Real-Time Systems
Hermann Härtig

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

- Concurrency and Synchronization/Communication
- Time
  - Access to
  - Control over (“timeout”, ...)
- Scheduling/Resource Management
  - Built in
  - Explicit
- Recurrent processes
RT Language Classes

Synchronous HLL (tick driven)
- Esterel
- Lustre
- (State Charts)

*Imperative* HLL with rt-extensions
- ADA
- RT-Java
- PEARL
- CHILL
- RT-Euclid (designed to enable static analysis)
For further study

Gerard Berry,
Esterel Language Primer
http://www.esterel-technologies.com/files/primer.zip

Video of Artist summer school 2008
http://www.artist-embedded.org/artist/Videos-Slides

(Prof. Christian Hochberger's “Embedded Systems”
moved on to another Uni)

Caveat:
This lecture: introduction into principles only
Ignored: Extensive Tool Chain, Verification, ...
Esterel at a glance

Starting Point (Berry):

“Ideal Systems produce their outputs synchronously with their inputs.”

Esterel:

Most statements are instantaneous

(starts and terminates at the same instant of time)

Stepwise execution,

everything completes in each step/cycle/tick

Time consumption explicit (e.g., “Pause“)
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
- \( S_1;S_2 \)
- \( S_1 || S_2 \)
- Loop S end
  - starts s, repeats if not terminated
  - (s must consume time)
<table>
<thead>
<tr>
<th><strong>Variable:</strong></th>
<th>Value of any type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Signal:</strong></td>
<td>Value and Status</td>
</tr>
<tr>
<td></td>
<td>Value of any type</td>
</tr>
<tr>
<td><strong>Status:</strong></td>
<td>Present/non present</td>
</tr>
<tr>
<td></td>
<td>Newly evaluated at every step</td>
</tr>
<tr>
<td></td>
<td>present when emitted</td>
</tr>
</tbody>
</table>
Signals

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

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).
More Statements

- `halt: loop pause end`
- `await \(\sigma\) : abort halt when \(\sigma\) end abort`
- `sustain x(t): loop emit x(t); pause end`
- `loop S each R`
  - restarts S at each occurrence of R
  - `every \(\sigma\) do S end every:
    - `await \(\sigma\); loop S each \(\sigma\)`
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:
01  input A, B, R;
02  output O;
03    loop
04        [ await A || await B ];
05        emit O
06    each R
07    end module
**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:**

```plaintext
01  input I;
02  output COUNT := 0 : integer;
03  every I do
04      emit COUNT(pre(?COUNT) + 1)
05  end every
06  end module
```

Beware:
“emit COUNT(?COUNT+1)” is tempting but incorrect.
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
            abort every Centimeter do
                Distance := Distance + 1
            end every
            when Second do
                emit Speed(Distance)
            end abort
        end var
    end loop
end module
ADA

Used intensively, e.g. Military, Aircraft (B777), Space
“most commonly used language in US weapons modernization”
Ada 83 - result of a competition ...
Ada 2005, includes Ravenscar: subset
Annex: Real-Time Systems
Few general points

Ada has “Annexes”: in this lecture: Real-Time Annex

Ada has “profiles”: relevant for this lecture “Ravenscar” reduced functionality for Hard-RT

Ada has “pragmas” (compiler directives)

CAVEAT: In this lecture: very limited extract relevant for RTS
Especially, not covered explicitly: Packages(library), OO, Type-System, Generics, exceptions .. we rely on your intuition
THE ADA-RT Book

Concurrent and Real-Time Programming in Ada

by

Alan Burns and Andy Wellings

Cambridge University Press
ISBN 978-0521866972

Most code examples taken from this source.

Many more resources available.

Image Source: amazon.com/dp/B001GS6TBO/
Concurrency

Tasks

- are entities whose execution may proceed in parallel.
- have a thread of control.
- proceed independently, except at points where they synchronize.
- are created and activated via
  - an object declaration or
  - created dynamically using an “access type”
    
    \[ \text{Ptr:} = \text{new} \ldots \]
Example: Operator/Subscriber

```vhdl
01 task type Subscriber;
02
03 task type Telephone_Operator is
04 entry Directory_Enquiry(Person : in Name; Addr : in Address;
05 Num : out Number);
06 end Telephone_Operator;
07
08 S1, S2, S3 : Subscriber; -- friends of Stuart Jones
09 An_Op : Telephone_Operator;
10
11 task body Subscriber is
12 Stuarts_Number : Number;
13 begin
14 ...
16 end Subscriber;
```
Termination of Tasks

Every task has a “master” and “depends” on it:

block, subprogram etc.
containing the declaration of the task object or
of the access object type

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

- Protected objects (ignored in this lecture) for synchronized access to shared data
- Rendezvous for synchronous communication between tasks
- Unprotected access to shared data (global variables)
The Rendezvous

Based on client/server model:

- One task (client) calls an entry of an other task
- Other task accepts a call (Server)
- Calling task placed on a queue
Example: Operator/Subscriber

```plaintext
01 task body Subscriber is
02 Stuarts_Number : Number;
03 begin
04 -- ...
06 "10 MAIN STREET, YORK", Stuarts_Number);
07 -- phone Stuart
08 -- ...
09 end Subscriber;

01 task body Telephone_Operator is
02 begin
03 loop
04 -- prepare to accept next call
05 accept Directory_Enquiry(Person : in Name;
06 Addr : in Address; Num : out Number) do
07 -- look up telephone number and
08 -- assign the value to Num
09 null; --RM
10 end Directory_Enquiry;
11 -- undertake housekeeping such as logging all calls
12 end loop;
13 end Telephone_Operator;
```
In detail

- task executing
- task executing
- a rendezvous
- task suspended
- data exchange

Time

Server
- accept Request
- end
- Server.Request
- end

Client 1
- accept Request
- end

Client 2
- Server.Request
- Server.Request
A task can have multiple entries:

```plaintext
01 select
02   when (expression) =>
03     accept E1 do bla end E1;
04 or
05   when (expression) =>
06     accept E2 do bla end E2;
07 or ...
08 end select;
09
```
Select Statement

- Arbitrary entry, whose expression is evaluated to true, is called.
- Exception if no expression evaluates to true.
- The Boolean expression is evaluated only once per execution of select
  (do not use global variables in when clause !)

More on “select” later (timings)
Priorities

01  pragma task_dispatching_policy(policy identifier);

Supported by ADA 205:

• Preemptive fixed priority
• Non-Preemptive fixed priority
• Round robin
• EDF
• And mixtures thereof
Fixed Priorities

- task (type) T is pragma Priority(P); ....
- Distinct Run Queue per (active) priority
  released tasks at the end of queue
  preempted tasks at the beginning
- Such priorities are called the base priorities of the task
  in contrast to active priority
Priority Ceiling

Implements the “immediate ceiling protocol”

- The object ceiling priority must be maximum priority of any calling task
- The task executing a protected operation executes at the ceiling priority of the protected object

Pragma Locking_Policy(Ceiling_Locking);
Protected object is pragma priority(...) ...

01
02
Active Priority

Base priority or

- Ceiling priority if calling a protected object
- The creating task's priority if higher than the base priority
- During rendezvous:
  the priority of the task executing the accept statement
  inherits the priority of the calling task
Run-Time Priorities

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

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

Resets base priority.
EDF Dispatching

How to set deadlines:

01  Package ada.dispatching.EDF is
02  procedure set_deadline (D: in deadline, 
03                     T: in TaskId)
04  Procedure DelayUntilAndSetDeadline(...)
05  Procedure GetDeadline (...)

Or

01  Pragma Relative_Deadline(Milliseconds(3))
02  & Explicit call to set first deadline of periodic task
EDF and Ceiling

- Implements “Preemption Level Control Protocol”
- look forward to: lecture on resource access control
- separates:
  - Urgency (EDF)
  - Preemption level

- Using Priority Ceiling of Protected Objects as Preemption Level

- Details: see Ted Baker(91) and (complicated)
Specify “priority partitions” to set scheduling disciplines

```plaintext
Pragma Priority_Specific_Dispatching
  (Fifo_Within_Priorities, 10, 16)

Pragma Priority_Specific_Dispatching
  (EDF_Across_Priorities, 2, 9)

Pragma Priority_Specific_Dispatching
  (RoundRobin_Within_Priorities, 1, 1)
```
Entry Queuing: Implicit Policies

Standard: FIFO

```c
01 pragma Queuing_Policy(Priority_Queuing);
```

“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
Explicit Request Ordering: Requeue

Explicit treatment of request orders, example:
A request enters entry or barrier
Parameters inspected in body code
Possible decision: requeue at different entry

Action:
Requeue a request of a caller to some entry or barrier

Not easy to use !!!
Example

```plaintext
01  protected body AirportGate is
02     entry EnterGateBusiness(Ticket)
03     begin
04         if Ticket.Economy then
05             requeue EnterGateEconomy;
06         end if;
07         HandleBusinessPassenger
08     end EnterGateBusiness;
09
10     entry EnterGateEconomy(Ticket)
11     when AllBusinessPassengersHaveEntered
12     begin HandleEconomyPassenger end EnterGateEconomy
13     end AirportGate;
14```

Timing events

Package Ada.Real_time.Timing_events is
   --...

Procedure Set_Handler(Event: in out Timing_Event;
       At-Time: Time; Handler: Timing_Event_Handler);
   --...
Procedure Set_Handler(Event: in out Timing_Event;
       At-Time: Time_Span; Handler: Timing_Event_Handler);

Causes Handler to be called at chosen times.
Handlers are called by Clock_Interrupt Handler
Must not block.
Used for periodic action and watchdogs
Example: Watchdog

```vhdl
01  protected Watchdog is
02    pragma Interrupt_Priority (Interrupt_Priority'Last);
03    entry Alarm_Control;
04      -- Called by alarm handling task.
05    procedure Call_In;
06      -- Called by application code every 50ms if alive.
07    procedure Timer(Event : in out Timing_Event);
08      -- Timer event code, ie the handler.
09  private
10      Alarm : Boolean := False;
11  end Watchdog;
12
13  Fifty_Mil_Event : aliased Timing_Event;
14  TS : Time_Span := Milliseconds(50);
15
16  Set_Handler(Fifty_Mil_Event, TS, Timer);
```
Example: Watchdog

```plaintext
01  protected body Watchdog is
02      entry Alarm_Control when Alarm is
03          begin
04              Alarm := False;
05          end Alarm_Control;
06
07  procedure Timer(Event : in out Timing_Event) is
08      begin
09          Alarm := True;
10          -- Note no use is made of the parameter in this example
11      end Timer;
12
13  procedure Call_in is
14      begin
15          Set_Handler(Fifty_Mil_Event, TS, Timer);
16          -- This call to Set_Handler cancels the previous call
17      end Call_in;
18
19  end Watchdog;
```
**Time: Delay Statement**

```plaintext
delay
  duration
  point in time

delay 5.0;  -- delay for at least 5 seconds

delay until A_Time; -- delay at least until A_Time
```

specifies minimum delay
Delay and Select, server side

```
01  select
02     accept An_Entry do  bla
03      end An_Entry;
04  or
05      delay 10.0;
06      Put("An_Entry: timeout");
07      end select;
```

Select terminates if entry is not called within 10 time units.
Delay and Select, client side(1)

01  select
02      Operator.Enquiry()
03  or
04      delay 10;
05  end select;

Select terminates if entry is not accepted within 10 time units.
Only one call alternative allowed
Client side (2): “Asynchronous” Select

01 select trigger
02 triggering_alternative --- (entry-call or delay)
03 then abort
04 abortable_part
05 end select;

• If delay or entry-call complete before the abortable part, the abortable part is aborted
• abortable_part must not an accept statement
Example

```plaintext
01 select
02   delay 5.0;  -- triggering alternative
03 then abort
04   CalculationComplete:= false;
05   Invert_Giant_Matrix(M);  -- abortable part
06   CalculationComplete:= true;
07 end select;
```

Careful: notice the race condition!
Example: Watchdog

task type Watchdog is
    entry All_Is_Well;
end Watchdog;

task body Watchdog is
begin
    loop
        select
            accept All_Is_Well;
        or
            delay 10.0;
            -- signal alarm, potentially the client has failed
            exit;
        end select;
    end loop;
    -- any further required action
end Watchdog;
Example: Operator/Subscriber

task type Subscriber;

task body Subscriber is
    Stuarts_Number : Number;
begin
    loop
        select
                "10 MAIN STREET, YORK", Stuarts_Number);
        -- log the cost of a directory enquiry call
        or
            delay 10.0;
        -- phone up his parents and ask them,
        -- log the cost of a long distance call
        end select;
    end loop;
end Subscriber;
Simple Periodic Task With Static Priority

```
01  task A is
02      pragma Priority(5);
03  end A;

04  task body A is
05      Next_Release: Real_Time.Time;
06      begin
07          Next_Release := Real_Time.Clock;
08          loop
09              -- code
10              Next_Release := Next_Release + Real_Time.Milliseconds(10);
11              delay until Next_Release;
12          end loop
13          end A;
14
```
Recurrent Tasks as Package (1)

```
with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Periodic_Scheduler is
  procedure Set_Characteristic(T : Task_Id; Period : Time_Span;
                              First_Schedule : Time);
  procedure Wait_Until_Next_Schedule; -- potentially blocking
end Periodic_Scheduler;

-- Periodic tasks can now be encoded as

task Periodic_Task;
task body Periodic_Task is
  begin
    loop
      -- statements to be executed each period
      Periodic_Scheduler.Wait_Until_Next_Schedule
    end loop;
  end Periodic_Task;
```
procedure Set_Characteristic(T : Task_Id; Period : Time_Span; 
First_Schedule : Time) is 
begin 
  Set_Value(((Period, First_Schedule), T )); 
end Set_Characteristic; 

procedure Wait_Until_Next_Schedule is 
  Task_Info : Task_Information := Value; 
  Next_Time : Time; 
begin 
  Next_Time := Task_Info. Period + 
  Task_Info.Next_Schedule_Time; 
  Set_Value(((Task_Info. Period, Next_Time))); 
  delay until Next_Time; 
end Wait_Until_Next_Schedule; 
end Periodic_Scheduler;
Missing in this RT-HLL lecture

- RT-Java and RT-Garbage Collection
- Language with built-in periodic processes
To take away ...

- Principles of synchronous languages
- Mechanisms to explicitly handle timing
- Mechanisms to handle asynchronous events
- “scheduling” of processes, queues, ...