Critical Systems

Lecture: *Spark -- Quick Tour*

Outline

- What is SPARK?
- Who might want to use SPARK?
- What do SPARK programs look like?
- What do SPARK specifications (procedure contracts, assertions) look like?
- What types of checking/verification can SPARK tools perform?
- Where has SPARK been used in the past?
What is SPARK?

- Developed by Praxis High Integrity Systems
  - [http://www.praxis-his.com/sparkada/](http://www.praxis-his.com/sparkada/)
- Marketed in a partnership with AdaCore
  - [http://www.adacore.com/](http://www.adacore.com/)
  - integrated with AdaCore GnatPro compiler and integrated development environment
- SPARK tools are GPL open source
  - Examiner is implemented in SPARK
Potential SPARK Users

- Developers of safety critical systems
  - since it is based on Ada, SPARK avoids many common errors associated with less safe languages such as C
  - SPARK analysis tools provide automatic checking for properties typically required in safety critical systems
    - simple static analysis: all variables initialized, no ineffective assignments, all input parameters actually used, etc.
    - automated reasoning: absence of run-time exceptions, checks of assertions and pre/post conditions
- Developers of security critical systems
  - SPARK allows specification and checking of information flow properties
- Researchers who want to implement cutting-edge verification techniques
  - in a simple, well-designed, imperative language
  - in a framework used by industry

SPARK Programs

SPARK programs are collections of annotated Ada packages

```ada
package MyPackage
  --# own G1, G2;
  is
  type MyPublicType is ...
  procedure P(in X, out Y);
  --# global in G1; out G2;
  --# derives Y from X,
  --# & G2 from G1;
  --# post Y = X + 1;
end MyPackage;
```

```ada
package body MyPackage
  is
  G1: --
  G2: --
  type MyPrivateType is ...
  procedure P(in X, out Y) is
    _P implementation...
  end P;
  begin
    _initialization...
  end MyPackage;
```

- The package specification declares the public interface of the package
  - Ada elements: types, procedures/functions, public global variables
  - SPARK elements: procedure contracts, indication of hidden global variables (own)
- The package body provides the implementations of procedures, initialization of package globals, and private types and variables
Example

**Package Specification**

```ada
critical
package Odometer
   --# own Trip, Total: Integer;
   is
   procedure Zero_Trip;
      --# global out Trip;
      --# derives Trip from;
      --# post Trip = 0;
   function Read_Trip return Integer;
      --# global in Trip;
   function Read_Total return Integer;
      --# global in Total;
   procedure Inc;
      --# global in out Trip, Total;
      --# derives Trip from Trip & Total from Total;
      --# post Trip = Trip~ + 1 and Total = Total~ + 1;
end Odometer;
```

*Note:* SPARK annotations are stated as Ada comments (“--”) with special leading character "#".

Critical Systems -- SPARK -- Quick Tour
Example

Package Specification

```
package Odometer
  --# own Trip, Total: Integer;
  is
  procedure Zero_Trip;
    --# global out Trip;
    --# derives Trip from;
    --# post Trip = 0;
  function Read_Trip return Integer;
    --# global in Trip;
  function Read_Total return Integer;
    --# global in Total;
  procedure Inc;
    --# global in out Trip, Total;
    --# derives Trip from Trip & Total from Total;
    --# post Trip = Trip~ + 1 and Total = Total~ + 1;
end Odometer;
```

Note: SPARK annotations are stated as Ada comments ("--") with special leading character ".#".

Critical Systems -- SPARK -- Quick Tour

---

Example

Package Specification

```
package Odometer
  --# own Trip, Total: Integer;
  is
  procedure Zero_Trip;
    --# global out Trip;
    --# derives Trip from;
    --# post Trip = 0;
  function Read_Trip return Integer;
    --# global in Trip;
  function Read_Total return Integer;
    --# global in Total;
  procedure Inc;
    --# global in out Trip, Total;
    --# derives Trip from Trip & Total from Total;
    --# post Trip = Trip~ + 1 and Total = Total~ + 1;
end Odometer;
```

Note: SPARK annotations are stated as Ada comments ("--") with special leading character ".#".

Critical Systems -- SPARK -- Quick Tour

---

Note: Functional contracts (pre/post conditions) state constraints on procedure inputs and outputs.

Note: Derives annotations in procedure contracts indicate how procedure flows from input parameters/globals to output parameters/globals in the procedure implementation. These information flow contracts clarify (non)dependences between variables.
Example

Package Body

```ada
package body Odometer is
  Trip, Total : Integer;

  procedure Zero_Trip is
  begin
    Trip := 0;
  end Zero_Trip;

  function Read_Trip return Integer is
  begin
    return Trip;
  end Read_Trip;

  function Read_Total return Integer is
  begin
    return Total;
  end Read_Total;

  procedure Inc is
  begin
    Trip := Trip + 1;
    Total := Total + 1;
  end Inc;

end Odometer;
```

Package bodies provide implementations of private globals and to functions/procedures.

SPARK Annotations in Detail

Starting from Ada

```ada
procedure Add(X: in Integer);
```

- Enough info to tell the compiler how to generate code to call the procedure
- Indicates X is an input parameter
- But...
  - doesn’t say what the procedure does
  - X might be ignored
  - implementation doesn’t have to “add” anything
**SPARK Annotations in Detail**

*global* indicates which global variables are used

```ada
procedure Add(X: in Integer);
--# global in out Total;
```

- Use of global variables is often presented as "bad style", yet SPARK annotations help remove some objections to the use of globals
- SPARK contract indicates...
  - exactly which globals are used and which are not
  - for those used, if they are read and/or written

**SPARK Annotations in Detail**

Post-conditions describe functional properties of outputs & guarantees made to clients

```ada
procedure Add(X: in Integer);
--# global in out Total;
--# post Total = Total~ + X;
```

- Pre-state - value of all variables just before the method begins execution
- Post-state - value of all variables just after the method completes execution

**Terminology**

- In SPARK, any *in* variable must be used; absence of *out* means *X* cannot be assigned to in procedure.
- Total is the one and only global variable used (let's us know that no other global variables are modified).
- Post-state - value of all variables just after the method completes execution
- Tilde (~) indicates value of Total in the procedure pre-state (before execution)
- Final value of Total equals initial value plus *X* (recall *X* cannot be changed in procedure)
SPARK Annotations in Detail

Pre-conditions state requirements/assumptions on clients

```plaintext
procedure Add(X: in Integer);
--# global in out Total;
--# pre X > 0;
--# post Total = Total~ + X;
```

Constraints on $X$ that the client must satisfy before calling procedure `Add`. Avoids errors in which clients call procedures with incorrect assumptions about parameter restrictions.

---

SPARK Annotations in Detail

Pre/post-conditions can be quite strong...

```plaintext
function FindSought
(A: Table; Sought: Integer) return Index;
--# pre for some M in Index => ( A(M) = Sought );
--# return Z => (( A(Z) = Sought) and
--#  (for all M in Index range 1 .. (Z - 1) =>
--#    (A(M) /= Sought))) ;
```

Simple pre-condition with existential quantification...

Post-condition constraining return value to inputs using universal quantification...
Critical Systems -- SPARK -- Quick Tour

**SPARK Annnotations**

Data and Information Flow Annotations

```ada
procedure Add(X: in Integer);
--# global in out Total;
--# derives Total from Total, X;

SPARK derives clauses are used in information flow analysis -- which captures coupling between input and output variables.
```

...for this example, the derives clause doesn’t add any new information, but...

Critical Systems -- SPARK -- Quick Tour

**SPARK Annnotations**

Data and Information Flow Annotations

```ada
procedure Add(X: in Integer);
--# global in out Total, Grand_Total;
--# derives Total from Total, X &
--#    Grand_Total from Grand_Total, X;
```

Here we capture the fact that Total is not used to calculate Grand_Total and that, e.g., Total never reads from Grand_Total.

Information annotations can play an important role in security critical systems by specifying and enforcing separated channels of information flow where no “leakage” occurs between channels.
Exposing Errors

When/how are bugs discovered in an implementation using conventional technology?

- By the compiler
  - usually straight-forward to deal with because the compiler tells us exactly what is wrong.
- At run time by a language check
  - e.g., array out of bounds error. Typically we obtain an error message saying what structure was violated and whereabouts in the program this happened.
- By testing
  - This means running various examples and poring over the (un)expected results and wondering where it all went wrong.
- By the program crashing.
  - Often difficult to diagnose.

SPARK aims to catch errors earlier in the development process by adding static analysis capabilities to be invoked before or along side of compiling. SPARK aims to move most or all of errors in last three categories into first category.

SPARK Tool Support

Correspondence With Ada

- SPARK is a true subset of Ada
  - any legal SPARK program is a legal Ada program
- Enables leverage of Ada development tools
  - compilers, debuggers, testing tools, IDEs, etc.
- Enables smooth implementation of systems with mixed criticality
  - use SPARK on critical subsystems
  - full Ada on non-critical subsystems
- Leverage existing developer knowledge of Ada
SPARK Tool Support

**Praxis static checking / verification tools**

<table>
<thead>
<tr>
<th><strong>Inputs</strong></th>
<th><strong>SPARK Tools</strong></th>
<th><strong>Outputs</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARK Code</td>
<td>SPARK Examiner</td>
<td>Flow Analysis Reports</td>
</tr>
<tr>
<td>Core Annotations</td>
<td>VC Generator Automatic Simplifier</td>
<td>Runtime Error Checks</td>
</tr>
<tr>
<td>Proofs Contexts</td>
<td>SPADE Proof Checker</td>
<td>Correctness Proofs</td>
</tr>
</tbody>
</table>

*Source: Adapted from Rod Chapman’s (Praxis HIS) slides on SPARK for AstreNet 2006*

---

The SPARK Examiner is the primary point of contact for the SPARK tool chain

- It checks conformance of the code to the rules of the kernel language.
- It checks consistency between the code and the embedded annotations by control, data and information flow analysis.
Uses of SPARK

SPARK has been (is being) used in a number of safety and security critical applications

- Several large scale security critical projects at Rockwell Collins such as the Janus crypto-graphic engine.
- iFACTS - United Kingdom next generation air-traffic control system (team of 100+ developers at Praxis).
- Avionics systems in the Lockheed C130J and EuroFighter Typhoon projects
- other safety-critical projects in railways, etc.

Brief History of SPARK

- Need for analysis motived by Bob Phillips from Royal Signals and Radar Establishment (UK)
- Group at Southampton University led by Bernard Carre became involved and developed tools for a subset of Pascal called SPADE (Southampton Program Analysis Development Environment).
- It was realized that Pascal was inadequate because it did not address separate compilation and information hiding.
- Ada was chosen, and a kernel identified (SPADE Ada Kernel = SPARK)
- SPARK semantics formally defined in Z.
- SPARK has evolved since the original definition to track the evolution of Ada (e.g., Ada 95) and to make the annotations more effective and easier to use.
Summary

SPARK is attractive to both practitioners and researchers of critical software, formal verification, and certification

- Well designed, semantically clean programming language and accompanying verification environment
- Used on a number of safety-critical and security critical projects in industry
- Supported by a leading vendor of Ada tools
- Open source -- available for experimentation
- Relatively simple language that can be used for research in formal verification

Acknowledgements

- Some material in this lecture is based on Chapter 1 from...
- A few slides are inspired by or adapted from Andrew Ireland’s (Heriot-Watt University) slides on SPARK
- A few slides are inspired by or adapted from Rod Chapman’s (Praxis HIS) slides on SPARK
- Web-site for ACM’s Special Interest Group for Ada (SIGAda) 
- Historical Information on Ada
- For more information on Ada in general for high integrity systems, see “Guide for the Use of the Ada Programming Language in High Integrity Systems” ISO/IEC TR 15942: 2000