Introduction to Ada & SPARK
In 1970s, US DoD was concerned by number of obsolete, hardware-dependent, non-modular languages.

Working group to formulate requirements for programming languages for DoD projects
- no existing language met the requirements
- one of four proposals selected as DoD’s language mandated for new projects
- called “Ada” after Ada Lovelace, world’s first programmer

http://en.wikipedia.org/wiki/Ada_Lovelace
Ada - Genesis

“…none of the evidence we have so far can inspire confidence that this language has avoided any of the problems that have afflicted other complex language projects of the past.

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.
Ada - Genesis

By 1987:

- Reduced the number of languages in DoD software from 450 to 37
- Ada was mandated for all projects where new code was 30% or more of total

Examples of systems programmed largely in Ada

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

Ada Information Clearing House http://www.adaic.org/
Also http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html
Ada - Overview

• Designed for large, long-lived applications,
  – Safety-critical / high-security
  – Embedded, real-time systems
  – e.g., commercial and military aircraft avionics,
    air traffic control, railroad systems, and medical
    devices.

• First internationally standardized (ISO)
  language (Ada 95, Ada 05, Ada 12)
Ada - Overview

• Strong typing with explicit scalar ranges
• Packages: Data abstraction
• Generic programming/templates
• Exception handling
• Concurrent programming
• Standard libraries for I/O, string handling, numeric computing, containers
Ada - Overview

- Facilities for modular organization of code
- Object orientated programming
- Systems programming
- Real-time programming
- Distributed systems programming
- Numeric processing
- Interoperability: Interfaces to other languages (C, COBOL, Fortran)
Goal: Correctness by construction

- Correct by virtue of techniques used for construction
- Design By Contract (DBC)
  - A program unit is both a
    - **client**, when using services provided by other entities
    - **supplier**, when providing services to other entities
  - Contracts specify the rights and responsibilities of both clients and suppliers:
    - Contract specifies the interface to a module: module is “correct” if it satisfies its contract
    - Compositional: rights may be assumed in order to discharge responsibilities
• Represent the effects of a sequence of components as the composition of the effects of its components

\[ \mathcal{A}(S_1; S_2; S_3; \ldots; S_n) \]

Sequence of statements: \( S_1 ; S_2 ; S_3 ; \ldots ; S_n ; \)
Abstraction & composition

Affects of a unit specified by its contract

**Pre-condition**

\[ M(\ldots,\ldots,\ldots) \{ \ldots \} \]

- Check that N’s precondition is satisfied...
- Assume N’s post-condition after call
- No need to check body of N when called from M.

**Post-condition**

\[ N(\ldots) \]

- Check that method conforms to its contract
Abstraction & composition

Affects of a unit specified by its contract

\[ \text{Pre-condition} \]

\[ \text{Post-condition} \]

- allows each method to be checked in isolation
- allows analysis without access to procedure bodies
  - early during development
  - before programs are complete or compile-able
- 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
Correctness by construction

• Need interface specs (contracts) that are:
  – Unambiguous (precise)
  – Complete (no exploitable “loop holes”)
  – Consistent (no contradictions)
  – Accurate (say what is “meant”)

• Would like static analysis that is
  – Sound (no false negatives)
  – Accurate (few false positives)
  – Deep (reveals subtle application-specific flaws)
  – Fast (scalable)
  – Modular (compositional)
What is Spark?

- Programming Language
  
  *Subset of Ada appropriate for critical systems -- no heap data, pointers, exceptions, gotos, aliasing*
What is Spark?

- Interface Specification Language
  - Aspects & pragmas for pre/post-conditions, assertions, loop invariants, information flow specifications

- Programming Language
  - Subset of Ada appropriate for critical systems -- no heap data, pointers, exceptions, gotos, aliasing
What is Spark?

**Interface Specification Language**
- Aspects & pragmas for pre/post-conditions, assertions, loop invariants, information flow specifications

**Programming Language**
- Subset of Ada appropriate for critical systems -- no heap data, pointers, exceptions, gotos, aliasing

**Automated Verification Tools**
- Flow analysis
  - static analysis to check aspects related to data flow, initialization of variables
- Dynamic analysis
  - dynamic check of pre/post-conditions, loop invariants, loop variants on an execution path
- Proof Checker
  - semi-automated framework for carrying out proof steps to discharge verification conditions.
SPARK 2014 Language: Guiding Principles

• Support the largest practical subset of Ada 2012 that is
  – Unambiguous & amenable to sound formal verification
  – DO-333 says: “…an analysis method can only be regarded as formal analysis if its determination of a property is sound. Sound analysis means that the method never asserts a property to be true when it is not true.”

• What does “unambiguous” mean in practice?
  – No erroneous behaviour, no unspecified lang. features.
  – Limit implementation-defined features to as small a set as possible, and allow these to be configured for a particular implementation.
SPARK 2014 Language: Guiding Principles

• Designed to be a “formal method” as defined by DO-333.

• Support both static and dynamic verification of contracts.

• Practical note: started with the full-blown GNAT compiler infrastructure, so many “difficult” language features are just removed or expanded out by the compiler.
The SPARK 2014 language
What is left out of Ada

• Things that make formal reasoning harder:
  – Access types (pointers)
  – Unstructured control flow (goto’s)
  – Exception handling
  – Aliasing of outputs of subprograms
  – Side-effects in expressions and functions
  – Tasks (concurrency)
  – Dynamic arrays?
Why no access types (pointers)

- Access types only make sense in connection with dynamic storage allocation.
- But dynamic allocation is a real problem, hard to prove that storage is never exhausted.
Why no goto’s?

• **Are inherently non-compositional**
  – The effect of a sequence of code cannot be represented as the composition of the effects of its components.

• **Not needed**

\[
A(S_1 ; S_2 ; S_3 ; \ldots ; S_n )
\]

\[
A(S_1) \circ A(S_2) \circ A(S_3) \circ \ldots \circ A(S_n)
\]

Sequence of statements: \( S_1 ; S_2 ; S_3 ; \ldots ; S_n ; \)
Why no exceptions handling

- Exception handling makes the control flow of a program much more complex
- Certifiable programs cannot have unexpected exceptions
Why no aliasing?

• Can lead to language ambiguities: e.g., `Multiply(A, B, A)`

```
procedure Multiply(X, Y : in Matrix; Z : out Matrix) is
begin
    Z := Matrix'(Matrix_Index => (Matrix_Index => 0));
    for I in Matrix_Index loop
        for J in Matrix_Index loop
            for K in Matrix_Index loop
                Z(I, J) := Z(I, J) + X(I,K) * Y(K, J);
            end loop;
        end loop;
    end loop;
end Multiply
```
Why no aliasing?

• Complicates analysis of procedure/function calls
  – Meaning of statements in body depends on calling context
  – Compromises compositional methods
  – e.g., \( x := y + 1; \ z := y + 1; \)
Why no side-effects in functions?

• Can lead to language ambiguities, e.g.,

\[
X : \text{Integer} := 1;
\]

\[
\text{function } F(Y : \text{Integer}) \text{ return Integer is}
\]

\[
X := Y + 1;
\]

\[
\text{return } X;
\]

\[
\text{end } F;
\]

\[
Y := F(X) + G(X)
\]

\[
\text{Y := F(X) + G(X)}
\]

\[
\text{function } G(Y : \text{Integer}) \text{ return Integer is}
\]

\[
\text{return 2 * Y}
\]

\[
\text{end } G;
\]

• Complicates analysis of function/procedure calls

\[
\text{foo( F(X), G(X) )}
\]
Why no dynamic arrays?

• Need to bound the amount of storage space a program uses to know it will function correctly
  – Sizes of arrays calculated statically
  – Bound on stack size calculated statically
Why no tasks (concurrency)?

• The effect of a sequence of code cannot be represented as the composition of the effects of its sequential components
  – Cannot reason about the effects of a module by examining its code in isolation
  – Need to consider potential “interference” from modules executed by other tasks

• Non-determinacy is a concern
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
Precise Interface Specifications

Producing appropriate interface specification is a key element of the design process

- Important properties should be exposed
  - usage 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 these tasks!
Ada / Spark Interfaces

Ada interfaces

- Interfaces and implementations are *lexically* distinct
- Parameters modes declare whether parameter is input, output, or both

SPARK interfaces

- Specify intended data and information flow
  - with Global …  with Depends …  with Abstract_State …
  - with Refined_Global …  with Refined_State …
  - with Refined_Depends …
- Specify intended behavior (for formal verification)
  - with Pre …  with Post …  pragma Assert …
  - pragma Loop_Invariant …  pragma Loop_Variant …
A SPARK program is a set Ada packages

**Package Specification**
```ada
package MyPackage
  with SPARK_mode
  is
    type MyPublicType is...
    G1: ...
    G2: ...

    procedure P(in X, out Y)
      with Global => ...,
      Pre => ...,
      Post => ...;

  end MyPackage;
```

**Package Body**
```ada
package body MyPackage
  is
    G3: ...

    type MyPrivateType is...

    procedure P(in X, out Y) is
      begin
        ...P implementation...
        end P;

    begin
      ...initialization...
    end MyPackage;
```

- Package specification declares the public interface of the package
  - Ada elements: types, procedures/functions, public global variables
  - SPARK elements: data flow and procedure contracts

- Package body provides the implementations of procedures, initialization of package globals, and private types and variables
Purpose of Contracts

• Make code clearer at specification level
  – more abstract (“what” not “how”)
• Introduce redundancy, compiler can check
• Allow error checks to be made
• Support verification
What is Spark?

**Interface Specification Language**

*Aspects & pragmas for pre/post-conditions, assertions, loop invariants, information flow specifications*

**Automated Verification Tools**

*Flow analysis*

*static analysis to check aspects related to data flow, initialization of variables*

*Dynamic analysis*

*dynamic check of pre/post-conditions, loop invariants, loop variants on an execution path*

**Proof Checker**

*semi-automated framework for caring out proof steps to discharge verification conditions.*

**Programming Language**

*Subset of Ada appropriate for critical systems -- no heap data, pointers, exceptions, gotos, aliasing*
Phase 1 of 2: frame condition computation...
Phase 2 of 2: analysis of data and information flow...
exchange.ads:5:23: warning: unused initial value of "X"
Tools in Action: Examine

```
package Exchange
  with SPARK_Mode
is

procedure Exchange(X, Y: in out Float)
  with Depends => (X => Y, Y => X);

end Exchange;
```

```
package body Exchange
  with SPARK_Mode
is

procedure Exchange(X, Y: in out Float) is begin
  X := Y;
  Y := X;

end Exchange;

end Exchange;
```

warning: unused initial value of "X"
warning: missing dependency "null => X"
warning: missing dependency "Y => Y"
warning: incorrect dependency "Y => X"
Tools in Action: Examine

```
package Exchange
  with SPARK_Mode
is
  procedure Exchange(X, Y: in out Float)
    with Depends => (X => Y, Y => X);
end Exchange;
```

```
package body Exchange
  with SPARK_Mode
is
  procedure Exchange(X, Y: in out Float)
    begin
      T := X;
      X := Y;
      Y := T;
    end Exchange;
end Exchange;
```

*Error: T is undefined*
Tools in Action: Examine

warning: unused initial value of "T"

... 

warning: missing dependency "T => X"
Phase 1 of 2: frame condition computation ...
Phase 2 of 2: analysis of data and information flow ...
Summary logged in . . .
process terminated successfully, elapsed time: 00.75s
Tools in Action: Prove

Phase 3 of 3: generation and proof of VCs ...
analyzing Exchange, 0 checks
analyzing Exchange.Exchange, 1 checks
exchange.ads:8:19: info: postcondition proved
Tools in Action: Prove

Phase 3 of 3: generation and proof of VCs ...
analyzing Inc, 0 checks
analyzing Inc.Inc, 1 checks
inc.adb:7:14: warning: range check might fail
Tools in Action: Prove

Type declarations are contractual:
- Inc has the right to assume no RTE at entry
- Inc has responsibility to guarantee no RTE while executing

type T is range -128 .. 128;

procedure Inc (X : in out T)

is begin
    X := X + 1;
end;

VC’s:
H1: x >= -128
H2: x <= 128

->
C1: x + 1 >= -128
C2: x + 1 <= 128

GNATProve

VC’s:
H1: x >= -128
H2: x <= 128

->
C1: true
C2: x <= 127

CSE 814 SPARK - Introduction
Phase 3 of 3: generation and proof of VCs...
analyzing Inc, 0 checks
analyzing Inc.Inc, 1 checks
inc.adb:7:14: info: range check proved
Tools in Action: Prove

Pre-condition is contractual:

- Inc has the right to assume
  - No RTE at entry
  - \( X < T’\text{Last} \) at entry
- Inc has responsibility to guarantee no RTE

VC’s:

\[
\begin{align*}
\text{H1: } & x \geq -128 \\
\text{H2: } & x < 128 \\
\text{C1: } & x + 1 \geq -128 \\
\text{C2: } & x + 1 \leq 128
\end{align*}
\]

GNATProve

VC’s:

\[
\begin{align*}
\text{H1: } & x \geq -128 \\
\text{H2: } & x \leq 128 \\
\text{C1: } & \text{true} \\
\text{C2: } & \text{true}
\end{align*}
\]
Acknowledgements & references


- Many slides adapted from

- Web-site for ACM’s Special Interest Group for Ada (SIGAda) [http://www.sigada.org/](http://www.sigada.org/)

- Historical Information on Ada

- Slides 16-18 from Altran Tutorial on Spark 2014.