ADACORE U

SPARK 2014: Systems Programming

University.adacore.com

Systems Programming – What is it?

- **Bare metal programming**
  - bare board applications (no Operating System)
  - Operating Systems (ex: Muen separation kernel)
  - device drivers (ex: Ada Drivers Library)
  - communication stacks (ex: AdaCore TCP/IP stack)
- **Specifics of Systems Programming**
  - direct access to hardware: registers, memory, etc.
  - side-effects (yes!)
  - efficiency is paramount (sometimes real-time even)
  - hard/impossible to debug

Systems Programming – How can SPARK help?

- **SPARK is a Systems Programming language**
  - same features as Ada for accessing hardware (representation clauses, address clauses)
  - as efficient as Ada or C
- **Side-effects can be modeled in SPARK**
  - reads and writes to memory-mapped devices are modeled
  - concurrent interactions with environment are modeled
- **SPARK can help catch problems by static analysis**
  - correct flows, initialization, concurrent accesses
  - absence of run-time errors and preservation of invariants

Systems Programming – A trivial example

```
X : Integer with Volatile,
  Address => Ext_Address;

procedure Get (Val : out Integer)
  with Global  => (In_Out => X),
       Depends => (Val => X,
                   X   => X);

procedure Get (Val : out Integer) is
begin
   Val := X;
end Get;
```

X is volatile

X is also an output

output X depends on input X

X is only read

Volatile Variables and Volatile Types

- **Variables whose reads/writes cannot be optimized away**

- **Identified through multiple aspects (or pragmas)**
  - aspect Volatile
  - but also aspect Atomic
  - and GNAT aspect Volatile_Full_Access
  - all the above aspects can be set on type or object

- **Other aspects are useful on volatile variables**
  - aspect Address to specify location in memory
  - aspect Import to skip definition/initialization

```
type T is new Integer with Volatile;

X : Integer with Atomic, Import, Address => … ;
```

Flavors of Volatile Variables – Async_Readers / Async_Writers

- **Boolean aspects describing asynchronous behavior**
  - Async_Readers if variable may be read asynchronously
  - Async_Writers if variable may be written asynchronously

- **Effect of Async_Readers on flow analysis**

- **Effect of Async_Writers on flow analysis & proof**
  - always initialized, always has an unknown value

```
X : Integer with … Async_Readers;
Y : Integer with … Async_Writers;

procedure Set is
   U, V : constant Integer := Y;
begin
   pragma Assert (U = V);
   X := 0;
   X := 1;
end Set;
```

✗ unprovable assertion

✓ assignment not useless

Flavors of Volatile Variables – Effective_Reads / Effective_Writes

- **Boolean aspects distinguishing values & sequences**
  - Effective_Reads if reading the variable has an effect on its value
  - Effective_Writes if writing the variable has an effect on its value

- **Effect of both on proof and flow dependencies**
  - Final value of variable is seen as a sequence of values it took

```
X : Integer with … Effective_Writes;
Y : Integer with … Effective_Reads;

procedure Set with
  Depends => (X => Y,
              Y => Y)
is
begin
   X := Y;
   X := 0;
end Set;
```

✓ despite final assign

✓ although only read

Combinations of Flavors of Volatile Variables

- **All four flavors can be set independently**
  - Default for Volatile/Atomic is all four True
  - When some aspects set, all others default to False
- **Only half the possible combinations are legal**
  - Async_Readers and/or Async_Writers is set
  - Effective_Reads = True forces Async_Writers = True
  - Effective_Writes = True forces Async_Readers = True
  - sensor: AW=True
  - actuator: AR=True
  - input port: AW=True, ER=True
  - output port: AR=True, EW=True

Constraints on Volatile Variables

- **Volatile variables must be defined at library level**

- **Expressions (and functions) cannot have side-effects**
  - read of variable with AW=True must appear alone on rhs of assign
  - a function cannot read a variable with ER=True

```
procedure Read_All is
   Tmp : Integer := 0;
begin
   Tmp := Tmp + AR;
   Tmp := Tmp + AW;
   EW := Tmp;
   Set (ER);
end Read_All;

function Read_ER return Integer is
   Tmp : Integer := ER;
begin
   return Tmp;
end Read_ER;
```

❌ AW not alone on rhs

❌ ER not alone on rhs

❌ ER output of Read_ER

Constraints on Volatile Functions

- **Functions should have mathematical interpretation**
  - a function reading a variable with AW=True is marked as volatile with aspect Volatile_Function
  - calls to volatile functions are restricted like reads of Async_Writers

```
function Read_Non_Volatile
  return Integer;
--  reads AR, AW, EW

function Read_Volatile
  return Integer
  with Volatile_Function;
--  reads AR, AW, EW

function Read_ER
  return Integer
  with Volatile_Function;
--  reads ER
```

✗ not a volatile function

✓ OK for volatile function

✗ ER output of Read_ER

State Abstraction on Volatile Variables

- **Abstract state needs to be identified as “External”**
- **Flavors of volatility can be specified**
  - Default if none specified is all True

```
package P1 with
   Abstract_State =>
     (S with External)
is …
```

✓ always OK

```
package P2 with
   Abstract_State =>
     (S with External =>
        (Async_Writers,
         Effective_Reads))
is …
```

✓ OK if refined into AW, ER

✗ not OK if refined into AR, EW

Constraints on Address Attribute

- **Address of volatile variable can be specified**

```
X : Integer with Volatile, Address => … ;

Y : Integer with Volatile;
for X’Address use … ;
```

- **Address attribute not allowed in expressions**

- **Overlays are allowed**
  - GNATprove does not check absence of overlays
  - GNATprove does not model the resulting aliasing

```
X : Integer := 1;
Y : Integer := 0
  with Address => X'Address;
pragma Assert (X = 1);
```

❌ assertion wrongly proved

Can something be known of volatile variables?

- **Variables with Async_Writers have no known value**
- **… but they have a known type!**
  - type range, ex: 0 .. 360
  - type predicate, ex: 0 .. 15 | 17 .. 42 | 43 .. 360
- **Variables without Async_Writers have a known value**
- **GNATprove also assumes all values are valid (X'Valid)**

```
X : Integer with Volatile, Async_Readers;

procedure Read_Value is
begin
   X := 42;
   pragma Assert (X = 42);
end Read_Value;
```

✓ proved

Other Concerns in Systems Programming

- **Software startup state → elaboration rules**
  - SPARK follows Ada static elaboration model
  - … with additional constraints for ensuring correct initialization
  - … but GNATprove follows the relaxed GNAT static elaboration
- **Handling of faults → exception handling**
  - raising exceptions is allowed in SPARK
  - … but exception handlers are SPARK_Mode=>Off
  - … typically the last-chance-handler is used instead
- **Concurrency inside the application → tasking support**
  - Ravenscar and Extended_Ravenscar profiles supported in SPARK

Quiz

Is this correct? 1/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
X : Integer with Volatile,
  Address => Ext_Address;

procedure Get (Val : out Integer)
  with Global  => (Input => X),
       Depends => (Val => X);

procedure Get (Val : out Integer) is
begin
   Val := X;
end Get;
```

Is this correct? 1/10

✖ NO

```
X : Integer with Volatile,
  Address => Ext_Address;

procedure Get (Val : out Integer)
  with Global  => (Input => X),
       Depends => (Val => X);

procedure Get (Val : out Integer) is
begin
   Val := X;
end Get;
```

X has Effective_Reads set by default, hence it is also an output

Is this correct? 2/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
X : Integer with Volatile, Address => Ext_Address,
  Async_Readers, Async_Writers, Effective_Writes;

procedure Get (Val : out Integer)
  with Global  => (Input => X),
       Depends => (Val => X);

procedure Get (Val : out Integer) is
begin
   Val := X;
end Get;
```

Is this correct? 2/10

✓ YES

✓

```
X : Integer with Volatile, Address => Ext_Address,
  Async_Readers, Async_Writers, Effective_Writes;

procedure Get (Val : out Integer)
  with Global  => (Input => X),
       Depends => (Val => X);

procedure Get (Val : out Integer) is
begin
   Val := X;
end Get;
```

`X has Effective_Reads=False, hence it is only an input`

Is this correct? 3/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
Speed : Float with Volatile, Async_Writers;
Motor : Float with Volatile, Async_Readers;

procedure Adjust with
   Depends => (Motor =>+ Speed)
is
   Cur_Speed : constant Float := Speed;
begin
   if abs (Cur_Speed) > 100.0 then
      Motor := Motor - 1.0;
   end if;
end Adjust;
```

Is this correct? 3/10

✓ YES

```
Speed : Float with Volatile, Async_Writers;
Motor : Float with Volatile, Async_Readers;

procedure Adjust with
   Depends => (Motor =>+ Speed)
is
   Cur_Speed : constant Float := Speed;
begin
   if abs (Cur_Speed) > 100.0 then
      Motor := Motor - 1.0;
   end if;
end Adjust;
```

✓

Speed is an input only, Motor is both an input and output. Note how the current value of Speed is first copied to be tested in a larger expression.

? Is this correct? 4/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
Raw_Data : Float with Volatile,
  Async_Writers, Effective_Reads;
Data     : Float with Volatile,
  Async_Readers, Effective_Writes;

procedure Smooth with
   Depends => (Data => Raw_Data)
is
   Data1 : constant Float := Raw_Data;
   Data2 : constant Float := Raw_Data;
begin
   Data := Data1;
   Data := (Data1 + Data2) / 2.0;
   Data := Data2;
end Smooth;
```

Is this correct? 4/10

✖ NO

```
Raw_Data : Float with Volatile,
  Async_Writers, Effective_Reads;
Data     : Float with Volatile,
  Async_Readers, Effective_Writes;

procedure Smooth with
   Depends => (Data => Raw_Data)
is
   Data1 : constant Float := Raw_Data;
   Data2 : constant Float := Raw_Data;
begin
   Data := Data1;
   Data := (Data1 + Data2) / 2.0;
   Data := Data2;
end Smooth;
```

Raw_Data has Effective_Reads set, hence it is also an output

Is this correct? 5/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
type Regval is new Integer with Volatile;
type Regnum is range 1 .. 32;
type Registers is array (Regnum) of Regval;

Regs : Registers with Async_Writers, Async_Readers;

function Reg (R : Regnum) return Integer is
   (Integer (Regs (R)))
  with Volatile_Function;
```

Is this correct? 5/10

NO

```
type Regval is new Integer with Volatile;
type Regnum is range 1 .. 32;
type Registers is array (Regnum) of Regval;

Regs : Registers with Async_Writers, Async_Readers;

function Reg (R : Regnum) return Integer is
  (Integer (Regs (R)))
  with Volatile_Function;
```

Regs has Async_Writers set, hence it cannot appear as the expression in an expression function

Is this correct? 6/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
type Regval is new Integer with Volatile;
type Regnum is range 1 .. 32;
type Registers is array (Regnum) of Regval;

Regs : Registers with Async_Writers, Async_Readers;

function Reg (R : Regnum) return Integer
  with Volatile_Function
is
   V : Regval := Regs (R);
begin
   return Integer (V);
end Reg;
```

Is this correct? 6/10

NO

```
type Regval is new Integer with Volatile;
type Regnum is range 1 .. 32;
type Registers is array (Regnum) of Regval;

Regs : Registers with Async_Writers, Async_Readers;

function Reg (R : Regnum) return Integer
  with Volatile_Function
is
   V : Regval := Regs (R);
begin
   return Integer (V);
end Reg;
```

Regval is a volatile type, hence variable V is volatile and cannot be declared locally

Is this correct? 7/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
type Regval is new Integer with Volatile;
type Regnum is range 1 .. 32;
type Registers is array (Regnum) of Regval;

Regs : Registers with Async_Writers, Async_Readers;

function Reg (R : Regnum) return Integer
  with Volatile_Function
is
begin
   return Integer (Regs (R));
end Reg;
```

Is this correct? 7/10

✓ YES

```
type Regval is new Integer with Volatile;
type Regnum is range 1 .. 32;
type Registers is array (Regnum) of Regval;

Regs : Registers with Async_Writers, Async_Readers;

function Reg (R : Regnum) return Integer
  with Volatile_Function
is
begin
   return Integer (Regs (R));
end Reg;
```

Regs has Effective_Reads=False hence can be read in a function.
Function Reg is marked as volatile with aspect Volatile_Function.
No volatile variable is declared locally.

Is this correct? 8/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
package P with
  Abstract_State => (State with External),
  Initializes => State
is …

package body P with
  Refined_State => (State => (X, Y, Z))
is
   X : Integer with Volatile, Async_Readers;
   Y : Integer with Volatile, Async_Writers;
   Z : Integer := 0;
end P;
```

Is this correct? 8/10

✗ NO

```
package P with
  Abstract_State => (State with External),
  Initializes => State
is …

package body P with
  Refined_State => (State => (X, Y, Z))
is
   X : Integer with Volatile, Async_Readers;
   Y : Integer with Volatile, Async_Writers;
   Z : Integer := 0;
end P;
```

X has Async_Writers=False, hence is not considered as always initialized. As aspect Initializes specifies that State should be initialized after elaboration, this is an error.

Note that is allowed to bundle volatile and non-volatile variables in an external abstract state.

Is this correct? 9/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
type Pair is record
   U, V : Natural;
end record
  with Predicate => U /= V;

X : Pair with Atomic, Async_Readers, Async_Writers;

function Max return Integer with
  Volatile_Function,
  Post => Max'Result /= 0
is
   Val1 : constant Natural := X.U;
   Val2 : constant Natural := X.V;
begin
   return Natural'Max (Val1, Val2);
end Max;
```

Is this correct? 9/10

NO

```
type Pair is record
   U, V : Natural;
end record
  with Predicate => U /= V;

X : Pair with Atomic, Async_Readers, Async_Writers;

function Max return Integer with
  Volatile_Function,
  Post => Max'Result /= 0
is
   Val1 : constant Natural := X.U;
   Val2 : constant Natural := X.V;
begin
   return Natural'Max (Val1, Val2);
end Max;
```

X has Async_Writers set, hence it may have been written between the successive reads of X.U and X.V

? Is this correct? 10/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
type Pair is record
   U, V : Natural;
end record
  with Predicate => U /= V;

X : Pair with Atomic, Async_Readers, Async_Writers;

function Max return Integer with
  Volatile_Function,
  Post => Max'Result /= 0
is
   P    : constant Pair := X;
   Val1 : constant Natural := P.U;
   Val2 : constant Natural := P.V;
begin
   return Natural'Max (Val1, Val2);
end Max;
```

Is this correct? 10/10

✓ YES

```
type Pair is record
   U, V : Natural;
end record
  with Predicate => U /= V;

X : Pair with Atomic, Async_Readers, Async_Writers;

function Max return Integer with
  Volatile_Function,
  Post => Max'Result /= 0
is
   P    : constant Pair := X;
   Val1 : constant Natural := P.U;
   Val2 : constant Natural := P.V;
begin
   return Natural'Max (Val1, Val2);
end Max;
```

values of P.U and P.V are provably different, and the postcondition is proved

university.adacore.com