An imperative synchronous language

Estérel

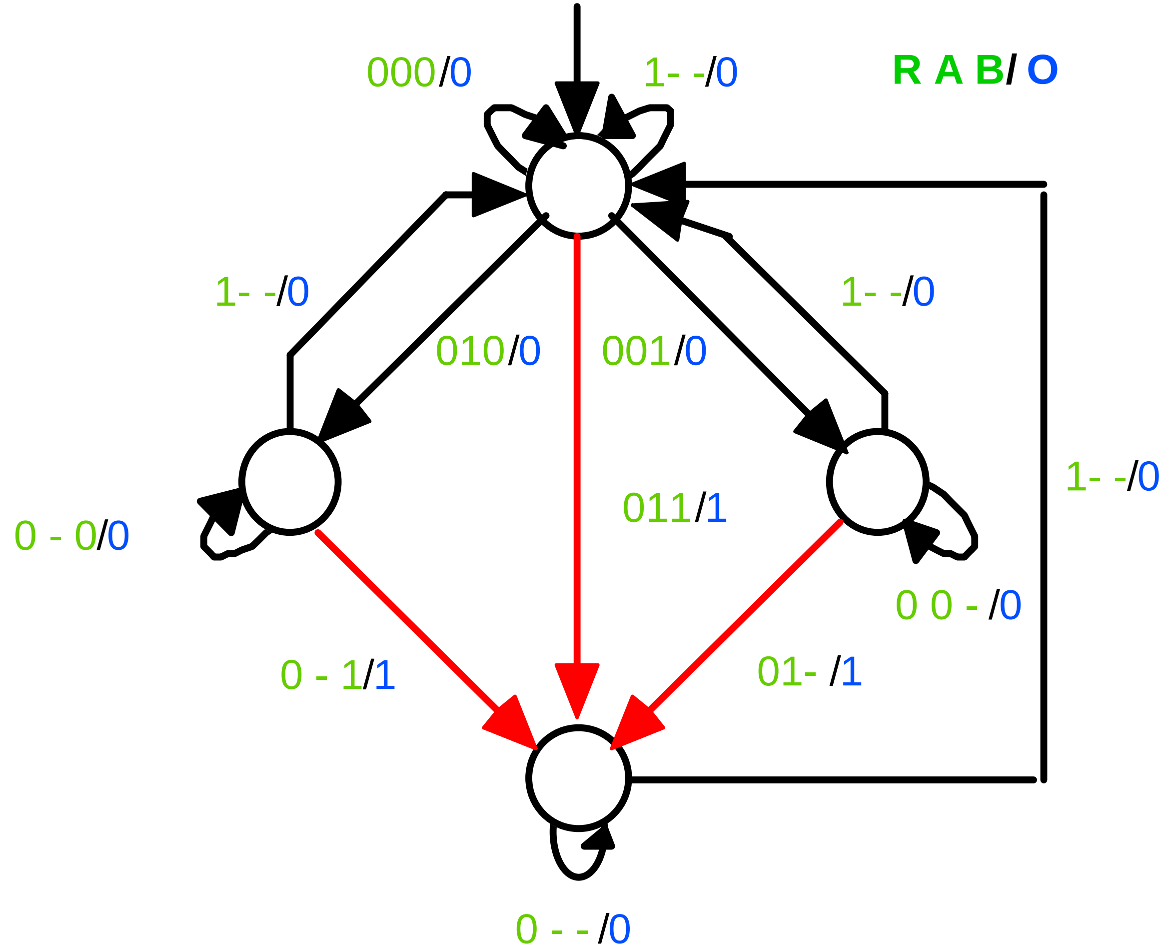
Introduction : ABRO

- **Input**: **A**, **B**, **R**
- **Output**: **O**
- **Behavior**:
  - **O** must be emitted as soon as both **A** and **B** have occurred since the last occurrence of **R**
  - **R** has priority over **A** and **B**. It cancels previous occurrences of **A** or **B**.

Tracing execution of ABRO

Instants	1	2	3	4	5	6	7	8	9	10
R	0	1	0	0	0	0	0	1	0	0
A	0	0	0	1	0	0	1	1	0	1
B	0	0	0	0	0	1	1	1	1	0
O	0	0	0	0	0	1	0	0	0	1

Mealy Machines: ABRO



Esterel : ABRO

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

Declarative part

Imperative part

Programming style

From this example

- A simple language:
  - A few constructs
  - Write Thing Once (replace replication by structure, wherever it is possible)
- Control
  - sequence (**;**), parallel composition (**||**)
  - preemption (**loop** … **each**)
  - communication (signal broadcast)
  - Explicit curly brackets (**{…}**) to group instructions

Valued Signals

- **Input**: **Revolution**, **Second**
- **Output**: **Rpm : integer**
- **Behavior**:

Pure signals

Valued signals

Whenever Second occurs, emit the number of rounds per minute (Rpm). That is, 60 times the number of rounds since the last occurrence of Second. Each round is signaled by an occurrence of Revolution

```
// Tachometer
module TACHO:
   input Revolution, Second;
   output Rpm: integer;
   loop
     var Rnb: integer := 0 in
       abort
         every Revolution do
           Rnb := Rnb + 1
         end every
       when Second do
         emit Rpm(Rnb*60)
       end abort
     end var
   end loop
end module
```

Comments, /* …*/

Valued signal

Variable and its scope

Preemptions

Information sharing (1)

```
module TAXI:
   input Km, Second, Go, Stop;
   output Charge:integer;
   host constant FixedCharge:integer;
   host constant  TUCharge:integer;
   host constant DUCharge:integer;
   await Go;
   // a taxi drive ...
end module
```

constants

No shared variables. Information sharing through Instantaneous broadcast of signals

Information sharing (2)

```
signal AddCharge: integer combine + in
   var Amount: integer:= FixedCharge in
      abort

every Km do                          every 30 Second do
   emit AddCharge(DUCharge)     ||      emit AddCharge(TUCharge)
end every                            end every

                 every AddCharge do
        ||          Amount := Amount + ?AddCharge
                 end every

      when Stop do
         emit Charge(Amount)
      end abort
    end var
end signal
```

Local signal

Signal value

Resulting value

Generic modules (1)

```
module TopMeter:
   generic type T;
   generic function Mult(integer,T):T;
   generic constant K:T;
   input Top, Get;
   output Got:T;
   // imperative code
end module
```

type
function

Abstract
for
Esterel

Generic modules (2)

```
// body
loop
   var Nb: integer:= 0 in

        abort
            every Top do
               Nb := Nb + 1
            end every
        when Get do
           emit Got( Mult(Nb,K) )
        end abort

   end var
end loop
```

Instantiation (1)

```
module GetSpeed:
 host function intMult(integer,integer):integer;
 host function floatMult(integer,float):float;
 input WheelRev, ShaftRev;
 input IgnitionON, IgnitionOFF, Second;
 output Rpm:integer, Speed:float;
 host constant TachoK:integer, SpeedK:float;
 // body
 end module
```

Instantiation (2)

```
await IgnitionON;
abort
   run TACHO/TopMeter [
        type integer/T;
        function intMult/Mult;
        signal ShaftRev/Top, Second/Get, Rpm/Got;
        constant TachoK/K ]
  ||
   run SPEED/TopMeter [
        type float/T;
        function floatMult/Mult;
        signal WheelRev/Top, Second/Get, Speed/Got;
        constant SpeedK/K ]
when IgnitionOFF
```

instantiations

Renaming:
actual/formal

Preemptions

- Upto now: strong abortive form

  `abort … when … do … end abort`

- Weak form of abortion

  `weak abort … when … do … end abort`

- Suspension

  `suspend … when …`

- Variant delayed/immediate

  `… when immediate guard-exp`

Runner (G. Berry)

```
module Runner:
  constant NumberOfLaps: unsigned=10;
  input Morning, Second, Meter;
  input Step, Lap;
  output Walk, Jump, Run;
  input relation Morning => Second,
                 Lap => Meter;
  // body
end module
```

```
every Morning do
   repeat NumberOfLaps times
      abort
         abort
            sustain Walk
         when 100 Meter;
         abort
            every Step do
               emit Jump
            end every
         when 15 Second;
         sustain Run
      when Lap
   end repeat
end every
```

A Tour of Esterel

Module

```
[main] module module-ident:
   declarations
   body
end [module]
```

Classical imperative statements

```
nothing
call proc-id (exp-list) //value or reference
stat1 ; stat2
stat1 || stat2
loop stat end loop
if bool-exp then stat1 else stat2 end if
trap trap-id in stat end trap
exit trap-id
var var-dcl in stat end var
var-id := expr
```

Primitive reactive statements

Signals

Local signal:

```
signal sig-dcl in stat end signal
```

Signal emission:

pure	valued
`emit sig-id`	`emit sig-id (expr)`

or

Testing the status of a signal:

```
if sig-id then
   stat1
else
   stat2
end if
```

Access to a signal’s value:

? *sig-id*

Primitive reactive statements

Signals

Signal emission (equational form):

```
pure   sig-id [<= bool-expr] [if bool-expr]
```

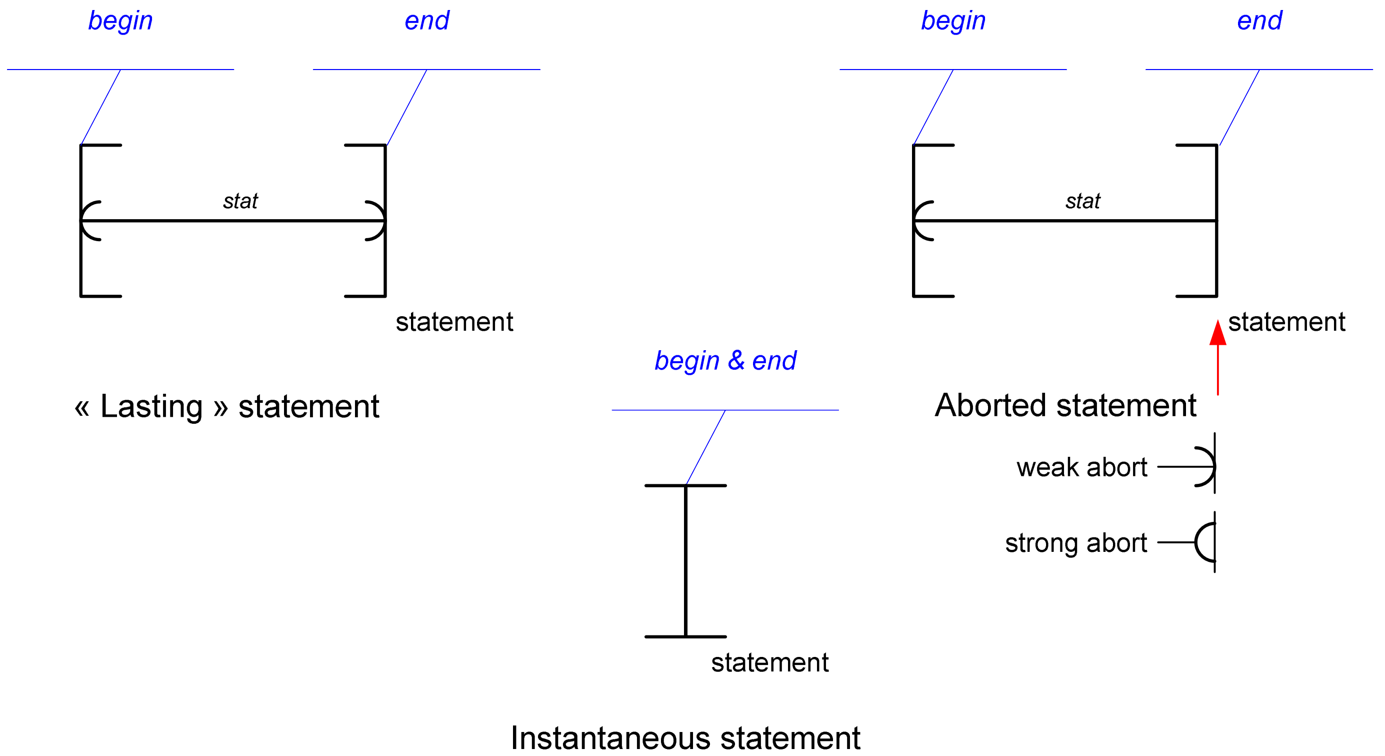
or

```
valued   ?sig-id <= expr [if bool-expr]
```

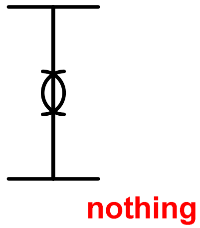
2 predefined signals: **tick** and **never**

Informal Semantics (1)

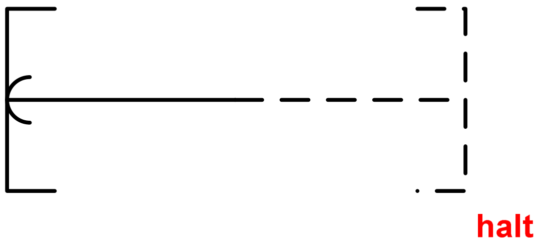
An Esterel statement has a temporal extension expressed as a logical time interval.
For convenience, we introduce a graphical notation.



Informal Semantics (2)

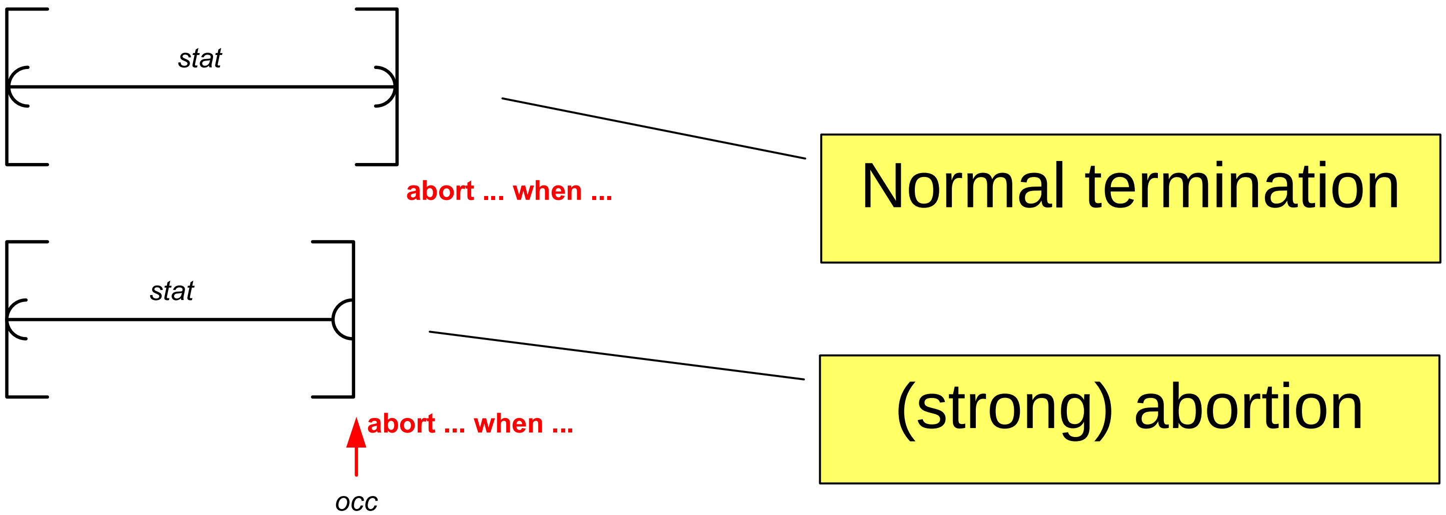


`nothing` // do nothing, instantaneously



`halt` // do nothing, forever

`abort` ***`stat`*** **`when`** ***`occ`*** // strong abortion



Charles André - University of Nice-Sophia Antipolis

Informal Semantics (3)

occ

In its simplest form, *occ* stands for **the presence of a signal**

```
abort
   …
when S
```

Complex occurrences are also available: use combination operators

`and or not xor`

```
abort
   …
when S and T
```

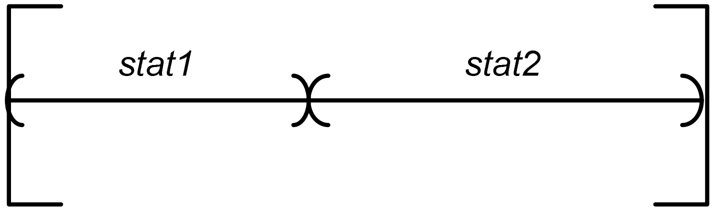
Complex guards with a repetition factor

`int-expr complex-occ`

```
abort
   …
when 5 S
```

Informal Semantics (4)

stat1* ; *stat2 // sequence

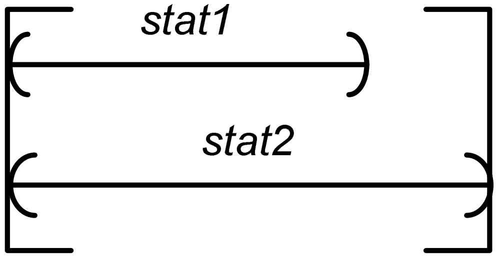


stat1 ; *stat2*

stat2 starts at the very same instant when *stat1* terminates

stat1* || *stat2 // parallel composition

Parallel composition terminates when all branches terminate

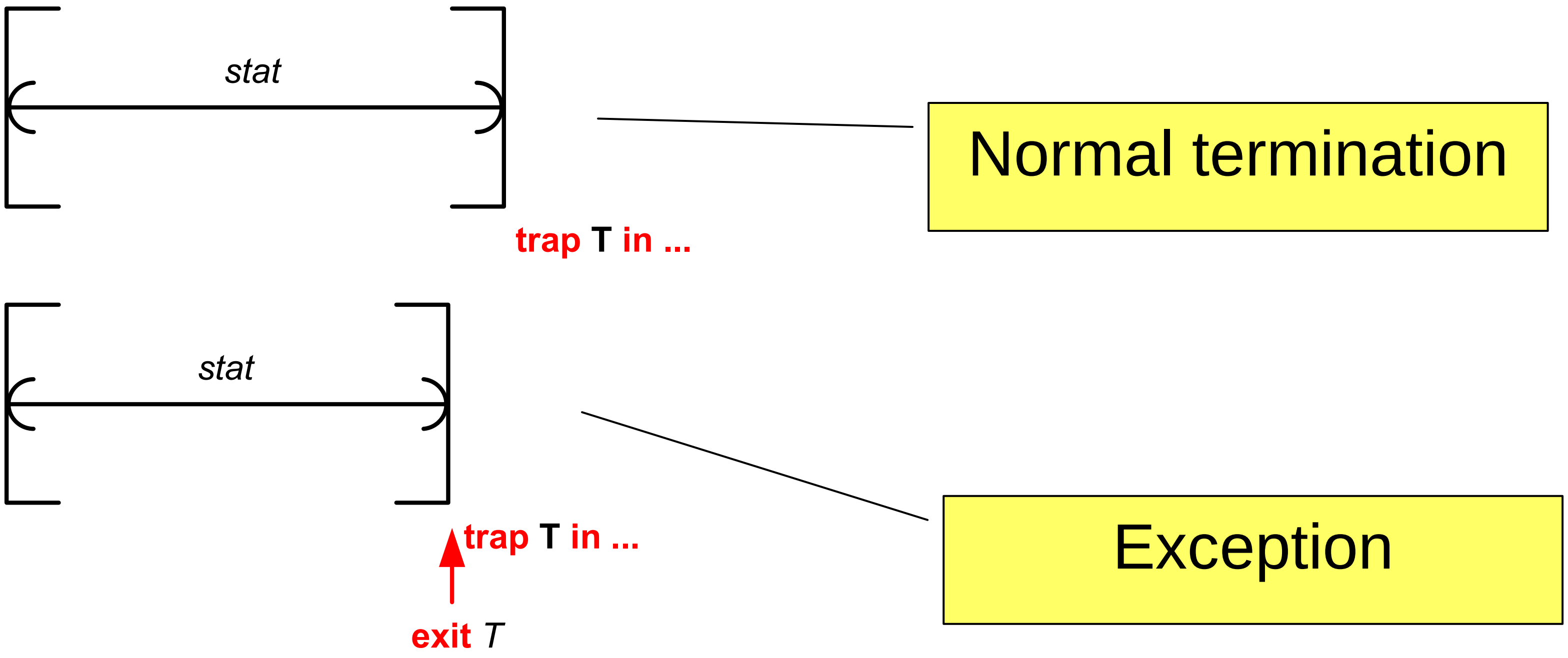


stat1 || *stat2*

Informal Semantics (5)

trap ***trap-id*** **in** ***stat*** **end trap**

exit ***trap-id***



Nested traps

```
trap T in
    trap U in
        …
        if A then exit T end if;
        …
        if B else exit U end if;
        …
    end trap ; // trap U
    … stat2
end trap ; // trap T
… stat3
```

Nested traps

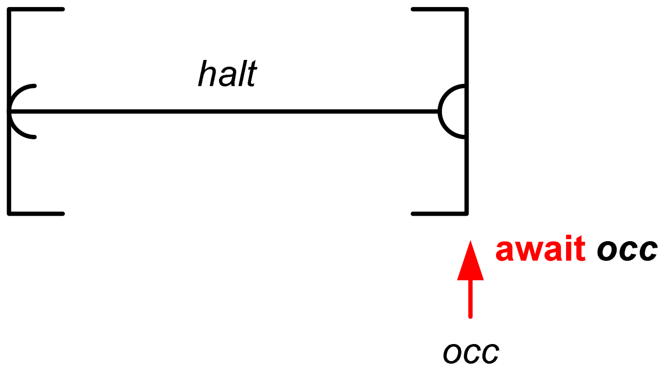
```
trap T in
    trap U in
        …
        if A then exit T end if;
        …
        if B else exit U end if;
        …
    end trap ; // trap U
    … stat2
end trap ; // trap T
… stat3
```

Nested traps

```
trap T in
    trap U in
        …
        if A then exit T end if;
        …
        if B else exit U end if;
        …
    end trap ; // trap U
    … stat2
end trap ; // trap T
… stat3
```

Derived reactive statements (1)

await *occ* ≡ **abort** *halt* **when** *occ*

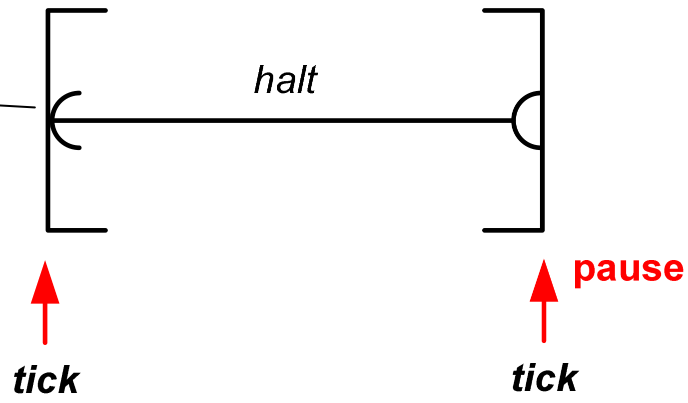


The simplest temporal statement

pause ≡ **await** *tick*

where *tick* is a predefined signal that represents the reaction clock of the reactive program

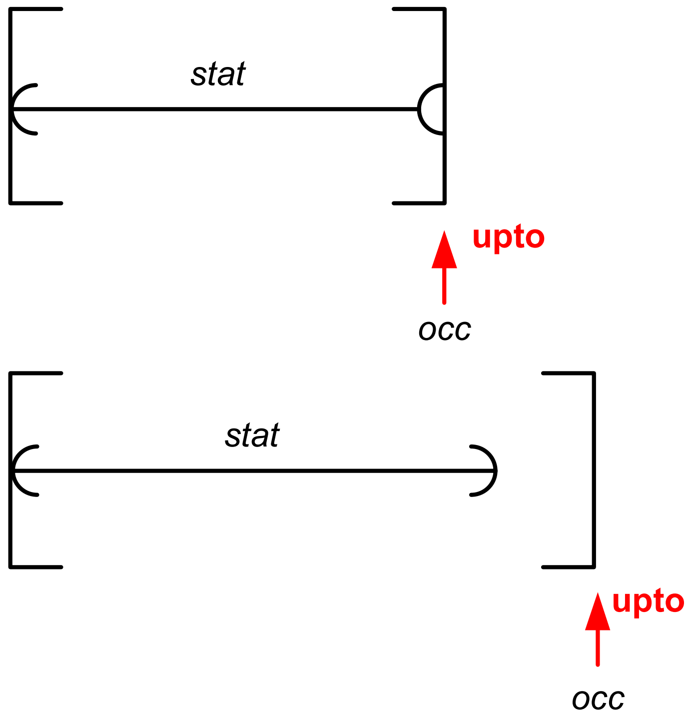
Wait for the next instant



Derived reactive statements (2)

`do` ***stat*** `upto` ***occ*** ≡ `abort` ***stat*** `; halt when` ***occ***

// a statement in use in early versions of the language



Mandatory occurrence of occ to terminate

Derived reactive statements (3)

```
abort
   stat
when occ
do
   handler
end abort
```

≡

```
trap T in
  abort
     stat ;
     exit T
  when occ ;
  handler
end trap
```

Exception clause

Derived reactive statements (4)

Trap with exception handler

```
trap trap-id in
   stat
handle trap-id do
   handler
end trap
```

Exception handler

Variant :
List of trap-ids
and several handlers

```
trap E in
   trap trap-id in
      stat ;
      exit E
   end trap ;
   handler
end trap
```

Temporal loops (1)

```
loop
    stat
each occ
```

≡

```
loop
    do stat upto occ
end loop
```

```
sustain S
```

≡

```
loop
    emit S
each tick
```

```
every occ do
    stat
end every
```

≡

```
await occ ;
loop
    stat
each occ
```

Temporal loops (2)

```
repeat [count-id:=] unsigned-expr times
   stat
end repeat
```

```
var v: unsigned := unsigned-expr in
   trap T in
      loop
         if v>0 then
            stat ; v := v - 1
         else
            exit T
         end if
      end loop
   end trap
end var
```

Advanced abort statements (1)

```
await
   case occ1 do
      stat1
   case occ2 do
      stat2
end await
```

Deterministic:

Ordered list of occurrences.
If several delays elapse at the same time, the first one in the list takes priority.

```
trap T1 in
   trap T2 in
      abort
         await occ2 ; exit T2
      when occ1 ; exit T1
   handle T2 do
      stat2
   end trap
handle T1 do
   stat1
end trap
```

Advanced abort statements (2)

```
abort
    stat
when
    case occ1 do stat1
    case occ2 do stat2
    …
    case occN do statN
end abort
```

Deterministic:

Ordered list of abortion cases.
If several *occs* elapse at the same time, the first one in the list takes priority.

Advanced abort statements (3)

```
weak abort            trap w in
    stat                stat ;
when occ      ≡         exit w
                       ||
                        await occ ;
                        exit w
                      end trap
```

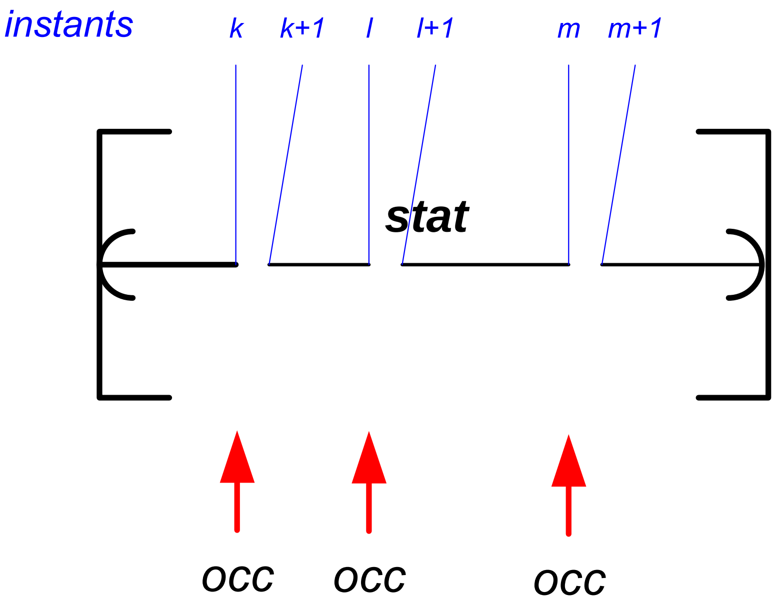
```
weak abort            trap T in
    stat                trap w in
when occ do               stat ;
    handler               exit T
end abort     ≡          ||
                          await occ ;
                          exit w
                        end trap;
                        handler
                      end trap
```

```
weak abort
    stat
when occ do
    handler
end abort
```

There exists also a version with multiple abortion cases

Suspension statement

```
suspend
    stat
when occ
```



When the suspend statement starts, *stat* is immediately started.

Then, at each instant:

- Either, *occ* is present, then *stat* is “frozen”
- Or, *occ* is absent, and then *stat* normally executes

suspend *stat* when *occ*

Test statements

Simple branching :

```
if bool-exp
then
   stat1
else
   stat2
end if
```

Multiple branching tests:

```
if
   case bool-exp1 do stat1
   ...
   case bool-expN do statN
   [default do stat]
end if
```

Module Instantiation

`run`

- Reuse of a module
- Possible renamings of
  - `types, signals,`
  - `constants, functions,`
  - `procedures, tasks,`
  - `module name`

Renaming syntax

`category new-id/old-id , … ;`

Tasks

Tasks are intended to **« lasting » activity** programming, while procedures and functions are supposed to be instantaneous.

A task runs asynchronously w.r.t. the synchronous code.

The synchronous code launches a task with the **`exec`** statement.

The task signals its completion by a special signal, called a **`return`** signal.

No longer supported in Esterel v7

Declarations :

```
task task-id(type-ref-list)(type-val-list);
return sig-id : type;
```

Invocation :

```
exec task-id(param-ref-list)(param-val-list)
     return sig-id
```

Incorrect programs (1)

An Esterel program must be
Reactive and **Deterministic.**

Due to instantaneous broadcast of signals; this is not the case for all syntactically correct Esterel programs.

In what follows, we present some instances of incorrect programs. We (informally) explain why they are incorrect.
The full characterization of correct programs relies on the mathematical semantics of the language and will be studied later.

Incorrect programs (2)

Instantaneous loop

```
~~loop~~
~~    emit S~~
~~end loop~~
```

```
loop
    emit S ;
    pause
end loop
```

Incorrect programs (3)

Causality cycle

Output signals can be « fed back » to inputs.
Non deterministic and/or non reactive behavior may result from this instantaneous feedback.

Non reactive programs

```
signal S in
  if S then
    nothing
  else
    emit S
  end if
end signal
```

```
signal S:integer
     combine + in
  ...
  emit ?S <= ?S+1;
  ...
end signal
```

Incorrect programs (4)

Non deterministic programs

```
signal S in
  if S then
    emit S
  else
    nothing
  end if
end signal
```

2 solutions :
S absent or
S present

```
signal S:integer in
    emit ?S <= ?S
end signal
```

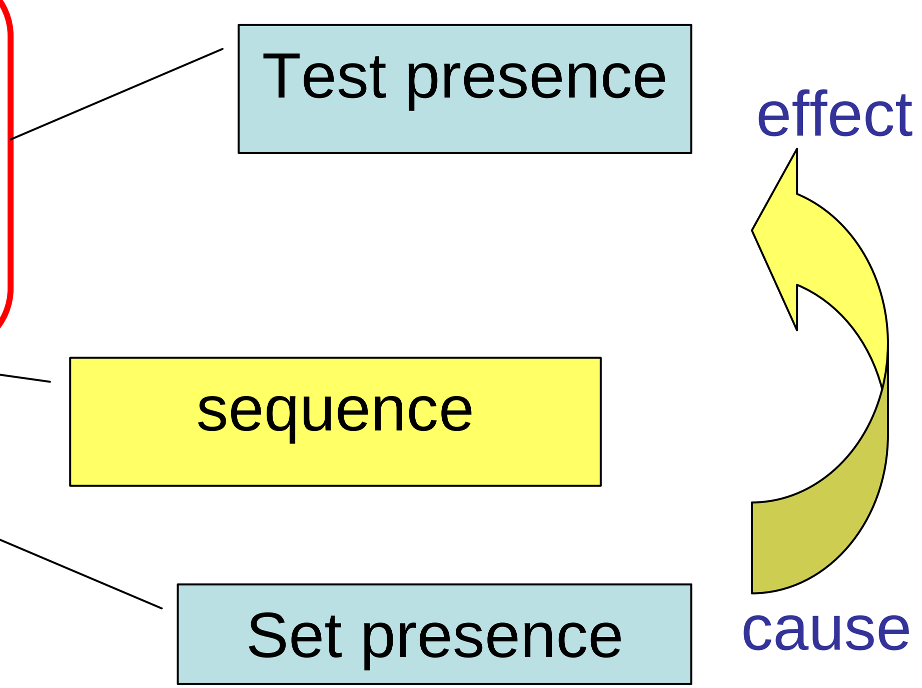
Infinitely many solutions

Incorrect programs (5)

Non constructive programs

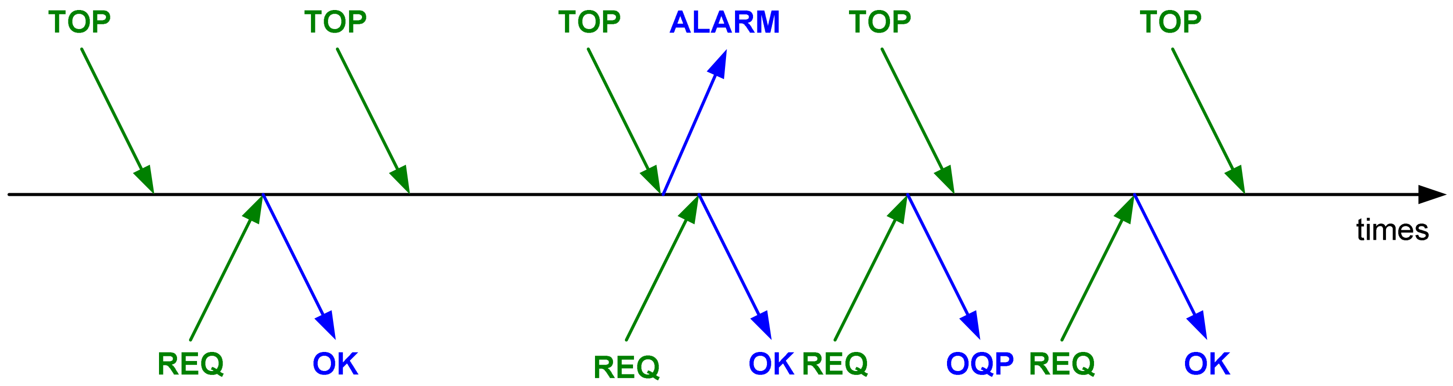
(See the Constructive Semantics of Esterel)

```
signal S in
  if S then
    emit S
  else
    emit S
  end if ;
  emit S
end signal
```



Examples

Application : TOP/REQ



TOP/REQ: program (2)

```
module TOPREQ:
   input TOP, REQ;
   output OK, OQP, ALARM;
   input relation TOP # REQ;
   signal DONE, QUERY in

      // TOP management
      ||
      // REQ memorize

   end signal
end module
```

TOP/REQ: program (2)

```
// TOP management
every TOP do
    emit QUERY;
    if DONE else
        emit ALARM
    end if
end every
```

```
// REQ memorize
loop
    await REQ;
    emit OK;
    abort
      every REQ do
          emit OQP
      end every
    when QUERY do
        emit DONE
    end abort
end loop
```

TOP/REQ: program (2)

```
% TOP management
every TOP do
    emit QUERY;
    if DONE else
       emit ALARM
    end if
end every
```

```
% REQ memorize
loop
   await REQ;
   emit OK;
   abort
    every REQ do
       emit OQP
    end every
   when QUERY do
      emit DONE
   end abort
end loop
```

TOP/REQ: program (2)

```
// TOP management
every TOP do
    emit QUERY;
    if DONE else
        emit ALARM
    end if
end every
```

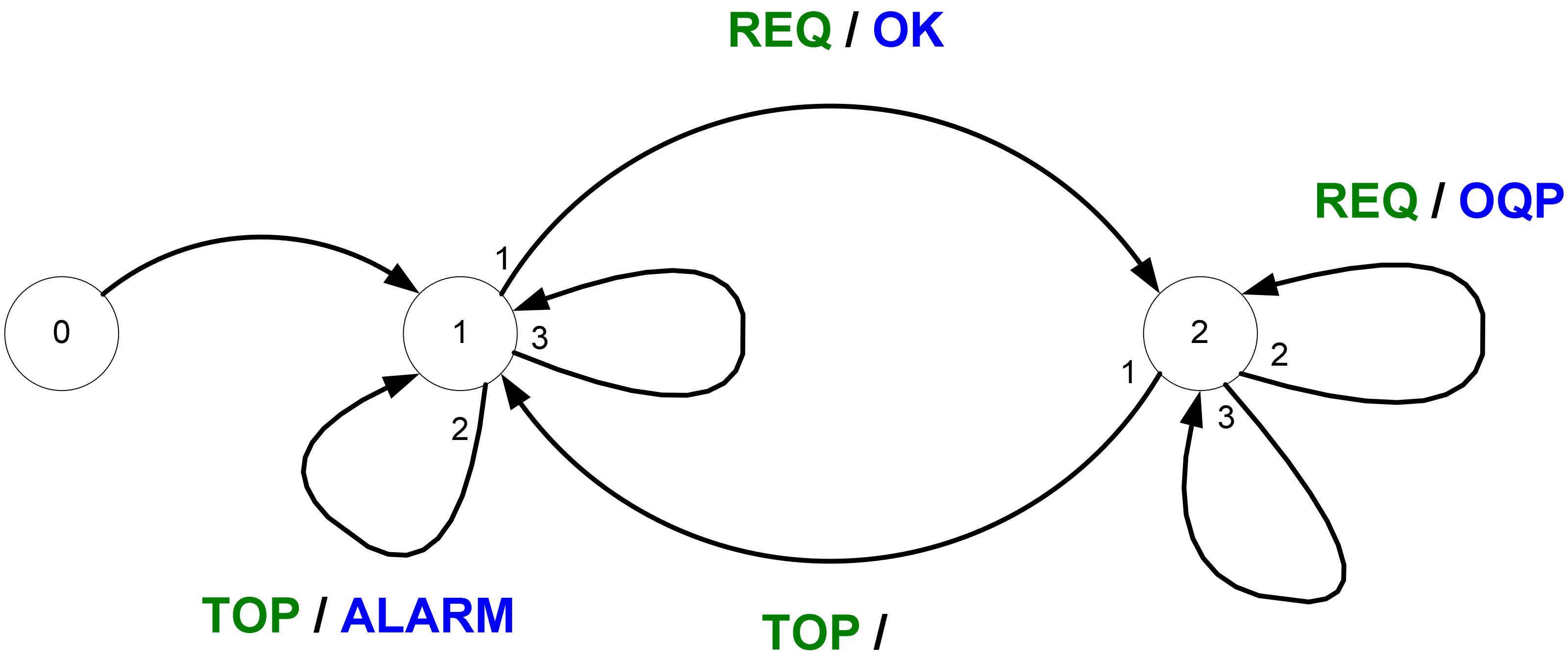
```
// REQ memorize
loop
   await REQ;
   emit OK;
   abort
    every REQ do
       emit OQP
    end every
   when QUERY do
       emit DONE
   end abort
end loop
```

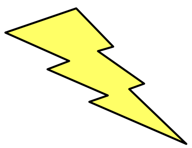
TOP/REQ: program (3)

```
// TOP management
// Equational form
every TOP do
    emit {
        QUERY,
        ALARM if not DONE
    }
end every
```

```
// REQ memorize
loop
   await REQ;
   emit OK;
   abort
     every REQ do
        emit OQP
     end every
   when QUERY do
      emit DONE
   end abort
end loop
```

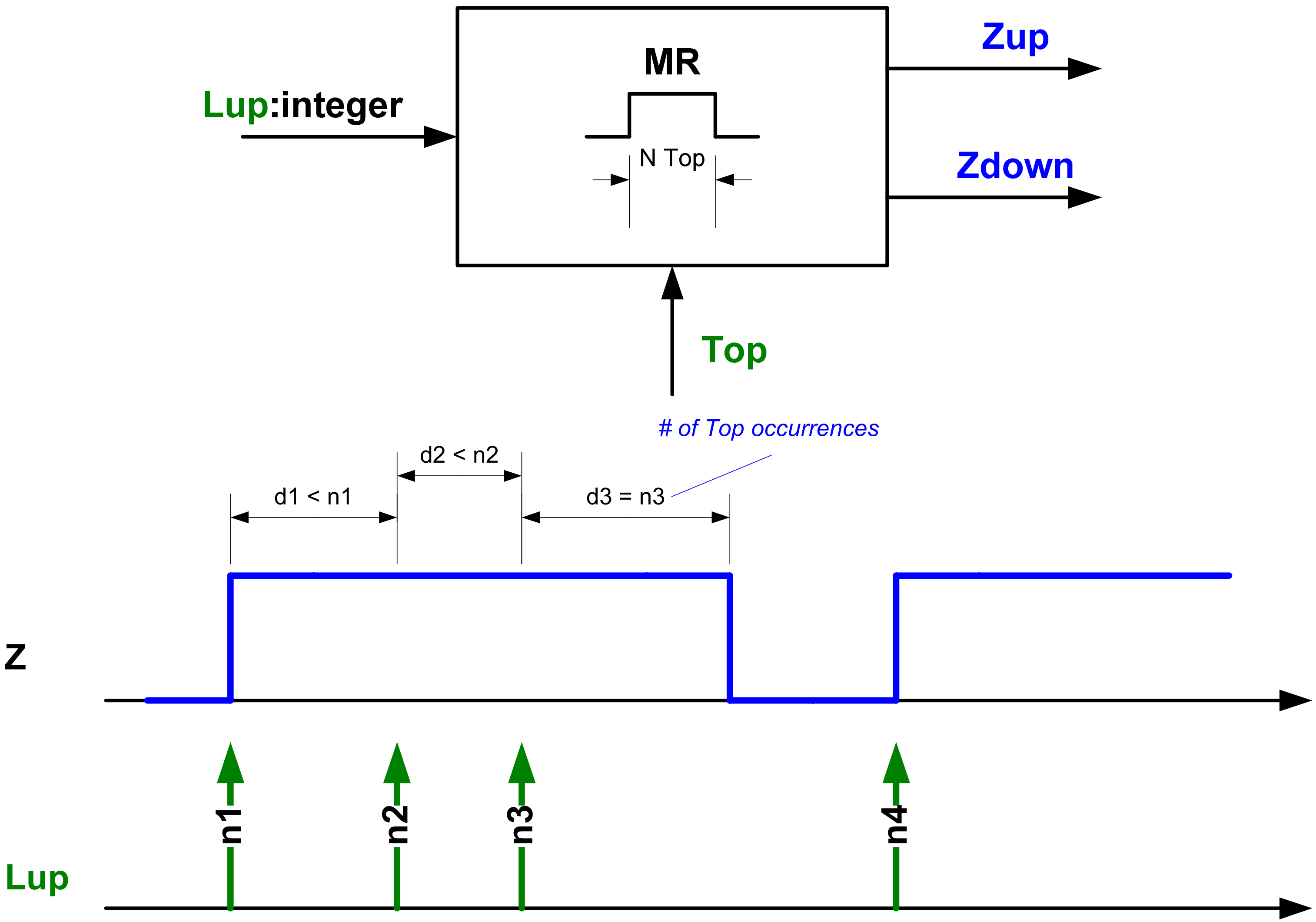
TOP/REQ: Mealy Machine



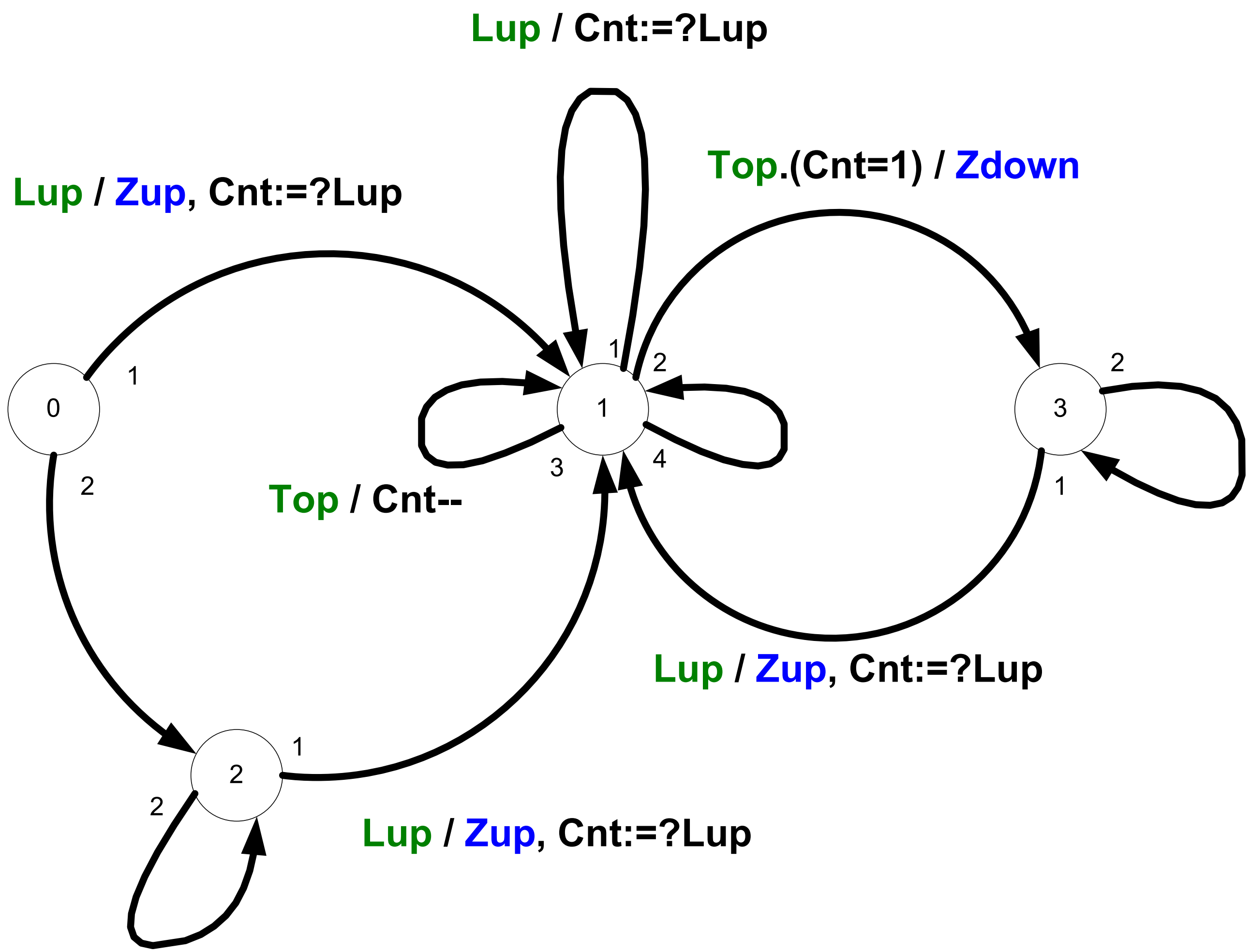


Priority over transitions: 1 > 2 > 3 > …

Application: monostable



Monostable: Mealy Machine



Other extensions

- Operator **pre** (previous)

  pre(S) // presence status

  pre(?S) // value

- **Finalize**

  finalize

  p

  with

  q

  end finalize

pre

```
every not pre(S) and S do
     emit risingEdgeS
end every

input Top;
output TopNb: integer init 0;
every Top do
     emit ?TopNb <= pre(?TopNb)+1
end every
```

Translation of pre(S)

```
trap T in
   signal S, preS in
      p;
      exit T
   ||
      loop
         if S then
            pause;
            emit preS
         else
            pause
         end if
      end loop
   end signal
end trap
```

Translation of pre(?S)

```
trap T in
   signal S: Type init v, preS: Type init v
in
      p; // a process
      exit T
   ||
      loop
         if S then
            var preval := ?S in
               pause;
               emit preS(preval)
            end var
         else
            pause
         end if
      end loop
   end signal
end trap
```

pre + suspend

```
module SuspPre:
input Top,A;
output rEdge;
suspend
   signal S in
      every A do emit S end every
      ||
      every S and not pre(S) do
         emit rEdge
      end every
   end signal
when immediate not Top
end module
```

Presence status at the previous instant when not suspended

Last wills

```
abort
      p
      ||
      abort
            q
      when B do
            emit b
      end abort
when A do
      emit a
end abort
```

Last will of q

Last will of p

When A is present then

is emitted a

but b **is not**

Finalize (1)

q must be instantaneous

finalize ***p*** **with** ***q*** **end finalize**

The finalizer code *q* is executed when :

1. The body (p) terminates,
2. The body raises an exception enclosing the finalize instruction
3. Some concurrent statement exits a trap that contains the finalize statement
4. The finalize instruction is preempted by an enclosing abortion

- When several finalizers are enclosed within each other and triggered, their finalizer codes are executed in the innermost to outermost finalizer order.
- Each finalizer is executed only once.

Finalize (2)

```
var X := 0 : integer in
   abort
      finalize
         abort
            finalize
               halt
            with
               X := 2;
               emit O1(X)
            end finalize
         when J do
            X := X*3;
            emit O2(X)
         end abort
      with
         X := X + 5;
         emit O3(X)
      end finalize
   when I do
      X := X * 7;
      emit O4(X)
   end abort
end var
```

if J is present

finalize: X := 2, O1(2)

handler: X := 6, O2(6)

Termination of the body:
finalize: X := 11, O3(11)

Finalize (3)

```
var X := 0 : integer in
   abort
      finalize
         abort
            finalize
               halt
            with
               X := 2;
               emit O1(X)
            end finalize
         when J do
            X := X*3;
            emit O2(X)
         end abort
      with
         X := X + 5;
         emit O3(X)
      end finalize
   when I do
      X := X * 7;
      emit O4(X)
   end abort
end var
```

If I is present

finalize: X := 2, O1(2)

finalize: X := 7, O3(7)

handler:X := 49, O4(49)