Programming for Safety Critical Systems

Andy Edmunds
ae2@ecs.soton.ac.uk

Some History

- Literature Review.

- **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)** referring to slides by C. Johnson. [cj]

… and so on:

- 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]

In summary:

- **Ada** is frequently used, but not exclusively.

- **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.

  - Lack of a large Ada skill-base is a factor in hindering widespread use.

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 of which 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?

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

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

- ... and others

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;
```

Certification

- Certification is required in many industries (which is hard for Java)
- This 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.

Why Use – Ada?

- A `better' language for SCS.
  - Strictly typed.
  - fewer bugs [nai]
  - Language designed for Real-time and High Reliability.
  - Projects delivered faster 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;
```

- This is like a thread, or a process
- Entries may have different implementations

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 more ...
- SPARKAda – An **Ada** Subset
- 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

- Annotations for information flow analysis.
  - Annotate the specification (.ads).

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

- Check that the implementation uses *tm* and *cttm* correctly in *tm := cttm*;
  - That is, *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 detailed specification. Is it implemented correctly by *tm := cttm* ?
- **Using proof –** The examiner generates verification conditions to be discharged. For the example we would need to show (given the hypotheses) that:

Using GSL for assignment, we have

[tm := cttm] tm = cttm

substituting, we obtain

cttm = cttm

So ...

- We have highlighted ways to address program correctness, where errors are introduced by the *programming* activity.

- If we use automatic code generation we could improve this situation.

Tomorrow's session ...

- Using Event-B tools we can generate code Automatically
  - and formal modelling can also help to highlight/remove systematic errors.

>>>>> It will be very useful to understand,

'***Shared Event Decomposition***'