Characteristics of a RTS

- Large and complex
- Concurrent control of separate system components
- Facilities to interact with special purpose hardware
- **Guaranteed response times**
- Extreme reliability
- Efficient implementation

Real-Time Facilities

- Goal
  - To understand the role that time has in the design and implementation of real-time systems
- Topics
  - Notion of time
  - Clocks, delays and timeouts
  - Specifying timing requirements
  - Temporal scopes
  - Fault tolerance

Real-Time Facilities: Requirements

- Interfacing with **time**
  - accessing clocks so that the passage of time can be measured
  - delaying processes until some future time
  - programming timeouts so that the non-occurrence of some event can be recognized and dealt with
- Representing timing requirements
  - specifying rates of execution
  - specifying deadlines
- Satisfying timing requirements — covered later

The Notion of Time

- Transitivity: $\forall x, y, z : (x < y \wedge y < z) \Rightarrow x < z$
- Linearity: $\forall x, y : x < y \vee y < x \Rightarrow x = y$
- Irreflexivity: $\forall x : not(x < x)$
- Density: $\forall x, y : x < y \Rightarrow \exists z : (x < z < y)$

Standard Time

Name	Description	Note
True Solar Day	Time between two successive culminations (highest point of the sun)	Varies through the year by 15 minutes (approx)
Temporal Hour	One-twelfth part of the time between sunrise and sunset	Varies considerably through the year
Universal Time (UT0)	Mean solar time at Greenwich meridian	Defined in 1884
Second (1)	1/86,400 of a mean solar day	
Second(2)	1/31,566,925.9747 of the tropical year for 1900	Ephemris Time defined in 1955

Name	Description	Note
UT1	correction to UTO because of polar motion	
UT 2	Correction of UT1 because of variation in the speed of rotation of the earth	
Seconds(3)	Duration of 9_192_631_770 periods of the radiation corresponding to the transition between two hyperfine levels of the ground state of the Caesium - 133 atom	Accuracy of current Caesium atomic clocks deemed to be one part of 10^13 (that is, one clock error per 300,000 years)
International Atomic Time (IAT)	Based upon Caesium atomic clock	
Coordinated Universial Time (UTC)	An IAT clock synchronized to UT2 by the addition of occasional leap ticks	Maximum difference between UT2 (which is based on astrological measurement) and IAT (which is based upon atomic measurements) is kept to below 0.5 seconds

Access to a Clock

- by having direct access to the environment's time frame
- by using an internal hardware clock that gives an adequate approximation to the passage of time in the environment

Calendar

```
package Ada.Calendar is
   type Time is private;
   subtype Year_Number is Integer range 1901..2099;
   subtype Month_Number is Integer range 1..12;
   subtype Day_Number is Integer range 1..31;
   subtype Day_Duration is Duration range 0.0..86_400.0;

   function Clock return Time;
   function Year(Date:Time) return Year_Number;
   function Month(Date:Time) return Month_Number;
   function Day(Date:Time) return Day_Number;
   function Seconds(Date:Time) return Day_Duration;
   procedure Split(Date:in Time; Year:out Year_Number;
                   Month:out Month_Number; Day:out Day_Number;
                   Seconds:out Day_Duration);
   function Time_Of(Year:Year_Number; Month:Month_Number;
   Day:Day_Number; Seconds:Day_Duration := 0.0) return Time;
```

Calendar II

```
   function "+"(Left:Time; Right:Duration) return Time;
   function "+"(Left:Duration; Right:Time) return Time;
   function "-"(Left:Time; Right:Duration) return Time;
   function "-"(Left:Time; Right:Time) return Duration;
   function "<"(Left,Right:Time) return Boolean;
   function "<="(Left,Right:Time) return Boolean;
   function ">"(Left,Right:Time) return Boolean;
   function ">="(Left,Right:Time) return Boolean;
   Time_Error:exception;
     -- Time_Error may be raised by Time_Of,
     -- Split, Year, "+" and "-"
private
   implementation-dependent
end Ada.Calendar;
```

Calendar III

- A value of the private type `Time` is a combination of the date and the time of day
- The time of day is given in seconds from midnight
- Seconds are described in terms of a subtype `Day_Duration`
- Which is, in turn, defined by means of `Duration`

Duration

- This fixed point type `Duration` is one of the predefined scalar types and has a range which, although implementation dependent, must be at least -86_400.0 .. +86_400.0
- The value 86_400 is the number of seconds in a day
- The accuracy of `Duration` is also implementation dependent but the smallest representable value `Duration'Small` must not be greater than 20 milliseconds
- It is recommended in the ARM that it is no greater than 100 microseconds

Example Use

```
declare
  Old_Time, New_Time : Time;
  Interval : Duration;
begin
  Old_Time := Clock;
  -- other computations
  New_Time := Clock;
  Interval := New_Time - Old_Time;
end;
```

- The other language clock is provided by the optional package `Real_Time`
- This has a similar form to `Calendar` but is intended to give a finer granularity
- The value of `Tick` must be no greater than one millisecond; the range of `Time` (from the epoch that represents the program's start-up) must be at least fifty years

Real-Time Clock

```
package Ada.Real_Time is
   type Time is private;
   Time_First: constant Time;
   Time_Last: constant Time;
   Time_Unit: constant := implementation_defined_real_number;
   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;
   function Clock return Time;
   function "+" (Left: Time; Right: Time_Span) return Time;
   function "+" (Left: Time_Span; Right: Time) return Time;
     -- similarly for "-", "<",etc
```

Real-Time Clock II

```
   function To_Duration(TS: Time_Span) return Duration;
   function To_Time_Span(D: Duration) return Time_Span;
   function Nanoseconds (NS: Integer) return Time_Span;
   function Microseconds(US: Integer) return Time_Span;
   function Milliseconds(MS: Integer) return Time_Span;
   type Seconds_Count is range implementation-defined;
   procedure Split(T : in Time; SC: out Seconds_Count;
                   TS : out Time_Span);
   function Time_Of(SC: Seconds_Count;
                    TS: Time_Span) return Time;
private
   -- not specified by the language
end Ada.Real_Time;
```

Metrics

- `Time_Unit` is the smallest amount of real time representable by the `Time` type
- The value of `Tick` must be no greater than 1 millisecond
- The range of `Time` (from the epoch that represents the program's start-up) must be at least 50 years
- Other important features of this time abstraction are described in the Real-Time Annex

Example: Timing a Sequence

```
declare
   use Ada.Real_Time;
   Start, Finish : Time;
   Interval : Time_Span := To_Time_Span(1.7);
begin
   Start := Clock;
   -- sequence of statements
   Finish := Clock;
   if Finish - Start > Interval then
      raise Time_Error; -- a user-defined exception
   end if;
end;
```

Clocks in Real-Time Java

- Similar to those in Ada
- `java.lang.System.currentTimeMillis` returns the number of milliseconds since 1/1/1970 GMT and is used by used by `java.util.Date`
- Real-time Java adds real-time clocks with high resolution time types

RT Java Time Types

```
public abstract class HighResolutionTime implements
       java.lang.Comparable
{
  public abstract AbsoluteTime absolute(Clock clock,
                                        AbsoluteTime destination);

  ...

  public boolean equals(HighResolutionTime time);

  public final long getMilliseconds();
  public final int getNanoseconds();

  public void set(HighResolutionTime time);
  public void set(long millis);
  public void set(long millis, int nanos);
}
```

```
public class AbsoluteTime extends HighResolutionTime
{
  // various constructor methods including
  public AbsoluteTime(AbsoluteTime T);
  public AbsoluteTime(long millis, int nanos);

  public AbsoluteTime absolute(Clock clock, AbsoluteTime dest);

  public AbsoluteTime add(long millis, int nanos);
  public final AbsoluteTime add(RelativeTime time);

  ...

  public final RelativeTime subtract(AbsoluteTime time);
  public final AbsoluteTime subtract(RelativeTime time);

}
```

```
public class RelativeTime extends HighResolutionTime
{
  // various constructor methods including
  public RelativeTime(long millis, int nanos);
  public RelativeTime(RelativeTime time);

  public AbsoluteTime absolute(Clock clock,
                               AbsoluteTime destination);

  public RelativeTime add(long millis, int nanos);
  public final RelativeTime add(RelativeTime time);

  public void addInterarrivalTo(AbsoluteTime destination);

  public final RelativeTime subtract(RelativeTime time);

  ...
}

public class RationalTime extends RelativeTime
{ . . .}
```

RT Java: Clock Class

```
public abstract class Clock
{
  public Clock();

  public static Clock getRealtimeClock();

  public abstract RelativeTime getResolution();

  public AbsoluteTime getTime();
  public abstract void getTime(AbsoluteTime time);

  public abstract void setResolution(RelativeTime resolution);

}
```

RT Java: Measuring Time

```
{
  AbsoluteTime oldTime, newTime;
  RelativeTime interval;
  Clock clock = Clock.getRealtimeClock();

  oldTime = clock.getTime();
  // other computations
  newTime = clock.getTime();

  interval = newTime.subtract(oldTime);

}
```

Clocks in C and POSIX

- ANSI C has a standard library for interfacing to “calendar” time
- This defines a basic time type `time_t` and several routines for manipulating objects of type time

- POSIX requires at least one clock of minimum resolution 50 Hz (20ms)

POSIX Real-Time Clocks

```
#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 */
```

Delaying a Process

- In addition to clock access, processes must also be able to delay their execution either for a relative period of time or until some time in the future
- **Relative delays**

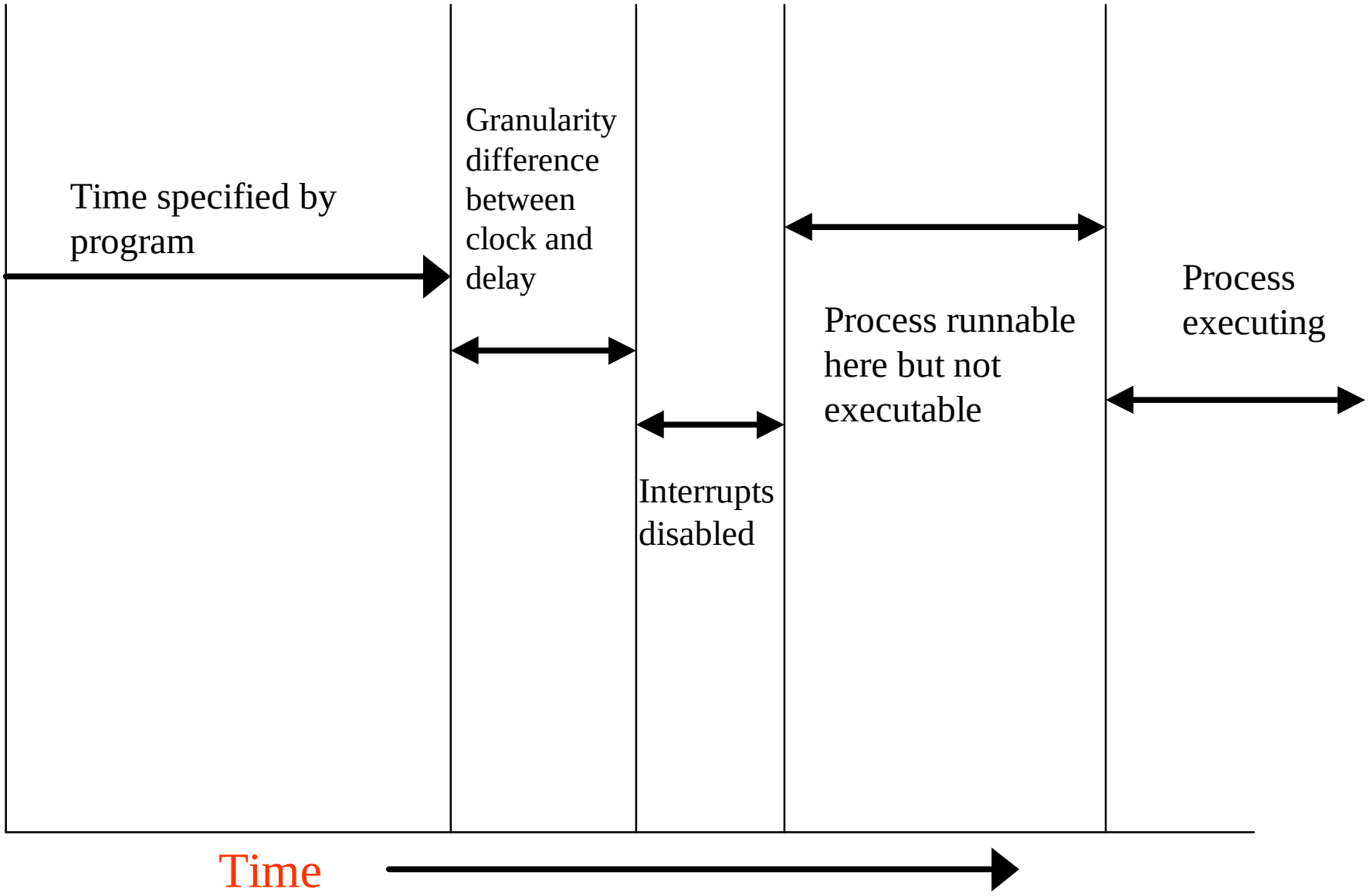
```
Start := Clock; -- from calendar
loop
   exit when (Clock - Start) > 10.0;
end loop;
```

- To eliminate the need for these busy-waits, most languages and operating systems provide some form of delay primitive
- In Ada, this is a delay statement

```
delay 10.0;
```

- In POSIX: `sleep` and `nanosleep`
- Java: sleep; RT Java provides a high resolution sleep

Delays



Absolute Delays

```
-- Ada
START := Clock;
FIRST_ACTION;
delay 10.0 - (Clock - START);
SECOND_ACTION;
```

- Unfortunately, this might not achieve the desired result

```
START := Clock;
FIRST_ACTION;
delay until START + 10.0;
SECOND_ACTION;
```

- As with **`delay`**, **`delay until`** is accurate only in its lower bound
- RT Java - sleep can be relative or absolute
- POSIX requires use of an absolute timer and signals

Drift

- The time over-run associated with both relative and absolute delays is called the **local drift** and it it cannot be eliminated
- It is possible, however, to eliminate the **cumulative drift** that could arise if local drifts were allowed to superimpose

Regular Activity

```
task T;

task body T is
begin
  loop
    Action;
    delay 5.0;
  end loop;
end T;
```

Cannot delay for less than 5 seconds

local and cumulative drift

Periodic Activity

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

If `Action` takes 6 seconds, the delay statement will have no effect

Will run on average every 5 seconds

local drift only

Control Example

```
with Ada.Real_Time; use Ada.Real_Time;
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures;
use Control_Procedures;
procedure Controller is


   task Temp_Controller;

   task Pressure_Controller;
```

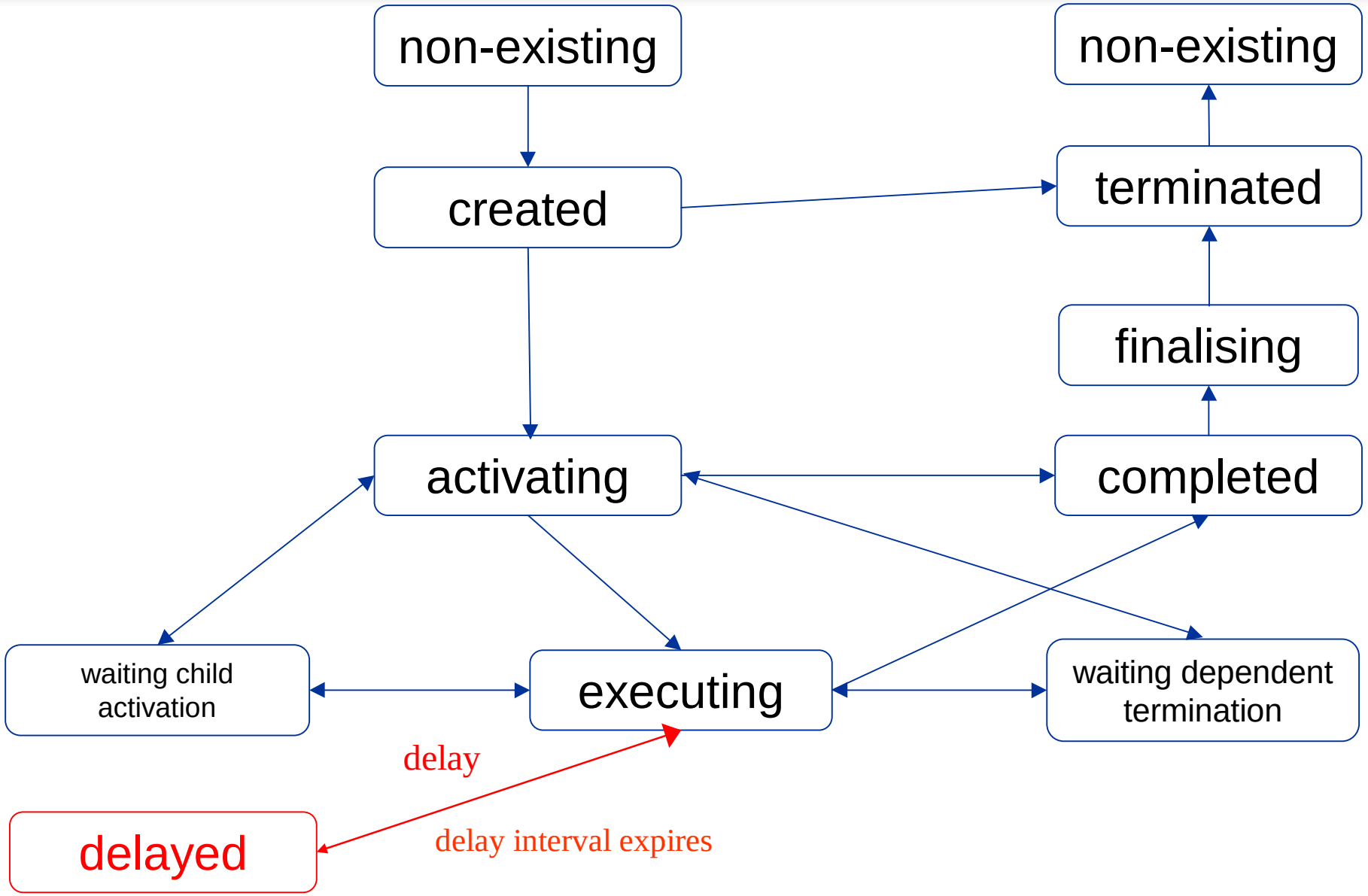
Control Example II

```
task body Temp_Controller is
    TR : Temp_Reading; HS : Heater_Setting;
    Next : Time;
    Interval : Time_Span := Milliseconds(30);
  begin
    Next := Clock;  -- start time
    loop
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS);
      Write(TR);
      Next := Next + Interval;
      delay until Next;
    end loop;
  end Temp_Controller;
```

Control Example III

```
  task body Pressure_Controller is
    PR : Pressure_Reading; PS : Pressure_Setting;
    Next : Time;
    Interval : Time_Span := Milliseconds(70);
  begin
    Next := Clock;  -- start time
    loop
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      Write(PR);
      Next := Next + Interval;
      delay until Next;
    end loop;
  end Pressure_Controller;
begin
  null;
end Controller;
```

Ada Task States



Timeouts: Shared Variable Communication

- Timeout can be applied to condition synchronization facilities:
  - semaphores, e.g. POSIX

```
if(sem_timedwait(&call, &timeout) < 0) {
  if ( errno ==  ETIMEDOUT) {
    /* timeout occurred */
  }
  else { /* some other error */ }
} else {
  /* semaphore locked */
};
```

  - conditional critical regions
  - condition variables in monitors, mutexes or synchronized methods
  - entries in protected object

- POSIX also allows a timeout whilst waiting for a mutex lock

Message-Passing and Timeouts

```
task Controller is
  entry Call(T : Temperature);
end Controller;

task body Controller is
  -- declarations, including
  New_Temp : Temperature;
begin
  loop
    accept Call(T : Temperature) do
       New_Temp := T;
    end Call;
    -- other actions
  end loop;
end Controller;
```

Message-Passing and Timeouts

```
task Controller is
  entry Call(T : Temperature);
private entry Timeout;
end Controller;

task body Controller is
  task Timer is
    entry Go(D : Duration);
  end timer;
  task body Timer is separate;
  -- other declarations
begin
  loop
    Timer.Go(10.0);
    select
      accept Call(T : Temperature) do
         New_Temp := T;
      end Call;
    or
      accept Timeout;
      -- action for timeout
    end select;
    -- other actions
  end loop;
end Controller;
```

```
task body timer is
  DU : Duration;
begin
  accept Go(D : Duration) do
     Timeout_Value := d;
  end Go;
  delay Timeout_Value;
  Controller.Timeout;
end timer;
```

Message-Passing and Timeouts

```
task Controller is
  entry Call(T : Temperature);
end Controller;

task body Controller is
  -- declarations
begin
  loop
    select
      accept Call(T : Temperature) do
        New_Temp := T;
      end Call;
    or
      delay 10.0;
      -- action for timeout
    end select;
    -- other actions
  end loop;
end Controller;
```

Message Passing: Absolute Delays

```
task Ticket_Agent is
  entry Registration(...);
end Ticket_Agent;

task body Ticket_Agent is
  -- declarations
  Shop_Open : Boolean := True;
begin
  while Shop_Open loop
    select
      accept Registration(...) do
        -- log details
      end Registration;
    or
      delay until Closing_Time;
      Shop_Open := False;
    end select;
    -- process registrations
  end loop;
end Ticket_Agent;
```

- Within Ada, it make no sense to mix an else part, a terminate alternative and delay alternatives
- These three structures are mutually exclusive; a select statement can have, at most, only one of them
- However, the select can have a number of delays but they must all be of the same kind (that is, **`delay`**s or **`delay until`**s).

Timeout on Message Send

```
loop
  -- get new temperature T
  Controller.Call(T);
end loop;

loop
  -- get new temperature T
  select
    Controller.Call(T);
  or
    delay 0.5;
    null;
  end select;
end loop;

select
  T.E   -- entry E in task T
else
  -- other actions
end select;
```

The **null** is not strictly needed but shows that again the delay can have arbitrary statements following, that are executed if the delay expires before the entry call is accepted

Timeouts and Entries

- The above examples have used timeouts on inter-task communication; it is also possible, within Ada, to do timed (and conditional) entry call on protected objects

```
select
  P.E ; -- E is an entry in protected object P
or
  delay 0.5;
end select;
```

Timeouts on Actions

```
select
  delay 0.1;
then abort
  -- action
end select;
```

- If the action takes too long, the triggering event will be taken and the action will be aborted
- This is clearly an effective way of catching *run-away code*

Imprecise Computation: Ada

```
declare
  Precise_Result : Boolean;
begin
  Completion_Time := ...
  --  compulsory part
  Results.Write(...); -- call to procedure in
                      -- external protected object
  select
    delay until Completion_Time;
    Precise_Result := False;
  then abort
    while Can_Be_Improved loop
      -- improve result
      Results.Write(...);
    end loop;
    Precise_Result := True;
  end select;
end;
```

Real-Time Java

- With Real-Time Java, timeouts on actions are provided by a subclass of `AsynchronouslyInterruptedException` called `Timed`

```
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);
}
```

Imprecise Computation: RT Java

```
public class PreciseResult
{
  public resultType value; // the result
  public boolean preciseResult; // indicates if it is imprecise
}

public class ImpreciseComputation {
  private HighResolutionTime CompletionTime;
  private PreciseResult result = new PreciseResult();

  public ImpreciseComputation(HighResolutionTime T)
  {
    CompletionTime = T; //can be absolute or relative
  }

  private resultType compulsoryPart()
  {
    // function which computes the compulsory part
  };
```

```
public PreciseResult Service()  // public service
{
  Interruptible I = new Interruptible()
  {
    public void run(AsynchronouslyInterruptedException exceptic
                    throws AsynchronouslyInterruptedException
    {
      // this is the optional function which improves on the
      // compulsory part
      boolean canBeImproved = true;

      while(canBeImproved)
      {
        // improve result
        synchronized(this) {
          // write result --
          // the synchronized statement ensures
          // atomicity of the write operation
        }
      }
      result.preciseResult = true;
    }

    public void interruptAction(
           AsynchronouslyInterruptedException exception)
    {
      result.preciseResult = false;
    }
  };
```

```
    Timed t = new Timed(CompletionTime);

    result.value = compulsoryPart(); // compute the compulsory part
    if(t.doInterruptible(I)) {
      // execute the optional part with the timer
      return result;
   } else { ... };
  }
}
```

POSIX

- POSIX does not support ATC and, therefore, it is difficult to get the same effect as Ada and RT Java
- POSIX does support Timers (see later)

Specifying Timing Requirements

- Work on a more rigorous approach to this aspect of real-time systems has followed two largely distinct paths:

1. The use of formally defined language semantics and timing requirements, together with notations and logics that enable temporal properties to be represented and analysed
2. A focus on the performance of real-time systems in terms of the feasibility of scheduling the required work load on the available resources (processors and so on)

Timing Verification

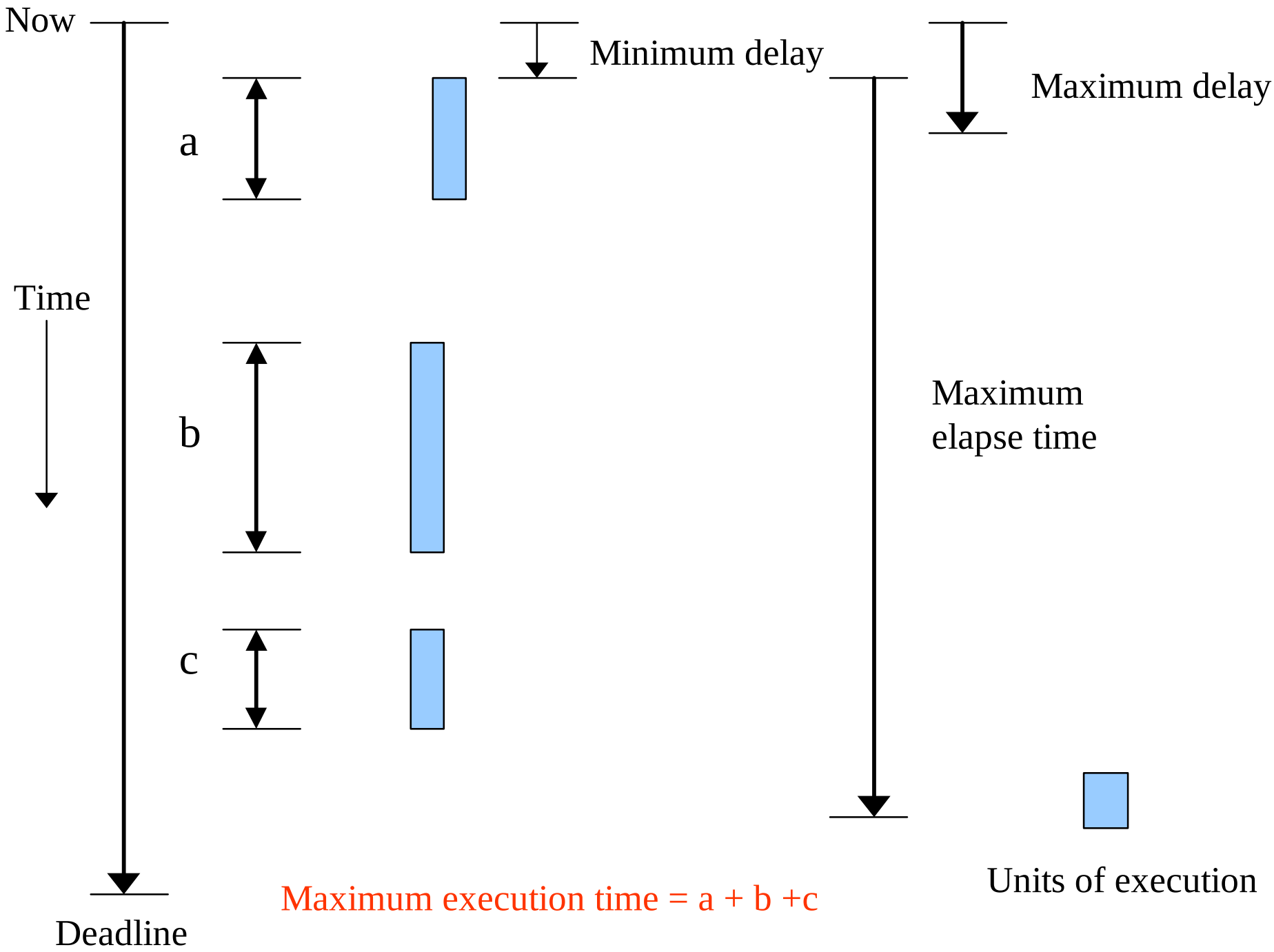
- The verification of a real-time system can thus be interpreted as requiring a two stage process:

1. verifying requirements — given an infinitely fast reliable computer, are the temporal requirements coherent and consistent, that is, have they the potential to be satisfied?
2. verifying the implementation — with a finite set of (possible unreliable) hardware resources, can the temporal requirements be satisfied?

Temporal Scopes

- deadline — the time by which the execution of a TS must be finished;
- minimum delay — the minimum amount of time that must elapse before the start of execution of a TS;
- maximum delay — the maximum amount of time that can elapse before the start of execution of a TS;
- maximum execution time — of a TS;
- maximum elapse time — of a TS.

Temporal scopes with combinations of these attributes are also possible


Now
Minimum delay
Maximum delay
a
Time
b
Maximum
elapse time
c
Deadline
Maximum execution time = a + b +c
Units of execution

Temporal Scopes

- Can be
  - Periodic
  - Sporadic
  - Aperiodic

- Deadlines can be:
  - Hard
  - Soft
  - Interactive — performance issue
  - Firm

Specifying Processes and TS

```
process periodic_P;
  ...
begin
  loop
    IDLE
    start of temporal scope
      ...
    end of temporal scope
  end;
end;
```

- The time constraints take the form of maximum and/or minimum times for `IDLE` and the requirement that the end of the temporal scope be by some deadline

Deadline

The deadline can itself be expressed in terms of either

- absolute time
- execution time since the start of the temporal scope, or
- elapsed time since the start of the temporal scope.

Aperiodic Processes

```
process aperiodic_P;
   ...
begin
   loop
      wait for interrupt
      start of temporal scope
         ...
      end of temporal scope
   end;
end;
```

Language Support for TS

- Ada and C/POSIX
- Real-Time Euclid and Pearl
- Real-Time Java
- DPS
- Esteral

Ada: Periodic Task

```
task body Periodic_T is
  Release_Interval : Duration := ...; -- or
  Release_Interval : Time_Span := Milliseconds(...);
begin
  -- read clock and calculate the next
  -- release time (Next_Release)
  loop
    -- sample data (for example) or
    -- calculate and send a control signal
    delay until Next_Release;
    Next_Release := Next_Release + Release_Interval;
  end loop;
end Periodic_T;
```

POSIX: Periodic Thread

```
#include <signal.h>
#include <time.h>
#include <pthread.h>
void periodic_thread() /* destined to be the thread */
{
  int signum;                    /* signal caught */
  sigset_t set;                  /* signals to be waited for */
  struct sigevent sig;           /* signal information */
  timer_t periodic_timer;        /* timer for a periodic thread */
  struct itimerspec required, old;  /* timer details */
  struct timespec first, period;    /* start and repetition */
  long Thread_Period = ....      /* actual period in nanoseconds */
```

```
/* set up signal interface */
sig.sigev_notify = SIGEV_SIGNALS;
sig.sigev_signo = SIGRTMIN; /* for example */

/* allow, e.g., 1 sec from now for system initialisation */
CLOCK_GETTIME(CLOCK_REALTIME, &first);  /* get current time */
first.tv_sec = first.tv_sec + 1;
period.tv_sec = 0;          /* set repetition value to period*/
period.tv_nsec = Thread_Period;
required.it_value = first;  /* initialise timer details */
required.it_interval = period;

TIMER_CREATE(CLOCK_REALTIME, &sig, &periodic_timer);
SIGEMPTYSET(&set);          /* initialise signal set to null */
SIGADDSET(&set, SIGRTMIN);  /* only allow timer interrupts*/
TIMER_SETTIME(periodic_timer, 0, &required, &old);
```

```
  /* enter periodic loop */
  while(1) {
    SIGWAIT(&set, &signum);
    /* code to be executed each period here */
  }
}

int init()
{
  pthread_attr_t attributes;      /* thread attributes */
  pthread_t PT;                   /* thread pointer */

  PTHREAD_ATTR_INIT(&attributes); /* default attributes */
  PTHREAD_CREATE(&PT, &attributes,
                 (void *) periodic_thread, (void *)0);
}
```

Ada: Sporadic Task

- A sporadic task that is triggered by an interrupt would contain no explicit time information but would, typically, use a protected object to handle the interrupt and release the task for execution

```
protected Sporadic_Controller is
  procedure Interrupt; -- mapped onto interrupt
  entry Wait_For_Next_Interrupt;
private
  Call_Outstanding : boolean := false;
end Sporadic_Controller;
```

```
protected Sporadic_Controller is
  procedure Interrupt is
  begin
    Call_Outstanding := True;
  end Interrupt;
  entry Wait_For_Next_Interrupt
    when Call_Outstanding is
  begin
    Call_Outstanding := False;
  end Wait_For_Next_Interrupt;
end Sporadic_Controller;

task body Sporadic_T is
begin
  loop
    Sporadic_Controller.Wait_For_Next_Interrupt;
    -- action
  end loop;
end Sporadic_T;
```

Real-Time Euclid

1. periodic **frameInfo** first activation **timeOrEvent**
2. atEvent **conditionId frameInfo**

- The clause **frameInfo** defines the periodicity of the process (including the maximum rate for sporadic processes).
- The simplest form this can take is an expression in real-time units:

```
frame realTimeExpn
```

- The value of these units is set at the beginning of the program

Periodic Process

- A periodic process can be activated for the first time by
  - having a start time defined
  - waiting for an interrupt to occur
  - waiting for either of above
- The syntax for **timeOrEvent** must, therefore, be one of the following

  atTime **realTimeExpn**

  atEvent **conditionId**

  atTime **realTimeExpn** or atEvent **conditionId**

- **conditionId** is a condition variable associated with an interrupt; it is also used with sporadic processes

RT Euclid: Example

- A cyclic temperature controller with periodicity 60 units (every minute if the time unit is set to 1 second) which become active after 600 units or when a **startMonitoring** interrupt arrives

```
realTimeUnit := 1.0  % time unit = 1 seconds

var Reactor: module  % Euclid is module based
var startMonitoring : activation condition atLocation 16#A10D
% This defines a condition variable which is
% mapped onto an interrupt

  process TempController : periodic
     frame 60 first activation
     atTime 600 or atEvent startMonitoring
  % import list
  %
  % execution part
  %
  end TempController
end Reactor
```

Note: no loop; scheduler controls the activation

Ada Equivalent

```
task body Temp_Controller is
  -- definitions, including
  Next_Release : Duration;
begin
  select
    accept Start_Monitoring;
      -- or a timed entry call
      -- onto a protected object
  or
    delay 600.0;
  end select;
  Next_Release := Clock + 60.0;
     -- take note of next release time
  loop
    -- execution part
    delay until Next_Release;
    Next_Release := Next_Release + 60.0;
  end loop;
end Temp_Controller;
```

Pearl

- Provides explicit timing information concerning the start, frequency and termination of processes

```
EVERY 10 SEC ACTIVATE T
```

- To activate at a particular point in time (say 12.00 noon each day):

```
AT 12:00:00 ACTIVATE LUNCH
```

- A sporadic task, S, released by an interrupt, `IRT`, is defined by

```
WHEN IRT ACTIVATE S;
```

- or if an initial delay of one second is required:

```
WHEN IRT AFTER 1 SEC ACTIVATE S;
```

- A task in Pearl can be activated by a time schedule or an interrupt but **not** both:

```
AFTER 10 MIN ALL 60 SEC ACTIVATE TempController;

WHEN startMonitoring ALL 60 SEC ACTIVATE TempController;
```

- The term `ALL 60 SEC` means repeat periodically, after the first execution, every 60 seconds

Real-Time Java

- Objects which are to be scheduled must implement the `Schedulable` interface; objects must also specify their:
  - memory requirements via the class `MemoryParameters`
  - scheduling requirements via the class `SchedulingParameters`
  - timing requirements via the class `ReleaseParameters`

```
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();

  // methods for setting the above
}
```

Release Parameters

- A schedulable object can have a deadline and a cost associated with each time it is released for execution
- The cost is the amount of execution time that a scheduler should give to the object
- If the object is still executing when either its deadline or its cost expire, the associated event handlers are scheduled
- Noted:
  - RTJ does not require an implementation to support execution time monitoring
  - RTJ does require an implementation to detect missed deadlines.
  - The release events for sporadic and aperiodic threads are currently not well-defined
- A program can indicate that it is not concerned with a missed deadline by passing a null handler

Periodic Parameters

```
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);
}
```

Aperiodic and Sporadic Release Parameters

```
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);
}
```

Real-Time Threads

```
public class RealtimeThread extends java.lang.Thread
       implements Schedulable
{
  public RealtimeThread(SchedulingParameters s, ReleaseParameters r);
  . . .

  // methods for implementing the Schedulable interface
  public synchronized void addToFeasibility();
  . . .

  public static RealtimeThread currentRealtimeThread();

  public synchronized void schedulePeriodic();
  // add the thread to the list of schedulable objects
  public synchronized void deschedulePeriodic();
  // remove the thread from the list of schedulable object
  // when it next issues a waitForNextPeriod
  public boolean waitForNextPeriod() throws ...;

  public synchronized void interrupt();
  // overrides java.lang.Thread.interrupt()

  public static void sleep(Clock c, HighResolutionTime time) throws ...;
}
```

RT Java: Periodic Thread

```
public class Periodic extends RealtimeThread
{
  public Periodic( PriorityParameters PP,
                   PeriodicParameters P)
  { ... };

  public void run()
  {
    while(true) {
      // code to be run each period
      ...
      waitForNextPeriod();
    }
  }
}
```

PriorityParameters are a subclass of SchedulingParameters -- see later

RT Java: Periodic Thread Cont.

```
{
  AbsoluteTime A = new AbsoluteTime(...);
  PeriodicParameters P = new PeriodicParameters(
          A, new RelativeTime(10,0),
          new RelativeTime(1,0), new RelativeTime(5,0),
          null,  null );

  PriorityParameters PP = new PriorityParameters(...);

  Periodic ourThread = new Periodic(PP, P); //create thread

  ourThread.start(); // release it

}
```

DPS

- Whereas Pearl, RT Euclid and RT Java have associate temporal scopes with processes, and, therefore, necessitate the specification of timing constraints on the process itself, other languages such as DPS provide local timing facilities that apply at the block level

- In general, a DPS temporal block (scope) may need to specify three distinct timing requirements (these are similar to the more global requirements discussed earlier):
  - delay start by a known amount of time;
  - complete execution by a known deadline;
  - take no longer than a specified time to undertake a computation

DPS: Coffee Making Example

```
get_cup
put_coffee_in_cup
boil_water
put_water_in_cup
drink_coffee
replace_cup
```

Instant coffee

- The act of making a cup of coffee should take no more than 10 minutes; drinking it is more complicated
- A delay of 3 minutes should ensure that the mouth is not burnt
- The cup itself should be emptied within 25 minutes (it would then be cold) or before 17:00 (that is, 5 o'clock and time to go home)

DPS: Coffee Example Continued

- Two temporal scopes are required:

```
start elapse 10 do
  get_cup
  put_coffee_in_cup
  boil_water
  put_water_in_cup
end

start after 3 elapse 25 by 17:00 do
  drink_coffee
  replace_cup
end
```

DPS: Coffee Example Continued

- For a temporal scope that is executed repetitively, a time loop construct is useful:

  `from <start> to <end> every <period>`

- For example, many software engineers require regular coffee throughout the working day:

```
from 9:00 to 16:15 every 45 do
   make_and_drink_coffee
```

Esterel

Synchronous Hypothesis: Ideal systems produce their outputs synchronously with their inputs

- Hence all computation and communication is assumed to take zero time (all temporal scopes are executed instantaneously)

```
module periodic;
input tick;
output result(integer);
var V : integer in
  loop
    await 10 tick;
    -- undertake required computation to set V
    emit result(v);
  end
end
```

- A sporadic module has an identical form

Esterel

- One consequence of the synchronous hypothesis is that all actions are atomic
- This behaviour significantly reduces nondeterminism
- Unfortunately it also leads to potential causality problems

```
signal S in
  present S else emit S end
end
```

- This program is incoherent: if S is absent then it is emitted; on the other hand if it were present it would not be emitted
- A formal definition of the behavioral semantics of Esterel helps to eliminate these problems

Fault Tolerance

A deadline could be missed in a `proven' system if:

- worst-case calculations were inaccurate
- assumptions made in the schedulability checker were not valid
- the schedulability checker itself had an error
- the scheduling algorithm could not cope with a load even though it is theoretically schedulable
- the system is working outside its design parameters

Fault Tolerance of Timing Failures

It is necessary to be able to detect:

- overrun of deadline
- overrun of worst-case execution time
- sporadic events occurring more often than predicted
- timeout on communications

- The last three failures in this list do not necessary indicate that deadlines will be missed;:
  - an overrun of WCET in one process might be compensated by a sporadic event occurring less often than the maximum allowed
- Hence, the damage confinement and assessment phase of providing fault tolerance must determine what actions to take
- Both forward and backward error recovery is possible

Deadline Overrun Detection and FER

- The Ada RTS is unaware of the timing requirements of its application tasks and has to provide primitive mechanisms to detect deadline overrun
- This is achieved using the asynchronous transfer of control facility
- A similar approach can be used to detect a deadline overrun in a sporadic task

```
task body Periodic_T is
  Next_Release : Time;
  Next_Deadline : Time;
  Release_Interval : constant Time_Span := Milliseconds(...);
  Deadline : constant Time_Span := Milliseconds(...);
begin
  -- read clock and calculate the Next_Releas and
  -- Next_Deadline
  loop
    select
      delay until Next_Deadline;
      -- deadline overrun detected here perform recovery
    then abort
      -- code of application
    end select;
    delay until Next_Release;
    Next_Release := Next_Release + Release_Interval;
    Next_Deadline := Next_Release + Deadline;
  end loop;
end Periodic_T;
```

Problem with Ada Approach

- It assumes that the recovery strategy requires the task to stop what it is doing
- This is one option but there are other approaches; for example, allowing the task to continue its execution at a different priority
- For these, a more appropriate response to detecting a deadline overrun is to raise an asynchronous event
- In Real-Time Java, the virtual machine will signal an asynchronous event when a periodic thread is still executing when its deadline has passed
- Sporadic event handlers, in Real-Time Java, have no explicit deadline overrun detection; they are assumed to be soft.

Timers in POSIX

```
#define TIMER_ABSTIME ..

struct itimerspec {
  struct timespec it_value; /* first timer signal */
  struct timespec it_interval; /* subsequent intervals */
};

typedef ... timer_t_t;

int timer_create(clockid_t clock_id, struct sigevent *evp,
                 timer_t *timerid);

int timer_delete(timer_t timerid);

int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *value,
                  struct itimerspec *ovalue);

int timer_gettime(timer_t timerid,
  struct itimerspec *value);

int timer_getoverrun(timer_t timerid);
```

Watchdog Timer in POSIX

```
#include <signal.h>
#include <timer.h>
#include <pthread.h>

timer_t timer;/* timer shared between monitor and server*/

struct timespec deadline = ...;
struct timespec zero = ...;

struct itimerspec alarm_time, old_alarm;

struct sigevent s;


void server(timer_t *watchdog)
{
  /* perform required service */
  TIMER_DELETE(*watchdog);
}
```

A monitor thread checks the progress on a server thread to ensure it meets its deadline

```
void watchdog_handler(int signum, siginfo_t *data,
                      void *extra)
{ /* SIGALRM handler */
  /* server is late */
  /* undertake some recovery */
}
void monitor()
{
  pthread_attr_t attributes;
  pthread_t serve;
  sigset_t mask, omask;
  struct sigaction sa, osa;
  int local_mode;
  SIGEMPTYSET(&mask);
  SIGADDSET(&mask, SIGALRM);
  sa.sa_flags = SA_SIGINFO;
  sa.sa_mask = mask;
  sa.sa_sigaction = &watchdog_handler;
  SIGACTION(SIGALRM, &sa, &osa); /* assign handler */
```

```
alarm_time.it_value = deadline;
alarm_time.it_interval = zero; /* one shot timer */

s.sigev_notify = SIGEV_SIGNAL;
s.sigev_signo = SIGALRM;

TIMER_CREATE(CLOCK_REALTIME, &s, &timer);

TIMER_SETTIME(timer, TIMER_ABSTIME, &alarm_time,
              &old_alarm);

PTHREAD_ATTR_INIT(&attributes);
PTHREAD_CREATE(&serve, &attributes,
              (void *)server, &timer);
```

RT Java: Timers

```
public abstract class Timer extends AsyncEvent
{
  protected Timer(HighResolutionTimer time, Clock clock,
                  AsyncEventHandler handler);
  public ReleaseParameters createReleaseParameters();

  public AbsoluteTime getFireTime();
  public void reschedule(HighResolutionTimer time);
  public Clock getClock();

  public void disable();
  public void enable();

  public void start(); // start the timer ticking
}
```

RT Java Timers Continued

```
public class OneShotTimer extends Timer
{
  public OneShotTimer(HighResolutionTimer time,
                      AsyncEventHandler handler);
}
public class PeriodicTimer extends Timer
{
  public PeriodicTimer(HighResolutionTimer start,
         RelativeTime interval, AsyncEventHandler handler);
  public ReleaseParameters createReleaseParameters();

  public void setInterval(RelativeTime interval);
  public RelativeTime getInterval();

  public void fire();
  public AbsoluteTime getFireTime();
}
```

Timing Errors and DPS

- With local time structures, it is also appropriate to associate timing errors with exceptions:

```
start <timing constraints> do
   -- statements
exception
   -- handlers
end
```

- In a time dependent system, it may also be necessary to give the deadline constraints of the handlers

```
start elapse 22 do
   -- statements
exception
   when elapse_error within 3 do
      -- handler
end
```

Coffee Example Revisited

```
from 9:00 to 16:15 every 45 do
  start elapse 11 do
    get_cup; boil_water
    put_coffee_in_cup; put_water_in_cup
  exception
    when elapse_error within 1 do
      turn_off_kettle  -- for safety
      report_fault; get_new_cup
      put_orange_in_cup; put_water_in_cup
    end
  end
  start after 3 elapse 26 do
    drink
  exception
    when elapse_error within 1 do empty_cup end
  end
  replace_cup
exception
  when any_exception do
    null   -- go on to next iteration
  end
end
```

Overrun of WCET

- The consequences of an error should be restricted to a well-defined region of the program
- A process that consumes more of the CPU resource than has been anticipated may miss its deadline
- If a high-priority process with a fair amount of slack time overruns its WCET, it may be a lower priority process with less slack available that misses its deadline
- It should be possible to catch the timing error in the process that caused it; hence it is necessary to be able to detect when a process overruns its worst-case execution time
- If a process is non pre-emptively scheduled (and does not block waiting for resources), its CPU execution time is equal to its elapse time and the same mechanisms that were used to detect deadline overrun can be used

CPU Time Monitoring in POSIX

- Uses the clock and timer facilities
- Two clocks are defined: `CLOCK_PROCESS_CPUTIME_ID` and `CLOCK_THREAD_CPUTIME_ID`
- These can be used in the same way as `CLOCK_REALTIME`
- Each process/thread has an associated execution-time clock; calls to:

```
clock_settime(CLOCK_PROCESS_CPUTIME_ID,
  &some_timespec_value);
clock_gettime(CLOCK_PROCESS_CPUTIME_ID,
  &some_timespec_value);
clock_getres(CLOCK_PROCESS_CPUTIME_ID,
  &some_timespec_value)
```

- will set/get the execution-time or get the resolution of the execution time clock associated with the calling process (similarly for threads)

CPU Time Monitoring Continued

- Two functions allow a process/thread to obtain and access the clock of another process/thread.

```
int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
int pthread_getcpuclockid(pthread_t thread_id,
                          clockid_t *clock_id);
```

- POSIX timers can be used to create timers which will generate signals when the execution time has expired
- As the signal generated by the expiry of the timer is directed at the process, it is application-dependent which thread will get the signal if a thread's execution-time timer expires
- As with all execution time monitoring, it is difficult to guarantee the accuracy of the execution-time clock in the presence of context switches and interrupts

WCET and RT Java

- Real-Time Java allows a *cost* value to be associated with the execution of a schedulable object
- If supported by the implementation, this allows an asynchronous event to be fired if the cost is exceeded

Overrun of Sporadic Events

- A sporadic event firing more frequently than anticipated has an enormous consequence for a system with hard deadlines
- It is necessary either to ensure that this is prohibited or to detect it when it occurs and take some corrective action
- There are essentially two approaches to prohibiting sporadic event overrun
  - If the event if from a hardware interrupt, the interrupt can be inhibited from occurring by manipulating the associated device control registers.
  - Another approach is to use sporadic server technology (see later)
- Alternatively, it is necessary to detect when they are occurring too frequently; most real-time languages and operating systems are woefully lacking in support for this

Timing Errors and BER

- see book

Summary

- The introduction of the notion of time into real-time programming languages has been described in terms of four requirements
  - access to a clock,
  - delaying,
  - timeouts,
  - deadline specification and scheduling.
- It is useful to introduce the notion of a temporal scope
  - deadline for completion of execution
  - minimum delay before start of execution
  - maximum delay before start of execution
  - maximum execution time
  - maximum elapse time
- Consideration was given as to how temporal scopes can be specified in programming languages

Summary

- The degree of importance of timing requirements is a useful way of characterising real-time systems
- Constraints that must be met are termed hard; those that can be missed occasionally, or by a small amount, are called firm or soft
- To be fault tolerant of timing failures, it is necessary to be able to detect:
  - overrun of deadline
  - overrun of worst-case execution time
  - sporadic events occurring more often than predicted
  - timeout on communications.
- Following detection, event-based reconfiguration may need to be undertaken