The big topics:

- What is time? / What is embodiment?
- Interfacing with time
- Specifying timing requirements
- Satisfying timing requirements

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.

Can time be interpolated between observable events?
Notions of time and space

What is time?

A physical notion of time:

• 1676: Rømer proofs 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.

A mathematical notion of time:

• Transitivity: \( x < y \land y < z \Rightarrow x < z \)
• Linearity: \( x < y \lor y < x \Rightarrow x \neq y \)
• Irreflexivity: \( \neg (x < x) \)
• Density: \( x < y \Rightarrow \exists z \mid x < z \land z < y \)
• Continuity: \( x < y \Rightarrow \forall z \mid x + z < y + z \)

Real clocks run asynchronously.

Real clocks have limited resolution.
Notions of time and space

What time is it?

“How real is real-time?”

“Real-time” clock usually understood as an external time source.

Real-time systems as an engineering discipline:

“Real-time” in engineering is the notion of an actual, accessible, generated or external, measured-by-events reference time.

Moon calendars since about 4000 BC
Sundials and water clocks since about 1500 BC
Solar calendars since about 500BC
Spring based clocks since about the 13th century
Gregorian calendar 1582: corrections to previous solar calendars
Pendulum clocks since 1656
Greenwich Mean Time GMT 1675: Royal Greenwich observatory founded
Universal time UT0 1884: Mean solar time at Greenwich meridian
Quartz crystal clocks since 1927
Universal time UT1 Continuously updated corrections to UT0, i.e. polar motion
Universal time UT2 Continuously updated corrections to UT1, i.e. variations in the speed of rotation of the earth

International Atomic Time TAI 1955: A caesium 133 atomic clock (current accuracies: one miss in $10^{13}$ ticks, e.g. approximately once every 300,000 years)
Ephemeris Time ET0 1961: Based on observing the motion of the solar system.
Universal Time Coordinated UTC since 1964: An TAI based clock which is synchronized to UT1 (by introducing occasional leap ticks). [UTC – TAI] < 0.5s
Ephemeris Time ET1 1967: Corrections to the ET0 model
Ephemeris Time ET2 1972: Corrections to the ET1 model

‘1 second’

1 / 86,400 of a mean solar day.
1 / 31,566,925.9747 of the tropical year for 1900
(Ephemeris Time as defined in 1955).
9,192,631,770 periods of the radiation corresponding to the transition between two hyper-fine levels of the ground state of the caesium 133 atom.

Generating a time frame

- by any local timer generating regular interrupts.
- by employing a RTC-module (a timer based on the notion of seconds).

Using an existing time-frame

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

Common programming languages guarantee a ‘resolution’ and ‘accuracy’ of time (in reference to ‘a second’) … not its origin or meaning.
Notions of time and 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)

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.

Husserl rejected pure abstract and formalized reasoning.
Notions of time and space

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’
  - 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’

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 embodied
- Embodied perception as a bi-directional sensation and a basis for empathy (perception in itself does not exist).
- Recent works in robotics (and insights about biological sensors) blur the line between action and perception even further.

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)

Implications:
- There is no such thing as Universal ‘intelligence’, ‘autonomy’, ‘conscience’ or any other cognitive process which is independent of a physical environments.
- There is no such thing as a Universal morphology (mechanical design, robot, device, …) which is useful or even operational in all physical environments.
Notions of time and space

What is embodiment?

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

Implications:

1. There is no such thing as a universal intelligence, autonomy, conscience or any other cognitive process which is independent of a physical environment.

2. An actual or potential existence of a universal intelligence or a universal robot can of course not be disproved.

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.

Meaningful embedded systems have a purpose and are situated, embodied, and self-sufficient.

The big topics:

What is time? / What is embodiment?

- Interfacing with time
- Specifying timing requirements
- Satisfying timing requirements

Embodied skills as part of a meaningful embedded system thus depend on

A tight coupling between perception and action

... up to the level where the distinction between both can become difficult.

This requires:

- To operate under real-time constraints: Real-time systems
- To construct meaningful morphologies: Embedded system
Notions of time in Ada

Ada.Real-Time.Timing_Events

package Ada.Real_TIME.Timing_Events is
    type Timing_Event is tagged limited private;
    type Timing_Event_Handler is access protected procedure (Event : in out Timing_Event);

    procedure Set_Handler (Event : in out Timing_Event;
                           At_Time : in Time;
                           Handler : in Timing_Event_Handler);

    procedure Set_Handler (Event : in out Timing_Event;
                           In_Time : in Time_Span;
                           Handler : in Timing_Event_Handler);

    function Current_Handler (Event : Timing_Event) return Timing_Event_Handler;

    function Time_Of_Event (Event : Timing_Event) return Time;

    procedure Cancel_Handler (Event : in out Timing_Event;
                             Cancelled : out Boolean);

    private ... -- not specified by the language
end Ada.Real_Time.Timing_Events;

Interface with time

What time is it in ...

<table>
<thead>
<tr>
<th></th>
<th>Syntactical resolution</th>
<th>Required range</th>
<th>Required resolution</th>
<th>Actual resolution detectable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>ms</td>
<td>undefined</td>
<td>undefined</td>
<td>no</td>
</tr>
<tr>
<td>RT Java</td>
<td>ms, ns</td>
<td>undefined</td>
<td>undefined</td>
<td>yes</td>
</tr>
<tr>
<td>Ada</td>
<td>ms, µs, ns</td>
<td>&gt; 50 years</td>
<td>&lt; 1 ms</td>
<td>yes</td>
</tr>
<tr>
<td>POSIX</td>
<td>int (as seconds)</td>
<td>undefined</td>
<td>&lt; 20 ms</td>
<td>yes</td>
</tr>
<tr>
<td>Hardware timers</td>
<td>1/6 seconds typically 10ns ... 1 µs</td>
<td>2/3 seconds typically 100ms ... 100's years</td>
<td>configurable</td>
<td>yes</td>
</tr>
</tbody>
</table>

Notions of time in RT-Java

RT-Java time classes

package Ada.Real-Time.Timing_Events is
    type Timing_Event is tagged limited private;
    type Timing_Event_Handler is access protected procedure (Event : in out Timing_Event);

    procedure Set_Handler (Event : in out Timing_Event;
                           At_Time : in Time;
                           Handler : in Timing_Event_Handler);

    procedure Set_Handler (Event : in out Timing_Event;
                           In_Time : in Time_Span;
                           Handler : in Timing_Event_Handler);

    function Current_Handler (Event : Timing_Event) return Timing_Event_Handler;

    function Time_Of_Event (Event : Timing_Event) return Time;

    private ... -- not specified by the language
end Ada.Real_Time.Timing_Events;
Interfacing with time

Programming primitive ‘Delay’

Semantic: Suspend the current task immediately and for (at least) a predefined time span or until (at least) a predefined absolute time.

“Local delay drift” summarizes all additional (unintended) delays.

Interfacing with time

Programming primitive ‘Timer’

Semantic: Activate a specified routine after a predefined time span or at a predefined absolute time.

“Local delay drift” summarizes all additional (unintended) delays.
Interfacing with time

Relative delay

(Ada)

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

This loop will delay for at least 5 seconds. 

$\Rightarrow$ Local and cumulative drift!

Absolute delay

(Ada)

```ada
task body T is
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. 

$\Rightarrow$ Local drift only!

Interfacing with time

Absolute delay implemented by relative delay?

(Ada)

```ada
task body T is
begin
  Loop
    Start_Time := Clock
    Action;
    delay Interval - (Clock - Start_Time);
  end loop;
end T;
```

Absolute delay implemented by relative delay?

(Ada)

```ada
task body T is
begin
  Loop
    Start_Time := Clock
    Action;
    delay Interval - (Clock - Start_Time);
  end loop;
end T;
```

$\Rightarrow$ Delay time calculation is not atomic!
Interfacing with time

Zero delay
(Ada)

```ada
task T;
task body T is
begin
  loop
    Action;
    delay 0.0; -- sec.
  end loop;
end T;
```

Delay statements activate the scheduler.

- `delay 0.0` allows the scheduler to activate a runnable task of at least the same priority.
- (Real-time systems do not schedule in a time-slicing fashion, but switch tasks on events only.)

Yield statements activate the scheduler.

- `Yield` statements allow the scheduler to activate a runnable task of at least the same priority.
- (Real-time systems do not schedule in a time-slicing fashion, but switch tasks on events only.)

Relative regular timer
(Ada)

```ada
protected Event_Handlers is
  procedure Timer_Routine (Event : in out Timing_Event) ;
private
  Interval : constant Duration := 5.0; -- sec.
end Event_Handlers;
protected body Event_Handlers is
  procedure Timer_Routine (Event : in out Timing_Event) is
    begin
      Action;
      Set_Handler (Event, Interval, Timer_Routine'access);
      end Timer_Routine;
    end Event_Handlers;
```

Timer_Routine will activate after at least 5 seconds.
- Local and cumulative drift!
Interfacing with time

Absolute regular timer

(Ada)

```ada
protected Event_Handlers is
  procedure Timer_Routine (Event : in out Timing_Event));
private
  Interval : constant Duration := 5.0; -- sec.
  Next_Time : Time := Clock + Interval;
end Event_Handlers;
protected body Event_Handlers is
  procedure Timer_Routine (Event : in out Timing_Event) is
    begin
      Action;
      Next_Time := Next_Time + Interval;
      Set_Handler (Event, Next_Time, Timer_Routine'access);
    end Timer_Routine;
end Event_Handlers;
```

Timer_Routine will activate on average every 5 seconds.

Local drift only!

If executed in a real-time system, the drift will be significantly smaller compared to delay statements.

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

Absolute & relative delays and timers are available in:

- Real-Time Java, Pearl, Ada, ... and many other real-time oriented languages.

Only absolute delays and timers are available in strict real-time systems:

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

Only relative delays or timers are available in ‘low-level’ environments:

- POSIX: `nanosleep` (absolute delays need to be constructed by employing timers and signals as well as setting interrupt masks).

Timeouts on semaphores

(POSIX)

Suspend current process until the semaphore call is open or the relative timeout has passed:

```c
if (sem_timedwait (&call, &timeout) < 0) { /* something went bad */
  if (errno == ETIMEDOUT) { /* let's check a global variable */
    /* timeout occurred, try something else */
  } else {
    /* some other error occurred - panic a little */
  }
} else {
  /* semaphore is locked successfully, go ahead */
}
```

Around this line you shall start praying for a thread-safe `errno`
### Interfacing with time

#### Timeouts on semaphores (POSIX)

Suspend current process until the semaphore call is open or the relative timeout has passed:

```c
if (sem_timedwait(&call, &timeout) < 0) { /* something went bad */
  if (errno == ETIMEDOUT) { /* let's check a global variable */
    /* timeout occurred, try something else */
  } else {
    /* some other error occurred - panic a little */
  }
} else {
  /* semaphore is locked successfully, go ahead */
}
```

#### Relative timeouts on entry calls

(works for task-entry calls and protected object calls)

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

#### Non-blocking entry calls

(works for task-entry calls and protected object calls)

```c
task body Sensor is
T : Temperature;
begin
  loop
    -- find temperature T somewhere
    select
      Controller.Call(T);
    or
      delay until Deadline; /* e.g. refine the measurement further */
    end select;
  end loop;
end Sensor;
```

---

### Exception handling

Exception handling is nice, isn't it?
Relative timeouts on incoming calls

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

Accept sequences of any number of calls with less than 1s interleaving.

Absolute timeouts on incoming calls

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

Accept sequences of any number of calls until an absolute time.

Non-blocking acceptance of incoming calls

```vhdl
task body Controller is
  Current_Temp : Temperature;
begin
  loop
    select
      accept Call (T: Temperature) do
        Current_Temp:= T;
        end Call;
      else
        -- action if the temperature was not available in time
        -- e.g. try reading from the alternative source
      end select;
      -- normal processing
    end loop;
  end Controller;
end Controller;
```

Only accepts calls if there are calls already waiting to be processed.

Timeout on actions

All timeout schemas introduced up to now suspend/activate processes at well-defined synchronization points.

What if we need to interrupt a running process due to a timeout?

- Asynchronous transfer of control methods
  - Inside a single process: "Timeout on actions" discussed briefly here and in-depth in the chapter about asynchronism.
  - Between processes: Part of scheduling methods discussed in the chapter about scheduling.
Interfacing with time

Relative timeout on actions

(Ada)

```ada
select
  delay 0.5;  -- seconds
  -- below computations did not finish in time: take measures
then abort
  -- hard to predict sequence of computations
end select;
```

Absolute timeout on actions

(Ada)

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

Externally triggered timeout on actions

(Ada)

```ada
select
  Get_New_Data (Current_Sensor_Data);
  -- below computations did not finish before new data arrived: take measures
then abort
  -- hard to predict sequence of computations
end select;
```

Timeout on actions example

Common real-time 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 then proceed with:
3. Improve the result and keep a record of improvements (while keeping an eye on the deadline) and end in either of the cases:
   3a. The most precise result is achieved before the deadline occurred.
   3b. Otherwise use the closest approximation achieved before the deadline occurred.
4. The deadline is fulfilled and there is a usable result in any case.
Interfacing with time

Timeout on actions example

Get a first approximation and employ spare time for refinements:

- Set an absolute deadline for the computations
- Compulsory computations (save first result)

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

G Similar semantic, yet not on compiler level, but on library level.

Timeout on actions example

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

```java
select Get_New_Data (Current_Sensor_Data);
-- employ results based on previous data
-- compulsory computations (save first result)
then abort
   while Result_Can_Be_Improved loop
      -- optimising computations (save results after each iteration)
      end loop;
   Precise_Result := True;
else
   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 safely before the deadline

If time left: improve, improve, improve

Continue process with the best result possible - given the deadline

Notions of time and space

The big topics:

- What is time? / What is embodiment?
- Interfacing with time
- Specifying timing requirements
- Satisfying timing requirements
Specifying timing requirements

Temporal scopes

Common attributes:
- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline
Specifying timing requirements

Common temporal scope attributes

Temporal scopes can be:

- **Periodic**: controllers, routers, schedulers, streaming processes, ...
- **Aperiodic**: periodic 'on average' tasks, i.e. regular but not rigidly timed, ...
- **Sporadic / Transient**: user requests, alarms, I/O interaction, ...

Deadlines can be:

- **"Hard"**: single failure leads to severe malfunction and/or disaster
- **"Firm"**: only multiple or permanent failures lead to malfunction
- **"Soft"**: results are still useful after the deadline

Specifying timing requirements

How to handle time-unbound primitives?

(loops, recursions, synchronizations, dynamic allocations, aborts)

1. Exclude them
   - **Real-time Euclid, Ada Ravenscar Profile**
2. Expand them to become individually safe
   - (e.g. by adding mandatory timeout)
3. Attribute the code with additional constraints, enabling a full pre-runtime analysis
   - **CRL, Pearl, DSP**

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>`

Specifying timing requirements

Language support levels

1. **Time scopes as part of the language**:
   - (Possible) schedulability analysis by the compiler
   - **Real-time Euclid, CRL, DSP, Pearl, ...**
2. **Time scopes as provided by (standardized) libraries**:
   - (Possible) schedulability analysis by means of external tools
   - **Real-time Java, Ada, ...**
3. **Signal relation primitives** as part of the language:
   - Causality analysis
   - **Esterel**
4. **Timing primitives** as part of the language:
   - Causality analysis (deadlock analysis) – no schedulability analysis
   - **Ada, C#, ...**
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 Real-time systems.

Constraints in CRL are:

- **Time**:
  ```
  timeconstraint
  [use | nosoonerthan | notlaterthan <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 <ConditionId> <Frameinfo>] endactivationdeactivationconstraint
  ```

- **Direct recursions**:
  assert lower and upper recursion limits (general recursions are not covered).
Specifying timing requirements

**CRL**
(a language for complex real-time systems)

Evaluating those constraints and assertions in CRL:

- **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 – Language Status**
(a language for complex real-time systems)

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:

```plaintext
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.

**DPS: Distributed Programming System**

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

```plaintext
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
```

- DPS-compiler: breaks all codes down to processes and schedules
Specifying timing requirements

Real-time Java

Real-time Java comes with libraries providing:

- Multiple sets of predefined time-scope parameters
- A scheduler class (with a predefined priority scheduler)

Schedulability (feasibility) analysis possible.

```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();
}
```

Specifying timing requirements

Real-time Java

Measuring execution time (cost) is not required by the language, i.e., overrunHandler might never be activated.

```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);
}
```

Specifying timing requirements

Real-time Java

```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

    The absence of range types makes priority an int

    Note that deadline is a RelativeTime.
```

The absence of range types makes priority an int.
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 …;
}

Specifying timing requirements
Real-time Java

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

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);
}

Specifying timing requirements
Real-time Java

public class NoHeapRealtimeThread extends RealtimeThread {
    public NoHeapRealtimeThread extends RealtimeThread {
        public RealtimeThread (SchedulingParameters s, ReleaseParameters r, MemoryParameters m, MemoryArea a) throws …;
    }
    public synchronized void addToFeasibility ();
    public synchronized void addIfFeasible ();
    public static RealtimeThread currentRealtimeThread () throws …;
    public boolean waitForNextPeriod () throws …;
    public synchronized void interrupt ();
    public synchronized void sleep (…) throws …;
}
Specifying timing requirements

Real-Time Java

- Stayed in an academic niche since its introduction in 2001. No practical deployment stories in high integrity systems are known.
- Original reference implementation by TimeSys. Last remaining, current implementation might be JamaicaVM.
- Real-time Java is based on the concept of scoped memory — yet scoped memory does not embed smoothly into the larger Java context, while a growing community is working on “real-time” garbage collectors.
- Real-time Java might be replaced by a new breed of Java virtual machines which come with “real-time” garbage collectors.

Specifying timing requirements

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);
    ...
}

Specifying timing requirements

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);
    ...
}

Specifying timing requirements

Ada

... only a few time scope expressions at language level!

Extensions are used to specify other time scopes, based on
- Tasks
- Schedulers (fixed priorities, dynamic priorities and earliest deadline first)
- Timers
- Asynchronous transfer of control
- Code attribution via contracts
Satisfying timing requirements

Two paths towards fulfilling real-time 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.
  - Gives a correct solution according to the specification.

- **Complex systems** 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.

Specifying timing requirements

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 as a sequence of events.
- Time-scopes translate to signal-relations and signal-counters.

Yet, 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 required for the validation of the zero-time assumption!

Notions of time and space

The big topics:

What is time? / What is embodiment?

- **Interfacing with time**
- **Specifying timing requirements**
- **Satisfying timing requirements**
Satisfying timing requirements

**Two paths towards fulfilling real-time requirements**

Real-time logic approach => chapter “Reliability”

1. 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.
   - Formulate the specification on the basis of the reduced, synchronous system.

2. Verify the reduced system:
   - Verify the reduced synchronous against the specification => Program verification methods.

3. Compile the reduced system to an actual system:
   - The resulting actual system will be executable on real machines and employ real devices.

4. Re-check the actual system:
   - Complex Systems approach.

**Summary**

- **What is time? / What is embodiment?**
  - Approaches by different faculties to understand the foundations of this course

- **Interfacing with time**
  - Formulating local, time-dependent constraints
  - Access time, delay processes, timers
  - Timeouts, asynchronous transfer of control

- **Specifying timing requirements**
  - Formulating global timing-constraints
  - Understanding time-scope parameters (and expressing them in different languages)

- **Satisfying timing requirements**
  - Real-time logic approach & Complex systems approach