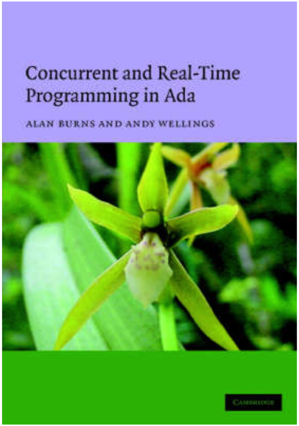
Programming Languages for Real-Time Systems

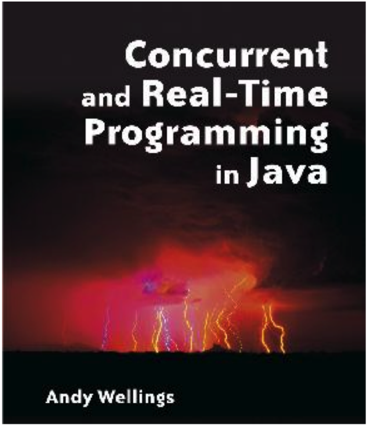
Prof. Dr. Jian-Jia Chen

LS 12, TU Dortmund

20 June 2016

References





Slides are based on Prof. Wang Yi, Prof. Peter Marwedel, and Prof. Alan Burns.

Terminologies

- Time-aware system makes explicit reference to time (e.g. open vault door at 9.00)
- Reactive system must produce output within a relative deadline (as measured from input)
  - Control systems are reactive systems
  - Required to constraint input and output (time) variability, input jitter and output jitter control
- Time-triggered computation is triggered by the passage of time
  - Release activity at 9.00
  - Release activity every 25ms, called a periodic activity
- Event-trigger computation is triggered by an external or internal event
  - The released activity is called sporadic, if there is a lower bound on the arrival interval of the event
  - The released activity is called aperiodic, if there is no such bound

Concurrent Programming

- The name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems
- Implementation of parallelism is a topic in computer systems (hardware and software) that is essentially independent of concurrent programming
- Concurrent programming is important because it provides an abstract setting to study parallelism without getting bogged down in the implementation details

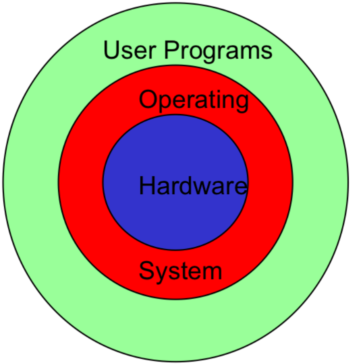
Why We Need It

- The alternative is to use sequential programming techniques
- The programmer must construct the system so that it involves the cyclic execution of a program sequence to handle the various concurrent activities
- This complicates the programmer's already difficult task and involves him/her in considerations of structures which are irrelevant to the control of the activities in hand
- The resulting programs will be more obscure and inelegant
- It makes decomposition of the problem more complex
- Parallel execution of the program on more than one processor will be much more difficult to achieve
- The placement of code to deal with faults is more problematic

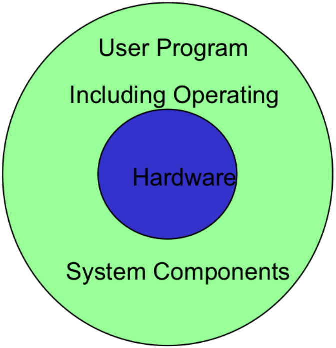
Programming Languages for Real-Time Systems

- Normally require operating system support
  - Assembly languages
  - Sequential systems implementation languages, e.g. C.
- No operating system support
  - High-level concurrent languages
  - For example, Ada, Real-Time Java, Real-Time POSIX, etc.
- Synchronous programming languages
  - Esterel, Lustre, Signal, etc.
- Model-based programming languages (from models to code)
  - Giotto, Real-Time UML, SimuLink, etc.

Real-Time Languages and OSes



Typical OS Configuration



Typical Embedded Configuration

Should concurrency be in a language or in the OS?

- Arguments for language-based concurrency:
  - It leads to more readable and maintainable programs
  - There are many different types of OSs; the language approach makes the program more portable
  - An embedded computer may not have any resident OS
  - Some compiler optimizations are invalid if using OS concurrency
  - It is easier to verify the satisfactions of the timing and safety requirements
- Arguments against:
  - It is easier to compose programs from different languages if they all use the same OS model
  - It may be difficult to implement a language's model of concurrency efficiently on top of an OSs model
  - OS standards are beginning to emerge
- The Ada/Java philosophy is that the advantages outweigh the disadvantages

Outline

Ada

Real-Time Java

Model-Based Design and Synchronous Programming

Ada

- After Ada Lovelace (regarded to be the 1st female programmer)
- The US Department of Defense (DoD) wanted to avoid multitude of programming languages obsolete or hardware-dependent
  - Reduced the number of programming languages used in these applications (fell from 450 in 1983 to 37 in 1996 by wiki)
  - Definition of requirements by a high order language working group
  - Selection of a language from a set of competing designs
  - selected design based on PASCAL
  - It has become a language for general-purpose computing with concurrent requirement
- Ada2005 now supports EDF, Fixed-Priority Scheduling, PIP/PCP, non-preemptive scheduling, Round-Robin, etc.

Real Time Programming: we need support for

- Concurrency (Ada tasking)
- Communication & synchronization (Ada Rendezvous)
- Consistency in data sharing (Ada protected data type)
- Real time facilities (Ada real time packages and delay statements)
  - accessing system time so that the passage of time can be measured
  - delaying processes until some future time
  - Timeouts: waiting for or running some action for a given time period

System Time

- A timer circuit programmed to interrupt the processor at a fixed rate.
- Each time interrupt is called a system tick (time resolution):
  - Normally, the tick can vary 1-50ms (or even microseconds) in RTOS
  - The tick may be selected by the user
  - All time parameters for tasks should be a multiple of the tick
  - System time = 32 bits
    - One tick = 1ms: system can run 50 days
    - One tick = 20ms: system can run 1000 days = 2.5 years
    - One tick = 50ms: system can run 2500 days= 7 years
- In Ada95 it is required that the system time should last at least 50 years

Real-Time Support in Ada

- Two pre-defined packages to access the system clock Ada.Calendar and Ada.Real_Time
  - Both based on the same hardware clock
- There are two delay-statements
  - Delay time (in seconds)
  - Delay until time
- The delay statements can be used together with select to program timeouts, timed entry etc.

Ada.Calendar

```
package Ada.Calendar is
  type Time is private;
      --- time is pre-defined based on the system clock
  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;
      --- Duration is pre-defined type (length of interval,
      --- expressed in sec's) declared in the package: Standard
  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);
```

Ada.Calendar (cont.)

```
function Time_Of(Year     : Year_Number;
                 Month    : Month_Number;
                 Day      : Day_Number;
                 Seconds  : Day_Duration := 0.0)
return Time;

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;
private
   -- not specified by the language
   -- implementation dependent
end Ada.Calendar;
```

Ada.Real_Time

```
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;
                  --- as Duration, a Time_Span value M representing
                   the length of an interval, corresponding to
                   the real time duration M*Time_Unit.
  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;
  function "-" (Left : Time; Right : Time_Span) return Time;
  function "-" (Left : Time; Right : Time) return Time_Span;
  function "<" (Left, Right : Time) return Boolean;
  function "<="(Left, Right : Time) return Boolean;
  function ">" (Left, Right : Time) return Boolean;
  function ">="(Left, Right : Time) return Boolean;
```

Ada.Real_Time (cont.)

```
function "+" (Left, Right : Time_Span) return Time_Span;
function "-" (Left, Right : Time_Span) return Time_Span;
function "-" (Right : Time_Span) return Time_Span;
function "*" (Left : Time_Span; Right : Integer) return Time_Span;
function "*" (Left : Integer; Right : Time_Span) return Time_Span;
function "/" (Left, Right : Time_Span) return Integer;
function "/" (Left : Time_Span; Right : Integer) return Time_Span;
function "abs"(Right : Time_Span) return Time_Span;
function "<" (Left, Right : Time_Span) return Boolean;
function "<="(Left, Right : Time_Span) return Boolean;
function ">" (Left, Right : Time_Span) return Boolean;
function ">="(Left, Right : Time_Span) return Boolean;
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;
```

Relative Delays

- Delay the execution of a task for a given period
- Relative delays (using clock access) – busy waiting

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

- To avoid busy-waiting, most languages and Operation Systems provide some form of delay primitive
  - In Ada, this is a delay statement delay 10.0;
  - In UNIX, sleep(10);

Absolute Delays

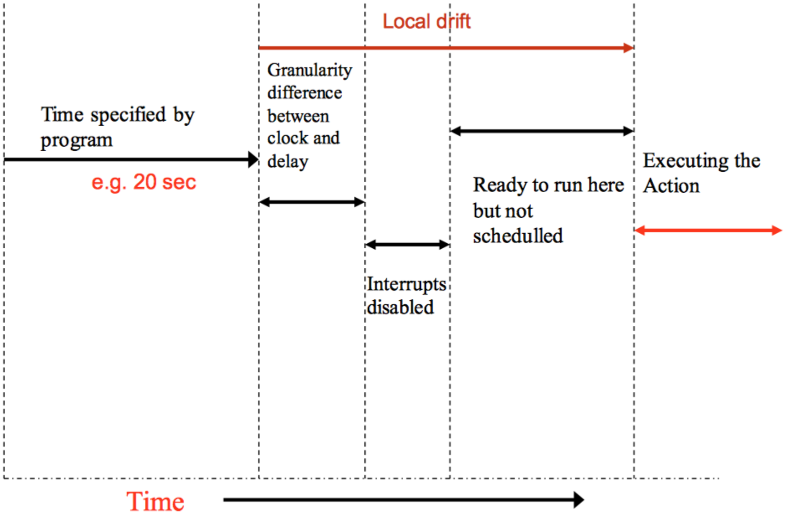
- To delay the execution of a task to a specified time point (using clock access) – busy waiting:

```
Start := Clock;
FIRST ACTION;
loop
exit when Clock > Start+10.0;
end loop;
SECOND Action;
```

- To avoid busy-wait:

```
Start := Clock;
FIRST ACTION;
delay until START + 10.0; (this is by interrupt)
SECOND Action;
```

Ada Delay


Local drift
Time specified by program
e.g. 20 sec
Granularity difference between clock and delay
Interrupts disabled
Ready to run here but not schedulled
Executing the Action
Time

Periodic Task

```
task body Periodic_T is
        Next_Release : Time;
        ReleaseInterval : Duration := 10
begin
        Next_Release := Clock  + ReleaseInterval;
        loop
                -- Action
                delay until Next_Release;
                Next_Release := Next_Release + ReleaseInterval;
        end loop;
end Periodic T;
```

Controller 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;
```

Controller Example (cont.)

```
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;
```

Controller Example (cont.)

```
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;
```

Outline

Ada

Real-Time Java

Model-Based Design and Synchronous Programming

Real-Time Specification for Java (RTSJ)

- Java was designed as a platform-independent language
- Especially the byte-code representation reduces the required space and can be used for embedded systems
- Java was also designed as a safe language, compared to C/C++, especially for memory protections
- Standard java is unfortunately not suitable for real-time embedded systems
  - The run-time library is too big
  - The garbage collection has to be handled carefully to avoid impact on the timing properties
  - Prioritization among threads is not well specified
- RTSJ
  - supports a fixed-priority based threading model
  - supports for PIP and PCP to handle priority inversions
  - garbage collector has to be run in a predictable way
  - Unlike Ada, Real-Time Java explicitly distinguishes between threads and real-time threads

ReleaseParameter Class

```
public class ReleaseParameters implements Cloneable{

  protected ReleaseParameters(RelativeTime cost,
            RelativeTime deadline,
            RelativeTime blockingTerm,
            AsyncEventHandler overrunHandler,
            AsyncEventHandler missHandler);
   ...
  // methods
  public RelativeTime getCost();
  public void setCost(RelativeTime cost);
  ...
}
```

Release Parameters

- The processing cost for each release and its blocking time
- Its relative deadline
- If the object is periodic or sporadic, then an interval is also given
- Event handlers can be specified for the situation when the deadline is missed or the processing cost consumed is larger than specified
- There is no requirement to monitor the processing time consumed by a schedulable object

An extract from the RealtimeThread Class

```
package javax.realtime;
public class RealtimeThread extends java.lang.Thread
       implements Schedulable {
  public RealtimeThread();
  public RealtimeThread(SchedulingParameters scheduling,
                        ReleaseParameters release);
  public RealtimeThread(SchedulingParameters scheduling,
      ReleaseParameters release, MemoryParameters memory,
      MemoryArea area, ProcessingGroupParameters group,
      Runnable logic);
  ...
  public void start();
  public void release();

  public static boolean waitForNextPeriod();
  public static boolean waitForNextRelease();
  // note there are AIE interruptible versions of the above
  ...
}
```

Remarks

- Scheduling Parameters
  - An empty class
  - Subclasses allow the priority of the object to be specified and, potentially, its importance to the overall functioning of the application
  - RTSJ specifies a minimum range of real-time priorities (28)
- MemoryParameters
  - the maximum amount of memory used by the object in an associated memory area
  - the maximum amount of memory used in immortal memory
  - a maximum allocation rate of heap memory.
- ProcessingGroupParameters
  - allows several schedulable objects to be treated as a group and to have an associated period, cost and deadline

PeriodicParameters Class

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

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

Periodic Task - Parameters

For period 10ms, relative deadline 5ms, execution time 1ms, starting at absolute time A, we have:

```
{
  AbsoluteTime A = new AbsoluteTime(...);
  PeriodicParameters P = new PeriodicParameters(
     A, // start time
     new RelativeTime(10,0), // period
     new RelativeTime(1,0),  // cost
     new RelativeTime(5,0),  // deadline
     null,  null ); // no deadline miss/cost overrun handlers

  Periodic ourThread = new Periodic(P); //create thread
  ourThread.start(); // release it
}
```

Periodic Task - Body

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

  public void run() {
   boolean deadlineMet= true;
   while(deadlineMe) {
     // code to be run each period
     ...
     deadlineMet = waitForNextPeriod();
   }
   // a deadline has been missed,
   // and there is no handler
   ...
  }
}
```

Semantics of waitForNextPeriod

- On a DEADLINE MISS
  - The RTSJ assumes that in this situation the thread itself will undertake some corrective action
  - If there are no handlers, waitForNextPeriod (wFNP) will not block the thread in the event of a deadline miss (it returns false immediately) .
  - Where the handler is available, the RTSJ assumes that the handler will take some corrective action and therefore it automatically deschedules the thread. If appropriate, the handler reschedules the thread
- If a deadline is met
  - wFNP returns true at the next release time

Outline

Ada

Real-Time Java

Model-Based Design and Synchronous Programming

RT Programming Languages

- Classic high-level languages with RT extensions e.g.
  - Ada
  - Real-Time Java, C + RTOS
  - SDL
- Synchronous Programming (from 1980s)
  - Esterel
  - Lustre
  - Signal
- Design, Modeling, Validation, and Code Generation (from models to code)
  - Giotto
  - Real-Time UML
  - SimuLink

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
```

Esterel (cont.)

- 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

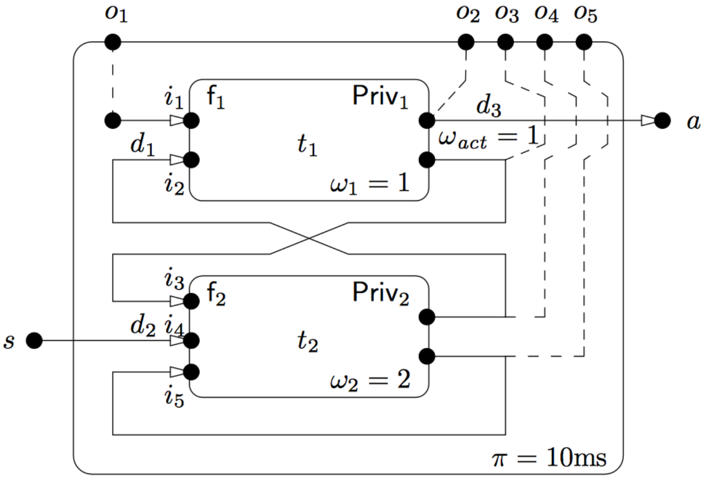
```
signal S in
  present S else emit S end
end
```

- 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

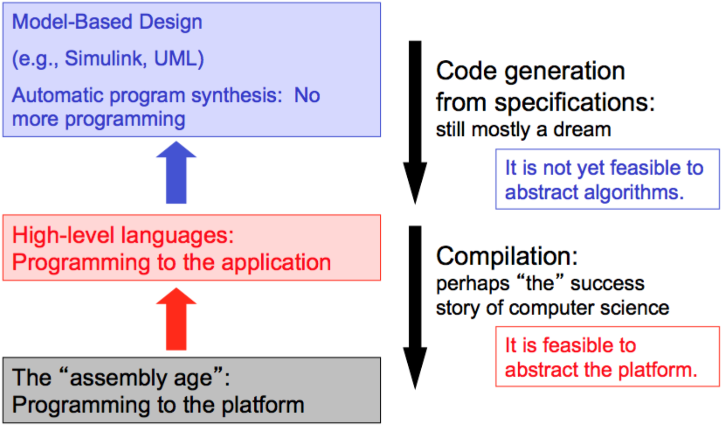
Giotto

- A language for control applications
  - A task may have an arbitrary number of input and output ports.
  - A task may also maintain a state, which can be viewed as a set of private ports whose values are inaccessible outside the task.
  - Giotto tasks are periodic tasks.
  - A Giotto program consists of a set of modes, each of which repeats the invocation of a fixed set of tasks. The Giotto program is in one mode at a time.
  - A mode switch describes the transition from one mode to another mode. For this purpose, a mode switch specifies a switch frequency, a target mode, and a driver.
- The periodic invocation of tasks, the reading of sensor values, the writing of actuator values, and the mode switching are all triggered by real time.
- A Giotto program does not specify where, how, and when tasks are scheduled.

Example of Giotto in One Mode



Lifting the Level of Abstraction



modified from Edward Lee’s slides