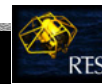


RES



Time & Space

Uwe R. Zimmer – The Australian National University



Real-Time & Embedded Systems



References for this chapter

[Ada95RM] (link to on-line version)

Ada Working Group
ISO/IEC JTC1/SC 22/WG 9
Ada 95 Reference Manual
– Language and Standard Libraries
ISO/IEC 8652:1995(E) with COR.1:2000,
June 2001

[Burns01]

Alan Burns and Andy Wellings
Real-Time Systems and Programming Languages
Addison Wesley, third edition, 2001

[Dourish01]

Paul Dourish
Where the Action Is
MIT Press, Cambridge, Massachusetts,
London, England, 2001

[Bollella01]

Greg Bollella, Ben Brosgol, Steve Furr, David Hardin, Peter Dibble, James Gosling, Mark Turnbull & Rudy Belliard
The Real-Time Specification for Java
<http://www.rti.org>

[Kligerman86]

E. Kligerman, A. D. Stoyenko
Real-Time Euclid:
A language for reliable real-time systems
IEEE transactions on Software Engineering SE-12(9): 941-949

all references and links are available on the course page



Real-Time & Embedded Systems



Notions of time and space

The big topics:

What is time? / What is embodiment?

Interfacing with time

Specifying timing requirements

Satisfying timing requirements



Real-Time & Embedded Systems



Notions of time and space

What is time?

Do we exist in time, or is time part of our existence?

☞ Is time an intrinsic property of nature? ☞ Platonism

Time is an external phenomenon. Thus simultaneous events happen at the same exact absolute time. There is the underlying assumption that time is progressing, even if no changes can be observed.

☞ Is time a construct which is based on observable events? ☞ Reductionism

Time is the observation of distinguishable events. If the observed events are 'regular', a useful time-reference can be constructed. If all possible observers detect one event before another one, they are said to be in sequence. If this cannot be assumed for all possible observers, they are said to be simultaneous. Therefore the notion of time is reduced to a **notion of causality**.

☞ Is time 'linear' between observable events?



What is time?

A mathematical notion of time

- Transitivity: $x < y \wedge y < z \Rightarrow x < z$
- Linearity: $x < y \vee x > y \Rightarrow x \neq y$
- Irreflexivity: $\neg(x < x)$
- Density: $x < y \Rightarrow \exists z | x < z \wedge z < y$

☞ Real clocks have **limited** resolutions and are running **asynchronously!**

What time is it?

UT0	1884: Mean solar time at Greenwich meridian
UT1	Corrected UT0, ☞ polar motion
UT2	Corrected UT1, ☞ variations in the speed of rotation of the earth
IAT (International Atomic Time)	a caesium 133 atomic clock (current accuracies: one miss in 10^{13} ticks, e.g. approximately once every 300000 years)
UTC (Universal Time Coordinated)	a IAT clock, which is synchronized to UT2 (by introducing occasional leap ticks) – difference between UTC and IAT is $< 0.5s$
'1 sec.'	1/86400 of a mean solar day ... or ... 1/31566925.9747 of the tropical year for 1900 (Ephemeris Time defined 1955) ... or ... 9192631770 periods of the radiation corresponding to the transition between two hyperfine levels of the ground state of the caesium 133 atom



What is time? – A physical notion of time

- 1676: **Rømer** proves the existence of the speed of light (and measures it).
- 1687: **Newton's** "Principia Mathematica" assumes an absolute time, independent of space itself and independent of events.
- 1905: The concept of absolute time is destroyed first by **Einstein** and (a few weeks later) by **Poincaré**.
- 1915: **Einstein's** general theory of relativity eliminates the independence of time (space) and events in time (space).
 - ☞ One principal consequence for measurements of time:

Clocks under higher gravity or in faster observation frames are slower
Practical consequences: clocks in satellites need to be adjusted accordingly



What is time?

How real is real-time?

- 'Real'-time ☞ usually: time as given by *external* sources.

Engineering: it is of no importance how time is defined and understood, as long as an '**external reference**' is given and used as '**real-time**'.



What time is it?

Time frames:

Generating a time frame?

- by a timer generating a regular interrupt
- by employing a RTC-module

Using an existing time-frame?

- by employing time-stamps or sequence numbers of received sensor-readings
- by a radio receiver for UTC or IAT (available in some countries)



What time is it?

Generating a time frame?

- by a timer generating a regular interrupt
- by employing a RTC-module

Using an existing time-frame?

- by employing time-stamps or sequence numbers of received sensor-readings
- by a radio receiver for UTC or IAT (available in some countries)

Common programming languages guarantee

a **'resolution'** and **'accuracy'** of time (in reference to 'a second'),
not its origin or meaning.



What is embodiment?

Working hypothesis:

- Embodied phenomena are those that by their very nature occur in real time and real space.



What is embodiment?

Phenomenology:

- The phenomena of experience as the central aspects and building blocks of understanding.
 (rather than: "finding the 'truth'")

applied to and trying to combine aspects of

- Ontology** (about the nature of being and categories of existence)
- Epistemology** (the study of knowledge)



What is embodiment?

Edmund Husserl (1859-1938, Vienna, Halle, Göttingen, Freiburg):

- Founder of the phenomenological tradition
... as a trial to establish modern science which is firmly grounded on the phenomena of experience (instead of being an abstract mathematical construct).
- Phenomenology originally as a method to examine the nature of intentionality
- Coined the terms
 - **Noema**: the objects of consciousness
 - **Noesis**: the mental experiences of those objects
 - **Lebenswelt** (life-world): the inter-subjective world of everyday-experience

☞ Husserl rejected pure abstract and formalized reasoning



What is embodiment?

Martin Heidegger (1889-1976, Freiburg):

- Moved phenomenology from a discussion about mental phenomena separated from the physical world (Cartesian dualism)
☞ to a discussion about **connected physical and mental phenomena**.
- Moved the central questions from epistemology to ontology ('*Being and Time*', 1927)

The meaning is not 'in the head' but in the world!

- Coined terms:
 - **Dasein** (being-in-the-world): 'being as inseparable from the world in which it occurs'
☞ being as always purposeful and active
☞ the world as an unconscious but accessible background
 - **Zuhanden** (ready-to-hand): 'equipment as a part of actual interaction with the world'
 - **Vorhanden** (present-at-hand): 'equipment as a conscious model'



What is embodiment?

Maurice Merleau-Ponty (1908-1961, Paris (Sorbonne)):

- 'The Phenomenology of Perception' (1945)
- Embodiment has three implications:
 - a body as a physical entity
 - a body as a set of physical skills and situated responses gained from the physical world
 - a body as a set of 'cultural skills' gained from the cultural world in which it is embedded
- Embodied perception as a bi-directional sensation and a basis for empathy (Perception in itself does not exist).

☞ see also: Phenomenology of Jean-Paul Sartre

Recent works in robotics (and insights about biological sensors) blur the line between action and perception even further.



What is embodiment?

back to the working hypothesis:

- ☞ Embodied phenomena are those that by their very nature occur in real time and real space.

refinement:

- ☞ Embodiment is the property of any engagement with the real world which (may) makes this engagement meaningful.

(Paul Dourish)

- ☞ Embodied phenomena are the essence of meaningful interaction

(Real-time and embedded systems are the technical instantiations of embodiment)

What is embodiment?

Implications:

- There is *no such thing* as 'intelligence', 'autonomy' or any other cognitive process, which is *independent of a physical environment*.
- There is *no such thing* as a universal system or body (mechanical design, robot, device, ...) which is *operational in all physical environments*.

What is embodiment?

Meaningfully embedded systems are part of an 'ecological niche' (Rolf Pfeifer):

- The operational environment** is supportive and employed by the system
 - The embedded system** is constructed as a part of the operational environment and according to the task
 - The task** is meaningful considering the morphology and cognitive ability of the system as well as the response from the environment
- i.e. being *situated, embodied, and self-sufficient*

What is embodiment?

- Embodied skills depend on **a tight coupling between perception and action**
... up to the level where the distinction between both can become difficult
- Tight coupling between perception and action means **to operate under real-time constraints**
... and to construct meaningful morphologies

The big topics:

What is time? / What is embodiment?

Interfacing with time

Specifying timing requirements

Satisfying timing requirements



Interfacing with time

What time is it in ...

	Resolution (syntactical)	Range (all time variables)	Requested resolution (clock ticks)	Actual resolution (detectable?)
JAVA	ms	undefined	undefined	no
RT JAVA	ms, ns	undefined	undefined	yes
Ada95	ms, μ s, ns	>50 years	<1ms	yes
POSIX threads	ms, ns	undefined	<20ms	yes
C	integer (as seconds)	undefined	undefined	no



Notions of time in Ada

Ada.Real-Time package

```
package Ada.Real_Time is
  type Time is private;
  Time_First : constant Time;
  Time_Last  : constant Time;
  Time_Unit  : constant := 10*1.0#E-9; -- ns

  type Time_Span is private;
  Time_Span_First : constant Time_Span;
  Time_Span_Last  : constant Time_Span;
  Time_Span_Zero  : constant Time_Span;
  Time_Span_Unit  : constant Time_Span;

  Tick : constant Time_Span; -- actual clock resolution < 1 ms

  function Clock return Time;

  (... operations on and conversions with time and time_span ...)
end Ada.Real_time
```



Notions of time in RT-Java

RT-Java time classes

Time root class:

```
public abstract class HighResolutionTime implements java.lang.Comparable
  direct known subclasses: AbsoluteTime, RelativeTime, RationalTime
```

- similar to Ada.Real-Time, but no requested accuracy
- adds the concept of frequency ('rational time'), but does not guarantee for equidistant instantiations.

Clock Class:

```
public abstract class Clock
{
  public static Clock getRealtimeClock ();
  public abstract RelativeTime getResolution ();
  public abstract void setResolution (RelativeTime resolution);
}
```



Notions of time in POSIX

Real-time clock interface in POSIX

```
#define CLOCK_REALTIME ...; // clockid_t type
struct timespec {
  time_t tv_sec; /* number of seconds */
  long tv_nsec; /* number of nanoseconds */
};

typedef ... clockid_t;

int clock_gettime (clockid_t clock_id, struct timespec *tp);
int clock_settime (clockid_t clock_id, const struct timespec *tp);
int clock_getres (clockid_t clock_id, struct timespec *res);

int clock_getcpuclockid (pid_t pid, clockid_t *clock_id);
int clock_getcpuclockid (pthread_t_t thread_id, clockid_t *clock_id);

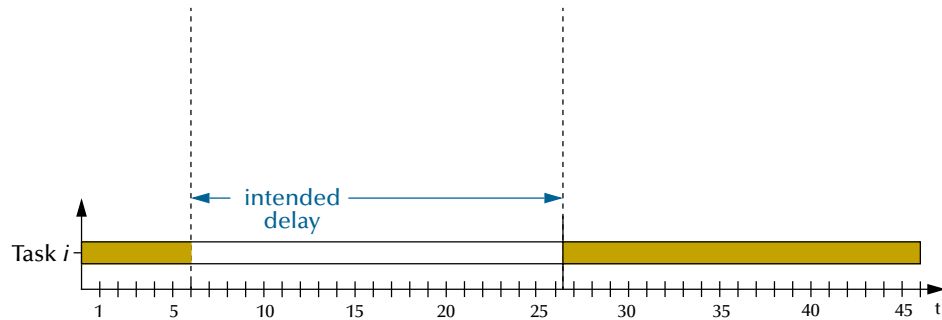
int nanosleep (const struct timespec *rqtp, struct timespec *rmtp);
/* nanosleep return -1 if the sleep is interrupted by a */
/* signal. In this case, rmtp has the remaining sleep time */
```



Interfacing with time

Programming primitive 'Delay'

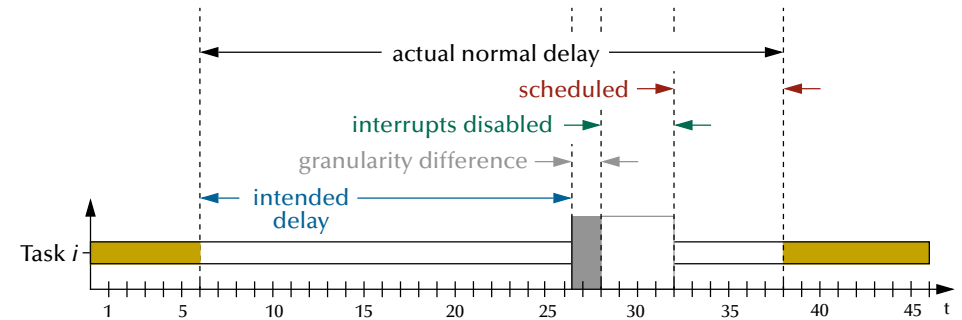
- Alternative for 'busy-wait' and interrupts: Suspend a task for a fixed time



Interfacing with time

Programming primitive 'Delay'

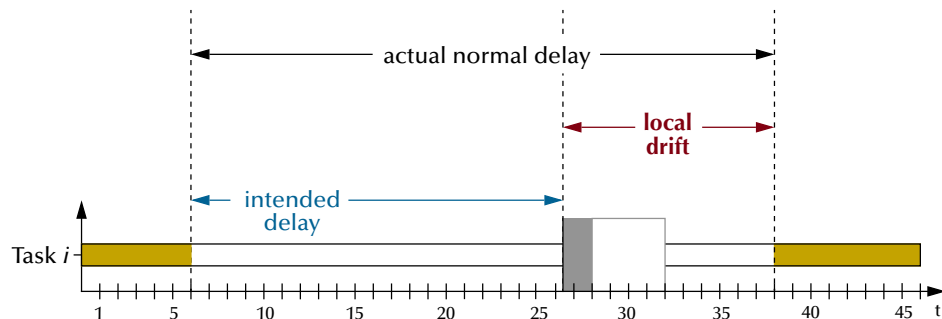
- Alternative for 'busy-wait' and interrupts: Suspend a task for a fixed time
 - but all these process or thread delays are precise only in their lower bound!



Interfacing with time

Programming primitive 'Delay'

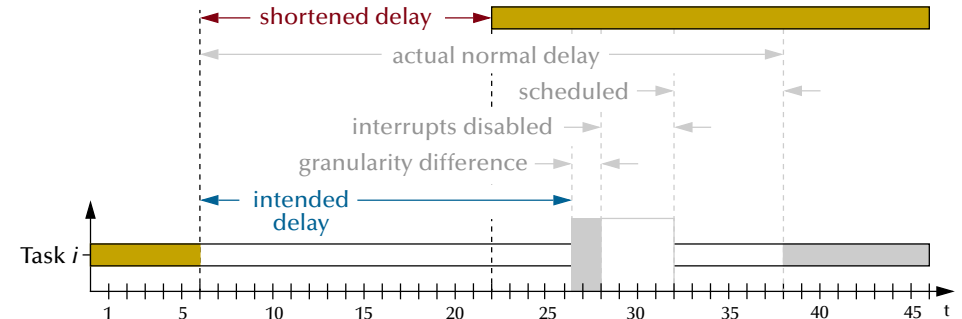
- Alternative for 'busy-wait' and interrupts: Suspend a task for a fixed time
 - local drift: sum of all additional actual delays



Interfacing with time

Programming primitive 'Delay'

- Alternative for 'busy-wait' and interrupts: Suspend a task for a fixed time
 - if task *i* is enabled for interrupts, it can also be activated earlier!



Interfacing with time

Relative delay

(Ada95)

```

task T;
task body T is
begin
  loop
    Action;
    delay 5.0; -- sec.
  end loop;
end T;

```

This loop will delay **at least** for 5 seconds:

local and **cumulative** drift effects here!

Interfacing with time

Relative as absolute delay?

(Ada95)

```

task body T is
Interval: constant Duration:= 5.0; -- sec.
Start_Time: Time;
begin
  loop
    Start_Time:= Clock;
    Action;
    delay Interval - (Clock - Start_Time);
  end loop;
end T;

```

Delay time calculation
is **not atomic!**

Interfacing with time

Relative as absolute delay?

(Ada95)

```

task body T is
Interval: constant Duration:= 5.0; -- sec.
Start_Time: Time;
begin
  loop
    Start_Time:= Clock;
    Action;
    delay Interval - (Clock - Start_Time);
  end loop;
end T;

```

Delay time calculation
is **not atomic!**

Interfacing with time

Absolute delay

(Ada95)

```

task body T is
Interval: constant Duration:= 5.0; -- sec.
Next_Time: Time;
begin
  Next_Time:= Clock + Interval;
  loop
    Action;
    delay until Next_Time;
    Next_Time:= Next_Time + Interval;
  end loop;
end T;

```

This loop will delay
on average for 5 seconds:

note that this also holds, if 'Action' is sporadically longer than 5 s

Interfacing with time

Zero delay?

(Ada95)

```

task body T is
begin
  loop
    Action;
    delay 0.0;
  end loop;
end T;

```

Allow explicitly
for a *task-switch*

- the delay statement does not only suspend the current task, but also potentially activates other tasks
- ☞ may be used to enable other processes on the same priority level

Interfacing with time

Delays

Absolute & relative delays are available in:

- Real-Time Java, Pearl, Ada95, ... and many other process-oriented languages

Only relative delays are available in some 'low-level' environments:

- POSIX: `nanosleep` (absolute delays need to be constructed employing timers and signals)

Only absolute delays are available in some 'harder' real-time environments:

- Occam2, Ada95 (Ravenscar profile), ...

Interfacing with time

Timeouts

As a third alternative to busy-waiting and infinite blocking, **timeouts** are implemented in:

- **Shared variable** communications
 - Semaphore
 - Conditional critical regions
 - Monitors
 - Protected objects
- **Message passing** between processes
 - Asynchronous and synchronous message transfers
 - Remote procedure calls
 - Remote objects
- **Actions**

Interfacing with time

Timeouts on semaphore

POSIX:

```

if (sem_timedwait (&call, &timeout) < 0) {
  if (errno == ETIMEDOUT) {
    /* timeout occurred, try something else */
  }
  else {
    /* some other error occurred, do something about it */
  }
}
else {
  /* semaphore is locked successfully, go ahead */
};

```

Suspend current process until the semaphore `call` is open or the time-span `timeout` has passed

Timeouts on entry calls

(same for task-entry calls (message passing) and protected object calls (monitors))

```
task body Sensor is
  T : Temperature;
begin
  loop
    -- find temperature T somewhere
    select
      Controller.Call(T);
    or
      delay 0.5; -- sec.
    -- action if temperature could not be delivered in time
    end select;
  end loop;
end Sensor;
```

Try calling for 500ms

Timeouts on incoming calls

```
task body Controller is
  Current_Temp : Temperature;
begin
  loop
    select
      accept Call (T: Temperature) do
        Current_Temp:= T;
      end Call;
    or
      delay 1.0; -- sec.
    -- action if the temperature was not available in time
    end select;
  -- normal processing
  end loop;
end Controller;
```

accept calls for a limited time-span of 1s

Timeouts on incoming calls

```
task body Controller is
  -- declarations
begin
  loop
    select
      accept Call (T: Temperature) do
        Current_Temp:= T;
      end Call;
    or
      delay until Deadline;
    -- no further calls before the deadline
    end select;
  -- normal processing
  end loop;
end Controller;
```

accept any number of calls until a closing time 'Deadline' for this entry

No-wait on incoming calls

```
task body Controller is
  -- declarations
begin
  loop
    select
      accept Shutdown do
        -- termination actions
        exit;
      end Shutdown;
    else
      -- normal operation
    end select;
  -- synchronize
  end loop;
end Controller;
```

synchronous alternative for an interrupt acceptance

Interfacing with time

No-wait on entry calls

(same for task-entry calls (message passing) and protected object calls (monitors))

```
task body Sensor is
  T : Temperature;
begin
  loop
    -- measure temperature T
    select
      Controller.Call(T);
    else
      -- action if temperature can not yet be delivered,
      -- e.g. further refine the measurement
    end select;
  end loop;
end Sensor;
```

Try delivering,
else refine the result further

Interfacing with time

Timeout on actions

- All above timeouts suspend / activate a process / task / thread at a synchronization point.
- Up to now, there wasn't any abortion of on-going actions, due to a timeout

to achieve this there need to be an

- **asynchronous change of control flows**
 - on the level of code-blocks: **Timeout on actions**
 - on the level of processes / tasks / threads: (investigated later)

Interfacing with time

Timeout on actions

(Ada95)

Monitor a code block:

```
select
  delay until Deadline;
  -- computations did not finish in time: take measures
then abort
  -- hard to predict sequence of computations
end select;
```

Interfacing with time

Timeout on actions

- Common RT-systems concept:

Timeliness is often more important than **Precision**

1. **Get a first approximation** in fixed amount of time and well before the deadline
2. Inspect the deadline and if there is enough spare time:
3. **Improve the result** and keep a record of improvements (while keeping an eye on the deadline)
 - 3-a If the most precise result is delivered before the deadline occurs: ✓
 - 3-b else use the closest approximation so far: ✓

- The deadline is fulfilled and there is a result in any case

Interfacing with time

Timeout on actions

Get a first approximation and employ spare time for refinements:

```
Deadline := ... -- set an absolute deadline for the computations
-- compulsory computations (save first result)
select
  delay until Deadline;
  Precise_Result := False;
then abort
  while Result_Can_Be_Improved loop
    -- optimising computations (save results after each iteration)
  end loop;
  Precise_Result := True;
end select;
-- use result
```

Take a first guess

improve, improve, improve, ...

Continue in time and with the best result possible

Interfacing with time

Timeout on actions

Time-base can also be given externally, e.g. via a protected call:

```
loop
  select
    Get_New_Data (Current_Sensor_Data);
    -- employ precise results based on previous data
    -- compulsory computations (save first result)
    -- employ first result on current data
  then abort
    while Result_Can_Be_Improved loop
      -- optimising computations (save results after each iteration)
    end loop;
  end select;
end loop;
```

New data? → abort current iterations

precise reactions

Take a first guess

first reactions

improve, – improve, – improve, ...

Interfacing with time

Timeout on actions

(RT-Java)

Similar methods in RT-Java:

```
public class Timed extends AsynchronouslyInterruptedException
  implements java.io.Serializable
{
  public Timed (HighResolutionTime time) throws IllegalArgumentException;
  public boolean doInterruptible (InterruptedException logic);
  public void resetTime (HighResolutionTime time);
}
```

☞ see full example of imprecise computations in *RT-Java* in the course textbook.

(timeouts on actions in *POSIX* need to be emulated employing timers and signals)

Notions of time and space

The big topics:

What is time? / What is embodiment?

Interfacing with time

Specifying timing requirements

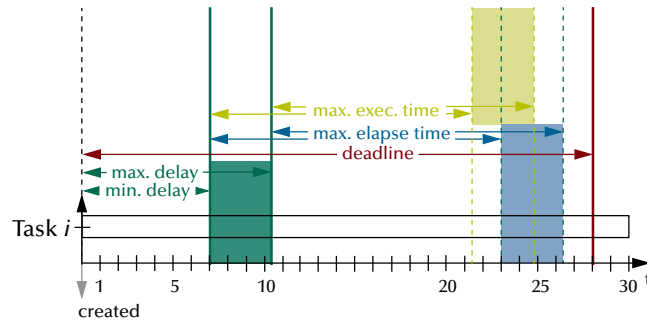
Satisfying timing requirements



Temporal scopes

Common attributes:

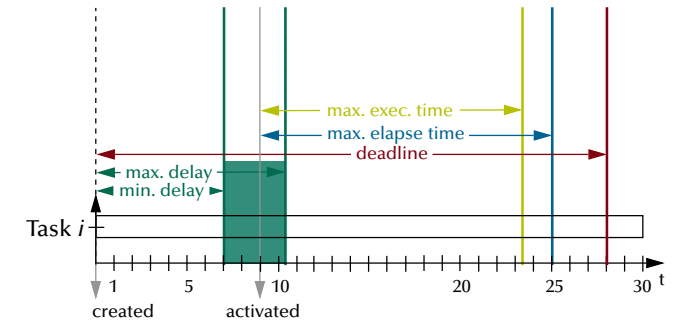
- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline



Temporal scopes

Common attributes:

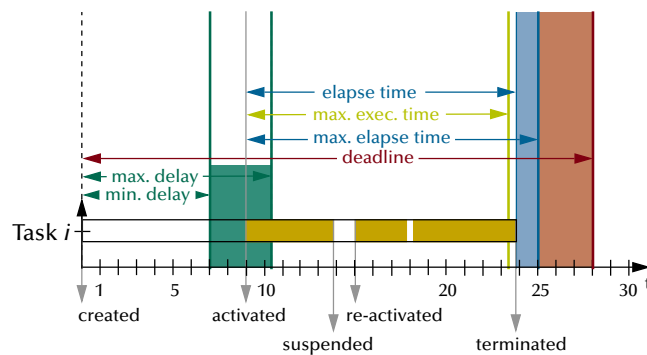
- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline



Temporal scopes

Common attributes:

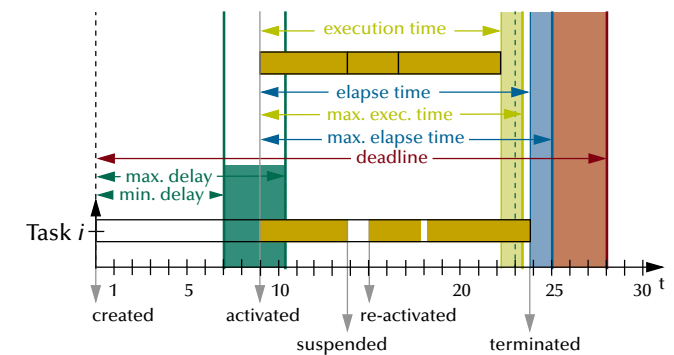
- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline



Temporal scopes

Common attributes:

- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline





Some common scope attributes

Temporal Scopes can be:

Periodic	– e.g. controllers, samplers, monitors
Aperiodic	– e.g. ‘periodic on average’ tasks, burst requests
Sporadic / Transient	– e.g. mode changes, occasional services

Deadlines (absolute, elapse, or execution time) can be:

Hard	– single failure leads to severe malfunction
Firm	– results are meaningless after the deadline – only multiple or permanent failures threaten the whole system
Soft	– results may still be useful after the deadline



Some common scope attributes

Temporal Scopes can be:

Periodic	– e.g. controllers, samplers, monitors
Aperiodic	– e.g. ‘periodic on average’ tasks, burst requests
Sporadic / Transient	– e.g. mode changes, occasional services

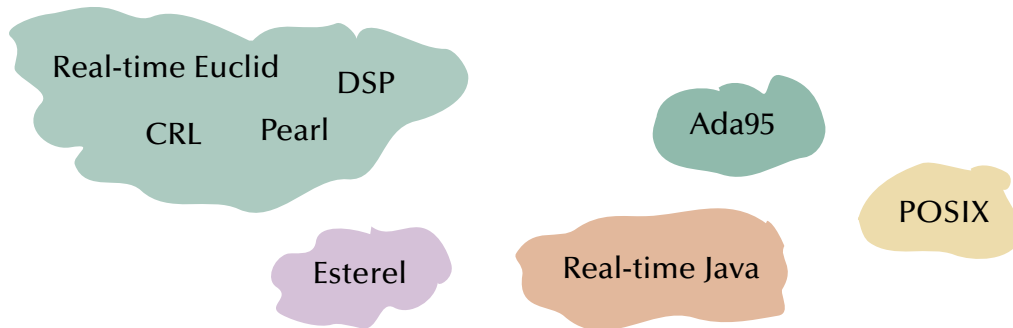
Deadlines (absolute, elapse, or execution time) can be:

Hard	– single failure leads to severe malfunction
Firm	– results are meaningless after the deadline – only multiple or permanent failures threaten the whole system
Soft	– results may still be useful after the deadline

distinction is not so obvious in practical systems



Language support for specifying temporal scopes



Real-time Euclid

Language features:

- Recursions and goto-statements are prohibited.
- Loops are restricted to time bounded loops.
- Processes are static and non-nested.

Time scopes:

- **periodic** <Frameinfo> **first activation** <TimeOrEvent>
- **atEvent** <ConditionalId> <Frameinfo>



Real-time Euclid

Periodic process example:

```

realTimeUnit := 1.0 % = 1 seconds
var Reactor      : module
var startMonitoring : activation condition atLocation 16#A10D
  process TempController : periodic
    frame 60
    first activation atTime 600
    or atEvent startMonitoring
  % import list
  %
  % execution part
  %
end TempController
end Reactor

```

Annotations:

- connect event to an interrupt-address (points to `atEvent startMonitoring`)
- time-scope specification (points to `frame 60`)
- no loop! (scheduler activates process) (points to `end TempController`)



Real-time Euclid

```

task body Temp_Controller is
  Next_Release : Duration;
begin
  select accept Start_Monitoring;
  or delay 600.0; -- sec.
  end select;
  Next_Release := Clock + 60.0; -- sec.
  loop
    -- execution part
    delay until Next_Release;
    Next_Release := Next_Release + 60.0; -- sec.
  end loop;
end Temp_Controller;

```

Ada-emulation of the above RT-Euclid program



Real-time Euclid

```

task body Temp_Controller is
  Next_Release : Duration;
begin
  select accept Start_Monitoring;
  or delay 600.0; -- sec.
  end select;
  Next_Release := Clock + 60.0; -- sec.
  loop
    -- execution part
    delay until Next_Release;
    Next_Release := Next_Release + 60.0; -- sec.
  end loop;
end Temp_Controller;

```

Annotations:

- Ada-emulation of the above RT-Euclid program (points to the code block)
- but: no formal time-scope specification! no schedulability analysis! (points to the code block)



Real-time Euclid

State:

Real-Time Euclid was suggested by Kligerman and Stoyenko in 1986

- Additional schedulability analysis modules became available
- Stayed in an academic context, but influenced many more recent RT-systems.



CRL

(a language for complex real-time systems)

Ways to handle 'dangerous constructs' (loops, recursions, synchronisations, ...):

- Exclude them
 - ☞ Real-time Euclid
- Expand them to become individually safe (e.g. by adding mandatory timeout)
- Attribute the code with additional constraints, enabling a full pre-runtime analysis ☞ CRL



CRL

(a language for complex real-time systems)

Constraints in CRL on:

- **Time:**
`timeconstraint`
`{use | nosoonerthan | nolaterthan <abs_time>} endtimeconstraint`
 (also relative constraints)
- **Iterations:** assert lower and upper limits for the number of iterations per block
- **Activations:**
`activationdeactivationconstraint`
`[periodic <Frameinfo> firstactive <TimeOrEvent>`
`| atEvent <ConditionalId> <Frameinfo>]`
`endactivationdeactivationconstraint`
- **Direct recursions:** assert lower and upper recursion limits (general recursions are not covered).



CRL

(a language for complex real-time systems)

Evaluation of the constraints and assertions:

- **Timing:**
 - verified at *compile-time*
- **Activation / Deactivation:**
 - checked for schedulability at *compile-time* and enforced by the scheduler at *run-time*.
- **Iterations and recursion:**
 - either verified at *compile-time* or checked at *run-time*.



CRL

(a language for complex real-time systems)

State:

CRL was suggested by Stoyenko, Marlowe and Younis in 1995

- Full featured language
- Compiled to attributed C++
- ☞ Stayed also in an academic context.



Pearl

Explicit time-scope expressions:

```

TaskStart ::= [StartCondition] ACTIVATE <task>
StartCondition ::= AT <time> [Frequency] |
                  AFTER <duration> [Frequency] |
                  WHEN <interrupt> [AFTER <duration> [Frequency]] |
                  Frequency
Frequency ::= ALL <duration>
            [{UNTIL <time> } |
             {DURING <duration>}]

```

- ☞ Schedulability analysis (at *compile-time* or *run-time*) possible (although not defined by the language)
- ☞ Pearl refinement: a combination of Pearl and Real-time Euclid ☞ **High-Integrity Pearl**



DSP: Distributed Programming System

Explicit time-scope expressions at the 'statement level':
e.g. time-scope for a software-engineer:

```

from 11:00 to 19:30 every 45 do
  start elapse 10 do
    setup_coffee_machine
    power_coffee_machine_up
    find_favorite_cup
    put_coffee_in_favorite_cup
    clean_coffee_machine
  end
  start after 3 elapse 25 by 20:00 do
    drink_coffee
  end
end

```

- ☞ DSP-compiler: breaks the sequences down in processes and schedules



Real-time Java

Real-time Java comes with:

- multiple sets of predefined time-scope parameters
- a scheduler class (with a predefined priority scheduler)
- ☞ Schedulability (feasibility) analysis possible.



Real-time Java

```

public abstract class Scheduler
{
  protected Scheduler ();
  protected abstract boolean addToFeasibility (Schedulable s);
  public abstract void fireSchedulable (Schedulable s);
  public abstract boolean isFeasible ();
  protected abstract boolean removeFromFeasibility (Schedulable s);
  public boolean setIfFeasible (Schedulable s,
                               ReleaseParameters r,
                               MemoryParameters m);
  ...
}

```

Formulates an on-line schedulability analysis!

Real-time Java

```
public class PriorityScheduler extends Scheduler
```

```
{
  public static final int MAX_PRIORITY;
  public static final int MIN_PRIORITY;
  protected PriorityScheduler ();
  protected boolean addToFeasibility (Schedulable s);
  public void fireSchedulable (Schedulable s);
  public boolean isFeasible ();
  protected boolean removeFromFeasibility (Schedulable s);
  public boolean setIfFeasible (Schedulable s,
                                ReleaseParameters r,
                                MemoryParameters m);
  ...
}
```

This PriorityScheduler is the only requested instantiation

Real-time Java

```
public class RealtimeThread extends java.lang.Thread
  implements Schedulable
```

```
{
  public RealtimeThread (SchedulingParameters s,
                        ReleaseParameters r,
                        MemoryParameters m,
                        MemoryArea a);
  public synchronized void addToFeasibility ();
  public synchronized void addIfFeasible ();
  public static RealtimeThread currentRealtimeThread () throws ...;
  public synchronized void schedulePeriodic ();
  public synchronized void deschedulePeriodic ();
  public boolean waitForNextPeriod () throws ...;
  public synchronized void interrupt ();
  public static void sleep (...) throws ...;
  ...
}
```

Priority

Periodic, aperiodic, or sporadic parameters

Real-time Java

```
public class NoHeapRealtimeThread extends RealtimeThread
```

```
{
  public RealtimeThread (SchedulingParameters s,
                        ReleaseParameters r,
                        MemoryArea a) throws ...;
  ...
}
```

This thread is allowed to interrupt any garbage collector at any time!
(since it doesn't depend on it itself)

Real-time Java

```
public abstract class SchedulingParameters
```

```
{
  public SchedulingParameters ();
}
public class PriorityParameters extends SchedulingParameters
{
  public PriorityParameters (int priority);
  public int getPriority ();
  public void setPriority (int priority) throws ...;
  ...
}
```

'Priority' is the only default scheduling parameter

Real-time Java

```
public abstract class ReleaseParameters
{
    protected ReleaseParameters
        (RelativeTime cost,
         RelativeTime deadline,
         AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

    public RelativeTime getCost();
    public AsyncEventHandler getCostOverrunHandler();
    public RelativeTime getDeadline();
    public AsyncEventHandler getDeadlineMissHandler();
}
```

Cost is an estimate of the max. execution time

Measuring execution time is *not* requested, i.e. the overrunHandler might never be activated!

Real-time Java

```
public class PeriodicParameters extends ReleaseParameters
{
    public PeriodicParameters
        (HighResolutionTime start,
         RelativeTime period,
         RelativeTime cost,
         RelativeTime deadline,
         AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

    public RelativeTime getPeriod();
    public HighResolutionTime getStart();
    public void setPeriod(RelativeTime period);
    public void setStart(HighResolutionTime start);
}
```

most frequently used release parameters

Real-time Java

```
public class AperiodicParameters extends ReleaseParameters
{
    public AperiodicParameters
        (RelativeTime cost,
         RelativeTime deadline,
         AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);
}
```

these are the minimum release parameters (while cost might be used for feasibility analysis only)

the deadline-missHandler need to be supplied in any implementation

Real-time Java

```
public class SporadicParameters extends AperiodicParameters
{
    public SporadicParameters
        (RelativeTime minInterarrival,
         RelativeTime cost,
         RelativeTime deadline,
         AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

    public RelativeTime getMinimumInterarrival();
    public void setMinimumInterarrival(RelativeTime minimum);
}
```

Sporadic events are not allowed to come in bursts!

Real-time Java

The minimal required implementation supplies:

- ☞ Priority scheduling
- ☞ On-line schedulability analysis
- ☞ Deadline violation handlers
- ☞ (max. execution time deadline checks are suggested but not required)
- ☞ a sporadic scheduler is not required (although the sporadic release parameter set is).

Hard real-time environments require the exclusive usage of
'No heap real-time threads'

(synchronization with an 'object-oriented' thread invalidates the feasibility assurances)

Ada95

Ada95 has *no* explicit time-scope expressions at task-level.

Ada95 offers ...

- tasking
- a priority scheduler (the only required scheduler)
- synchronisation and communication primitives
- asynchronous transfer of control, timed calls, timeout on actions ... etc. pp.
- ... **but no hardware timers!**

... to create the basis for most kinds of hard real-time-scopes manually.

☞ **but no automatic schedulability analysis!**

Esterel

Since Esterel is a synchronous language, ...

... all actions and communications take zero time by definition.

- ☞ There is no expression for continuous, non-zero time-scopes.
- ☞ Time is interpreted in the reductionist way as a sequence of events
- ☞ Time-scopes translate to signal-relations and signal-counters

Continuous time scopes need to be taken into account while

1. **analysing and reducing** the problem to a zero-time atomic system
2. **implementing** the synchronous system on an actual system.

- ☞ Continuous time-scopes for the ...

... **validation** of the zero-time assumption!

POSIX

the usual: ☞ *use timers!*

common combination:

- ☞ usage of **Ada95** together with **POSIX timers** as a basis for hard-real-time-scopes and schedulers



The big topics:

What is time? / What is embodiment?

Interfacing with time

Specifying timing requirements

Satisfying timing requirements



Two paths towards fulfilling rt-requirements:

☞ **Real-time logic approach**

- formal, correct in its specifications, & offers calculus for asynchronous, real-time systems
- ↳ needs to ignore most real world effects, like jitters, drifts, failures, interferences, etc. pp.
- gives a **correct solution** according to the specification

☞ **Complex systems oriented approach**

- deals with existing computer systems, sensors, & offers a set of approximating methods
- ↳ not complete or correct in any formal sense
- deals with **real-world systems**, gives 'robust' systems, passes rigorous experiments



Fulfilling rt-requirements:

Complex systems oriented approach:

- **System identification and compile-time analysis:**
 - Calculate or limit statement durations
 - Calculate or limit iterations and recursions
 - Analyse potential dead- or life-locks ☞ **chapter 8**
 - Calculate schedulability ☞ **chapter 7**
- **Run-time analysis and checks:**
 - Dynamic scheduling schemes: Re-validate schedulability ☞ **chapter 7**
 - Check for all constraints and assertions at run-time ☞ **chapter 9**
- **Supply fault-tolerant behaviours:**
 - Error recoveries, mode changes, ... ☞ **chapter 9**



Fulfilling rt-requirements:

Real-time logic approach:

- **Reduce the problem:**
 - Reduce any asynchronous, analogue, dynamical, fractal, jitter-, drift, or failure-affected parts of the system to a fully synchronous and discrete system ☞ **chapters 5 and 6**
 - Formulate the specification on the basis of the reduced synchronous system.
- **Verify the reduced system:**
 - Verify the reduced synchronous against the specifications ☞ not covered in this course
- **Compile the reduced system to an actual system:**
 - The resulting actual system will be executable on a real machine and employ real devices
- ☞ **Re-check the actual system** (e.g. by means of a complex systems-approach) ☞ ...



Summary

Time & Space

- **What is time? / What is embodiment?**
 - Approaches by different faculties to understand the basis for this course
- **Interfacing with time**
 - Formulating local time-dependent constraints
 - Access time, delay processes, detect timeouts (in different languages)
- **Specifying timing requirements**
 - Formulating global timing-constraints
 - Understanding time-scope parameters (and expressing them in different languages)
- **Satisfying timing requirements**
 - Real-time logic and complex systems approach