CMPE 117

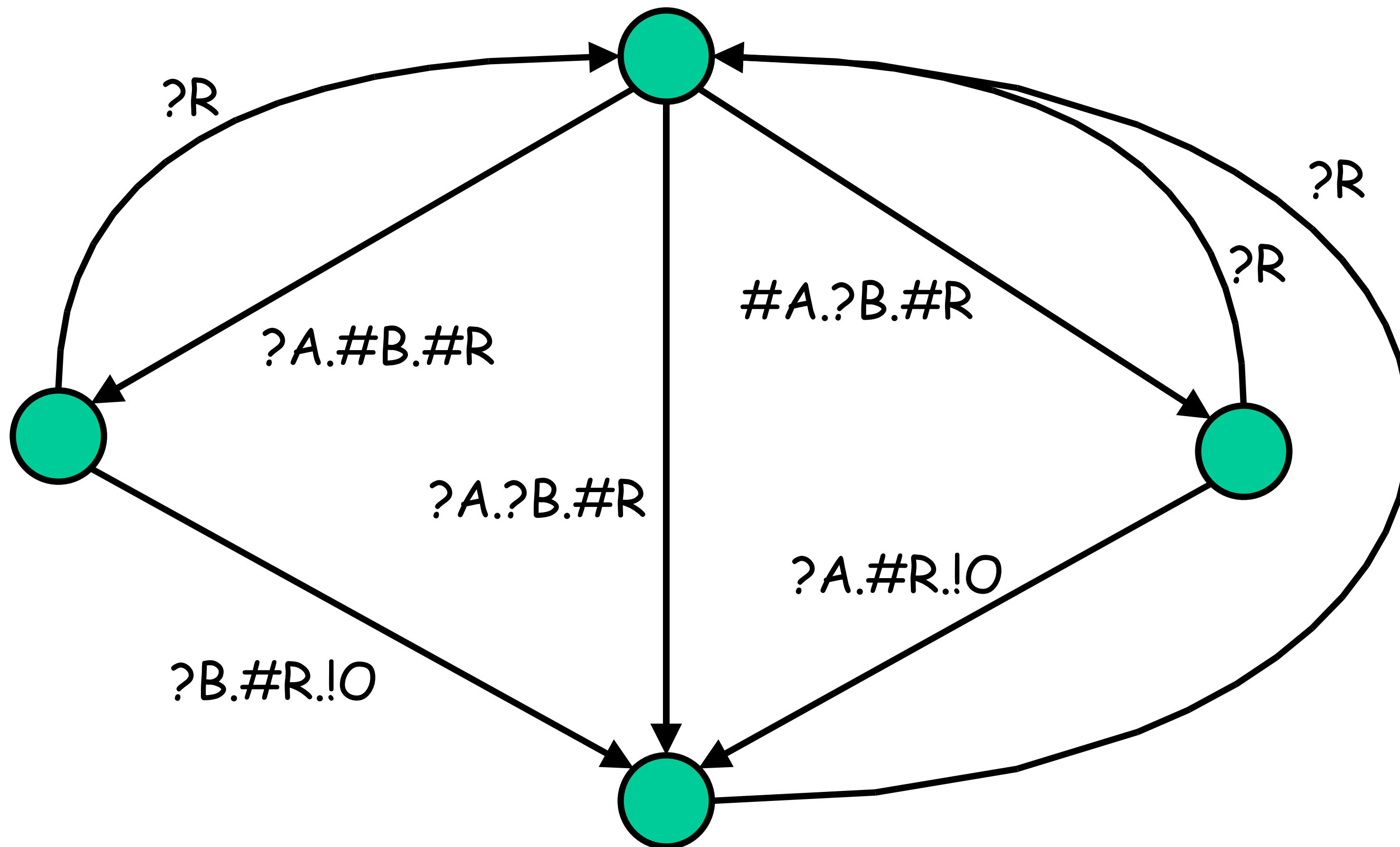
Esterel

© Luca de Alfaro, 2007

Part I

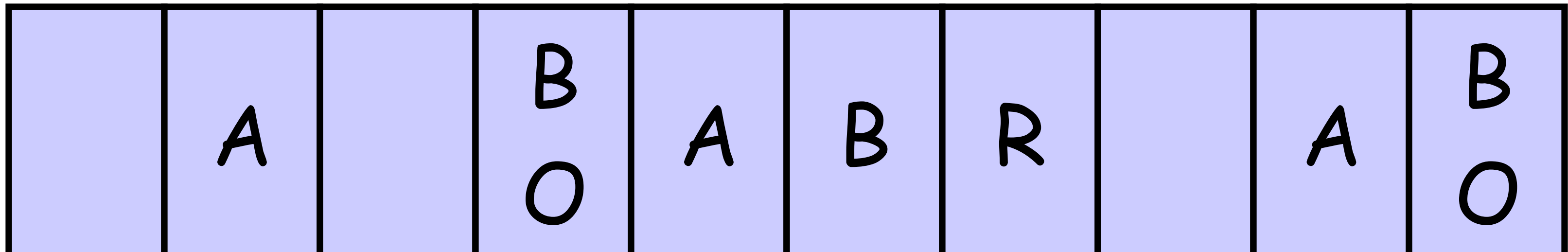
Preview

The ABRO Mealy Machine



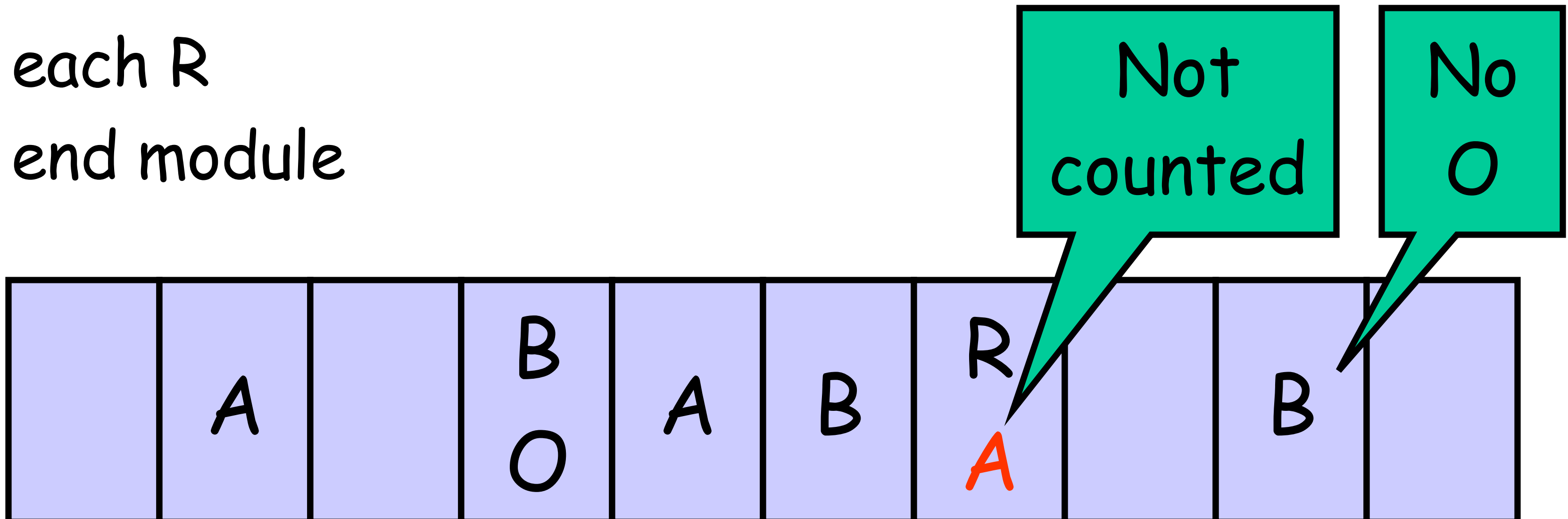
Module ABRO

```
module ABRO:
input A, B, R;
output O;
loop
   [ await A || await B ];
   emit O
each R
end module
```



Module ABRO

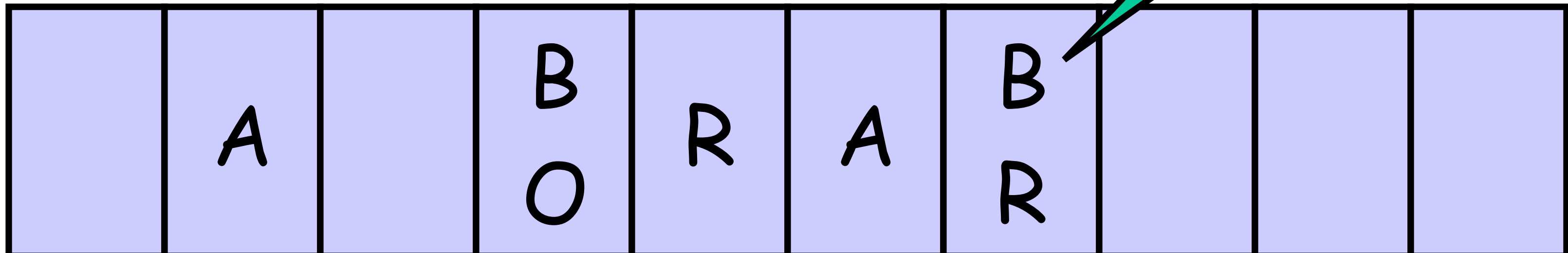
```
module ABRO:
input A, B, R;
output O;
loop
   [ await A || await B ];
   emit O
each R
end module
```



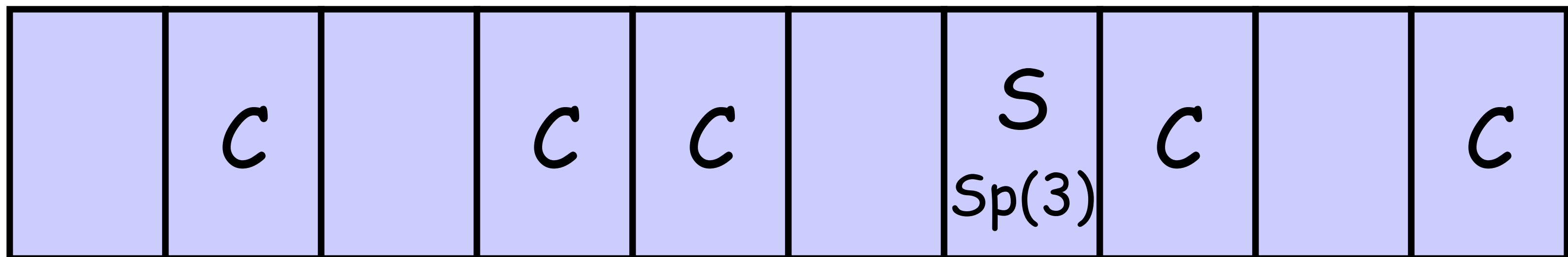
Module ABRO

```
module ABRO:
input A, B, R;
output O;
loop
   [ await A || await B ];
   emit O
each R
end module
```

Not counted due to R, so no O



```
module SPEED:
input Centimeter, Second;
relation Centimeter # Second;
output Speed: Integer;
loop
    var Distance := 0 : integer in
            abort
                    every Centimeter do
                            Distance := Distance + 1
                    end every
            when Second do
                    emit Speed(Distance)
            end abort
    end var
end loop
end module
```



Esterel:
The basic principles

Concurrency and Communication

- **Variables:**
  - Local to a thread, cannot be used for communication
  - Can change value sequentially (as in PLs) within a time-blob
- **Signals:**
  - Global, can be used for communication
  - They are either present or absent: no temporal ordering within a time-blob ("instant")

Concurrency and communication

```
    x := 0;
    x := x + 1
||
    x := 1
```

Meaning???

Concurrency and communication

```
   x := 0;
   x := x + 1;
   emit A(x)
||
   y := ?A;
   emit B(y+1)
```

Meaning:

- A is present, with value 1
- B is present, with value 2
- A and B are both present, and they are NOT ordered.

Concurrency and communication

```
   x := 0;
   x := x + 1;
   emit A(x)
||
   y := ?A;
   emit A(y+1)
```

Meaning:

- Illegal: causality loop.

Concurrency and communication

```
output A;
output B: combine Integer with +;
   x := 0;
   x := x + 1;
   emit B(x);
   emit A(x);
||
   y := ?A;
   emit B(y+1)
```

Meaning:

- A present, value 1.
- B present, value 1 + 2 = 3

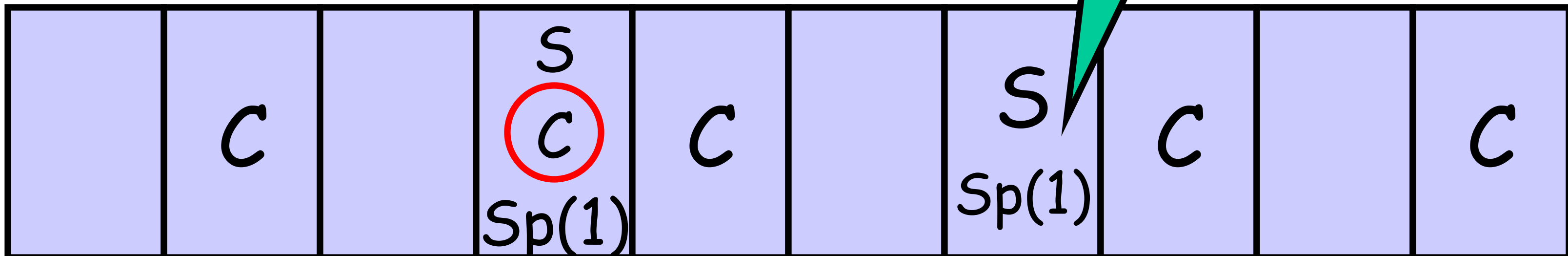
Concurrency and communication

emit *A*(?*A* + 1);

Illegal! Causality loop.

(We will learn later what causality is – for now, use your intuition).

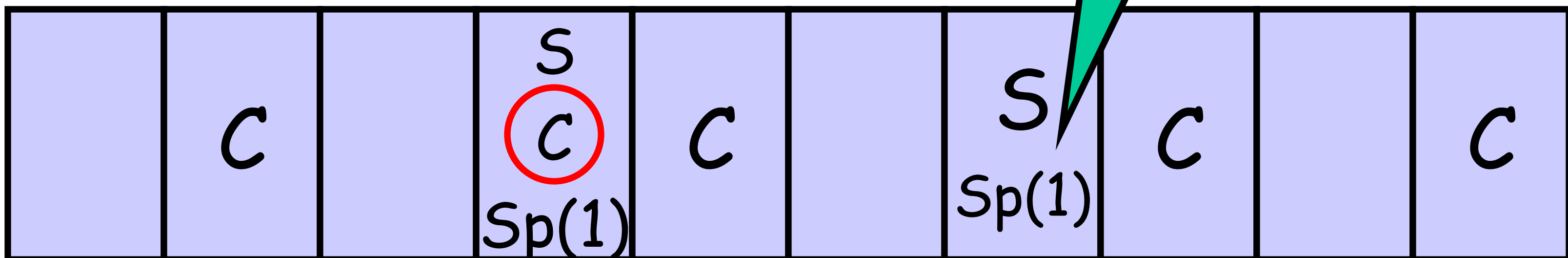
```
module SPEED:
input Centimeter, Second;        % Now they can occur together
output Speed: Integer;
loop
    var Distance := 0 : integer in
        abort
            every Centimeter do
                Distance := Distance + 1
            end every
        when Second do
            emit Speed(Distance)
        end abort
    end var
end loop
end module
```



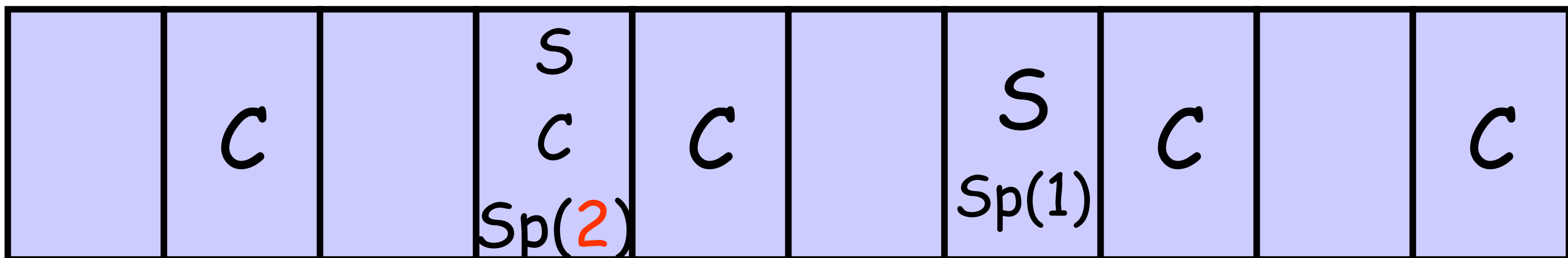
abort p when S

- In the starting instant, p is immediately started, the initial presence or absence of S being ignored.
- If p terminates before S occurs, then the whole abort statement terminates at the same time.
- If S occurs while p has not yet terminated, then the abort statement immediately terminates and p does not receive the control in the current instant.

```
module SPEED:
input Centimeter, Second;        % Now they can occur together
output Speed: Integer;
loop
    var Distance := 0 : integer in
        abort
            every Centimeter do
                Distance := Distance + 1
            end every
        when Second do
            emit Speed(Distance)
        end abort
    end var
end loop
end module
```



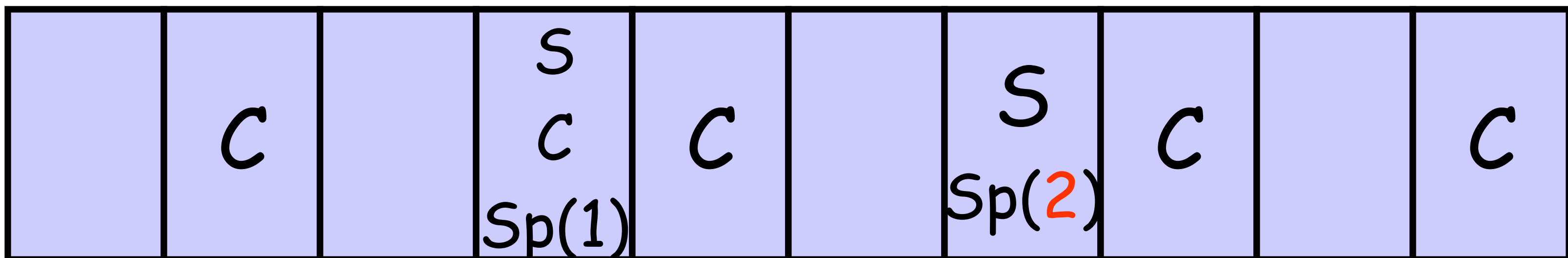
```
module SPEED:
input Centimeter, Second;         % Now they can occur together
output Speed: Integer;
loop
    var Distance := 0 : integer in
          weak abort
                    every Centimeter do
                              Distance := Distance + 1
                    end every
          when Second do
                    emit Speed(Distance)
          end abort
    end var
end loop
end module
```



weak abort p when S

- In the starting instant, p is immediately started, the initial presence or absence of S being ignored.
- If p terminates before S occurs, then the whole abort statement terminates at the same time.
- If S occurs while p has not yet terminated, then p receives control for one last instant, then the abort statement terminates.

```
module SPEED:
input Centimeter, Second;        % Now they can occur together
output Speed: Integer;
loop
    var Distance := 0 : integer in
        abort
            every immediate Centimeter do
                Distance := Distance + 1
            end every
        when Second do
            emit Speed(Distance)
        end abort
    end var
end loop
end module
```

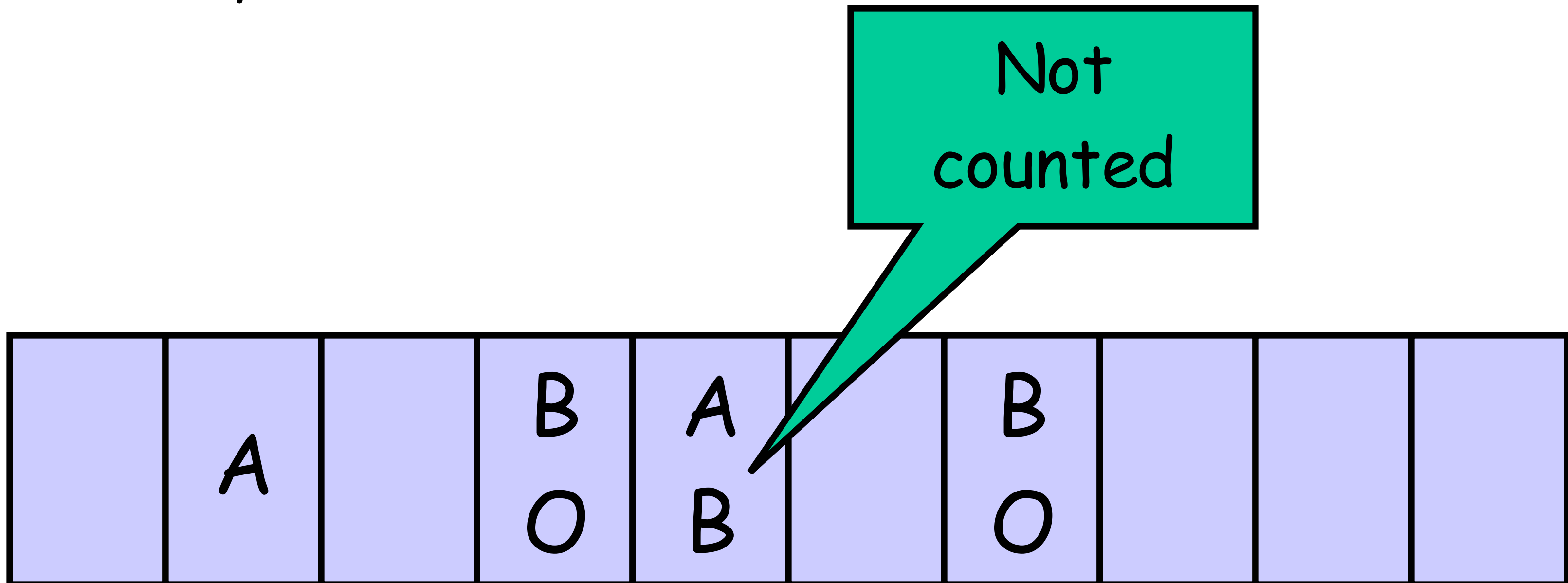


Immediate await

- The ‘every’ and ‘await’ statements usually await for the *next* occurrence of a signal: not the occurrence in the current instant.
- The statements can be preceded by ‘immediate’ to denote that an occurrence in the current instant satisfies them.

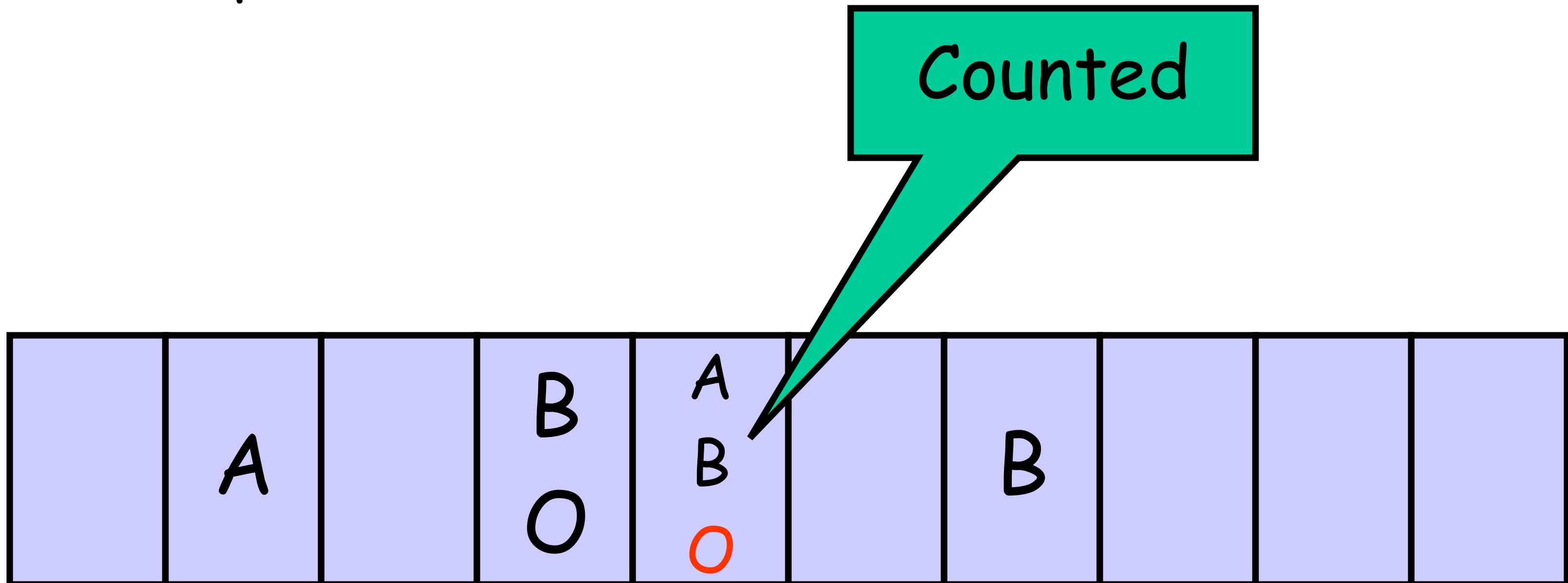
Immediate await

```
loop
    await A;
    await B;
    emit O;
end loop
```



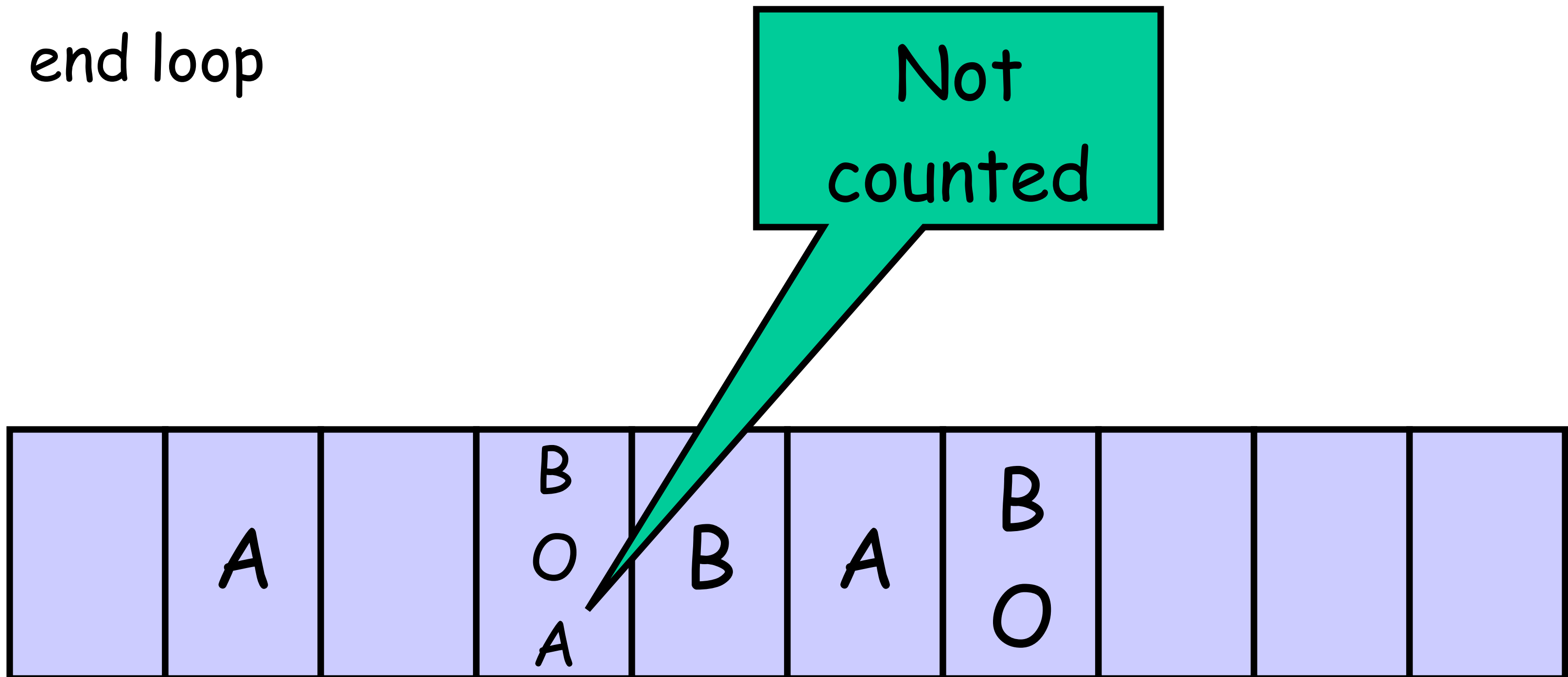
Immediate await

```
loop
    await A;
    immediate await B;
    emit O;
end loop
```



Immediate await

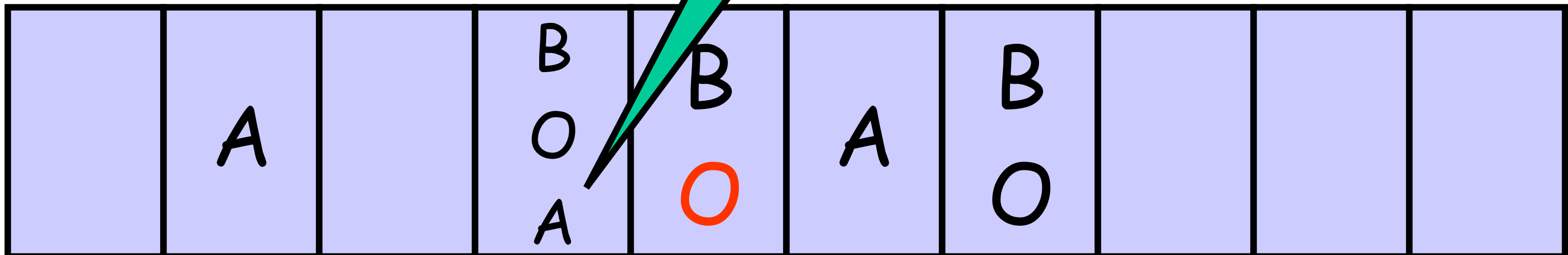
```
loop
   await A;
   await B;
   emit O;
end loop
```



Immediate await

```
loop
    immediate await A;
    await B;
    emit O;
end loop
```

Counted



Immediate await

```
loop
    immediate await A;
    immediate await B;
    emit O;
end loop
```

Illegal: Every loop must be guaranteed to take at least one instant, and the one above can execute in zero instants, if A and B both present.

loop ... each

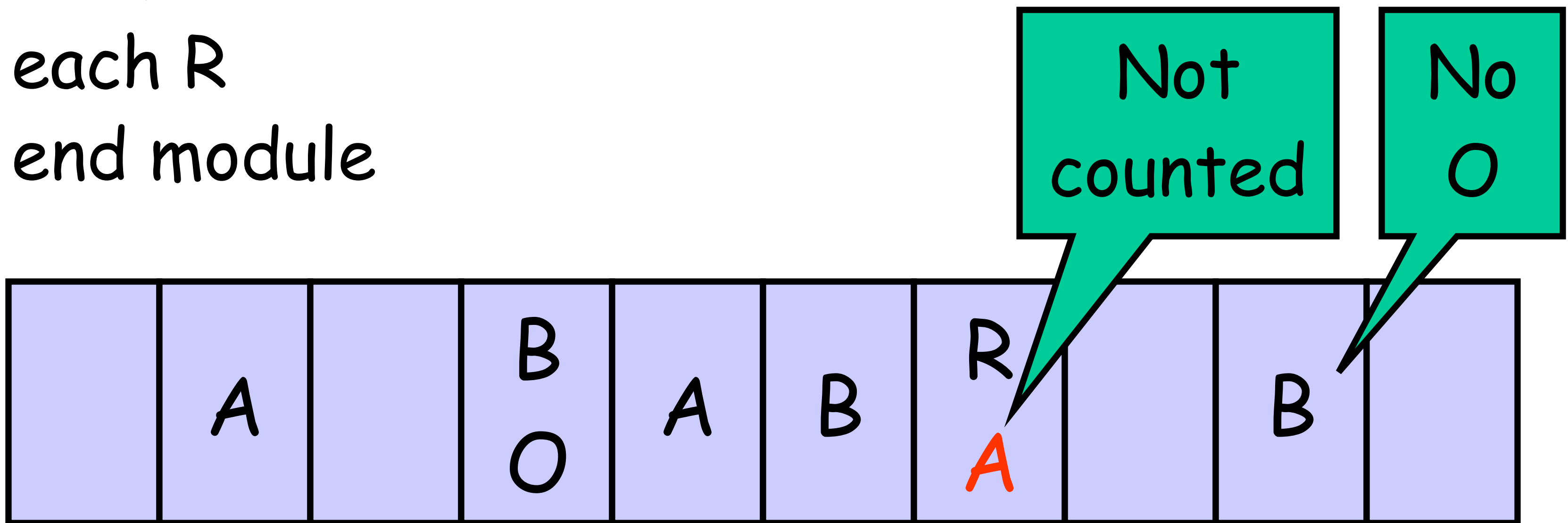
```
loop
    p
each A

is equivalent to:

loop
    abort
        p; halt
    when A
end loop
```

Module ABRO

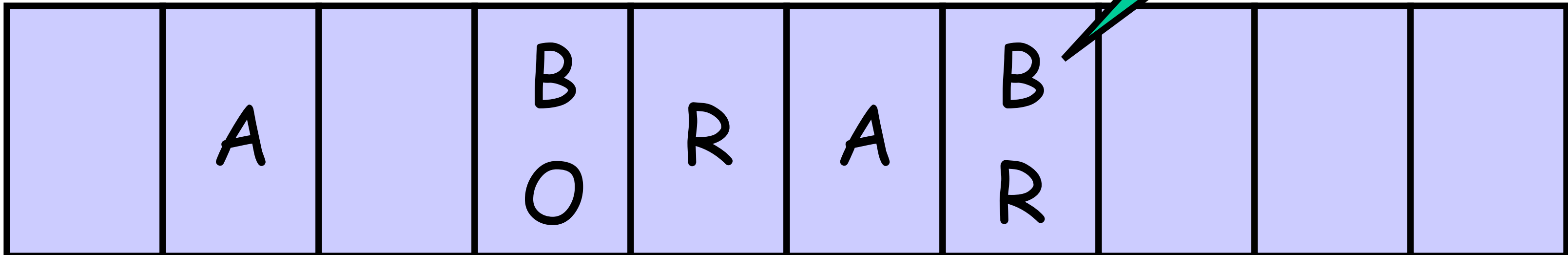
```
module ABRO:
input A, B, R;
output O;
loop
   [ await A || await B ];
   emit O
each R
end module
```



Module ABRO

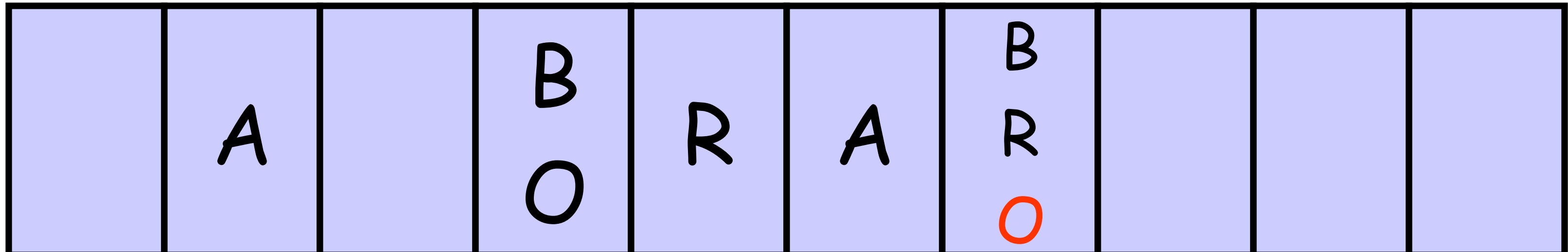
```
module ABRO:
input A, B, R;
output O;
loop
    abort
        [ await A || await B ];
        emit O;
        halt
    when R
end loop
end module
```

Not counted due to R, so no O



Module WABRO

```
module WABRO:
input A, B, R;
output O;
loop
   weak abort
      [ await A || await B ];
      emit O;
      halt
   when R
end loop
end module
```



Reuse

- What if we have to await A, B, C, then emit O, with reset R?
- One solution is to write a new WABCRO module from scratch.
- Another is to instantiate twice the module WABRO.

WABCRO, reusing WABRO

```
module WABCRO:
input A, B, C, R;
output O;
signal D in
   run WABRO [ signal D / O]
||
   run WABRO [ signal D / A,   C / B]
end signal
end module
```

sustain

```
module TWO_STATES_SIMPLE:
input On, Off;
output IsOn, IsOff;
loop
   abort
        sustain IsOff
   when On;
   abort
        sustain IsOn
   when Off
end loop
end module
```

		On		Off	On	Off		On Off	
IsOff	IsOff	IsOn	IsOn	IsOff	IsOn	IsOff	IsOff	?	

sustain

```
module TWO_STATES_SIMPLE:
input On, Off;
output IsOn, IsOff;
loop
   abort
      sustain IsOff
   when On;
   abort
      sustain IsOn
   when Off
end loop
end module
```

		On		Off	On	Off		On Off	
IsOff	IsOff	IsOn	IsOn	IsOff	IsOn	IsOff	IsOff	IsOn	IsOn

suspend

```
suspend
   sustain Regul (RegFun (?Speed, ?Gaspedal))
when Coast
```

suspend

```
suspend
   sustain Regul (RegFun (?Speed, ?Gaspedal))
when immediate Coast
```

suspend

```
suspend
   <put here your code to seek light>
when immediate Bump
```

Note: you can suspend complex activities that take multiple instants to complete, not only "sustain" statements. Try to do this in C!

suspend

```
signal Coast in
   suspend
        sustain Regul (RegFun (?Speed, ?Gaspedal))
   when coast
||
   loop
        await CoastOn;
        abort
                sustain Coast
        when Coastoff
   end loop
end signal
```

suspend

```
signal Coast in
    suspend
        sustain Regul (RegFun (?Speed, ?Gaspedal))
    when coast
||
    signal IsOff in
        run TWO_STATES_SIMPLE
                [signal CoastOn / On,
                        CoastOff / Off,
                        Coast / IsOn ]
    end signal
end signal
```

present

```
module TWO_STATES_SIMPLE:
input On, Off;
output IsOn, IsOff;
loop
    abort
        sustain IsOff
    when On;
    abort
        sustain IsOn
    when Off
end loop
end module
```

What if we want it to start with IsOn?

		On		Off	On	Off			
IsOff	IsOff	IsOn	IsOn	IsOff	IsOn	IsOff	IsOff	IsOff	IsOff

present

```
module TWO_STATES_SIMPLE:
input StartOn, On, Off;
output IsOn, IsOff;
loop
    present StartOn else
        abort
            sustain IsOff
        when On;
    end present
    abort
        sustain IsOn
    when Off
end loop
end module
```

StartOn IsOn	IsOn	On IsOn	IsOn	Off IsOff	On IsOn	Off IsOff	IsOff	IsOff	IsOff

Running external code

```
module REGUL:
function RegFun (integer, integer): integer;
input Centimeter, Second;
sensor GasPedal: integer;
relation Centimeter # Second;
output Regul: integer;
signal Speed: integer in
   run SPEED
||
   await Speed;
   sustain Regul (RegFun (?Speed, ?GasPedal))
end signal
end module
```

The Esterel Language

Part II

The details

Global Structure of a Program

- A program consists of several modules:

module *name:*

interface declaration

statement

end module

Modules

- A program consists of several modules, of which one is the designated (main) one.
- A module can be instantiated (run) in another module via the **run** statement.

Data

- Esterel knows only about few types:
  - **boolean** (constants: true, false; ops: and, or, not)
  - **integer**
  - **float**
  - **double**
  - **string**
  - user types
- **=, < >** (and also **+, -, *, /, <=, <, >, >=**)
- No type conversions!
- No other operations!
- Do all that in the host language.

User Types

- Users can declare types, e.g.:

  type Time

  type Beep
- Same for constants:

  constant Noon, Winter

All of this is implemented in the host language.

Functions and Procedures

- **function** CompareTime (Time, Time): boolean;
- **procedure** IncrementTime (Time) (integer);
- **procedure** Translate (Rectabgle) (Coord, integer);

- Both are instantaneous.
- Functions return a value.
- Procedures get two lists
  - list of arguments passed by reference
  - list of arguments passed by value

Tasks

- Very similar to procedures, but <u>not instantaneous.</u>
- Executed with the **exec** statement.

  task MoveRobot (Coord) (Rectangle);

- More on tasks later.

Signals and Sensors

- Signals can be **input**, **output**, **inputoutput**, **return**.
- The **return** signal is used for tasks (more on this later).
- **inputoutput** is for signals that can be generated by more than one module.
- There is a specific signal "tick"

Sensors

- A sensor is a signal that is always present, and that carries a value --- think at is as a sensor!

sensor Temperature: **integer;**

Examples

- **sensor** Temperature: **integer;**
- **output** YesVotes := 0 : **combine integer with +;**

Input Relations

- **relation** A # B;

  A and B never both present
- **relation** A => B;

  If A is present, then B is also present

Local Declarations

```
signal Alarm in
   p
end signal

var x: double in
   p
end var
```

Data Expressions

- Very limited: +, -, /, etc

Signal Expressions

- Combine signals with **and**, **or**, **not:**
  - Meter **and not** Second
  - B1 **and** B2 **and not (**B3 **or** B4**)**
  - **not** tick
- We will see later that *constructive causality* applies.

Delay Expressions

- Meter **and not** Second
- **immediate [**Meter **and not** Second**]**
- 3 Second
- 3 **[**Second **and not** Meter**]**
- The multiplicators are evaluated just once.
- If the delay multiplicator evaluates to 0, it is set to 1 (only **immediate** causes immedate wait)
- No nesting of multiplicators and boolean (computers can make sense of this, but not humans).

Statements

- Parallel:

  p

||

  q

Terminates when both p and q terminate.

Basic Statements

- **nothing** *(does nothing in 0 time)*
- **pause** *(does nothing in 1 tick)*
- **halt** *(does nothing forever)*

- *Assignment:* x := *expr;*
- **call** P (x, y) (*expr1, expr2);*

Signal Emission

- **emit** *A*
- **emit** B(10)

- And remember that the value of B can then be accessed via ?B.

Sequencing

- p ; q

Looping

```
loop
   p
end loop
```

- The body of the loop must be statically guaranteed to take at least 1 time unit under all circumstances.
- No deep analysis is carried out to ensure this.

Loop Delays – invalid example

```
loop
   present I then
      present J else
         p
      end present
   else
      q
   end present
end loop
```

Even if p, q each have delays, there is a potentially 0-delay path in the loop.

repeat

repeat *<integer expr>* **times**

  p

end repeat

- The expression is evaluated, if •0 then the body is skipped.
- Esterel always considers the statement as potentially immediate (even if the expression is "1").

repeat – illegal example

```
loop
   repeat 3 times
      ...
   end repeat
end loop
```

Not allowed by the compiler.

Conditionals – signals and data

- Signals:

  present *A* **then** p **else** q **end present**

- Data

  if expr **then** p **else** q **end if**

- Similar, but very different in intent.

Await

await [immediate] delayexpr

Example:

await immediate [Second and Meter]

Abort

[weak] abort

p

when *cond* *(this can be an immediate condition)*

[**do** *q*]

end abort

Temporal Loops

```
loop
   p
each d
```

Equivalent to:

```
loop
   abort
      p; halt
   when d
end loop
```

Temporal Loops

every d
   p
end

<u>Equivalent to:</u>

await immediate d;
loop
   p
each d

Suspend

suspend

p

when [immediate] I

Suspends the execution of p whenever I is present. If "immediate" is used, then the presence of I is tested also at the first instant.

Suspend - example

```
suspend
    abort
        sustain O
    when J
when I
```

What does this do?

Suspend - example

```
suspend
    abort
        sustain O
    when J
when I
```

Emits O at all instants, till J is present. BUT, if I is present, all this activity (emission of O, detection of J) is "suspended" - as if those instants did not exist.

Traps

trap T **in**

p

end trap

If a statement "**exit** T" is executed, then the trap is exited.

Traps - example

```
trap T in
    p;
    exit T
||
    await S;
    exit T
end trap
```

What does this mean? (you have seen it already...)

Traps - example

trap T **in**

    p;

    exit T

||

    await S;

    exit T

end trap

Meaning: "**weak abort** p **when** S"

Nested Traps and Handlers

- Traps can be nested; the outer one has priority.
- Traps can have handlers:

```
trap T in
    p
handle T do
    q
end trap
```

MultipleTraps

```
trap T, U in
    p
handle T do
    q
handle U do
    r
end trap
```

Multiple Traps - example

A translation of: "**weak abort** p **when** S **do** q **end**"

```
trap T, W in
    p;
    exit T
||
    await S;
    exit W
handle W do
    q
end trap
```

Parallel statement

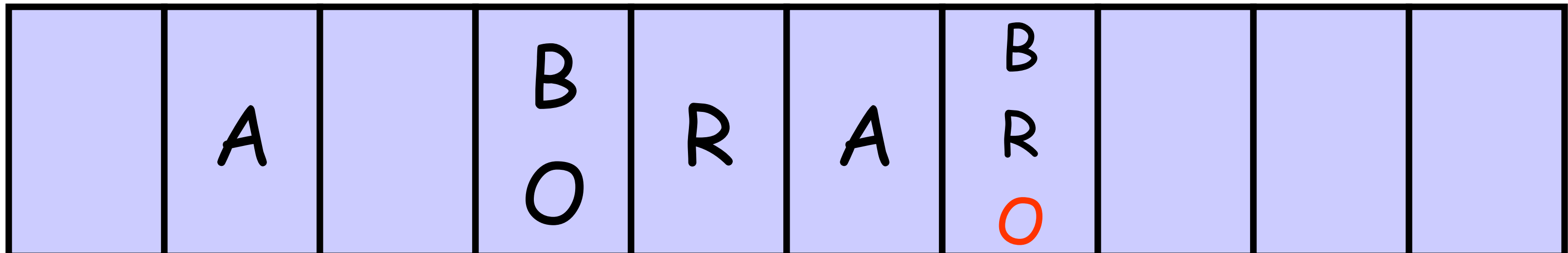
p || q

Both p and q are immediately started.

The parallel statement terminates when both p and q terminate.

Module WABRO, again

```
module WABRO:
input A, B, R;
output O;
loop
    weak abort
        [ await A || await B ];
        emit O;
        halt
    when R
end loop
end module
```



The **run** statement

```
module WABCRO:
input A, B, C, R;
output O;
signal D in
   run WABRO [ signal D / O]
||
   run WABRO [ signal D / A,   C / B]
end signal
end module
```

Some example tasks

- Write a “Flip-flop”: the inputs are S(et), R(eset). It should sustain outputs Q or NQ (negated Q) according to what is the last received.
- Write an arbiter module, that emits OA or OB according to whether *A* or B is received first, and emits S if *A* and B occur at the same time; the reset input is R.

Some example tasks

- Write a module that runs tasks p, q, r, switching from one to the next in round-Robin fashion whenever input A is received.

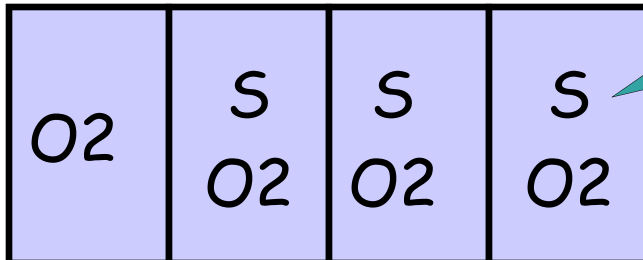
Constructive Causality

- The goal of Esterel is to establish with signals are present in each instant.
- **Constructive Causality:** *A set of rules that tells us whether a program is able to unambiguously define a value for all signals.*
- It is equivalent, in sequential hardware, to checking the absence of race conditions.

Reincarnation

Signal life does not extend beyond the syntactic scope:

```
loop
    signal S in
        present S then emit O1 else emit O2 end;
        pause;
        emit S
    end signal
end loop
```



This incarnation of S is not the one being tested to emit O2

Patological programs

Nondeterminism:

```
module P:
output O;
present O then emit O end
end module
```

Patological programs

Contradiction:

```
module P:
output O;
present O else emit O end
end module
```

Patological programs

Dependency cycles:

```
module P:
output O1, O2;
    present O1 then emit O2 end
||
    present O2 then emit O1 end
end module
```

Acyclicity?

Acyclicity has been considered as a possible solution, but it is rather restrictive:

```
module P:
input I;
output O1, O2;
present I then
    present O1 then emit O2 end
else
    present O2 then emit O1 end
end present
end module
```

Another cyclic example

```
module P:
output O1, O2;
present O1 then emit O2 end;
pause;
present O2 then emit O1 end
end module
```

Logical correctness?

Can we use logical correctness as the solution?

- Problem: highly illogical behavior! (pun intended)

```
module P:
output O1, O2;
    present O1 then emit O1 end
||
    present [O1 and not O2] then emit O2 end
end module
```

Logical correctness?

Can we use logical correctness as the solution?

- Problem: highly illogical behavior! (pun intended)

```
module P:
output O;
present O then emit O else emit O end
end module
```

The solution: Constructiveness

- Constructiveness is a list of rules that enables us to set signals present or absent.
  - Initially, all signals are set to the unknown state.
  - Rules are then applied, trying to determine the absence or presence of all signals.
  - If the presence/absence of all signals can be determined, then the module is *constructively causal;* otherwise, it is rejected by the compiler.
- Constructive causality corresponds to the absence of race conditions in hardware.

Constructive Causality: Rules

1. An unknown signal can be set present if it is emitted.
2. An unknown signal can be set absent if no emitter can emit it.
3. The **then** branch of a test can be executed if the test is executed and the signal is present.
4. The **else** branch of a test can be executed if the test is executed and the signal is absent.
5. The **then** branch of a test cannot be executed if the signal is absent.
6. The **else** branch of a test cannot be executed if the signal is present.

Examples

```
module P:
input I;
output O;
signal S1, S2 in
    present I then emit S1 end
||
    present S1 else emit S2 end
||
    present S2 then emit O end
end signal
```

A more complex example

```
module P:
output O;
signal S in
    emit S;
    present O then
        present S then
            pause
        end present;
        emit O
    end present
end signal
end module
```

Preemption

This is not constructive:

```
module P:
output O;
abort
    sustain O
when O
```

(this is similar to "present O else sustain O")

Preemption

This is constructive:

```
module P:
output O;
weak abort
    sustain O
when O
```

Lego and Esterel

```
module lego1 :
input TOUCH_1, TOUCH_3;
output MOTOR_A_SPEED(integer), MOTOR_C_SPEED(integer),
       MOTOR_A_DIR(integer), MOTOR_C_DIR(integer),
       CPUTS(string);
relation TOUCH_1 # TOUCH_3;

constant MOTOR_FWD, MOTOR_REV, MAX_SPEED : integer;
```

```
var t : integer in
   emit MOTOR_A_SPEED(MAX_SPEED/2);
   emit MOTOR_C_SPEED(MAX_SPEED/2);
   loop
      emit MOTOR_A_DIR(MOTOR_FWD);
      emit MOTOR_C_DIR(MOTOR_FWD);
      emit CPUTS("fwd");
      await [TOUCH_1 or TOUCH_3];
      present TOUCH_1 then
         t:=1;
      else
         t:=3;
      end present;
      emit MOTOR_A_DIR(MOTOR_REV);
      emit MOTOR_C_DIR(MOTOR_REV);
      emit CPUTS("rev");
      await 100 tick;        % 1 tick = 1 ms
      if t=1 then
         emit MOTOR_A_DIR(MOTOR_FWD);
         emit CPUTS("right");
      else
         emit MOTOR_C_DIR(MOTOR_FWD);
         emit CPUTS("left");
      end if;
      await 100 tick;  % 1 tick = 1 ms
end loop
end var.
```