Programming for Safety Critical Systems

Andy Edmunds
ae2@ecs.soton.ac.uk

Background

- *Division and Modulus are simple, right?*

  Division and Modulus for Computer Science

  Daan Leijen. [daanj]

- **The Choice of Computer Languages for use in Safety-Critical System (1991)** by W.J. Cullyer, S.J. Goodenough and B.A. Wichmann. [cgw]

- **An informal survey - languages used in SCSs (2006)** by C. Johnson. [cj]

… more references:

- By 2006: *mostly* Ada, C/C++,- Assembly Code?
  - **Software for Dependable Systems: Sufficient Evidence? (2007)** by Daniel Jackson et al. [dj]

  - **An Introduction to Safety Critical Systems. (2011)** by IPL, Information Processing Ltd. [ipl]

Certification

- Certification is required in many industries (which is hard to obtain for Java)
- It requires proof of adherence to prescribed standards, for engineering processes, and artefacts. See again [ipl]

- Formal Methods
- is recommended in some standards
- mandated in others e.g. Def-Stan 00-55.

In summary:

- Typically for Embedded Systems

- **C**/**C++** is used – despite all the criticism.

  - Use of *Guidelines* like MISRA-C/C++ can mitigate shortcomings.

  - Certification is used to check compliance to various safety standards.

But What about Java?

Java obtained a bad reputation

- Its Memory Model was broken!
- The specification was vague.
- Garbage collection for limited memory?

- In particular for critical systems (many, as we Pointed out are embedded):
  - The JVM
    - is an unnecessary processing overhead.
    - is an additional source of errors.
    - portability through byte-code + interpreter is not necessary.

Can’t we do something with Java?

- open Safety Critical Java [oscj]
- from Purdue
- JML Extended – Safe JML
- 'Safe' JVM
- Translates to c and uses gcc?

- Java Modelling Language [jml]
- Design by contract style.
- Use an extended static checker to ensure conformity.
- Runtime assertion checking.

A JML Example
(source: IBM JML *Tutorial*)

A specification modelling popping off a stack

```
/*@
  @ public normal_behavior
  @   requires ! isEmpty();
  @   ensures
  @     elementsInQueue.equals(((JMLObjectBag)
  @          \old(elementsInQueue))
  @                        .remove(\result)) &&
  @     \result.equals(\old(peek()));
  @*/
Object pop() throws NoSuchElementException;
```

How about using Ada?

- A `better’ language for SCS.
  - Strictly typed.
  - fewer bugs [nai]
  - Language designed for Real-time and High Reliability.
  - Projects delivered faster than with C. [nai]
  - It is well established, particularly in Defense.
  - It has a safe subset (SPARKAda).

- The GNAT Compiler is free.

Language Elements

- Separation of
  - Specification (.ads) and
  - implementation (.adb)
- Packages: Spec and Body.
- Tasks.
- Protected Objects.
- Procedures and functions.
- Task entry and rendezvous.

Ada Task Spec

```
procedure Heating_Controller_Main is
  task T is                        -- interface (like a header in C)
    entry Sense_PressInc(state_inc:  out boolean);
    entry Sense_PressDec(state_dec:  out boolean);
    ...
  end T;
```

- A Task is like a thread.

- Entries allow access to internal state (Rendezvous)

Ada Task Body

```
task body T is                          -- denotes the implementation
  ts1 : Integer := 0; ...               -- local declarations part
  begin
    loop
      if ((inc_flag = false)) then
        ...
        put("ts1 =  "); put(ts1); New_Line;
        select                          -- rendezvous communication
          accept Sense_PressInc(state_inc:  out boolean) do
          begin
            state_inc := inc_flag;
          end;
          end Sense_PressInc;
        or
          accept Sense_PressDec(state_dec:  out boolean) do
          ...
```

Ada Protected Spec

```
package Shared is
  protected type Shared_Object is          -- interface
    procedure Get_Temperature1(tm:  out Integer);
    procedure Set_Temperature(tm:  in Integer);
    …
end Shared_Object;

private                                    -- encapsulated data
     ctm : Integer := 20;
     shss : boolean := false;
     cttm : Integer := 20;
end Shared;
```

Ada Protected Body

```
package body Shared is
  protected body Shared_Object is      -- implementation

  procedure Get_Temperature1(tm:  out Integer) is
    begin
      tm := cttm;
    end Get_Temperature1;

    procedure Set_Temperature(tm:  in Integer) is
    begin
      cttm := tm;
    end Set_Temperature;
  end Shared_Object;
end Shared;
```

SPARKAda

- For the highest assurance of correctness
  - standard **Ada** is still not good enough!

- But there is a safe subset … **SPARKAda**
- Annotated Ada Specification (.ads)
- Additional Static Checking.
- Design by Contract.
- Pre and Post Conditions.
- Uses Proof to show that a program satisfies its contracts.

Static Checks: Data Flow

```
procedure Get_Temperature1(tm:  out Integer);
```

- The SPARK Examiner:
  - Performs language conformance checks.
  - Does *data flow analysis.*
- Data flow: parameter checks:
  - '**out**' parameters are *initialised*
  - … and not read before that.
  - '**in**' parameters are not assigned to, but read.
  - '**in out**' parameters are assigned to, and read.
- Same check for Global Variables.

Information Flow

- Annotate the **specification** (.ads) before implementation.

```
procedure Get_Temperature1(tm:  out Integer);
    --# derives tm from cttm;
    begin
        tm := cttm;
    end Get_Temperature1;
```

- Check that the implementation uses *tm* and *cttm* Correctly. (*tm* appears on the left of an assignment, and *cttm* on the right).

Proof: Pre and Post Conditions

```
procedure Get_Temperature1(tm:  out Integer);
--# derives tm from cttm;
--# pre cttm  >  0
--# post tm = cttm
```

- A more precise specification. Is it implemented correctly by *tm := cttm* ?
- **Using proof –** The examiner generates Verification Conditions (to be discharged).
- For the post condition we would need to show: using the Generalised Substitution for assignment,

[tm := cttm] tm = cttm

Which is *true*, since,

cttm = cttm

So ...

- We have looked at
  - shortcomings of some languages.
  - ways to address program correctness, where errors are introduced by the *programming* activity.

- Using automatic code generation we could improve this situation.

- Formal modelling can help to highlight/remove the *systematic* errors.

Tomorrow's session ...

- Using Event-B tools

  - we can generate code automatically.

  - we obtain the benefits of formal modelling.

>> It would be useful to review <<
'***Shared Event Decomposition***'.