CIS 890: Safety Critical Systems

Lecture: *Objectives of Spark*
Outline

- Brief notes about the history and use of Ada
- SPARK Overview
  - The principle of “Correctness By Construction”
  - Rationale for SPARK
  - SPARK language features
  - Tool support
  - Examples
  - History
In 1970s, US DoD was concerned by number of obsolete, hardware-dependent, non-modular languages

Commissioned a working group to formulate requirements for programming languages for DoD projects

- concluded that no existing language met the requirements
- after a competition between four proposals, a language (subsequently named “Ada”) was selected as DoD’s language mandated for new projects

Overall effort had the effect of reducing number of languages used in DoD software from 450 to 37.

1987 - DoD mandates Ada for all projects where new code is 30% or greater of total

- NATO countries had similar mandates

1997 - Mandate dropped as DoD began to emphasize use of COTS software (to reduce costs)

- had the effect of dramatically reducing support and use of Ada in the US

http://en.wikipedia.org/wiki/Ada%28programming_language%29
Ada - Characteristics

**Ada 83**
- Strong, static typing
- Modularity mechanisms (e.g., packages)
- Run-time checks
- Parallel programming constructs (tasks)
- Generics

**Ada 95**
- Object-oriented programming
- Dynamic dispatch
Ada - Use

Examples of systems that are programmed largely in Ada

- Boeing 777 -- nearly all software in Ada
- French TGV automatic train control system (Alys World Dialogue, vol. 8, no. 2, Summer 1994)
- European Space Agency GPS Receiver for space applications
- Swiss Postbank Electronice Funds Transfer system
- Commercial launch platforms (Ariane 4, Ariane 5, Atlas V)
- Most satellites and space probes from the European space agency
- US DoD weapons platforms such as Crusader, HIMARS, Tomahawk, B1-B Bomber, Patriot Missile Defense System, etc.

For more information, see the Ada Information Clearing House website (http://www.adaic.org/)

Also see Michael Feldman (Chair, SIGAda Education Working Group) list http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html

A lot of information on the early use of Ada was publicized by the Ada Information Clearing House (AdaIC) -- but some of that material is out of date.
Assessment

- In 1985, Ada was a tremendous improvement over languages being used in DoD and safety-critical systems.
- Today, Ada is still an improvement over most other languages being used in DoD and safety-critical systems!! (notably C, C++)
- Common criticisms
  - bloated feature set
    - one effect was that Ada compilers were harder to write, thus more expensive, which led to poorer adoption at low-end of the market
    - However, “Ada is a large woman, but once you get your arms around her, you learn to really love her” -- quoted by Mathew Heaney
  - early tasking model was un-implementable
  - pre-Ada 95 was not “truly object-oriented” (is that bad?, or is the main point to have useable abstractions?)
Famous Criticism

Tony Hoare strongly criticized the proposed Ada definition (three years before its subsequent release and pruning), but made the following statement...

“It is not too late! I believe that by careful pruning of the Ada language, it is still possible to select a very powerful subset that would be reliable and efficient in implementation and safe and economic to use.”

Professor Tony Hoare
1980 ACM Turing Award Lecture

...some people argue that perhaps the SPARK subset corresponds to what he might have had in mind.

Source: Andrew Ireland’s (Hariot Watt University) slides on SPARK
CIS 890 -- Objectives of SPARK (Barnes Ch. 1)
SPARK

- An annotated language (based on Ada)
  - completely unambiguous
  - free from undefined and implementation-dependent language
  - formally defined
- SPARK language is open and widely available
- SPARK can be used as a
  - programming language or
  - precise executable design language from which other languages can be generated
- Supported by analysis tools (including proof)

Source: Rod Chapman’s (Praxis HIS) slides on SPARK for AstreNet 2006
The language is designed to provide *constructive* (evidence producing) static analyses that are:

- **Deep**
  - tells you something useful about semantics...
- **Sound**
  - never says “OK” when there could be a spec violation
- **Complete**
  - as few false reports of errors as possible
- **Modular**
  - works on incomplete programs

**Source:** Rod Chapman’s (Praxis HIS) slides on SPARK for AstreNet 2006
Therefore analysis can ensure:
- freedom from language misuse
- freedom from data and information flow errors
- freedom from run-time errors
- specified safety/security properties are satisfied

That’s why the analysis tool is called the SPARK Examiner

Source: Rod Chapman’s (Praxis HIS) slides on SPARK for AstreNet 2006
Key Approach of SPARK

Unit Interface enhanced with contract information is central to SPARK

- To manage complexity (ability to effectively reason about a system) and also to drive down costs, software is developed in terms of units.
- We want to reason about those units independently:
  - independent of the implementation inside (we like to be able to change the implementation)
  - independently of the context
- This requires:
  - Interface definition separate from implementation
    - Interface summarizes “relevant” properties of unit and hides “irrelevant”
      - a key aspect of design is deciding what is relevant and what is irrelevant
  - Techniques for checking that the code complies to the interface
  - Techniques for checking composition (this may be the same thing as above).
Benefits of Contracts

Contracts enable compositional checking

- **Pre-condition**
  - `M(...) { ...
    N(.....)
    ...
    Check that N’s precondition is satisfied...
    ...
  }

- **Post-condition**
  - Assume N’s post-condition after call

- **Pre-condition**
  - No need to check body of N when called from M.

- **Post-condition**
  - Check that method conforms to its contract
Benefits of Contracts

Compositional checking is one key to scalability...

- allows each method to be checked in isolation
- allows analysis without access to procedure bodies
  - early during development
  - before programs are complete or compilable
- if a method is changed, only need to check that one method (not the entire code base)
- enables checking to be carried out in parallel
Appropriate Interface Specifications

Producing appropriate interface specification is a key element of the design process (and thus somewhat of an art)

- Important functional properties should be exposed
  - clients need understand requirements/guarantees of the unit
  - in some domains, non-functional properties such as worse-case execution time and use of system resources (e.g., threads) are also important
- Implementation details should be hidden
  - hide (if at all possible) data structure choices

...a good programming language should facilitate the above tasks!
Ada / Spark Interfaces

Ada interfaces

- Interfaces and implementations are *lexically* distinct
- Simple things such as requiring parameters to be declared as input/output go beyond other languages

SPARK interfaces

- Adds support for contracts in the form of pre/post-conditions
- Add extra annotations for, e.g.,
  - specifying dependences (*derives*)
  - indicating which global variables are accessed (*global*)
  - indicating the possibility of hidden state changes (*own*)
SPARK Contracts

Starting from Ada

```plaintext
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**

- **global** indicates which global variables are used

**procedure** 

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

- **SPARK annotations are stated in Ada comments**
- **Total is read in the implementation**
- **Total is written to (assigned) in the implementation**
- **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).**

- 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

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

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

Final value of Total equals initial value plus X (recall X cannot be changed in procedure)
Tilde (~) indicates value of Total in the procedure pre-state (before execution)

Terminology

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

**Pre-conditions state requirements/assumptions on clients**

```haskell
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`.*
SPARK Annotations

Data and Information Flow Annotations

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

Ada modes indicate direction of data flow at procedure boundaries for parameters.

SPARK also requires explicit listing of any globals used along with mode information for those globals.

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...
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.
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 invoke before or along side of compiling. SPARK aims to move most or all of errors in last three categories into first category.
Rationale for SPARK

Logical Soundness

- SPARK seeks to avoid ambiguities
  - \( Y := F(X) + G(X); \)
  - If side effects are allowed and order of execution is undefined, the result of the statement above is unclear.

- SPARK strategies for avoiding ambiguity
  - functions cannot have side effects
  - *need to add more here*
Rationale for SPARK

Simplicity of Language Definition

- Choose a language subset that makes reasoning as easy as possible
  - simpler to understand for developers
  - simpler to check/verify using automated tools
- Since SPARK is targeted to embedded domains (which usually have fixed resources), it can eliminate features like dynamic data creation that also make reasoning more difficult
- Sometimes this strategy can have unintended consequences
  - e.g., developers end up simulating heap allocated data in arrays; without associated language support, this can lead to worse problems with reasoning, etc.!
Expressive Power

- In SPARK, “expressive power” tends to mean things like providing enhanced support for...
  - abstraction & information hiding
  - specifying flow relationships between variables
  - capturing and checking software contracts
  - stronger assertions
SPARK Rationale

Security and Integrity

- Well-formedness rules (syntax, etc.) must be checkable in reasonable (polynomial) time
- No constructs such as pointers, pointer arithmetic, computed gotos, etc. that can lead to exploits
- In addition, we will see (this is a primary topic of SAnToS research) how SPARK’s emphasis on information flow annotations can be used to enforce important security policies.
SPARK Rationale

Verifiability

- Safety-critical programs have to be shown to be correct
- Thus, language constructs must allow rigorous mathematical/logical analysis
  - troublesome language features should be eliminated
  - *note:* advances in logics for heap data (e.g., separation logic) may provide a justification for expanding SPARK
- Enabling modular verification through interface contract increases scalability of analysis and reuse of both implementation and specification artifacts
SPARK Rationale

Bounded Space and Time Requirements (for targeting the domain of embedded systems)

- Safety-critical domains must avoid errors associated with exhausting resources
- Thus, it must be possible to calculate maximum amount of space statically (prior to execution)
  - no dynamic storage allocation
  - no dynamic array bounds
  - no recursion (unbounded call stack storage)
- Thus, it is desirable to be able to establish bounds on execution time
  - SPARK is designed so that manual worst-case execution time analysis is possible if manual analysis of loop bounds is carried out (the SPARK Examiner does not implement this form of analysis).
SPARK Rationale

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 Rationale

Correspondence With Ada (continued)

- SPARK omits...
  - exceptions, goto statements
  - generics
  - access (pointer) types
  - multi-tasking
    - however, multi-tasking is included in a recent extension of SPARK called RavenSPARK that is not yet widely used
Relationship between SPARK and Ada

Ada
Specialized annexes

Remainder of Ada core

The common kernel

SPARK Core annotations

SPARK proof annotations

Ada

SPARK

Source: Barnes “High Integrity Software: The SPARK Approach” (p. 11)
SPARK Rationale

**Verifiability of Compiled Code**

- SPARK does not have its own compiler suite
- Rather, it leverages the use of existing Ada compilers. The hope is that...
  - existing stringent qualification of Ada compilers plus their wide-spread use will provide correctness/reliability in compilation
SPARK Rationale

Complexity of the Run-Time System

- Final deployed executable code image includes
  - machine instructions corresponding to source code written by programmer
  - machine code implementing a run-time system for managing system resources, etc.
- In safety-critical applications, run-time system must also be verified/certified,
  - can be quite costly
- SPARK has been designed so that it demands a very small run-time system
Overarching Principles

Make programs as explicit as possible (to simplify reasoning -- both human and mechanical)

- Overloading should be avoided as far as possible.
- Scope and visibility rules should be such that each entity has a unique name at a given place.
- All subtypes (types with constraints) should be named.
- Operations on complete arrays should be explicit wherever possible, implicit operations between arrays with different bounds (sliding) should be avoided.
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.
SPARK Tool Support

SPARK Support Tools

- SPARK code
- Core Annotations
- Proof Contexts

SPARK Examiner

VC Generator
Automatic Simplifier

SPADE Proof Checker

Flow Analysis Reports
Run-time Error Checks
Correctness Proofs

CIS 890 -- Objectives of SPARK (Barnes Ch. 1)
Example

Package Specification

```plaintext
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;
```

CIS 890 -- Objectives of SPARK (Barnes Ch. 1)
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;
Another Example

SPARK automatically detects potential “problematic” aliasing

type Matrix_Index is range 0 .. 9;
type Matrix is array (Matrix_Index; Matrix_Index) of Integer;

...

procedure Multiply(X, Y: in Matrix; Z: out Matrix)
--# derives Z from X, Y;
is
begin
Z := Matrix’(Matrix_Index => (Matrix_Index => 0));  -- zero Z
for I in Matrix_Index loop
  for J in Matrix_Index loop
    for J in Matrix_Index loop
      Z(I, J) := Z(I, J) + X(I, K) * Y(K, J);
    end loop;
  end loop;
end loop;
end multiply;

...

Multiply(A,A,A);

Semantic Error: 165: This parameter is overlapped by another one which is exported.
Brief History of SPARK

- Need for analysis motivated 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 Kernal = 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.
Acknowledgements

- The 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) http://www.sigada.org/
- 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