

Real-Time Programming Languages (ADA and Esterel as Examples)

RT Language Classes

HLL suitable for RT-Analysis (e.g., rms or time-driven)

- Synchronous HLL (clock driven)

- 
- Esterel
 - Lustre
 - (State Charts)

- RT-Euclid

Normal HLL with some rt-extensions

- 
- ADA
 - RT-Java
 - PEARL
 - CHILL

Synchronous Systems \Rightarrow Synchronous Languages

Gerard Berry:

Our approach is based on the standard principle of separation of concerns. First, write the program at the logical level and make use of all the mathematics available there. Then, implement the program using the best available automatic synthesis tools, and check that the result is practically OK.

Ideal Systems produce their outputs synchronously with their inputs.

Implementation

Gerard Berry:

For example, assume we want to trigger a reaction with two inputs A and B present.

We execute the following [compiler generated] C code:

```
M_I_A();
```

```
M_I_B();
```

```
M();
```

If X is emitted by the reaction, the function M() will call an output function M_O_X() we have to supply. The reaction is subject to an essential atomicity condition: during the execution of the main reaction function M(), it is forbidden to call any input function. This is the software counterpart to the hardware requirement that input voltages must be kept stable during a clock cycle.

Esterel

Most statements are instantaneous
(starts and terminates at the same instant of time)

Stepwise execution,
everything completes in each step/cycle

Time consumption explicit (e.g., „Pause“)

Esterel „Data“: Variables and Signals

Variable: Value

Signal: Value vs Status

Value of any type

Status: Present/non present

Newly evaluated at every step
present when emitted

Signals vs Variables

Emit Count(pre(?Count) + 1) vs V:= V+1;

Beware:

writing “emit COUNT(?COUNT+1)” is tempting but incorrect. Since ?COUNT is the current value of COUNT, it cannot be incremented and reemitted right away as itself. It is necessary to use the previous value pre(?COUNT).

Esterel: „Statements“

- Consume no time (unless explicitly said otherwise)
 - Await A: „consumes one A“
 - Pause: „consumes one time step“ (tick)

 - $X := Y$: assigns values to variables
 - $S1; S2$
 - $S1 \parallel S2$
 - Loop S end
starts s, repeats if not terminated
(s must consume time)
 - Loop S each R
restarts S at each occurrence of R
-

Signals

- Emit $x(y)$: sets signal x present, assigns value y
- $?S$ current value:
value just emitted (if so) or value of previous instant (otherwise)
 $pre(?S)$: previous value
- Present \square then $s1$ else $s2$ end (conditional)
- Abort S when \square do R end abort;
starts S , terminates when σ becomes active, does R
- Suspend S when \square
suspends S when \square active
no emission when suspended
- Trap \square in S end trap
starts S , aborts when \square present

More Statements

- **halt**: loop pause end
- **await** \square : abort halt when \square end abort
- **sustain** $x(t)$: loop emit $x(t)$; pause end
- **every** \square do S end every:
 await \square ; loop S each \square

Examples (all by Berry): ABRO

Specification ABRO:

Emit an output O as soon as two inputs A and B have occurred.
Reset this behavior each time the input R occurs.

```
module ABRO:
```

```
input A, B, R;
```

```
output O;
```

```
loop
```

```
  [ await A || await B ];
```

```
  emit O
```

```
each R
```

```
end module
```

Counting

Specification COUNT:

Count the number of occurrences of the input I seen so far, and broadcast it as the value of a COUNT signal at each new I.

module COUNT:

input I;

output COUNT := 0 : integer;

every I do

 emit COUNT(pre(?COUNT) + 1)

end every

end module

Beware:

“emit COUNT(?COUNT+1)” is tempting but incorrect.

Speed

Specification SPEED:

Count the number of centimeters run per second, and broadcast that number as the value of a Speed signal every second.

```
module SPEED:
input Centimeter, Second;
relation Centimeter # Second;
output Speed : integer;
loop
  var Distance := 0 : integer in
    abort
      every Centimeter do
        Distance := Distance+1
      end every
    when Second do
      emit Speed(Distance)
    end abort
  end var
end loop
end module
```

RT-Extensions for Standard HLL: Requirements

standard goodies +

exception handling

tasks + communication

priorities

a notion of time

access to low level hardware

portability

efficiency

predictability

ADA as an Example

Used intensively, e.g. Military, Aircraft (B777), Space
“most commonly used language in US weapons
modernization”

Ada 83 - result of a competition ...

Ada 95 - major redesign (ISO/IEC 8652: 1995)

Ravenscar: subset

Ada 2005

Annex: Real-Time Systems

(here: very limited excerpts ...

Especially, not covered at all:

packages

Exception Handling

Exceptions are declared and may be raised.

```
Buffer_Full: exception;  
begin  
...  
if BufferSize > Limit then raise Buffer_Full end if;  
...  
exception  
  when Buffer_Full => reset;  
  when others => bla ;  
end;
```

Tasks

Tasks are entities whose execution may proceed in parallel.

A task has a thread of control.

Different tasks proceed independently, except at points where they synchronize.

... declared by an object declaration or created dynamically.

... Every task has a “master”: block, subprogram etc. containing the declaration of the task object (or access object).

Before leaving the master, the parent task waits for all dependent tasks.

Example

task type Babbler is

 entry Start(Message : Unbounded_String; Count :
 Natural);

end Babbler;

task body Babbler is

 Babble : Unbounded_String;
 Maximum_Count : Natural;

begin

-- We're done, exit task.

end Babbler;

Communication

- **Activation and termination (see previous slide)**
- **protected objects for synchronized access to shared data**
- **rendezvous for synchronous communication**
- **unprotected access to shared data (global variables)**
-

The Rendezvous

- One task calls an entry (Client)
- Other task accepts a call (Server)
- calling task placed on a queue

Rendezvous, example

```
task body Babbler is
  Babble : Unbounded_String;
  Maximum_Count : Natural;
begin
  accept Start(Message : Unbounded_String; Count :
  Natural) do
    Babble := Message;    -- Copy the rendezvous data to
    Maximum_Count := Count; -- local variables.
  end Start;
  -- babble ...
end Babbler;
```

procedure Noise is

Babble_1 : Babbler; -- Create a task.

Babble_2 : Babbler; -- Create another task.

begin

-- At this point we have two active tasks, but both of them

-- are waiting for a "Start" message. So, send them a Start.

Babble_1.Start(U("Hi, I'm Babble_1"), 10);

Babble_2.Start(U("And I'm Babble_2"), 6);

end Noise;

Entries and Exceptions

- What happens if an exception is raised within an accept statement ?
- Exception is raised twice, at caller and callee tasks

Select Statement

select

**when (expression) =>
accept E1 do bla end E1;**

or

**when (expression) =>
accept E2 do bla end E2;**

or ...

end select;

Arbitrary entry, whose expression is evaluated to true, is called.

Exception if no expression evaluates to true.

Time

Access to time: Calendar object.

delay

duration

point in time

delay 5.0; -- delay for 5 seconds

delay until A_Time; -- delay until it is ...

Delay and Select

select

```
accept An_Entry do bla  
end An_Entry;
```

or

```
delay 5.0;  
Put("An_Entry: timeout");  
end select;
```

Protected Objects

export

functions

read-only access to shared data

functions can be executed concurrently

procedures

read/write access to shared data

Procedures: executed under mutual exclusion

entries

procedures with barriers (boolean expressions)

synchronization operations automatic

evaluate barriers at each call to and at each return from

an entry

```
protected type Resource is
  entry Seize;      -- Acquire this resource exclusively.
  procedure Release; -- Release the resource.
private Busy : Boolean := False;
end Resource;
```

```
protected body Resource is
  entry Seize when not Busy is
  begin
    Busy := True;
  end Seize;

  procedure Release is begin Busy := False end Release;
end Resource;
```

Reordering Requests: Requeue

Flexible, explicit treatment of request orders
example:

An request enters entry or barrier

Parameters inspected

Decision: wait for another chance

Action:

Requeue a request of a caller to some entry or barrier

Not easy to use !!!

Example: Requeue Statement

```
protected body AirportGate is
  entry EnterGateBusiness(Ticket)
  begin
    if Ticket.Economy then
      requeue EnterGateEconomy;
    end if;
    CountBusinessPassengers
  end EnterGate;

  entry EnterGateEconomy(Ticket)
  when AllBusinessPassengersHaveEntered
  begin ... end

end Event;
```

Timeouts on Actions

select

delay 5.0; -- triggering alternative

CalculationComplete:= false;

then abort

Invert_Giant_Matrix(M); -- abortable part

CalculationComplete:= true;

end select;

also possible:

an entry call instead of delay (intricate semantics)

Real-Time Annex

- **Priorities**
- **Priority Scheduling**
- **Ceiling**
- **Queuing Policies**
- **Dynamic priorities**
-

Time

Access to Real-Time: `Real_Time.Time`

finer grained

monotonically non-decreasing
(time zone, correction, summertime)

loop

 delay until `Poll_Time`;

 --- poll

`Poll_Time := Poll_Time + Period`;

end loop;

Priorities

Task/protected type T is

pragma Priority(P);

pragma task_dispatching_policy(fifo within priorities);

evaluated at init time

task gets priority of creator if no pragma is present

“Dynamic” Priorities

```
procedure Set_Priority(Priority: Any_Priority;  
                      T: Task_ID := Current_Task);
```

```
function Get_Priority(T: Task_ID := Current_Task)  
  return Any_Priority;
```

Priority Ceiling

Pragma Locking_Policy(Ceiling_Locking);

“the task executing a protected operation inherits the priority of the protected object”

dynamic priorities higher than ceiling

Entry Queuing Policies

Standard: FIFO

```
pragma Queuing_Policy(Priority_Queueing);
```

“the user can override the default FIFO policy with the
pragma Queuing_Policy”

per partition (not per entries or tasks)

passing of dynamic priorities as implicit parameters

ADA

No support

- for explicit specification of periodic processes !!!
- for static analysis for feasible schedules !!!

Material

- Esterel:
Gerard Berry, Esterel Language Primer
<http://www.esterel-technologies.com/files/primer.zip>
- ADA: lots ...
Used for this lecture:
- Burns/Wellings: RT systems and Programming Languages
- <http://www.adahome.com/Ammo/Cplpl2Ada.html>
(Ada for C programmers)
- <http://www.adahome.com/Tutorials/Lovelace>
- <http://atlas.otago.ac.nz:800/staff/PaulG/Ada-Information/Ada95/Rationale>
- **ADA 2005 and Ravenscar:**
http://www.adacore.com/home/ada_answers/ada_2005