Introduction to Spark

Christoph Ambühl
Spark Programming Language

• subset of Ada

• allows static analysis (as opposed to testing)

• proof of absence of run-time errors
  – overflows
  – out of memory
  – division by zero
  – dangling pointers
Restrictions in Spark Language

- no pointers (called access types in Ada)
- no dynamic allocation of memory (no new statement like in Java)
- no labels and goto statements
- no recursion
- array bounds must be static (known at compile time)
- no exceptions
Who Uses Spark?

- Aircraft industry
- Weapons industry
- Engine Controllers
- High Integrity Software
- when a certification requires formal methods.

US National Security Agency (NSA)
EuroFighter Consortium
BAE Systems
Ultra Electronics
Lockheed Martin
ALSTOM Signalling Ltd
Rolls Royce Marine Power
Westinghouse Signals Ltd
Smiths Industries USA
Boeing North America
General Dynamics UK
Alenia Marconi Systems
MATRA BAe Dynamics
Alenia Difesa
AerMacchi
ARINC
Mondex International
NATO C3 Agency
Data Systems and Solutions
AWE
BAE Australia
Spark Tools: Examiner

- Verifies Correct Ada syntax.
- Verifies Spark syntax.
  - lots of warnings and error messages
- Data and Information flow analysis
  - direction of information flow
  - dependencies between variables.
procedure Exchange(a,b: in out Integer)
  --# derives a from b &
  --#       b from a;
is
begin
  a:=b;
  b:=a;
end Exchange;
Information Flow Analysis
Example: Exchange

procedure Exchange(a,b:in out Integer)
  --# derives a from b &
  --# b from a;
  is
  begin
    a:=b;
    b:=a;
  end Exchange;

!!! ( 1) Flow Error : 35: Importation of the initial value of variable a is ineffective.
!!! ( 2) Flow Error : 50: b is not derived from the imported value(s) of a.
??? ( 3) Warning : 601: b may be derived from the imported value(s) of b.
procedure Exchange(a,b: in out Integer)  
  --# derives a from b &  
  --#     b from a;  
  is  
    t: Integer;  
  begin  
    t:=a;  
    a:=b;  
    b:=t;  
    end Exchange;
Information Flow Analysis
Example: Exchange

procedure Exchange(a,b: in out Integer)
  --# derives a from b &
  --#  b from a;
  is
    t: Integer;
  begin
    t:=a;
    a:=b;
    b:=t+1000;
  end Exchange;

Flow analysis is not good enough!!
Proving Partial Correctness of Subprograms

Example: Exchange Function

```pascal
procedure Exchange(X,Y: in out Integer)
  --# derives X from Y & Y from X;
  --# pre true;
  --# post X=Y~ and Y=X~;
  is
  t:INTEGER;
begin
  t := X;
  X := Y;
  Y := t;
end Exchange;
```

Initial value of X
procedure Exchange(X,Y: in out Integer)
    --# derives X from Y & Y from X;
    --# post X=Y~ and Y=X~;
    is
        t:INTEGER;
    begin
        {pre: true}
        t := X;
        X := Y;
        Y := t;
        {post: X=Y~ and Y=X~}
    end Exchange;
What is the weakest condition that has to hold before the assignment such that the \(5 \cdot x > 1\) holds after the assignment?
Assignment Axiom of Hoare Logic

\( \{5 \cdot (2 \cdot t - 5) > 1\} \quad x := 2 \cdot t - 5; \quad \{5 \cdot x > 1\} \)

\( \{\varphi[e/v]\} \quad v := e; \quad \{\varphi\} \)
Proving Partial Correctness of Subprograms
Example: Exchange Function

procedure Exchange(X,Y: in out Integer)
    --# derives X from Y & Y from X;
    --# post X=Y~ and Y=X~;
    is
        t:INTEGER;
    begin
        {pre: true}
        t := X;
        X := Y;
        Y := t;
        {post: X=Y~ and Y=X~}
    end Exchange;
Proving Partial Correctness of Subprograms

Example: Exchange Function

procedure Exchange(X,Y: in out Integer)
  --# derives X from Y & Y from X;
  --# post X=Y~ and Y=X~;
  is
    t:INTEGER;
  begin {pre: true}
    {Y=Y and X=X}   (weakest precondition)
    {Y=Y~ and X=X~} (here Y is Y~)
    t := X;
    {Y=Y~ and t=X~}
    X := Y;
    {X=Y~ and t=X~}
    Y := t;
    {post: X=Y~ and Y=X~}
  end Exchange;
Examiner/ Simplifier / Proof Checker

• Examiner generates verification conditions.

• Simplifier simplifies the verification conditions generated by the examiner. Proves the easier ones (often about 90%).

• Proof checker is an interactive tool based on Prolog for proving the remaining verification conditions.

• If all VCs are proved, the program is partially correct.
If-Then-Else Example: Max procedure

procedure Int_Max(X, Y: in Integer; Z: out Integer)
  --# derives Z from X,Y;
  --# post (X>=Y -> Z=X) and (Y>=X -> Z=Y);
  is
    begin
      if X > Y then
        Z := X;
      else
        Z := Y;
      end if;
    end Int_Max;
There is a verification condition for each execution path

procedure Int_Max(X, Y: in Integer; Z: out Integer)
  --# derives Z from X, Y;
  --# post (X>=Y -> Z=X) and (Y>=X -> Z=Y);
  is
  begin
    if X > Y then
      Z:= X;
    else
      Z:= Y;
    end if;
  end Int_Max;
Need Weakest Preconditions for Both Paths

compute weakest precondition that guarantees that the postcondition will hold if we follow the red path.

compute weakest precondition that guarantees that the postcondition will hold if we follow the blue path.
Need Weakest Preconditions for Both Paths

\[ X > Y \]
\[ \rightarrow \]
\[ (X \geq Y \rightarrow X=X) \text{ and } (Y \geq X \rightarrow X=Y) \]

\[ \text{post: } (X \geq Y \rightarrow Z=X) \text{ and } (Y \geq X \rightarrow Z=Y) \]
The verification conditions

For path(s) from start to finish:

procedure_int_max_3.
H1: true . (precondition of procedure)
H6: x > y .

->
C1: (x >= y) -> (x = x) .
C2: (y >= x) -> (x = y) .

procedure_int_max_4.
H1: true . (precondition of procedure)
H6: not (x > y) .

->
C1: (x >= y) -> (y = x) .
C2: (y >= x) -> (y = y) .

A -> (B -> C)
is equiv.
(A and B) -> C
function Pow(A,B :in Integer) return Integer
  --# pre B>=0;
  --# return A**B;
  is
    c,d,e:INTEGER;
  begin
    c:=1; d:=A; e:=B;
    while e/=0 loop
      c:=c*d;
      e:=e-1;
    end loop;
    return c;
  end Pow;
function Pow(A,B :in Integer) return Integer
  --# pre B>=0;
  --# return A**B;
is
  c,d,e:INTEGER;
begin
  c:=1; d:=A; e:=B;
  while e/=0 loop
    --# assert c*(d**e)=A**B;
    c:=c*d;
    e:=e-1;
  end loop;
  return loop;
end Pow;
From start to finish: precondition implies postcondition if we do not enter the loop.

From start to assert statement: precondition implies assertion

From assert to assert: assertion implies assertion in next loop.

From assert to finish: assertion implies postcondition
\{ \text{assert } c \cdot (d^{**e}) = A^{**B} \text{ and } e-1 \neq 0 \} \\
\rightarrow \\
\quad (c \cdot d) \cdot (d^{**(e-1)}) = A^{**B} \}

\{ c \cdot (d^{**e}) = A^{**B} \rightarrow \\
\quad (e-1 \neq 0 \rightarrow \\
\quad \quad (c \cdot d) \cdot (d^{**(e-1)}) = A^{**B}) \}

\{ e-1 \neq 0 \rightarrow \\
\quad (c \cdot d) \cdot (d^{**(e-1)}) = A^{**B} \}

\{ e-1 \neq 0 \rightarrow \\
\quad c \cdot (d^{**(e-1)}) = A^{**B} \}

\{ e \neq 0 \rightarrow \\
\quad c \cdot (d^{**e}) = A^{**B} \}

\{ c \cdot (d^{**e}) = A^{**B} \}

\text{assert } c \cdot (d^{**e}) = A^{**B} \\
\quad c := c \cdot d \\
\quad e := e - 1 \\
\quad \text{return } c
Four VCs

H1: \( b \geq 0 \).
H14: \( \neg (b \not= 0) \).
\( \rightarrow \)  
C1: \( 1 = a^{\ b} \).

H1: \( c \cdot d^{\ e} = a^{\ b} \).
H20: \( e - 1 \not= 0 \).
\( \rightarrow \)  
C1: \( c \cdot d \cdot d^{\ (e - 1)} = a^{\ b} \).

H1: \( b \geq 0 \).
H14: \( b \not= 0 \).
\( \rightarrow \)  
C1: \( 1 \cdot a^{\ b} = a^{\ b} \).
C2: \( b > 0 \).

H1: \( c \cdot d^{\ e} = a^{\ b} \).
H20: \( \neg (e - 1 \not= 0) \).
\( \rightarrow \)  
C1: \( c \cdot d = a^{\ b} \).
Nested Subprograms

procedure Exchange(X, Y : in out Integer)
--# derives X from Y & Y from X;
--# pre true;
--# post X = Y~ and Y = X~;
is
  T : Integer;
begins
  T := X;
  X := Y;
  Y := T;
end Exchange;

procedure CAB(A, B, C : in out Integer)
--# derives A from C & B from A & C from B;
--# post A = C~ and B = A~ and C = B~;
is
begin
  Exchange(A,B);
  Exchange(A,C);
end CAB;
Nested Subprograms
Proof of CAB

procedure Exchange(X, Y : in out Integer)
--# derives X from Y & Y from X;
--# pre true;
--# post X = Y~ and Y = X~;

procedure CAB(A, B, C : in out Integer)
--# derives A from C & B from A & C from B;
--# post A = C~ and B = A~ and C = B~;
begin
   begin
      Exchange(A,B);
      Exchange(A,C);
   end CAB;

   3 VCs
Nested Subprograms
Proof of CAB

procedure Exchange(X, Y : in out Integer)
--# derives X from Y & Y from X;
--# pre true;
--# post X = Y~ and Y = X~;

procedure CAB(A, B, C : in out Integer)
--# derives A from C & B from A &
--# C from B;
--# post A = C~ and B = A~ and C = B~;
is begin
  Exchange(A,B);
  Exchange(A,C);
end CAB;

weakest precond.:

H12:  true .
H13:  a__1 = b .
H14:  b__1 = a .
H27:  true .
H28:  a__2 = c .
H29:  c__2 = a__1.

->

C1:  a__2 = c~ .
C2:  b__1 = a~ .
C3:  c__2 = b~ .
Nested Subprograms
Proof of CAB

procedure Exchange(X, Y : in out Integer) 
--# derives X from Y & Y from X;  
--# pre true; 
--# post X = Y~ and Y = X~;

procedure CAB(A, B, C : in out Integer) 
--# derives A from C & B from A &  
--# C from B; 
--# post A = C~ and B = A~ and C = B~;

is
begin

Exchange(A,B);  
\{ a \} b \} c

Exchange(A,C);  
\{ a_1 \} b_1 \} c_2
end CAB;

For path(s) from start to finish:

procedure_cab_5.
H1:  true .
H12:  true .
H13:  a1 = b .
H14:  b1 = a .
H27:  true .
H28:  a2 = c .
H29:  c2 = a1 .

->

C1:  a2 = c .
C2:  b1 = a .
C3:  c2 = b .
Proof Functions

```plaintext
--# function Fact (N : Integer) return Integer;

function Factorial (N : Integer) return Integer
--# pre N >= 0;
--# return Fact(N);
is
    r : Integer;
    i : Integer;
begin
    r := 1;
    i := 0;
    loop
        exit when i=N;
        i := i+1;
        r := r*i;
        --# assert i > 0 and r = Fact(i);
    end loop;
return r;
end Factorial;
```

Rules for Proof Checker to prove VCs:

- fact(0) may_be_replaced_by 1.
- fact(N) may_be_replaced_by N*fact(N-1) if N>0.
function Array_Max2(A: in Vector) return Integer
  --# return X => (for all M in Index_Range => (A(M)<=X)) and
  --# (for some M in Index_Range => (A(M)=X));
  is
    i:Integer;
    M:Integer;
  begin
    M:=Vector'First;
    i:= Vector'First +1;
    while i<=Vector'Last loop
      --#assert i in Index_Range and M in Index_Range and
      --#(for all J in Index_Range range Vector'First..i-1
      --#  => (A(J)<=A(M)));
      if A(M)<A(i) then
        M:=i;
      end if;
    i:=i+1;
    end loop;
    return A(M);
  end Array_Max2;
Conclusions

• Spark is used for quite large projects in Industry.

• VC generation works fine and is fast.

• Simplifier often proves 90%-95% of VCs.

• VCs can have hundreds of hypotheses in practice.

• Proof Checker is bottleneck. Specialist skills needed to use it.