Introduction & Languages

What is a real-time system?

Features of a Real-Time System?

- Fast context switches? Should be fast anyway!
- Small size? Should be small anyway!
- Quick responds to external interrupts? Predictable! – not ‘quick’
- Multitasking? Real time systems are often multitasking systems
- ‘Low level’ programming interfaces? Needed in many systems!
- Interprocess communication tools? Needed in any distributed sys.!
- High processor utilization? Needed in many distributed systems!
- Fast systems? Predictable! – not ‘fast’
What is a real-time system?

Definition of a Real-Time System

The correctness of a real-time systems depends on:
1. The logical correctness (accuracy) of the results as well as ...
2. The time when the result is delivered.
with respect to the specification.

Typical characteristics of a Real-Time System

- **Adherence to set time constraints.** Not too early – not too late
- **Predictability**
  - Repeatable results in time and value
- **Fault tolerance**
  - Robustness in the presence of foreseeable faults
- **Accuracy**
  - Results are precise enough to drive i.e. a physical systems
- **Frequently concurrent and distributed**

Real-Time Systems Scenarios

- **All size and complexities:** From heating regulators over mobile phones to high speed trains, aircraft, satellites, space station(s) ...
- **Embedded:** Almost always part of or coupled to a physical system.
- **Relevant:** Vital components of our traffic and communica-tion infrastructure among many other essential systems.
- **Dangerous:** Failures often lead to loss of life, or environmental damage.

⇔ Real-Time Systems require a specific understanding and skill set.

Simple example

Latencies:

- **Small & constant.**
- **Significantly shorter than the driver's response time.**

Reliability:

- **Provide robustness under all foreseeable and “manageable” failures.**

Efficient design:

- **Mass producible?**
Introduction & Languages

Simple example

Ways to define reliability

- Full fault tolerant, graceful degradation or fail safe?
- Redundancies?
- Testing?
- Verification?
- (Physical) Modularization?

Is Use an algebraic framework?
- Proof correctness?
Is Use a predictable runtime environment and language?
- Test all relevant cases?
Is Assume things will still go bad? Provide fall-backs?

Simple example

Ways to implement this (software)

- Sequential, concurrent or distributed?
- Shared memory or message passing?
- Synchronous or asynchronous communication?
- Dynamic or fixed schedule?
- Imperative, functional, or dataflow programming?
- Predefined communication channels or client/server models?
- Data driven or (global) clock synchronized?
- Polling, interrupt driven, or event driven?
- Globally synchronous, individually synchronous, or asynchronous I/O channels?
- Verifiable or testable software (models)?

Simple example

Ways to implement this (hardware)

The brakes:
- Mechanical (hydraulic) + overwrite valves
- Digitally controlled main valve, no mechanical (hydraulic) connection to brake pedal

The controller(s):
- Single CPU
- Multiple CPUs + shared memory
- Multiple CPUs + point-to-point connections
- Multiple CPUs + communication system (e.g. a bus system)

What is a real-time system?

Typical Real-Time Operating System

Often implemented as an integrated run-time environment, i.e. there is ‘no operating system’ (as embedded systems).

RT-OSs provide:
- Predictability
- Passivity
- Small footprint
- Instrumentation
Real-Time Systems Components

Embedded Real-Time Systems Components

Real-Time Programming Languages

Requirements for Real-Time Languages / Environments

- **Predictability**
  - No operations shall lead to unforeseeable timing behaviours (e.g., garbage collection)
- **Time**
  - Specified granularity, operations on time, scheduling
- **High integrity**
  - Complete, unambiguous language definition, and strong compilers and runtime environments detecting faults as early as possible
- **Concurrency and Distribution**
  - Solid, high-level synchronization and communication primitives, automated data marshaling
- **Concrete yet Scaling**
  - Mapping physical interfaces into high-level data types plus programming “in the very large”
Real-Time Programming Languages

Requirements for Real-Time Languages / Environments

- **Predictability**
  - No operations shall lead to unforeseeable timing behaviours (e.g., garbage collection)
- **Time**
  - Specified predictability operations on time, scheduling
- **High integrity**
  - Complete, unambiguous language definition, and strong compilers and runtime environments detecting faults as early as possible
- **Concurrency and Distribution**
  - Solid, high-level synchronization and communication primitives, automated data marshalling
- **Concrete yet Scaling**
  - Mapping physical interfaces into high-level data-types plus programming “in the very large”

Real-Time Languages, Operating Systems and Libraries

What if you “cannot / want-not” use a real-time language and you need to formulate some/all real-time constraints outside the programming language?

Real-time operating systems:

- Scheduling, interrupt handling, (potentially other features) migrate from the compiler environment into the operating system.
- Compiler level analysis is replaced by equivalent tools on the OS level.

Libraries:

- Loss of all compiler-level checks
- Loss of block structure and scoping

Languages explicitly supporting concurrency: e.g. Ada2005

Ada2005 is an **ISO standardized** (ISO/IEC 8652:1995/Amd 1:2007) ‘general purpose’ language which “promotes reliability and simplify maintenance” while keeping maximal efficiency and provides core language primitives for:

- Strong typing, separate compilation (specification and implementation), object-orientation,
- Concurrency, message passing, synchronization, monitors, rpcs, timeouts, scheduling, priority ceiling locks, hardware mappings, fully typed network communication
- Strong run-time environments (up to stand-alone execution)

… as well as standardized language-annexes for

- Additional real-time features, distributed programming, system-level programming, numeric, informations systems, safety and security issues.

Assignments are designed so that they can be solved with reasonable effort in Ada.
A simple queue specification

package Queue_Pack_Simple is

  QuasSize : constant Positive := 10;
  type Element is new Positive range 1_000..40_000;
  type Marker is mod QuasSize;
  type List is array (Marker'Range) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Elements : List;
  end record;

  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free - 1;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top - 1;
  end Dequeue;

end Queue_Pack_Simple;

A simple queue implementation

package body Queue_Pack_Simple is

  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free - 1;
    and Enqueue;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top - 1;
    and Dequeue;
  end Dequeue;

end Queue_Pack_Simple;
A queue specification with proper exceptions

package Queue_Pack_Exceptions is
  QueueSize : constant Integer := 10;
  type Element is (Up, Down, Spin, Turn);
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;

procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  if Queue.State = Filled and Queue.Top = Queue.Free then
    raise Queueoverflow;
  end if;
  Queue.Elements (Queue.Free) := Item;
  Queue.Free := Marker'Pred(Queue.Free);
  Queue.State := Filled;
end Enqueue;

procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  if Queue.State = Empty then
    raise Queueunderflow;
  end if;
  Item := Queue.Elements (Queue.Top);
  Queue.Top := Marker'Pred (Queue.Top);
  if Queue.Top = Queue.Free then
    Queue.State := Empty;
  end if;
end Dequeue;
end Queue_Pack_Exceptions;

A simple queue test program

with Queue_Pack_Simple; use Queue_Pack_Simple;
procedure Queue_Test_Simple is
  Queue : Queue_Type;
  Item  : Element;
begin
  Enqueue (2000, Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue); — will produce an unpredictable result!
end Queue_Test_Simple;

Ada2005

... introducing:
• exception handling
• enumeration types
• type attributed operators

A queue implementation with proper exceptions

package body Queue_Pack_Exceptions is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
    begin
      if Queue.State = Filled and Queue.Top = Queue.Free then
        raise Queueoverflow;
      end if;
      Queue.Elements (Queue.Free) := Item;
      Queue.Free := Marker'Pred (Queue.Free);
      Queue.State := Filled;
    end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
    begin
      if Queue.State = Empty then
        raise Queueunderflow;
      end if;
      Item := Queue.Elements (Queue.Top);
      Queue.Top := Marker'Pred (Queue.Top);
      if Queue.Top = Queue.Free then
        Queue.State := Empty;
      end if;
    end Dequeue;
  end Queue_Pack_Exceptions;
with Queue_Pack_Exceptions; use Queue_Pack_Exceptions;
with Ada.Text_IO; use Ada.Text_IO;

procedure Queue_Test_Exceptions is
  Queue : Queue_Type;
  Item  : Element;
begin
  Enqueue (Turn, Queue);  Dequeue (Item, Queue);  Dequeue (Item, Queue); — will produce a 'Queue underflow' exception
exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow  => Put ("Queue overflow");
end Queue_Test_Exceptions;

package body Queue_Pack_Private is
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    State     : Queue_State := Empty;
    Elements  : List;
    end record;
end Queue_Pack_Private;

package body Queue_Pack_Exceptions is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
    begin
      Queue.Elements (Queue.Free) := Item;
      Queue.Free := Marker (Queue.Free);
      Queue.State := Filled;
    end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker (Queue.Free);
    Queue.State := Filled;
    end Dequeue;

  procedure Queue (Queue: in out Queue_Type) is
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
      end if;
      Queue.Elements (Queue.Free) := Item;
      Queue.Free := Marker (Queue.Free);
      Queue.State := Empty;
    end if;
    end Queue;

end Queue_Pack_Exceptions;
A queue test program with proper information hiding

with Queue_Pack_Private; use Queue_Pack_Private;
with Ada.Text_IO; use Ada.Text_IO;
procedure Queue_Test_Private is
    Queue, Queue_Copy : Queue_Type;
begin
    instantiation of generic packages
    Queue_Copy := Queue; —  — "left hand of assignment must not be limited type"
    Enqueue (Item => 1, Queue => Queue);
    when Queueunderflow => Put ("Queue underflow");
    when Queueoverflow  => Put ("Queue overflow");
end Queue_Test_Private;

A generic queue implementation

generic
    type Element is private;
package Queue_Pack_Generic is
    type Queue_Type is limited private;
    procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
    Queueoverflow, Queueunderflow : exception;
private
    type Marker is mod QueueSize;
    type List is array (Marker'Range) of Element;
    type Queue_State is (Empty, Filled);
    type Queue_Type is record
        Top, Free : Marker := Marker’First;
        State : Queue_State := Empty;
        Elements : List;
        end record;
end Queue_Pack_Generic;

... introducing:
• specification of generic packages
• instantiation of generic packages

A generic queue implementation

package body Queue_Pack_Generic is
    procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
        if Queue.State = Filled and Queue.Top = Queue.Free then
            raise Queueoverflow;
        end if;
        Queue.Elements (Queue.Free) := Item;
        Queue.Free := Marker’Pred (Queue.Free);
    end Enqueue;
    procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
        if Queue.State = Empty then
            raise Queueunderflow;
        end if;
        Item := Queue.Elements (Queue.Top);
        Queue.Top := Marker’Pred (Queue.Top);
        if Queue.Top = Queue.Free then
            Queue.State := Empty;
        end if;
        end Dequeue;
end Queue_Pack_Generic;
A protected queue specification

package Queue_Pack_Protected is
  QueueSize : constant Integer := 10;
  subtype Element is Character;
  type Queue_Type is
    limited private
  protected type Protected_Queue is
    entry Enqueue (Item: in Element)
      when Queue.State = Empty or Queue.Top /= Queue.Free
      begin
        Queue.Elements (Queue.Free) := Item;
        Queue.Free  := Queue.Free - 1;
        Queue.State := Filled;
      end Enqueue;
    entry Dequeue (Item: out Element)
      when Queue.State = Filled
      begin
        Item := Queue.Elements (Queue.Top);
        Queue.Top := Queue.Top - 1;
        if Queue.Top = Queue.Free then
          Queue.State := Empty;
        end if;
      end Dequeue;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  type Marker      is mod QueueSize;
  type List        is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    State     : Queue_State := Empty;
    Elements  : List;
  end record;
end Queue_Pack_Protected;

A protected queue implementation

package body Queue_Pack_Protected is
  protected body Protected_Queue is
    entry Enqueue (Item: in Element) when Queue.State = Empty or Queue.Top /= Queue.Free
    begin
      Queue.Elements (Queue.Free) := Item;
      Queue.Free := Queue.Free - 1;
      Queue.State := Filled;
      end Enqueue;
    entry Dequeue (Item: out Element) when Queue.State = Filled
    begin
      Item := Queue.Elements (Queue.Top);
      Queue.Top := Queue.Top - 1;
      if Queue.Top = Queue.Free then
        Queue.State := Empty;
      end if;
    end Dequeue;
end Protected_Queue;
end Queue_Pack_Protected;

... introducing:

- protected objects
- tasks (definition, instantiation and termination)
- task synchronisation
- entry guards
- entry calls
- accept and selected accept statements
Abstract types & dispatching

... introducing:

- abstract tagged types
- abstract subroutines
- concrete implementation of abstract types
- dispatching to different packages, tasks, and partitions according to concrete types

package Queue_Pack_Abstract is
  subtype Element is Character;
  type Queue_Type is abstract tagged limited private;
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is abstract;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is abstract;
private
  type Queue_Type is abstract tagged limited null record;
end Queue_Pack_Abstract;

... obviously this does not require an implementation package (as all procedures are abstract)
A dispatching test program with Queue_Pack_Abstract; use Queue_Pack_Abstract;
with Queue_Pack_Concrete; use Queue_Pack_Concrete;

procedure Queue_Test_Dispatching is
  type Queue_Class is 
    access all Queue_Type'class;
  task Queue_Holder is — could be on an individual partition / computer
    entry Queue_Filled; end Queue_Holder;
  task Queue_User is — could be on an individual partition / computer
    entry Transmit_Queue (Remote_Queue: in Queue_Class);
  end Queue_User;
end Queue_Test_Dispatching;

A concrete queue specification

with Queue_Pack_Abstract; use Queue_Pack_Abstract;
package Queue_Pack_Concrete is
  QueueSize : constant Integer := 10;
  type Real_Queue is
    new Queue_Type with private;
  procedure Enqueue (Item: in Element; Queue: in out Real_Queue);
  procedure Dequeue (Item: out Element; Queue: in out Real_Queue);
  Queueoverflow, Queueunderflow : exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Real_Queue is
    new Queue_Type with record
      Top, Free : Marker := Marker'First;  
      State     : Queue_State := Empty;  
      Elements  : List;
    end record;
end Queue_Pack_Concrete;

A concrete queue implementation

package body Queue_Pack_Concrete is
  procedure Enqueue (Item: in Element; Queue: in out Real_Queue) is
    begin
      if Queue.State = Filled and Queue.Top = Queue.Free then
        raise Queueoverflow;
      end if;
      Queue.Elements (Queue.Free) := Item;
      Queue.Free := Queue.Free - 1;
      Queue.State := Filled;
    end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Real_Queue) is
    begin
      if Queue.State = Empty then
        raise Queueunderflow;
      end if;
      Item := Queue.Elements (Queue.Top);
      Queue.Top := Queue.Top - 1;
      if Queue.Top = Queue.Free then
        Queue.State := Empty;
      end if;
      end Dequeue;
  end Queue_Pack_Concrete;

A dispatching test program (cont.)

task body Queue_Holder is
  Local_Queue : Queue_Class := new Real_Queue; — any Queue_Type'class instance
begin
  Queue_User.Transmit_Queue (Local_Queue); — entry call between tasks
  accept Queue_Filled do
    Dequeue (Item, Local_Queue.all); — Item will be 'r'
  end Queue_Filled;
  end Queue_Holder;
end Queue_Holder;

task body Queue_User is
  Local_Queue : Queue_Class := new Real_Queue; — any Queue_Type'class instance
begin
  accept Transmit_Queue (Remote_Queue: in Queue_Class) do
    Enqueue ('r', Remote_Queue.all); — potentially a remote procedure call
    Enqueue ('l', Local_Queue.all); — local procedure call
    end Transmit_Queue;
  end Queue_Holder.Queue_Filled;
  Dequeue (Item, Local_Queue.all); — Item will be 'l'
  end Queue_User;
begin null; end Queue_Test_Dispatching;
Introduction & Languages

Ada2005

Ada2005 language status

- Established language standard with free and commercial compilers available for all major OSs and platforms.
- Emphasis on maintainability, high-integrity and efficiency.
- Stand-alone runtime environments for embedded systems.
- High integrity real-time profiles part of the language standard (Ravenscar profile).
- Used in many large scale and/or high integrity projects
  - frequently in the avionics industry, high speed trains, metro-systems, space programs ...
  - but also increasingly on small platforms / micro-controllers

Real-Time Java

Specific Java engines and class libraries enhance:

- **Threads**: Priorities, scheduling, and dispatching
- **Memory**: Controlled garbage collection and physical memory access
- **Synchronization**: Ordered queues, and priority ceiling protocols
- **Asynchronism**: Generalized asynchronous event handling, asynchronous transfer of control, timers, and an operational implementation of thread termination
  - All current real-time Java extensions keep the underlying, consequent object orientation.
  - Predictability often questionable.
  - Some restrict the language standard, some extend it.

Ada2005 language status

- Established language standard with free and commercial compilers available for all major OSs and platforms.
- Emphasis on maintainability, high-integrity and efficiency.
- Stand-alone runtime environments for embedded systems.
- High integrity real-time profiles part of the language standard (Ravenscar profile).

Real-Time Java

Specific Java engines and class libraries enhance:

- **Threads**: Priorities, scheduling, and dispatching
- **Memory**: Controlled garbage collection and physical memory access
- **Synchronization**: Ordered queues, and priority ceiling protocols
- **Asynchronism**: Generalized asynchronous event handling, asynchronous transfer of control, timers, and an operational implementation of thread termination
  - All current real-time Java extensions keep the underlying, consequent object orientation.
  - Predictability often questionable.
  - Some restrict the language standard, some extend it.

Summary

Ada refresher / intro course

- Specification and implementation (body) parts, basic types
- Exceptions
- Information hiding in specifications (‘private’)
- Generic programming
- Tasking
- Abstract types and dispatching

Real-Time Java

Real-Time Specification for Java

Versions 1.0.1 (2002) and 1.1 alpha 6 (2009)

- Enhanced thread model (memory attributes, more precise specs)
- Enabling powerful and highly adaptive scheduling policies
- Introducing scoped, immortal (keep the garbage collector out), and physical memory to Java (map to a physical architecture)
- Introducing timers, interrupts, and more exceptions
- Higher resolution time model
- Optional support for POSIX signals
Real-Time Java

Real-Time Specification for Java

- Standard library classes still rely on garbage collection!
  - i.e. any usage of standard libraries becomes a gamble.
- RTSJ is backwards compatible
  - i.e. no syntactical extensions and no additional compiler checks.
- Allows for different Java-engine implementations:
  - In terms of completeness: e.g. scheduling is not mandatory.
  - In terms of semantics: e.g. “instantiations per time span” can but does not need to imply equal distance intervals.
- Concept is still based on strict object oriented programming
  - Inheritance anomaly in concurrent systems needs to be considered.

Esterel

Control-oriented ↔ Dataflow-oriented

- Dataflow-oriented:
  Continuous data-streams, functional processing (DSPs, filters, ...)
  - high bandwidth
- Control-oriented:
  Discrete signals controlling data-streams and processes
  - low bandwidth

Esterel: Control-dominated Reactive Systems

- Real-Time process control:
  - reaction to (sparse) stimuli in specified time-spans by emitting control signals.
- Communication protocols:
  - control/protocol part of communication systems.
- Embedded systems / device control:
  - local, discrete device control.
- Human-machine interface:
  - switching modes, event and emergency handling.
- Control logic (hardware):
  - glue logic, interfaces, pipe control, state machines.

Transformational ↔ Interactive ↔ Reactive

- Transformational (functional) systems:
  Generating outputs based on input and stop, utilizing no or a small number of internal states.
- Interactive systems:
  I.e. servers and other systems in long-term operation, requesting occasional inputs, and accepting service-calls, when there are resources to do so.
- Reactive (reflex) systems:
  Systems which are reacting to external stimuli only (by generating other stimuli). Can be viewed as a predictable, functional system, which is listening to inputs continuously while holding enough resources to ensure specified reaction times.
Esterel: Strong synchrony or “zero delay” assumption

In logical terms:

- All operations are instantaneous!

In physical terms:

- There is no observable delay between stimuli and reaction!

In implementation terms:

- All operations are finished before the next input sampling, i.e., there is a base frequency which is high enough so that the sampling can be considered instantaneous with respect to the physical system.

A simple, reactive integrator

Module Speed:

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

These are exclusive

Immediate Reactions

by default all synchronization points:

- `await <signal>`;
- `abort ... when <signal>`;
- `every <signal> do ... end every`;
- `loop ... each <signal>`;
- `wait for the next signal occurrence ('rising edge trigger')`.

... yet with an additional 'immediate':

- `await immediate <signal>`;
- `abort ... when immediate <signal>`;
- `every immediate <signal> do ... end every`;
- `loop ... each immediate <signal>`;
- `a currently active signal will trigger these statements ('level trigger')`.

A simple, reactive, pure-signal example

Mealy machine

Module in Esterel:

```haskell
module A_and_B_gives_O;
  input A, B, R;
  output O;
  loop
    await A || await B;
    emit O; each R;
  end loop;
end module;
```
### A simple, reactive, wrong integrator

Specification: a module should count the number of metres per second and emit this number as ‘speed’ once per second.

```esterel
Module Speed;
  input Metre, Second;
  output Speed: integer;
  loop
    var Distance := 0: integer in
      abort
      every Metre do
        Distance := Distance + 1;
      end every;
      when Second do
        emit Speed (Distance);
      end abort;
      end var;
    end loop;
  end module;
```

These are no longer exclusive

### A simple, reactive, simultaneous signals integrator

Métre’ and ‘Second’ occur potentially simultaneously

- using ‘weak abort’ to handle the simultaneous case:

```esterel
Module Speed;
  input Metre, Second;
  output Speed: integer;
  loop
    var Distance := 0: integer in
      weak abort
      every Metre do
        Distance := Distance + 1;
      end every;
      when Second do
        emit Speed (Distance);
      end abort;
      end var;
    end loop;
  end module;
```

These are not exclusive

### A simple, reactive, simultaneous signals integrator

Métre’ and ‘Second’ occur potentially simultaneously

- using ‘every immediate’ to handle the simultaneous case:

```esterel
Module Speed;
  input Metre, Second;
  output Speed: integer;
  loop
    var Distance := 0: integer in
      abort
      every immediate Metre do
        Distance := Distance + 1;
      end every;
      when Second do
        emit Speed (Distance);
      end abort;
      end var;
    end loop;
  end module;
```

Immediate attribute takes every metre into account.

Above block is hard aborted with every ‘Second’ signal
Synchronous languages

Causality and Synchronous Languages

General terms:

‘The future should not influence the past’

Technically:

Causal synchronous programs are:

1. **Reactive** provide a well-defined output for each signal sequence.
2. **Deterministic** provide exactly one output for each signal sequence.

Causality in Synchronous Languages

Cyclic dependencies can cause causality problems in synchronous languages (similar to potential dead-locks in asynchronous languages).

- **Strict synchronous languages** avoid all cyclic dependencies in signals.
- **Esterel**: fully acyclic programs are considered too restrictive, since cyclic dependencies can make programs more intuitive in places.

- **Cyclic programs** can still be reactive and deterministic.

Non-causality in Synchronous Languages

- **Non-reactive programming**
  ```
  module non-reactive;
  output O;
  present O else emit O end;
  end module;
  ```

- **Non-deterministic programming**
  ```
  module non-deterministic;
  output O;
  present O then emit O else emit O end;
  end module;
  ```

- **Cyclic dependencies with multiple signals**
  ```
  module cyclic_dependency;
  output A, B;
  present A then emit B else emit A end;
  end module;
  ```

- All examples contain a reference to “the future”, i.e. are “cyclic”.

Strong synchrony or “zero delay” assumption

The system is assumed ‘synchronous’ or ‘instantaneous’, if
the total worst case computation time is smaller than the minimal time between two observable changes in the environment.

Synchronous systems assume a logical or discrete rather than continuous time.

Enables:

- Strong analysis and simplification tools.
- Significantly easier program verification.
- Straight forward hardware implementations.
**Introduction & Languages**

**VHDL state machine example**

```vhdl
entity enum is port (Clock, Reset : in Std_Logic;
                    A, B : in Boolean;
                    Out : out Std_Logic);
end enum;
architecture A_Simple_Moore_Machine of enum is
  type States is (Start, A_Detected, B_Detected, AB_Detected);
  attribute enum_encoding : string;      attribute enum_encoding of States : type is "gray";
  signal Current_State, Next_State: States;
begin
  Synchronous_Proc: process (Clock, Reset)
  begin
    if (Reset='1') then
      Current_State <= Start;
    elsif (Clock'event and Clock = '1') then
      if A and not B then Next_State <= A_Detected;
      elsif not A and B then Next_State <= B_Detected;
      elsif A and B then Next_State <= AB_Detected;
      else                     Next_State <= Current_State;
      end if;
    end if;
  end process; —  —   End Synchronous_Proc

  Combinatorial_Process: process (Current_State, A, B)
  begin
    case Current_State is
      when Start => Out <= '0' after 1 ns;
      when A_Detected => Out <= '0' after 1 ns;
      when B_Detected => Out <= '0' after 1 ns;
      when AB_Detected => Out <= '1' after 1 ns;
      else                     Next_State <= Current_State;
    end case;
  end process; —  —   End Combinatorial_Process
  end;
end architecture A_Simple_Moore_Machine;
```

**VHDL**

- Standardized hardware description language (IEEE 1076-2008)
- Can be data-flow or control-flow oriented.
- Programming in the large (packages, generics).
- Entities, architectures (processes), and configurations are separate.
- Signals can be digital or analog.
- Strong typing (all basic Ada types plus low level types: std_logic).
- Modules can be clocked independently or not at all (combinatorial).
- Highly concurrent.
- High level synchronization primitives (protected types).
**Process Algebra**

**Timed CSP**

Timed CSP (1986 onwards) is an extension of Communication Sequential Processes (CSP, introduced by Hoare in 1978).

- Everything is a process constructed via a small set of primitives.
- Events are instantaneous.
- All communications are synchronous.
- Processes progress until synchronization points or terminate.
- Unlimited concurrency.
- Traces denote possible chains of events.
- Proofs that certain traces can / cannot happen are constructed via algebraic transformations.

It forms a conceptual/algebraic basis for several concurrent languages incl. Esterel, Ada, Occam, VHDL, Go, VerilogCSP.

---

**Process and Experiment Automation Realtime Language**

**PEARL**

- Simple and ‘readable’ language for small projects.
- Supports tasking and timed activations.
- Supports interrupts, signals, semaphores, and bolt variables.
- Lacks support for ‘programming in the large’.

Is a settled standard:
- DIN 66253-1: Basic PEARL 1981
- DIN 66253-3: PEARL for distributed systems 1989

---

**Introduction & Languages**

**PEARL**

- Simple and ‘readable’ language for small projects.
- Supports tasking and timed activations.
- Supports interrupts, signals, semaphores, and bolt variables.
- Lacks support for ‘programming in the large’.

Is a settled standard:
- DIN 66253-1: Basic PEARL 1981
- DIN 66253-3: PEARL for distributed systems 1989

Currently maintained by a German special-interest community and one company (IEP).
POSIX

Portable Operating System Interface for Unix

- IEEE/ANSI Std 1003.1 and following.
- Library Interface (API) [C Language calling conventions – types exit mostly in terms of (open) lists of pointers and integers with overloaded meanings].
- More than 30 different POSIX standards (and growing / changing).
  - a system is ‘POSIX compliant’, if it implements parts of one of them!
  - a system is ‘100% POSIX compliant’, if it implements one of them!

POSIX - some of the relevant standards...

- **POSIX 1003.1**
  - OS Definition
  - single process, multi process, job control, signals, user groups, file system, file attributes, file device management, file locking, device I/O, device-specific control, system database, pipes, FIFO, ...

- **POSIX 1003.1b**
  - Real-time Extensions
  - real-time signals, priority scheduling, timers, asynchronous I/O, prioritized I/O, synchronized I/O, file sync, mapped files, memory locking, memory protection, message passing, semaphore, ...

- **POSIX 1003.1c**
  - Threads
  - multiple threads within a process; includes support for: thread control, thread attributes, priority scheduling, mutexes, mutex priority inheritance, mutex priority ceiling, and condition variables

- **POSIX 1003.1d**
  - Additional Real-time Extensions
  - new process creation semantics (spawn), sporadic server scheduling, execution time monitoring of processes and threads, I/O advisory information, timeouts on blocking functions, device control, and interrupt control

- **POSIX 1003.1j**
  - Advanced Real-time Extensions
  - typed memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues

- **POSIX 1003.21**
  - Distributed Real-time
  - builker management, send control blocks, asynchronous and synchronous operations, bounded blocking, message priorities, message labels, and implementation protocols

POSIX - support by different operating systems

<table>
<thead>
<tr>
<th>Standard</th>
<th>Solaris</th>
<th>IRIX</th>
<th>LynxOS</th>
<th>QNX Neutrino</th>
<th>Linux</th>
<th>VxWorks</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX 1003.1</td>
<td>Full support</td>
<td>Full support</td>
<td>Full support</td>
<td>Partial support (no memory locking)</td>
<td>Full support</td>
<td>Partial support (different process model)</td>
</tr>
<tr>
<td>POSIX 1003.1b</td>
<td></td>
<td>Conformant</td>
<td>Full support</td>
<td>Partial support (no timers, no message queues)</td>
<td>Full support</td>
<td></td>
</tr>
<tr>
<td>POSIX 1003.1c</td>
<td></td>
<td>Conformant</td>
<td>Full support</td>
<td>Conformant</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


POSIX - example: setting a timer (cont.)

```c
/* set the initial expiration and frequency of timer */
tmr_setting.it_value.tv_sec = 1;   tmr_setting.it_value.tv_nsec = 0;
tmr_setting.it_interval.tv_sec = num_secs;
tmr_setting.it_interval.tv_nsec = num_nsecs;
if ( timer_settime(timer_h, 0, &tmr_setting,NULL) < 0)
    perror('settimer');
/* wait for signals */
sigset_t allsigs;
while (1) {
    sigsuspend(&allsigs);
}
/* routine that is called when timer expires */
void timer_intr(int sig, siginfo_t *extra, void *cruft){
    /* perform periodic processing and then exit */
}
```

POSIX - example: setting a timer

```c
void timer_create(int num_secs, int num_nsecs)
{
    struct sigaction sa;
    struct sigevent sig_spec;
    sigset_t allsigs;
    struct itimerspec tmr_setting;
    timer_t timer_h;
    /* setup signal to respond to timer */
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = timer_intr;
    if (sigaction(SIGRTMIN, &sa, NULL) < 0)
        perror('sigaction');
    sig_spec.sigev_notify = SIGEV_SIGNAL;
    sig_spec.sigev_signo = SIGRTMIN;
    /* create timer, which uses the REALTIME clock */
    if (timer_create(CLOCK_REALTIME, &sig_spec, &timer_h) < 0)
        perror('timer create');
}
```

POSIX - other languages

POSIX is a ‘C’ standard...

... but bindings to other languages are also (suggested) POSIX standards:

- **Ada**: 1003.5*, 1003.24
  (some PAR approved only, some withdrawn, some (partly) implemented)

- **Fortran**: 1003.9 (6/92)
  **Fortran90**: 1003.19 (withdrawn)

... and there are POSIX standards for task-specific POSIX profiles, e.g.:

- **Super computing**: 1003.10 (6/95)
- **Realtime**: 1003.13, 1003.13b (3/98) - profiles 51-54
  combinations of the above RT-relevant POSIX standards — RT-Linux

- **Embedded Systems**: 1003.1 (PAR approved only)

Assembler level programming

**Macro-Assemblers**

- Closest to hardware.
- Predictable results (as predictable as the underlying hardware).
- Small footprint.
- As sequential or concurrent as the underlying hardware.
- No abstraction or support for large systems.
- Basic types are defined by the deployed processor (similar to C).
- Hard to read.
- Used mostly in very small applications.
Real-Time Programming Languages

Languages mentioned so far

- Real-Time Java (Real-Time Specification for Java 1.1) — for soft real-time applications
- Esterel (Esterel v7) — an alternative for high-integrity applications
- VHDL — compile real-time data and control paths into hardware
- Timed CSP (as used and developed since 1986) — an algebraic approach
- PEARL (PEARL-90) — a traditional language, specialized on plant modelling
- POSIX (POSIX 1003.1b, ...) — the world of bare bone integers and semaphores
- Assemblers — the world of bare bone words

Assignments are designed so that they can be solved with reasonable effort in Ada.

Summary

Introduction & Real-Time Languages

- Features (and non-features) of a real-time system
  - Features, definitions, scenarios, and characteristics.
- Components of a real-time system
  - Converters, interfaces, sensors, actuators, communication systems, controllers, ...
- Software layers of a real-time system
  - Algorithms, operating systems, protocols, languages, concurrent and distributed systems.
- Real-time languages criteria
  - Mostly high integrity, predictable languages with means for explicit time scopes.
- Examples of actual real-time languages