

# Synchronous Languages—Lecture 04

Prof. Dr. Reinhard von Hanxleden

Christian-Albrechts Universität Kiel  
Department of Computer Science  
Real-Time Systems and Embedded Systems Group

21 April 2020

*Last compiled: April 21, 2020, 11:15 hrs*



*Esterel II—Pragmatics*

## The 5-Minute Review Session

1. What is the difference between *transformational/interactive/reactive* systems?
2. What is *perfect synchrony*? What is the *synchronous model of computation*?
3. What is the motivation for the Esterel language?
4. What is the *multiform notion of time*?
5. What does it mean for an Esterel statement to be *instantaneous*? Name some instantaneous and non-instantaneous statements.

## The 5-Minute Review Session

1. What is a *signal* in Esterel?
2. What are the *signal coherence rules*?
3. What are the differences between *signals* and *variables*?
4. What is the *WTO principle*?
5. What *control flow constructs* does Esterel have?

## The 5-Minute Review Session

1. What is a *signal resolution function*? What are its requirements?
2. What is the difference between *immediate* and *non-immediate* abort?
3. What is the difference between *strong* and *weak abort*?
4. What is the difference between *strong* and *weak suspend*?
5. What is the difference between traps and weak aborts?

# Overview

## Examples

- People Counter Example
- Vending Machine Example
- Tail Lights Example
- Traffic-Light Controller Example

## Interfacing with the Environment

## People Counter Example

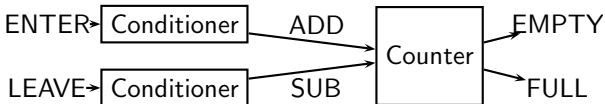
Construct an Esterel program that counts the number of people in a room.

- ▶ People enter the room from one door with a photocell that changes from 0 to 1 when the light is interrupted, and leave from a second door with a similar photocell. These inputs may be true for more than one clock cycle. The two photocell inputs are called ENTER and LEAVE.
- ▶ There are two outputs: EMPTY and FULL, which are present when the room is empty and contains three people respectively.

Source: Mano, *Digital Design*, 1984, p. 336

*Thanks to Stephen Edwards (Columbia U) for providing this and the following examples*

## Overall Structure



Conditioner detects rising edges of signal from photocell.  
Counter tracks number of people in the room.

## Implementing & Testing the Conditioner

```
module CONDITIONER:  
  input A;  
  output Y;  
  
  loop  
    await A; emit Y;  
    await [not A];  
  end  
  
end module
```

```
% estereel -simul cond.str1  
% gcc -o cond cond.c -lcsimul # may need -L  
% ./cond  
CONDITIONER> ;  
--- Output:  
CONDITIONER> A; # Rising edge  
--- Output: Y  
CONDITIONER> A; # Doesn't generate a pulse  
--- Output:  
CONDITIONER> ; # Doesn't generate a pulse  
--- Output:  
CONDITIONER> ; # Sensitive to A again  
--- Output:  
CONDITIONER> A; # Another rising edge  
--- Output: Y  
CONDITIONER> ;  
--- Output:  
CONDITIONER> A;  
--- Output: Y
```



## Implementing & Testing the Counter: First Try

```

module COUNTER:
  input ADD, SUB;
  output FULL, EMPTY;

  var count := 0 : integer in
    loop
      present ADD then if count < 3 then
        count := count + 1 end end;
      present SUB then if count > 0 then
        count := count - 1 end end;
      if count = 0 then emit EMPTY end;
      if count = 3 then emit FULL end;
      pause
    end
  end

end module
    
```

```

COUNTER> ;
--- Output: EMPTY
COUNTER> ADD SUB;
--- Output: EMPTY
COUNTER> ADD;
--- Output:
COUNTER> SUB;
--- Output: EMPTY
COUNTER> ADD;
--- Output:
COUNTER> ADD;
--- Output:
COUNTER> ADD;
--- Output: FULL
COUNTER> ADD SUB;
--- Output: # Oops!
    
```

## Implementing & Testing the Counter: Second Try

```

module COUNTER:
input ADD, SUB;
output FULL, EMPTY;

var c := 0 : integer in
  loop
    present ADD then
      present SUB else
        if c < 3 then c := c + 1 end end
      else
        present SUB then
          if c > 0 then c := c - 1 end end;
        end;
        if c = 0 then emit EMPTY end;
        if c = 3 then emit FULL end;
        pause
      end
    end
  end module

```

```

COUNTER> ;
--- Output: EMPTY
COUNTER> ADD SUB;
--- Output: EMPTY
COUNTER> ADD SUB;
--- Output: EMPTY
COUNTER> ADD;
--- Output:
COUNTER> ADD;
--- Output:
COUNTER> ADD;
--- Output: FULL
COUNTER> ADD SUB;
--- Output: FULL # Working
COUNTER> ADD SUB;
--- Output: FULL
COUNTER> SUB;
--- Output:
COUNTER> SUB;
--- Output:
COUNTER> SUB;
--- Output: EMPTY
COUNTER> SUB;
--- Output: EMPTY

```

## Assembling the People Counter

```
module PEOPLECOUNTER:
input ENTER, LEAVE;
output EMPTY, FULL;

signal ADD, SUB in
  run CONDITIONER[signal ENTER / A, ADD / Y]
||
  run CONDITIONER[signal LEAVE / A, SUB / Y]
||
  run COUNTER
end

end module
```

## Vending Machine Example

Design a vending machine controller that dispenses gum once.

- ▶ Two inputs, **N** and **D**, are present when a nickel and dime have been inserted.



- ▶ A single output, **GUM**, should be present for a single cycle when the machine has been given fifteen cents.



- ▶ No change is returned.

Source: Katz, *Contemporary Logic Design*, 1994, p. 389

## Vending Machine Solution

```
module VENDING:
input N, D;
output GUM;

loop
  var m := 0 : integer in
    trap WAIT in
      loop
        present N then m := m + 5; end;
        present D then m := m + 10; end;
        if m >= 15 then exit WAIT end;
        pause
      end
    end;
    emit GUM; pause
  end
end
end module
```

## Alternative Solution

```
loop
  await
  case immediate N do await
    case N do await
      case N do nothing
        case immediate D do nothing
        end
      case immediate D do nothing
    end
  case immediate D do await
    case immediate N do nothing
    case D do nothing
  end
end;
emit GUM; pause
end
```

## Tail Lights Example

Construct an Esterel program that controls the turn signals of a 1965 Ford Thunderbird.



Source: Wakerly, *Digital Design Principles & Practices*, 2ed, 1994, p. 550

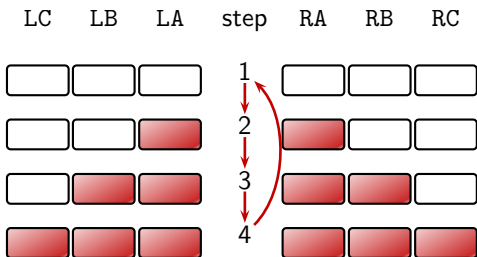
## Tail Light Behavior





## Tail Lights

- ▶ There are three inputs, which initiate the sequences: **LEFT**, **RIGHT**, and **HAZ**
- ▶ Six outputs: **LA**, **LB**, **LC**, **RA**, **RB**, and **RC**
- ▶ The flashing sequence is



## A Single Tail Light

```
module LIGHTS:  
output A, B, C;  
  
    loop  
        pause;  
        emit A; pause;  
        emit A; emit B; pause;  
        emit A; emit B; emit C; pause  
    end  
  
end module
```

## The T-Bird Controller Interface

```
module THUNDERBIRD :  
input LEFT, RIGHT, HAZ;  
output LA, LB, LC, RA, RB, RC;  
  
...  
  
end module
```

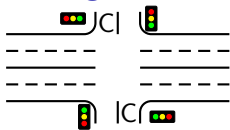
## The T-Bird Controller Body

```
loop
  await
  case immediate HAZ do
    abort
    run LIGHTS[signal LA/A, LB/B, LC/C]
    ||
    run LIGHTS[signal RA/A, RB/B, RC/C]
  when [not HAZ]
  case immediate LEFT do
    abort
    run LIGHTS[signal LA/A, LB/B, LC/C]
  when [not LEFT]
  case immediate RIGHT do
    abort
    run Lights[signal RA/A, RB/B, RC/C]
  when [not RIGHT]
  end
end
```

## Comments on the T-Bird

- ▶ This solution uses Esterel's innate ability to control the execution of processes, producing succinct easy-to-understand source but a somewhat larger executable.
- ▶ **An alternative:** Use signals to control the execution of two processes, one for the left lights, one for the right.
- ▶ **A challenge:** Synchronizing hazards.
- ▶ Most communication signals can be either level- or edge-sensitive.
- ▶ Control can be done explicitly, or implicitly through signals.

## Traffic-Light Controller Example



Control a traffic light at the intersection of a busy highway and a farm road.

Source: Mead and Conway, *Introduction to VLSI Systems*, 1980, p. 85.

- ▶ Normally, the highway light is green
- ▶ If a sensor detects a car on the farm road:
  - ▶ The highway light turns yellow then red.
  - ▶ The farm road light then turns green until there are no cars or after a long timeout.
  - ▶ Then, the farm road light turns yellow then red, and the highway light returns to green.
- ▶ **Inputs:** The car sensor **C**, a short timeout signal **S**, and a long timeout signal **L**.
- ▶ **Outputs:** A timer start signal **R**, and the colors of the highway and farm road lights **HG**, **HY**, **HR**, **FG**, **FY**, and **FR**.

# The Traffic Light Controller

```

module TLC:
input C, SEC;
output HG, HY, HR,
        FG, FY, FR;

signal R, L, S in
    run TIMER
||
    run FSM
end

end module
    
```

```

module TIMER:
input R, SEC;
output L, S;

loop
    weak abort
        await 3 SEC;
        [
            sustain S
            ||
            await 5 SEC;
            sustain L
        ]
    when R;
end

end module
    
```

```

module FSM:
input C, L, S;
output R, HG, HY, HR,
        FG, FY, FR;

loop
    emit HG; emit FR; emit R;
    await [C and L];
    emit HY; emit R;
    await S;
    emit HR; emit FG; emit R;
    await [(not C) or L];
    emit FY; emit R;
    await S;
end

end module
    
```

# Overview

## Examples

### Interfacing with the Environment

Available Alternatives

Handling Inconsistent Outputs

Events vs. State



## Interfacing with the Environment

- ▶ At some point, our reactive system must control real-world entities
- ▶ There are usually different options for the interface—differing in
  - ▶ Ease of use
  - ▶ Ease of making mistakes!
- ▶ **Example:** External device that can be ON or OFF
- ▶ **Options:**
  1. Single pure signal
  2. Two pure signals
  3. Boolean valued signal

## Different Modes of Motor Control

### Option 1: Single pure signal

- ▶ Motor is running in every instant which has the **MOTOR** signal present

### Pro:

- ▶ Minimal number of signals

### Con:

- ▶ High number of signal emissions (signal is emitted in every instant where the motor is on)—may be unnecessary run-time overhead
- ▶ Somewhat heavy/unintuitive representation

```
input BUMPER;  
output MOTOR;  
  
abort  
  sustain MOTOR  
when BUMPER
```

## Different Modes of Motor Control

### Option 2: Two pure signals

- ▶ Motor is switched on with signal **MOTOR\_ON** present
- ▶ Motor is switched off with signal **MOTOR\_OFF** present
- ▶ If neither **MOTOR\_ON** or **MOTOR\_OFF** is present, motor keeps its previous state

### Pro:

- ▶ Signal emissions truly indicate significant change of external state
- ▶ Simple representation in Esterel

### Con:

- ▶ No way to control inconsistent outputs
- ▶ No memory

```
input BUMPER;  
output MOTOR_ON,  
       MOTOR_OFF;  
  
emit MOTOR_ON;  
await BUMPER;  
emit MOTOR_OFF;
```

## Inconsistent Outputs

- ▶ Problem with MOTOR\_ON and MOTOR\_OFF: undefined behavior with both signals present
- ▶ Can address this at host-language level
- ▶ Can (and should) also address this at Esterel-level:

```
[
  present BUMPER else
    emit MOTOR_ON;
    await BUMPER
  end present;
  emit MOTOR_OFF
]
||
[
  await immediate MOTOR_ON and MOTOR_OFF;
  exit INTERNAL_ERROR
]
```

## Valued Signal for Motor Control

### Option 3: Boolean valued signal

- ▶ Merge pure signals `MOTOR_ON` and `MOTOR_OFF` into one valued signal `MOTOR`
- ▶ Motor is switched on if every emit-statement in that instant emits true

```
input BUMPER;  
output MOTOR combine BOOLEAN with and;  
  
emit MOTOR(true);  
await immediate BUMPER;  
emit MOTOR(false);
```

- ▶ Here: In case of conflicting outputs, motor stays switched off

## Valued Signal for Motor Control

Option 3 contd.

Pro:

- ▶ Again only one signal for motor control
- ▶ Explicit control of behavior for inconsistent outputs
- ▶ Valued signal has memory—can be polled in later instances, after emission
- ▶ Easy extension to finer speed control

Con:

- ▶ Inconsistent outputs are handled deterministically—but are not any more detected and made explicit
- ▶ For certain classes of analyses/formal methods that we may wish to apply, valued signals are more difficult to handle than pure signals

## Events vs. State

- ▶ Excessive signal emissions
  - ▶ make the behavior difficult to understand
  - ▶ cause overhead if fed to the external environment
- ▶ State:
  - ▶ “Robot is turning left”
  - ▶ “Motor is on”
  - ▶ Esterel:
    - ▶ waiting for some signal
    - ▶ terminated thread
    - ▶ value of valued signal
- ▶ Event:
  - ▶ Change of State
  - ▶ “Turn motor on”
  - ▶ Esterel:
    - ▶ emit pure signal
    - ▶ change value of signal

## Summary

- ▶ Esterel allows to specify precisely what happens if inputs arrive in combinations—but must consider this from application perspective as well
- ▶ Can memorize state in signal/variable values or as program state
- ▶ Several choices when interfacing with environment—must consider simplicity, robustness