'Avoiding Programming' for Safety Critical Systems

Andy Edmunds

ae2@ecs.soton.ac.uk
In the last Session ...

- We showed how errors can be introduced by the **programming** activity.

- We showed some examples of attempts to improve programming languages.

- We suggested that Event-B could help.
What can 'we' do?

- With Event-B tools (+ Tasking Event-B)
  - we can generate code automatically.
  - formal modelling helps to highlight/remove systematic errors.

- Using automatic code generation we
  - do less coding.
  - encourage re-use (using code templates).
How to do this ...

- As you know, Event-B is modelling, not programming.
- Developers focus on the design, not code.

- To produce source code, we add 'extra' information to Event-B.
  … and still we need a trusted compiler.
  … and, ideally, 'certify' the translator.

- We could still verify the code with JML, SPARKAda etc
Targets for Translation ...

Targets: Ada, OpenMP C, FMI C, Java ....
- The approach is suitable for
  - single threaded implementations.
  - multi-threaded implementations (using decomposition).
- not currently OO, but could be done.

Current Focus is on embedded systems.
- 'Implementable' controller code
- Environment simulation.
Event-B at the implementation level

- Tasking Event-B
- Event-B models:
  - Controller Tasks (AutoTask Machine).
  - Shared Protected Objects (Shared Machine).
  - Environment Tasks (Environ Machine).

- Use Decomposition to partition the system.
- Shared Event Style.
- Shared Events model communication, between
  - Controller tasks and Environment tasks.
  - Controller tasks and Protected Objects.
  - Environment tasks and Protected Objects.
Where Tasking Event-B Fits in.
Shared Event Decomposition

Tool-driven decomposition
Event 'Synchronization'

Machine m
Variables v1 v2
events
e =
any p, q
where g(v1,v2,p,q)
then a(v1,v2,p,q)
end

Machine ma
Variables v1
events
e a =
any p
where g(v1,p)
then a(v1,p)
end

Machine mb
Variables v2
events
e b =
any q
where g(v2,q)
then a(v2,q)
end

Composed Machine
Preparing for Decomposition

A Problematic Decomposition

Machine M0

Cannot Decompose !!

Evt = A := B

Vars A: x, B: x

Composed Machine

Refines

Machine M1

Vars A: x
Evt = A := ?

Machine M2

Vars B: x
Evt = ? := B
Preparing for Decomposition

Introduce Parameters

Machine M1

Vars A: x, B: x
Evt = ANY p
WHEN p = B
THEN A := p

Refines M0

Was A := B
A Model of Communication

Machine M1

Vars A: x, B: x
Evt = ANY p
WHEN p = B
THEN A := p

Machine M2

Vars A: x
Evt = ANY p
WHEN p = B & p: x
THEN A := p

Machine M3

Vars B: x
Evt = ANY p
WHEN p = B & p: x
THEN SKIP

Decompose

Refines

Composed Machine
A Model of Communication

Composed Machine

Machine M2

Vars A: x
Evt = ANY p?
WHEN p = B & p: x
THEN A := p

Machine M3

Vars B: x
Evt = ANY p!
WHEN p = B & p: x
THEN SKIP
An Implementation of the Communication

Incoming parameter

Machine M2

Vars A: x
Evt = ANY p?
WHEN p = B & p: x
THEN A := p

Outgoing parameter

Machine M3

Vars B: x
Evt = ANY p!
WHEN p = B & p: x
THEN SKIP

subroutine
Evt(p: x){
    A := p
}

call
Evt(B);
Tasking Event-B

Adds 'Tasking' Implementation Information to Event-B

Task Body Syntax:
- Allows use of Branches, Sequence and Loops.
- Has an 'Output' to console.
This example is from the Tasking Event-B wiki tutorial.
Heater Controller Example

Another View

Graphical Representation of an Event-B Development for the Heating Controller
A Tasking Machine
Implementation level Specification

AutoTasks Machines and Environ Machines

Events:
Used in a Sequence, Branch, Loop, Output

```
Get_Target_Temperature1;
Sense_PressIncrease_Target_Temperature;
if Raise_Target_Temperature
  else Raise_Target_TemperatureBlocked;
Sense_PressDecrease_Target_Temperature;
if Lower_Target_Temperature
  else Lower_Target_TemperatureBlocked;
Set_Target_Temperature;
Display_Target_Temperature
```
'in'/'out' annotations

```
Get_Target_Temperature1 <=
   COMBINES EVENT
   Shared Object IMPL.Get Target Temperature1 ||
   Display_Update_Task_IMPL.Get_Target_Temperature1
REFINES
   Get_Target_Temperature1

Get_Target_Temperature1 <=
   REFINES
   Get_Target_Temperature1
   ANY
   in tm
   WHERE
   grd1 : tm ∈ ℤ TYPING
   THEN
   act1 : cttm1 := tm
   END
```

synchronization
Parameter
direction
Code Generation
Generated Code

In the Display Task:

```
Shared_Object: Shared_Object_IMPL; ...

task body Display_Update_Task_IMPL is
  cttm1 : Integer := 0;
  period: constant Time_Span := To_Time_Span(0.5);
  nextTime: Time := clock + period;
begin
  loop
    delay until nextTime;
    Shared_Object.Get_Temperature1(cttm1);
  end loop;
...
```

In the Protected Object:

```
procedure Get_Temperature1(tm: out Integer) is
begin
  tm := cttm;
end Get_Temperature1;
```
Types and Translations.

So far,

- translations for built-in Event-B types are restricted to INTs and BOOLs.
- and Event-B INTs are not bounded (wrap-around in implementations?).
- we don't even have arrays as standard in Event-B.
Extending Event-B: with New Types, and Translations.

- Use the Theory Plug-in

- Theories are used to define new
  - datatypes
  - operators
  - rewrite rules
  - inference rules

We also use it for code generation,
- to translate predicates and expressions.
Defining a Translator:
From Event-B to a 'new' Target Language

```
THEORY AdaRules
TRANSLATOR Ada
Metavariables : a ∈ Z, b ∈ Z, c ∈ Q, d ∈ Q
Translator Rules
...
trns2:     a - b → a - b
trns9:     c = d → c = d
trns19:    a ≠ b → a /= b
trns21:    a mod b → a mod b
trns22:    ¬c → not(c)
trns23:    c v d → (c) or (d)
trns24:    c ∧ d → (c) and (d)
trns25:    c → d → not(c) or (d)
Type Rules
  typeTrns1:  Z → Integer
  typeTrns2:  BOOL → boolean
```
Adding new Types

THEORY Array
TYPE PARAMETERS T
OPERATORS

•array : array(s : P(T))
direct definition
array(s : P(T)) ≡ { n, f · n ∈ N ∧ f ∈ 0 · (n−1) → s | f }

•arrayN : arrayN(n : Z, s : P(T))
well-definedness condition n ∈ N ∧ finite(s)
direct definition
arrayN(n : Z, s : P(T)) ≡ { a | a ∈ array(s) ∧ card(s) = n }
Adding a Translation for the new Type
(In a theory)

• update : update(a : Z ↔ T, i : Z, x : T)
• lookup : lookup(a : Z ↔ T, i : Z)
• newArray : newArray(n : Z, x : T)

TRANSLATOR Ada
Metavariables s ∈ P(T), n ∈ Z, a ∈ Z ↔ T, i ∈ Z, x ∈ T
Translator Rules
  trns1 : lookup(a, i) ⇒ a(i)
  trns2 : a = update(a, i, x) ⇒ a(i) := x
  trns3 : newArray(n, x) ⇒ (others ⇒ x)
Type Rules
  typeTrns1 : arrayN(n, s) ⇒ array (0..n-1) of s
Using a new Type

VARIABLES
    cbuf  private
    a    private
    b    private

INVARIANTS
    inv1: cbuf ∈ arrayN(maxbuf, Z) not theorem TYPING Typing
    inv2: a ∈ Z not theorem TYPING Typing
    inv3: b ∈ Z not theorem TYPING Typing
    inv4: a ∈ 0..maxbuf-1 not theorem TYPING NonTyping
    inv5: b ∈ 0..maxbuf not theorem TYPING NonTyping
    inv6: ∀i. i ∈ (0..seqSize(abuf)) ⇒ prj2(abuf)(i) = cbuf((a+i) mod

EVENTS

INITIALISATION: internal not extended ordinary

THEN
    act1: cbuf = newArray(maxbuf, 0)
    act2: a = 0
    act3: b = 0

END
Tasking Event-B - restrictions

- AutoTasks do not communicate with each other.
- Communicate through Shared Machines.
- No nesting, in the Tasking Event-B syntax.
- One machine per 'Object'.
And finally … (almost)

- Writing code for Safety Critical Systems is hard.
- The existing code can be augmented by additional notations for extended static-checking (JML), static checking + proof (SPARK Ada)
- Use safe language subsets.
- Place restrictions on the implementation.
  - esp. for timing, and concurrency.

- Use Formal Modelling with automatic code gen.
- also, use Model-checking, SAT/SMT etc.
  to help discover errors.
... and finally (actually)

If you write code **manually**
- much of the development effort is invested in eliminating coding errors.

With **automatic code generation**
- The modelling process helps to eliminate **systemic** errors.
- If the translator is 'trusted', coding errors should be absent.

- Certifying a translator is possible, but expensive.