X is type Integer.
  - *May* read or update any other visible global variable in your program (its doesn't say which ones…)
  - *May* terminate with an unhandled exception (it doesn't say…)

The need for annotations – an example

- Consider the following *Ada* procedure specification:

```
procedure Inc (X : in out Integer);
```

- Pretty weak really!
- What *can* we do with this specification? Not much, other than *generate code* to call it…
- Moral: don't let compiler writers design programming languages!

The need for annotations – an example

- Here it is in bare-minimum SPARK:

```
procedure Inc (X : in out Integer);
--# global in out CallCount;
```

- In SPARK, this means:
  - It *must* read X and either update X or preserve the initial value of X
  - Ditto for global variable CallCount
  - No other global variables are accessed at all
  - *If it terminates,* then the final value of X is type Integer.
  - It never raises any exceptions
- Somewhat more useful!
- These properties will be checked when we (eventually) present the body of Inc for analysis.

Going further with annotations:

- We can (optionally) add more:

```
procedure Inc (X : in out Integer);
--# global in out CallCount;
--# derives Callcount from CallCount &
--#              X from X;
```

- This adds an *information-flow contract*, that additionally states:
  - The final value of CallCount depends on the initial value of CallCount, but NOT the initial value of X, and
  - The final value of X depends on the initial value of X, but NOT the initial value of CallCount.

Going further with annotations (2):

- We can (optionally) add (even) more:

```
procedure Inc (X : in out Integer);
--# global in out CallCount;
--# derives Callcount from CallCount &
--#               X from X;
--# post (X~ < Integer'Last ->
--#                X = X~ + 1) and
--#          (X~ = Integer'Last ->
--#                X = X~);
```

- Aha! It's a saturating Incrementer!
- Final value of CallCount remains unspecified.

An example (detection of erroneous constructs)

```
procedure Inc (X : in out Integer);
--# global in out Callcount;
```

detection of function side-effect

```
function AddOne (X : Integer)
   return Integer is
  XLocal : Integer := X;
begin
  Inc (Xlocal); -- illegal in SPARK
  return XLocal;
end AddOne;
```

Nb: this analysis is achieved without *"looking in the body"* of Inc.

An example (detection of erroneous constructs)

```
procedure Inc (X : in out Integer);
--# global in out Callcount;
```

detection of aliasing

```
Inc (CallCount); -- illegal in SPARK
```

Nb: this analysis is achieved without *“looking in the body”* of Inc*.*

Annotations: summary

- Annotations provide the information needed for verification *where it's needed* – nearly always on the *specification* of a unit.
  - SPARK *never* "looks in the body" of a unit to see what it does…
- This has a huge impact on efficiency of analysis. e.g. Aliasing analysis is sound and done in Polynomial time.
- This also allows for modular and constructive analysis, since you probably haven't written the body yet anyway!

Agenda

- SPARK – Rationale, Goals and Language
- Coffee
- SPARK Design, Verification and Security Topics

Agenda

- SPARK – Rationale, Goals and Language
- Coffee
- SPARK Design, Verification and Security Topics

SPARK Design Mini-Agenda

- Building blocks
  - Abstract data types (OO and non-OO)
  - Abstract state machines
  - Input/Output
  - Protected types and objects
  - Tasks

Abstract Data Types (ADTs)

- ADTs group a type, its basic operations, and contracts together using a package.
- These can be “tagged” (I.e. OO – like a “class” in other languages) or non-tagged.
- Let’s start with the non-OO version:

The ubiquitous “Stack” ADT specification…

```
package Stack is
   type Number is range 0 .. 20;
   type T is limited private;
   function EmptyStack (S : in T) return Boolean;
   function FullStack  (S : in T) return Boolean;

   procedure ClearStack(S : out T);
   --# derives S from ;

   procedure Push(S : in out T; X : in Number);
   --# derives S from S, X;

   procedure Pop(S : in out T;  X : out Number);
   --# derives S, X from S;
private
--# hide Stacks;
end Stack;
```

Stack ADT – refining the types

- We need to complete the "private part" with the detail of how the Stack is to be represented. For example:

```
private
   StackSize : constant := 100;
   type PointerRange is range 0 .. StackSize;
   subtype IndexRange is PointerRange range 1 .. StackSize;
   type Vector is array(IndexRange) of Number;
   type T is
      record
         StackVector  : Vector;
         StackPointer : PointerRange;
      end record;
end Stacks;
```

Stack ADT – completing the body

```
package body Stack is
   function EmptyStack(S : T) return Boolean is
   begin
      return S.StackPointer = 0;
   end EmptyStack;

   function FullStack(S : T) return Boolean is
   begin
      return S.StackPointer = StackSize;
   end FullStack;

   procedure ClearStack(S : out T)
   is
   begin
      S := T'(Vector'(others => 0), 0);
   end ClearStack;
```

Stack ADT – completing the body

```
procedure Push(S : in out T; X : in Number)
is
begin
   S.StackPointer := S.StackPointer + 1;
   S.StackVector(S.StackPointer) := X;
end Push;

procedure Pop(S : in out T; X : out Number)
is
begin
   X := S.StackVector(S.StackPointer);
   S.StackPointer := S.StackPointer - 1;
end Pop;

end Stack;
```

Tagged ADTs

- Consider a base “Object” ADT for a geometry program:

```
package Object is
   type T is tagged record
      X, Y : Float;
   end record;

   procedure Init (X, Y : in     Float;
                   Obj  :    out T);
   --# derives Obj from X, Y;

   function Area (Obj : in T) return Float;
end Object;
```

Tagged ADTs (2)

- Since Object.T is "tagged", we may inherit from it and extend it to define a new type:

```
with Object;
--# inherit Object;
package Circle is
   -- Extend Object.T with a new field Radius
   type T is new Object.T with record
      Radius : Float;
   end record;

   -- Init procedure inherited implicitly here

   -- Override inherited Area function
   function Area (Obj : in T) return Float;
end Object;
```

SPARK Design Mini-Agenda

- Building blocks
  - Abstract data types (OO and non-OO)
  - Abstract state machines
  - Input/Output
  - Protected types and objects
  - Tasks

Abstract State Machines (ASMs)

- ASMs declare a single persistent state variable (or an abstraction of several states), and operations that act on it.

- Two annotations help to specify ASMs
  - The “own variable” annotation “announces” that a package has a persistent state variable, and names it for use in other annotations.
  - The “initializes” annotation declares that an own variable is (or isn’t) initialized by the package at program startup time.

Abstract State Machines

```
package KeyStore
--# own State;
is
   type Key is …; -- whatever…

   procedure ClearAll;
   --# global out State;
   --# derives State from ;

   procedure LoadKey (K : in Key);
   --# global in out State;
   --# derives State from State, K;
end KeyStore;
```

Abstract State Machines

- Notes on KeyStore:
  - KeyStore.State is NOT initialized by default. (Absence of an initializes annotation tells us this)
  - KeyStore.Load *reads* the initial value of KeyStore.State
  - Therefore, any attempt to call KeyStore.Load *before* KeyStore.State has been properly initialized will result in a *data-flow error*
  - In other words – you *must* call KeyStore.ClearAll *before* any call to KeyStore.Load
  - Of course, the tool checks this...

Abstract State Machines

- Notes on KeyStore:
  - “derives State from ;”

  means
  - “The final value of State is derived from the initial value of <NO VARIABLES>”

  - What sort of expression has no variables?
  - Answer: A constant!

  - So...ClearAll initializes State to a well-defined and constant value.

  - The tool checks this as well…

SPARK Design Mini-Agenda

- Building blocks
  - Abstract data types (OO and non-OO)
  - Abstract state machines
  - Input/Output
  - Protected types and objects
  - Tasks

Input/Output

- It's really easy to write correct, safe, and secure code if it does no I/O!
- I/O is *hard*, especially for a static verification tool:
  - Devices may fail
  - Inputs may be malicious
  - Inputs may look OK, but be out of expected range and/or type
  - Inputs are *Volatile* – the outside world keeps changing 'em!

Input/Output

- SPARK offers a special type of own-variable for modelling I/O – the *external* variable.
- These act like a *stream* of values flowing to/from your program.
  - Aside: compare with functional programming I/O approaches.
- External own variables have a “mode” that indicates the direction of the stream of values.

Input/Output

```
-- Example:
-- A device driver for an Input device…

package Temperature
--# own in Values; -- external own var
is
   type Celsius is range 0 .. 100;

   function Read return Celsius;
   --# global in Values;

end Temperature;
```

Input/Output

- Analysis of external variables is subtly different from normal variables:
  - The information-flow analyser knows that such variables are *volatile* – i.e. reading an input twice *doesn't* necessarily yield the same value!
  - The Proof System *doesn't trust* any value coming from an external variable, so you can't prove anything until you've checked the validity of the data…
    - …it forces you to remember to validate input data…cool! ☺

SPARK Design Mini-Agenda

- Building blocks
  - Abstract data types (OO and non-OO)
  - Abstract state machines
  - Input/Output
  - Protected types and objects
  - Tasks

Protected types and objects

- In 2003, we added implemented RavenSPARK – “SPARK with Tasking”, based on the Ada95 “Ravenscar Profile”

- Ravenscar is a very simple, light concurrency model suitable for hard real-time, embedded systems.

Ravenscar Profile

- A Ravenscar program has:
  - A fixed set of library-level (I.e. “global”) tasks.
    - No nested tasks or dynamic creation of tasks…
    - Tasks may be “periodic” (e.g. activated every N milliseconds) or “sporadic” (e.g. tied to an interrupt)
  - A fixed of “protected objects” that are used for inter-task communication and synchronization.

Ravenscar Profile

- Protected Objects
  - Are basically like ASMs, but where operations are guaranteed to be executed in mutual exclusion.
  - Like a classical Hoare Monitor (a la Modula-1), but with
    - A single guarded “entry” that a single task may “block” upon.
    - Clever scheduling semantics…

Ravenscar Profile

- Scheduling in Ravenscar…
  - …Is fixed-priority pre-emptive, with mutual exclusion in PO's implemented by "immediate priority ceiling inheritance"…
- What does that mean in English?!?!
  - It's *very* simple to implement on a single processor (no semaphores at all…)
    - Implementations with evidence suitable for DO-178B Level A are commercially available and fielded.
  - Mutual exclusion and deadlock freedom are guaranteed
  - It's amenable to static analysis of schedulability – aka "Rate Monotonic Analysis"

RavenSPARK

- So…

  RavenSPARK =

    Sequential SPARK +

    Ada95 Ravenscar Tasking +

    A few more annotations +

    More verification

  e.g. inter-task information-flow analysis…

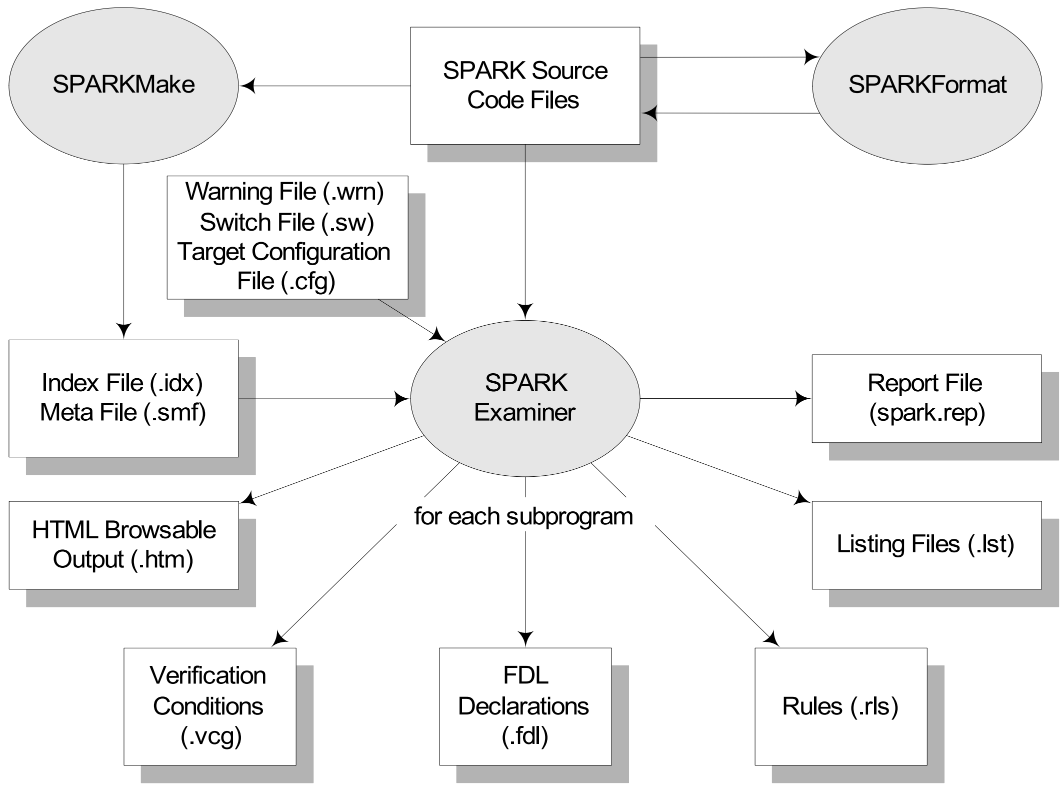
SPARK Verification and Analyses

- Mini Agenda
  - Tools
  - Examiner analyses
  - Simplifer and Checker
  - Security properties

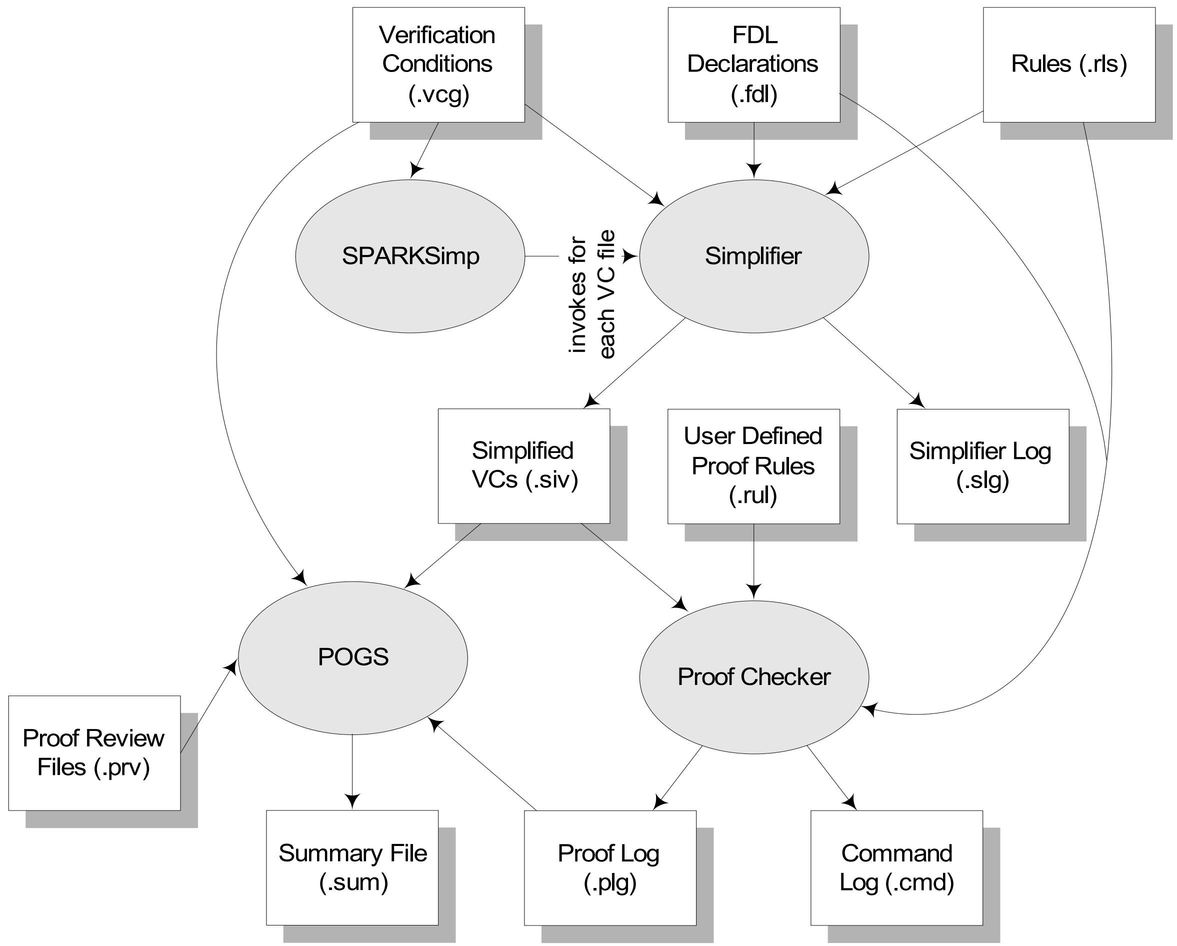
SPARK Tools

- The main tools:
  - The *Examiner* is the main static verification tool.
  - The *Simplifier* is an automatic theorem-prover.
  - The *Checker* is a user-assisted theorem-prover.
- They fit together like this:

SPARK Tools (1)



SPARK Tools (2)



SPARK Tools (3)

- There are a few other supporting tools:
  - SPARKFormat – a pretty-printer for annotations
  - SPARKMake – an analysis-order generator
  - SPARKSimp – a parallel “make” tool for the Simplifier.
  - POGS – Proof Obligation Summarizer

The Examiner

- The Examiner is kind of structured like a compiler at first…lexical analysis, parsing etc…
- Then…
  - Subset analysis
  - Static semantics (e.g. type checking)
  - Aliasing analysis
  - Side-effects analysis
  - Information flow analysis
- If *all* of the above are OK then we can enable the biggie – Verification Condition Generation (VCG).

Information flow analysis

- Based on the classic Denning/Denning paper from 1977, and extended by Bergeretti and Carré (ACM TOPLAS Jan 1985)
  - Subsumes traditional data-flow analysis, and is *sound* and *fast.*
- Eliminates *all* possibility of undefined behaviour (e.g. a read of an uninitialized variable) – another pre-requisite for the VCG to work.
- Also finds *ineffective* statements and *invariant* expressions.
- Verifies that specified information flow (the "derives" annotation) is actually implemented by the code.

The Verification Condition Generator

- Basically, this generates Verification Conditions (VCs) - conjectures about your program, the proof of which demonstrate certain properties, such as:
  – Type safety (aka “No runtime errors”)
  – Partial correctness with respect to pre- and post-conditions
  – Invariants pertaining to program state, inputs and outputs

Type Safety and "Runtime errors"

- These VCs are generated "for free" – no annotations are needed, since they are implicit in the semantics. You get VCs to show the absence of
  - Arithmetic overflow
  - Division by zero
  - Array index range error ("buffer overflow")
  - And many more…
  - …for every statement in your program…

Type Safety and “Runtime errors”

- Lots and lots of VCs…but they *should* be easy to prove…just right for an automated theorem prover!
- Quiz: sounds like a new-fangled idea, right?
  - Nope…
  - When was this approach first published?

Type Safety and "Runtime errors"

- In industrial applications, we find that the Simplifier should be able to prove over 95% of the "runtime error" VCs automatically….
- If not…
  - Your program is too complex (or just wrong)!
  - Go back and correct it!
- We do this *before* code review – why not…
  - "…We have the technology…"

SPARK and Secure Systems

- SPARK is mostly know for its use in the safety-critical arena.
- Ironic, actually, since most of the background research came from the ComSec community.
- Does it work with Secure Systems?

SPARK and Secure Systems

- Useful properties of SPARK for security:
  - Information flow analysis
    - No uninitialized variables…good… these can form a covert channel!
    - Information flow analysis can be used to verify MILS Properties e.g. “no secret info leaking to unclassified output” – we’re working on this now…

SPARK and Secure Systems

- Useful properties of SPARK for security:
  - Verification Conditions and Proof
    - Validation of Input Data is pretty much mandatory if you want to prove anything…
    - Proof of "No runtime errors" is *really* useful…
    - Proof of partial correctness can be useful as an aid to…
    - …security properties can be proved if they can be expressed as assertions.
  - What can be proved?
    - Basically, anything that can be expressed as an assertion in first-order predicate logic…

SPARK and Secure Systems

- Lack of runtime-library
  - Can be useful in high-grade applications, where evaluation of any COTS component could be impossible or prohibitively expensive.
  - SPARK answer: don't have a run-time library at all!
  - You *can* account for *every* byte of object code in the system.
- Not to everyone's taste (especially if you're used to Java… ☺)

SPARK and Secure Systems

- The real bottom line:
  1) SPARK strongly encourages you to *think* and to construct programs in a *rigorous* and *disciplined* fashion.

  2) SPARK programs exhibit a remarkably low pre-test defect rate.

- Ask the SEI about the correlation between pre-test defect rate and project over-spend and/or over-run…

SPARK and Secure Systems

- Some real SPARK security projects
  - Built by Praxis
    - MULTOS CA (published in IEEE Software)
    - Tokeneer ID Station (published in ISSSE Conference)
  - Built by others using SPARK
    - NATO C3 Agency (unpublished)
    - Rockwell Collins (see our press release)

Agenda

- One final topic…
- Adopting SPARK…

Adopting SPARK…

- …is non-trivial.
- It *isn’t* a “quick fix” that you can just plug in to your existing software process.

Adopting SPARK…

- SPARK has an impact on many other areas of software development:
  - – Design approach
  - – Review criteria and check-lists
  - – Testing (e.g. don’t do so much!)
  - – Generation of evaluation evidence
- It works best on “Green field” projects where you can start with a clean slate.

Adopting SPARK…

- SPARK is *highly* “culturally compatible” with mature software processes, especially in the world of high-integrity systems – for example:
  - CMM Levels 4+
  - SEI’s PSP and TSP

- (I took my PSP training using SPARK…it works… ☺)

Adopting SPARK - barriers

- Technically, SPARK is a no-brainer…
- Commercially, delivering <0.1 defects per kloc ought to be a no-brainer…
- The biggest barrier remains cultural and political inertia.
  - Change is seen as risky…
  - Spending more money "up front" (I.e. in design and code) scares project managers…
  - You don't get fired for just doing the same thing as the last project (no matter how badly it screwed up…)

Adopting SPARK

- "it's like dieting…"
  - Lots and lots of potions, magics, pills and "easy" solutions (and a multi-billion dollar market for them…)
  - To really change, you have to change your life-style…

Conclusions

- A precise programming language, designed for analysis completely changes the way we build software
- Emphasis on error prevention rather than error detection
- Replace “seeking suspicious constructs” with “prove system has desired properties”
- Modifies engineers’ behaviour towards rigour and discipline
- We call it “Correctness by Construction”
- It can be better and cheaper.

Final mandatory 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

Professor of Software Engineering, Oxford University (and the founder of Praxis…)

Contacts and Questions

In the USA:

Pyrrhus Software

www.pyrrhusoft.com

sparkinfo@pyrrhusoft.com

Rest of the world:

Praxis High Integrity Systems

www.praxis-his.com

www.sparkada.com

sparkinfo@praxis-his.com

Resources

- Cook, David. *Evolution of Programming Languages and Why a Language Is Not Enough to Solve Our Problems*. Crosstalk Dec 99. pp 7-12 (http://www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1999/12/cook.asp)
- Amey, Peter. *Correctness by Construction - Better Can Also be Cheaper.* Crosstalk March 2002 pp 24 -28. (http://www.praxis-his.com/pdfs/c_by_c_better_cheaper.pdf)
- ISO/IEC JTC1/SC22/WG9. Programming Languages - *Guide for the Use of the Ada Programming Language in High Integrity Systems.* (www.dkuug.dk/jtc1/sc22/wg9/n359.pdf)
- German, Andy, *Software Static Code Analysis Lessons Learned*. Crosstalk Nov 2003. pp 13-17. (http://www.stsc.hill.af.mil/crosstalk/2003/11/0311German.pdf)
- Hall, Anthony and Chapman, Roderick: “*"Correctness By Construction: Developing a Commercial Secure System“*, IEEE Software, Jan/Feb 2002, pp18-25 (http://www.praxis-his.com/pdfs/c_by_c_secure_system.pdf)
- King, Steve; Hammond, Jonathan; Chapman, Rod and Pryor, Andy: *"Is Proof More Cost Effective Than Testing?”*, IEEE Transactions on Software Engineering, Volume 26 Number 8 (http://www.praxis-his.com/pdfs/cost_effective_proof.pdf)

Resources (contd.)

- Butler, Ricky W., and George B. Finelli, eds. *"The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software."* IEEE Transactions on Software Engineering 19(1): 3-12. (http://shemesh.larc.nasa.gov/paper-nonq/nonq-paper.pdf)
- Littlewood & Strigini"*Validation of Ultrahigh Dependability for Software-based Systems*".. CACM Nov 1993 (http://www.csr.city.ac.uk/people/lorenzo.strigini/ls.papers/CACMnov93_limits/CACMnov93.pdf)
- Amey, Peter. "*A Language for Systems not Just Software*". ACM SigAda 2001. (http://www.praxis-his.com/pdfs/systems_not_just_sw.pdf)
- Chapman, Rod., Amey, Peter. *"Industrial Strength Exception Freedom".* Proceedings of ACM SigAda 2002. (http://www.praxis-his.com/pdfs/Industrial_strength.pdf)
- Chapman, Rod; Hilton, Adrian: "*Enforcing Security and Safety Models with an Information Flow Analysis Tool*". Proceedings of ACM SIGAda 2004 (http://www.praxis-his.com/sparkada/pdfs/infoflow_paper.pdf)
- Peter Amey, Rod Chapman, Neil White: "*Smart Certification Of Mixed Criticality Systems*". Ada Europe 2005 (http://www.praxis-his.com/sparkada/pdfs/Smart_Certification.pdf)
- Janet Barnes, Rod Chapman: "*Engineering the Tokeneer Enclave Protection Software*". Proceedings of IEEE ISSSE 2006

Resources (contd.)

- Amey, Peter,. and White, Neil. *"High Integrity Ada in a UML and C World".* Lecture Notes in Computer Science 3063
  A. Llamosi, A. Strohmeier (Eds.): Reliable Software Technologies – Ada-Europe 2004 9th Ada-Europe International Conference, La Palma de Mallorca, June 2004, pp. 225-236. (http://www.praxis-his.com/sparkada/pdfs/ada_uml_and_c.pdf)

- See also www.sparkada.com