Course Outline (lectures)

- Introduction
  - Characteristics of RTS
- Real Time Operating Systems (RTOS)
  - OS support: tasking, scheduling, resource handling, OSEK
- Real Time Programming Languages
  - Language support, e.g. Ada tasking
- Scheduling and Timing Analysis
  - Worst-case execution and response time analysis
  - Resource reservation and resource servers
- Distributed real time systems
  - Real Time Communication: CAN Bus
- Workload Models (advanced topic)
  - Graph-based task models
- Multiprocessor real-time systems (advanced topics)
  - Architectures and real-time scheduling
- Design and Validation (advanced topics)
  - Modeling and Verification
Today’s topic
Modeling Real-Time Systems
Plan

- Motivation: Why Modeling?
  - Basic concepts
- Finite Automata: State Machines
  - Examples
- Timed Automata: State Machines with clocks
- Modeling Ada Programs as Automata
Motivation: Why modeling?
Traditional software development

The Waterfall Model

Analysis → Design → Implementation → Testing

Problem Area

Errors are detected late or never:
30-50% of time for testing
Errors detected: the late the more expensive
Introducing, Detecting and Correcting errors
Finding errors as early as possible!

HOW?
Example: the Ada assignment
(all 82 students were wrong!)

When the number of messages received is over 100, Consumer should tell Buffer and Producer to stop
Example: the Ada assignment (all 82 students were wrong!)

When the number of msgs received is over 100
Example: the Ada assignment (all 82 students were wrong!)

When the number of msgs received is over 100
Example: the Ada assignment (all 82 students were wrong!)

Producer

put

Consumer

get

When the number of msgs received is over 100

stop

The run-time system will generate an Exception !!!!
Example: the Ada assignment
(all 82 students were wrong!)

Buffer

Producer

put

Consumer

When the number of msgs received is over 100

stop

Solution?
Example: the Ada assignment (all 82 students were wrong!)

When the number of msgs received is over 100

Deadlock! 😞

If Buffer if full, Producer is trying to put and blocked, Buffer is waiting for Consumer to empty one-slot and Consumer is waiting for Producer to stop
Here is the deadlock!

- **Producer**
  - B.put
  - accept(stop)

- **Buffer**
  - B.put
  - accept(put)
  - accept(get)

- **Consumer**
  - B.get
  - If m<100 Then B.get
  - If m=100 then P.stop
You can find the Bug using UPPAAL

Which is a software tool for modeling and Verification developed by Uppsala Uppsala (Sweden) & Aalborg University (Denmark)
Software development

- Problem Area
- Modelling and verification
- Implementation
- Testing
- Running System

Analysis Design Implementation Testing
Modeling and Verification

- "modeling" is a design process: describe systematically the abstract behaviour of a system
  - It is to create a model or a design proposal of the system to be developed

- "verification" is to check whether the model satisfies given requirement specifications
  - It is similar to testing, but testing on the model
Modeling and Verification with Finite Automata (i.e. State Machines)

- Modeling
  - Use finite automata to describe all the possible executions of system components
  - Model a system as a network of automata

- Verification
  - Check properties of the state machines using software tools like UPPAAL
Finite Automata: Examples of Modeling
Intelligent Light Control

**WANT:** if press is issued twice quickly then the light will get brighter; otherwise the light is turned off.

The double circle is the initial state
Finite Automata

It has 3 states and 3 transitions

And it accepts sequences out of the language:
\[ c! + c!d? + c!d?c! + c!(d?c!)*a! \]
Input and Output Actions

We use ? and ! to denote the pairs of complementary actions (i.e. call and accept) in rendezvous communication.
Automata as tasks

- We view an automaton as a task where the action labels stand for synchronization actions e.g. Ada’s rendezvous.
- The previous example could be a network protocol where c! and d? stand for “connect!” and “disconnect?” and a! for “abort!”
Networks of automata
(think about Ada Tasking, entry and call etc)

The tasks can synchronize on c and then a
The static architecture of a concurrent system: $A \ || \ B$

where $c$, $a$, $d$ stand for "ports" or "channels"
The static architecture of a concurrent system: $A \parallel B \parallel C$
Networks of automata
Networks of automata: initial state
Networks of automata: enabled transition
Networks of automata: transition taken

A --> c! -> B
A --> d? -> B
B --> c? -> C
B --> a? -> B
B --> a! -> A
C --> d! -> C
Networks of automata: nondeterministic choices (transitions)
Networks of automata: non-deterministic transitions (1)
Networks of automata: non-deterministic transition (2)
Networks of automata: deadlock!
Networks of automata: deadlock!

Unless there is another task willing to synchronize on any of the actions
Timed Automata:
State Machines with Timing Information
Finite Automata

No information about when the transitions take place!
Let’s add data types: Clock and Integer
**Intelligent Light Control**

**WANT:** if press is issued twice quickly then the light will get brighter; otherwise the light is turned off.
Intelligent Light Control (with timer)

**Solution:** Add real-valued clock $x$
Clocks and timing constraints

- Now we assume that the system has a finite number of clocks
  - The clocks start to increase from 0 when the system starts, and they run at the same rate
  - The clocks can be tested using e.g. \( x < 100 \)
  - The clocks can be reset to 0 on a transition using an assignment: \( x := 0 \)
Timed automata = Finite Automata with **clock type**

1. connect and reset $x$ to 0
2. If succeed within 100ms, then go to initial
3. abort after 100ms!
Timed automata with **Integers**

1. connect and reset $x$ to 0
2. If succeed within 100ms, then go to initial
3. abort after 100 ms!
Network of Timed automata

c! $x:=0$, $i:=3$

$x<100$  s? $i:=i+1$

x=>100

a!

i:=0

x:=0, y:=0

C?

y<10

i==3

s!

x:=0, y:=0
Network of Timed automata: initial states

- $c! x:=0, i:=3$
- $x<100 \quad s? \quad i:=i+1$
- $x=>100 \quad a!$
- $i:=0$

- $y<10 \
i==3$
- $s!$
- $x:=0, y:=0$

- $C?$
Network of Timed automata:
enabled transition

c!  x:=0, i:=3

x<100  s?  i:=i+1

x=>100  a!
i:=0

C?

y<10  i:=i  
i==3

s!
x:=0, y:=0
Network of Timed automata:

transition taken

c!
\[ x := 0, i := 3 \]

\[ x < 100 \]
\[ s? \]
\[ i := i + 1 \]

\[ x \geq 100 \]
a!
\[ i := 0 \]

y < 10
i == 3
s!
\[ x := 0, y := 0 \]
Network of Timed automata: enabled transition

This depends on the values of $x$, $y$, and $i$
Network of Timed automata: transition taken

This depends on the values of $x$, $y$, and $i$
Timed automata (definition)

- a timed automaton is a finite graph
  - a finite many nodes \( N \) with an initial node
  - a finite many edges \( E \) between nodes
  - an edge may be labelled with three elements
    - guard
    - action (a?, a!, or nothing)
    - assignment
      (may be skipped if none)
Guard (i.e. Conditional Expression)

- a clock constraint
  - \( g ::= x \leq n \mid x \geq n \mid x < n \mid x > n \mid g \land g \)
  - where \( n \) is any natural number

- a predicate over data variables
  - "any logical expression" you may write in C
assignment

- a clock reset: \( x := 0 \) for any clock \( x \)

- a sequence of assignments in the form \( i := e \) (or most of the programming constructs in C)
Timed Automata: Syntax

- **Clocks:** $x, y$
- **Data variable:** $i$
- **Guard:** clock constraint or predicate on variables
- **Reset/assignment:** on clocks and/or data variables
- **Nodes/locations:** $m, n$
- **Edges/transitions:** $m \to n$

**Action used for synchronization**
- $x \leq 5 \& y > 3$
- $i < 100$
- $x := 0$
- $i := i + 1$

Initial node is a double circle
Initial values of all clocks and variables are 0
Examples of Timed Automata
Timed Automata: Example

\[
\begin{align*}
X &= 0 \\
X &\geq 2
\end{align*}
\]
Timed Automata: Example

- $X \geq 2$
- $X := 0$

Graph showing the value of $x$ over time from 2 to 10.
Timed Automata: Example

$2 \leq x \leq 3$

$x := 0$

value of $x$

2 4 6 8 10

2 3 4
Timed Automata: Example
Modelling of Real-Time Systems

- Real Time
  - Clocks
- Synchronization
  - Rendezvous
  - Shared variables
- Concurrency
  - Networks of timed automata
Networks of Timed Automata

Example transitions

\((l1, m1, \ldots, x=2, y=3.5, i=3, \ldots) \longrightarrow (l2, m2, \ldots, x=0, y=3.5, i=7, \ldots)\)
Modeling Ada Programs
Rendezvous

task body A is
begin
...
B.Call;
...
end A

task body B is
begin
...
accept Call do
....
end Call
...
end A
Rendezvous

task body A is begin
... B.Call;
... end A

task body B is begin
... accept Call do
.... end Call
... end A

---

task body A is begin
... B.Call;
... end A

task body B is begin
... accept Call do
.... end Call
... end A

---

Task A

Task B

call!
call?
Producer, Buffer and Consumer

Producer -> Buffer
Buffer <- Consumer

put
get
Buffer

task buffer is
  entry put(X: in integer)
  entry get(x: out integer)
end;

task body buffer is
  v: integer;
  begin
  loop accept put(x: in integer) do v:= x end put;
      accept get(x: out integer) do x:= v end get;
  end loop;
end buffer;

---

buffer.put(...)  ←----------------------------- other tasks (users)!!
buffer get(...)
Buffer

task buffer is
  entry put(x: in integer)
  entry get(x: out integer)
end;

task body buffer is
  v: integer;
  begin
  loop accept put(x: in integer); v := x end put;
    accept get(y: out integer); y := v end get;
  end loop;
end buffer;

buffer.put(...) ← other tasks (users)!!
buffer.get(...)
task Producer;
task body Producer is
  m: integer;
  begin
    loop
      m := foo() -- produce a new message;
      Buffer.put(m);
      end loop;
  end Producer;
Consumer

task Consumer;
task body Consumer is
  m: integer;
  begin
    loop
      Buffer.get(m);  -- get a new message;
      foo(m);         -- do something with m;
    end loop;
  end Consumer;
Producer, Buffer and Consumer

Producer -> Buffer
Producer:
- `put! x:=m`
- `m:= foo()`
Buffer:
- `get?`
- `y:=v`
- `v:=x`
Consumer:
- `get?`
- `m:= y`

Example:
- `foo(m)`
Timeout and message passing

```plaintext
loop
  select
    accept set(T : temperature) do
      New_temp:=T;
      end Call;
    or
      delay 10.0;
      --action for timeout
  end select;
  --other actions
end loop;
```
Timeout and message passing

```
loop
    select
        accept set(T: temperature) do
            New_temp := T;
        end Call;
        or
            delay 10.0;
            -- action for timeout
    end select;
end loop;
```
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;
Periodic Activity

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