The Esterel language

Pascal Raymond, Verimag-CNRS

Introduction

- The first synchronous language (early 80’s)
- Gérard Berry and his team

  (École des Mines de Paris / INRIA Sophia-Antipolis)
- Imperative, sequential style (i.e. structure reflects control flow)
- Communication by synchronous broadcasting of *signal*

Communication by signal broadcasting

- Elementary information: either present or absent
- A signal can be pure (just here or not),
  or valued (either absent, or present with a value)

Elementary behaviours

- Related to signal: emit, wait, test a signal

Composition statements

- run several behaviours in sequence,
- run several behaviours concurrently,
- repeat a behaviour,
- interrupt a behavior etc.

Example: a speedometer

Specification

- **receives signals *second* and *centimeter***
- **each *second*, emit a signal *speed* carrying the number of *centimeters* received since the last *second***

Hints on the implementation

- **Use a *classical* variable `cpt` to count the occurences of *centimeter***

```
module SPEEDOMETER:
input sec, cm;      % pure signals
output speed : integer; % valued signal
```

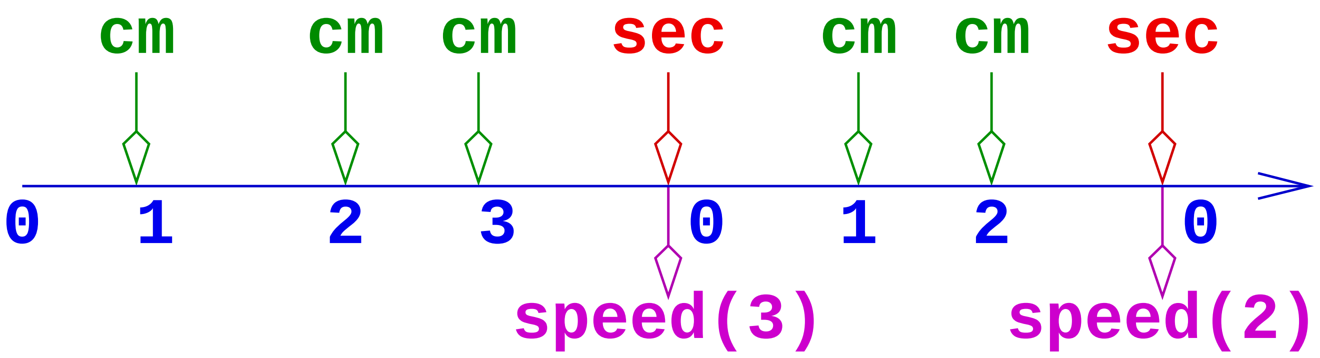
```
module SPEEDOMETER:
input sec, cm;          % pure signals
output speed : integer; % valued signal
 loop  % infinite behaviour









 end loop.
```

```
module SPEEDOMETER:
input sec, cm;      % pure signals
output speed : integer; % valued signal
 loop % infinite behaviour
  var cpt := 0 : integer in % internal variable



  end var
 end loop.
```

```
module SPEEDOMETER:
input sec, cm;      % pure signals
output speed : integer; % valued signal
 loop % infinite behaviour
  var cpt := 0 : integer in % internal variable

      loop             % normal behaviour:
        await cm ;     % each cm,
        cpt := cpt + 1 % increment cpt
      end loop



  end var
 end loop.
```

```
module SPEEDOMETER:
input sec, cm;      % pure signals
output speed : integer; % valued signal
 loop % infinite behaviour
  var cpt := 0 : integer in % internal variable
    abort    % terminate the folowing behavior:
      loop             % normal behaviour:
        await cm ;     % each cm,
        cpt := cpt + 1 % increment cpt
      end loop
    when sec do       % ... when sec arrives,
      emit vit(cpt)   % emit the value of cpt on signal speed
    end abort
  end var
 end loop.
```

Temporal behaviour



Why is it synchronous?

- Almost all statements are instantaneous:
  - ⋆ sequence, assignment, emission ...
- Exceptions are:
  - ⋆ `await cm`: waits for a strictly future occurence of `cm`
  - ⋆ `abort ... when sec`: terminates on the strictly future occurence of `sec`

Conclusion of the example

- Imperative language “relativelly” classical ...
- but with a synchronous semantics
- Lots of constructs (variables, signals, interrupts ...)
- Semantics a little bit complex (at least unusual)

⇒ Let’s study in detail a sub-language (pure Esterel):

- only pure signals,
- no variable and assignments,
- only a few statements

Statements related to signals

Await

- `await S`
- halts as soon as it takes control, will terminate (and pass the control in sequence) on the next occurrence of S

Emission

- `emit S`
- emits S and terminates immediately

Test

- `present S then c1 else c2 end`
- if `S` is present, behaves as `c1`, otherwise behaves as `c2`
- Degenerated forms:
  - ⋆ `present S then c1 end`
  - ⋆ `present S else c2 end`

Composition of behaviours

Sequence

- `c1 ; c2`
- passes immediately the control to `c1`,
- if and when `c1` terminates, passes immediately the control to `c2`,
- terminates if and when `c2` terminates

Composition of behaviours

Sequence

- `c1 ; c2`
- passes **immediately** the control to `c1`,
- if and when `c1` terminates, passes **immediately** the control to `c2`,
- terminates if and when `c2` terminates

Unbounded loop

- `loop c end`
- recursively equivalent to “`c ; loop c end`”
- never terminates

Parallelism

- `[ c1 || c2 ]`
- passes **immediately** the control to both `c1` and `c2`,
- terminates if and when the last of them terminates

Remark:

- Several concurrent behaviours may emit the same signal
- For a pure signal:
  - ⋆ no problem, the signal is present if emited at least once
- For a valued signal:
  - ⋆ values are combined by an associative, commutative operator
  - ⋆ Typically: `or` for Booleans, `+` for integers ...
  - ⋆ quite dangerous feature!

Synchronous semantics

How to give events a date?

- **There exists an implicit basic discrete clock**
- **Any event takes place at some instant of this clock**
- **In particular, input signals are occurring on the basic clock**

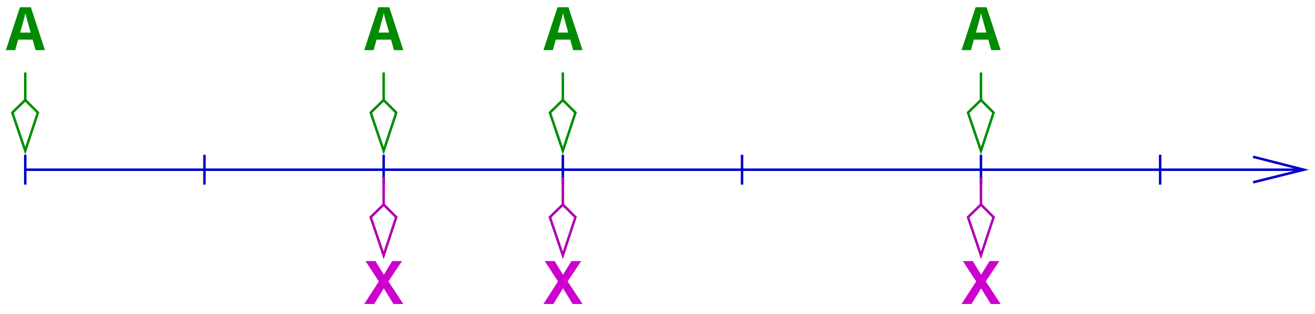
Synchronous semantics

How to give events a date?

- There exists an implicit basic discrete clock
- Any event takes place at some instant of this clock
- In particular, input signals are occurring on the basic clock

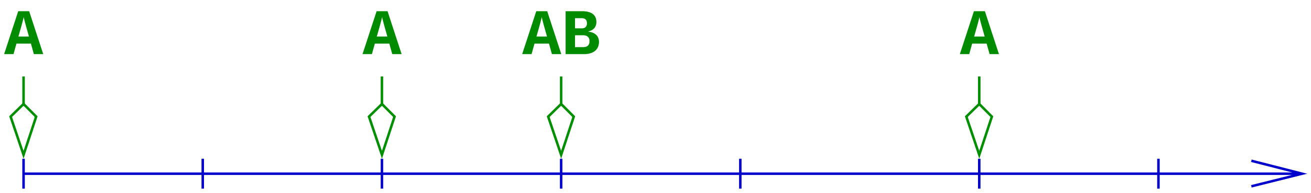
A simple example

- `loop await A ; emit X end`



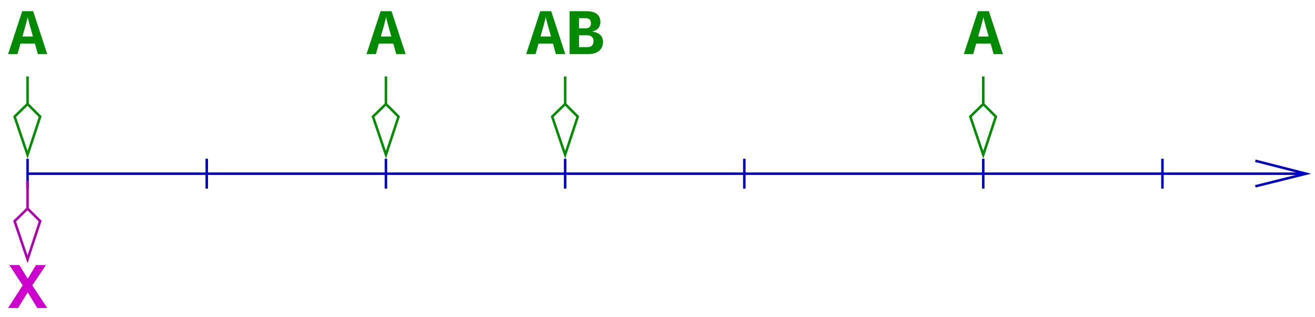
Another example

```
module Foo:
input A,B;
output X,Y,Z;
loop
   emit X;
   await A;
   emit Y;
   present B then emit Z end
end loop.
```



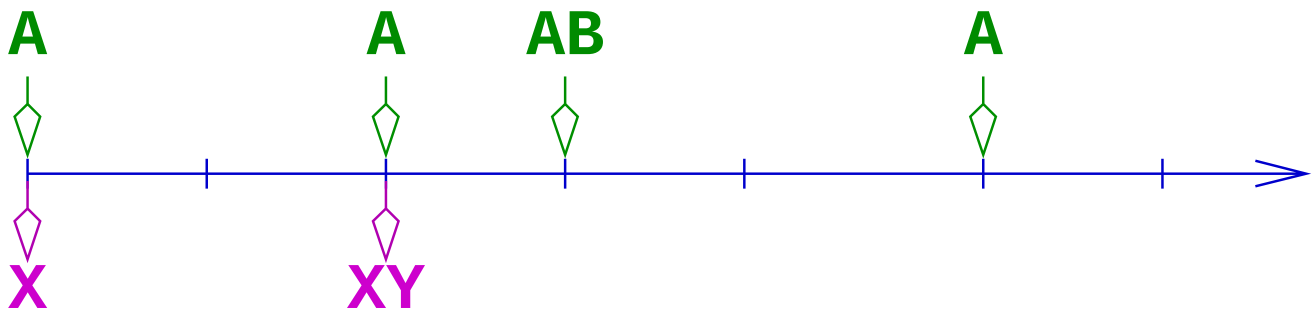
Another example

```
module Foo:
input A,B;
output X,Y,Z;
loop
   emit X;
   await A;
   emit Y;
   present B then emit Z end
end loop.
```



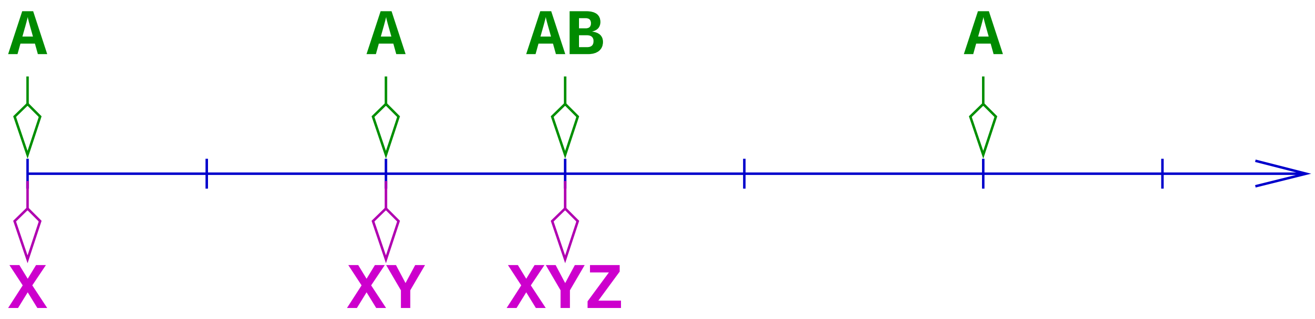
Another example

```
module Foo:
input A,B;
output X,Y,Z;
loop
   emit X;
   await A;
   emit Y;
   present B then emit Z end
end loop.
```



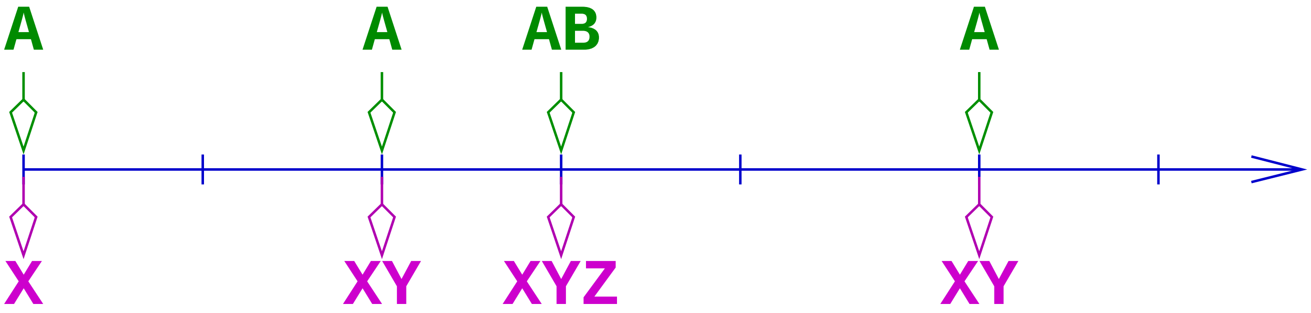
Another example

```
module Foo:
input A,B;
output X,Y,Z;
loop
   emit X;
   await A;
   emit Y;
   present B then emit Z end
end loop.
```



Another example

```
module Foo:
input A,B;
output X,Y,Z;
loop
   emit X;
   await A;
   emit Y;
   present B then emit Z end
end loop.
```



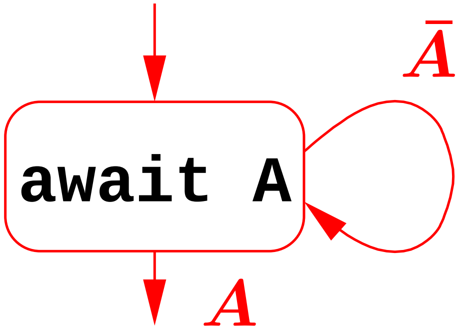
Esterel and Mealy machines

Principle

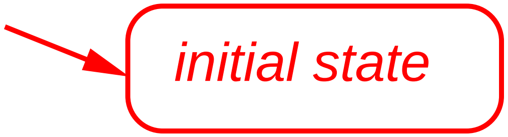
- **An Esterel program is a finite automaton**
- **More precisely, a Mealy machine (events are occuring on transitions)**

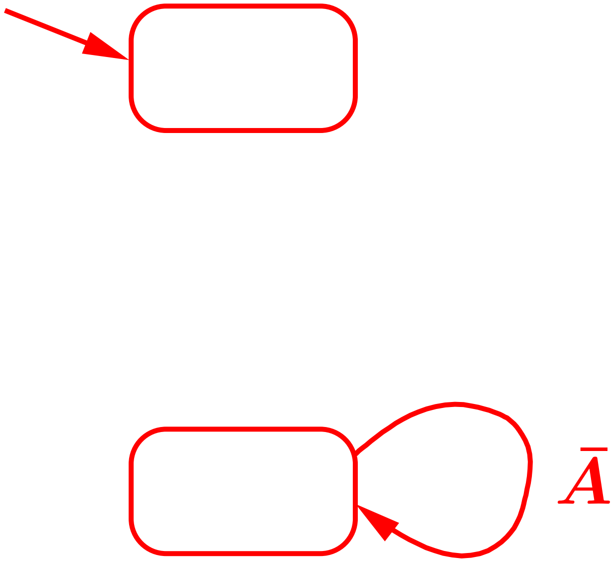
Control points (states)

- **At the very beginning (initial state)**
- **On each statement that *takes time***
- **Transition: condition/emission for going from one state to another**

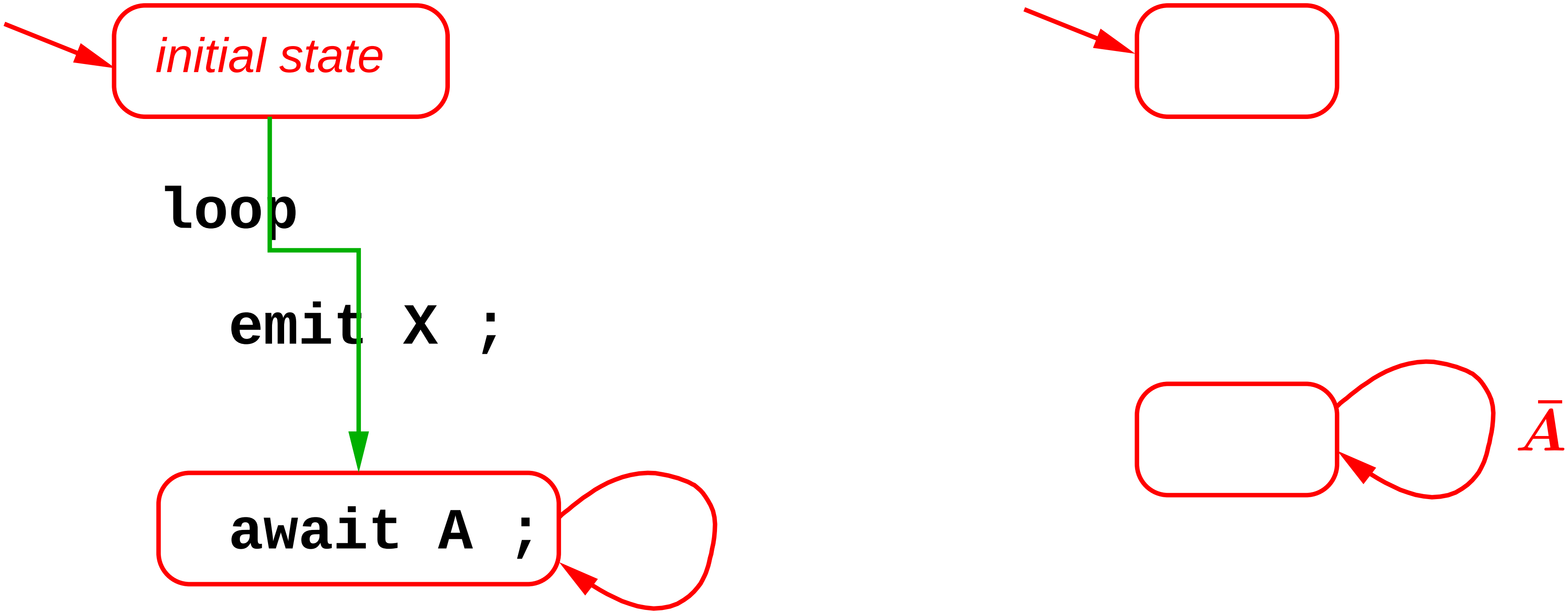


```
loop
  emit X ;

  await A ;
  emit Y ;
  present B then emit Z end
end.
```

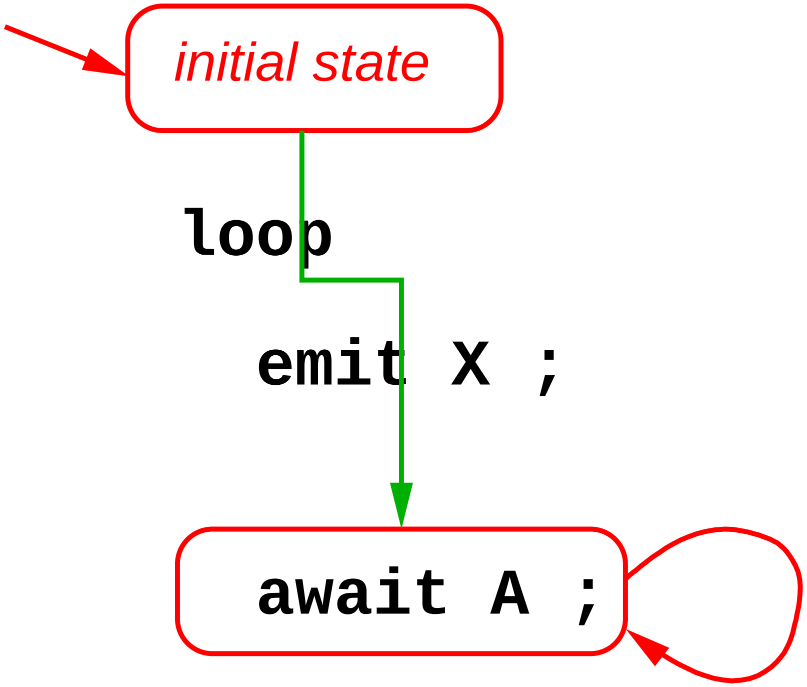




```
loop
  emit X ;

  await A ;

  emit Y ;
  present B then emit Z end
end.
```

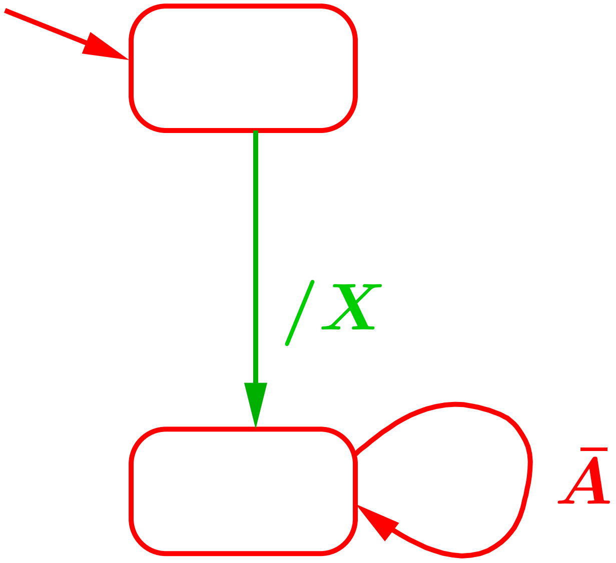


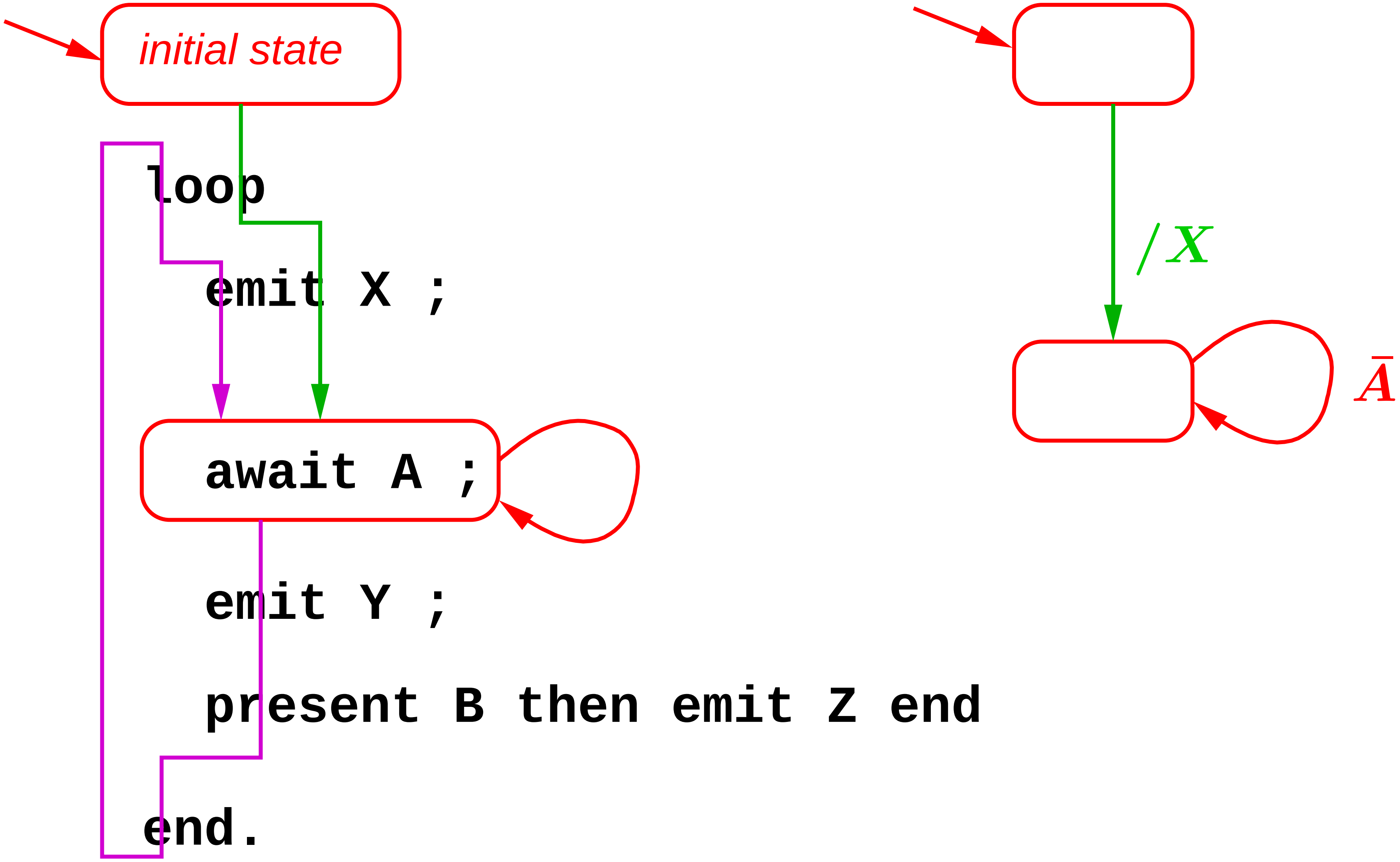
```
loop
   emit X ;

   await A ;

   emit Y ;
   present B then emit Z end
end.
```



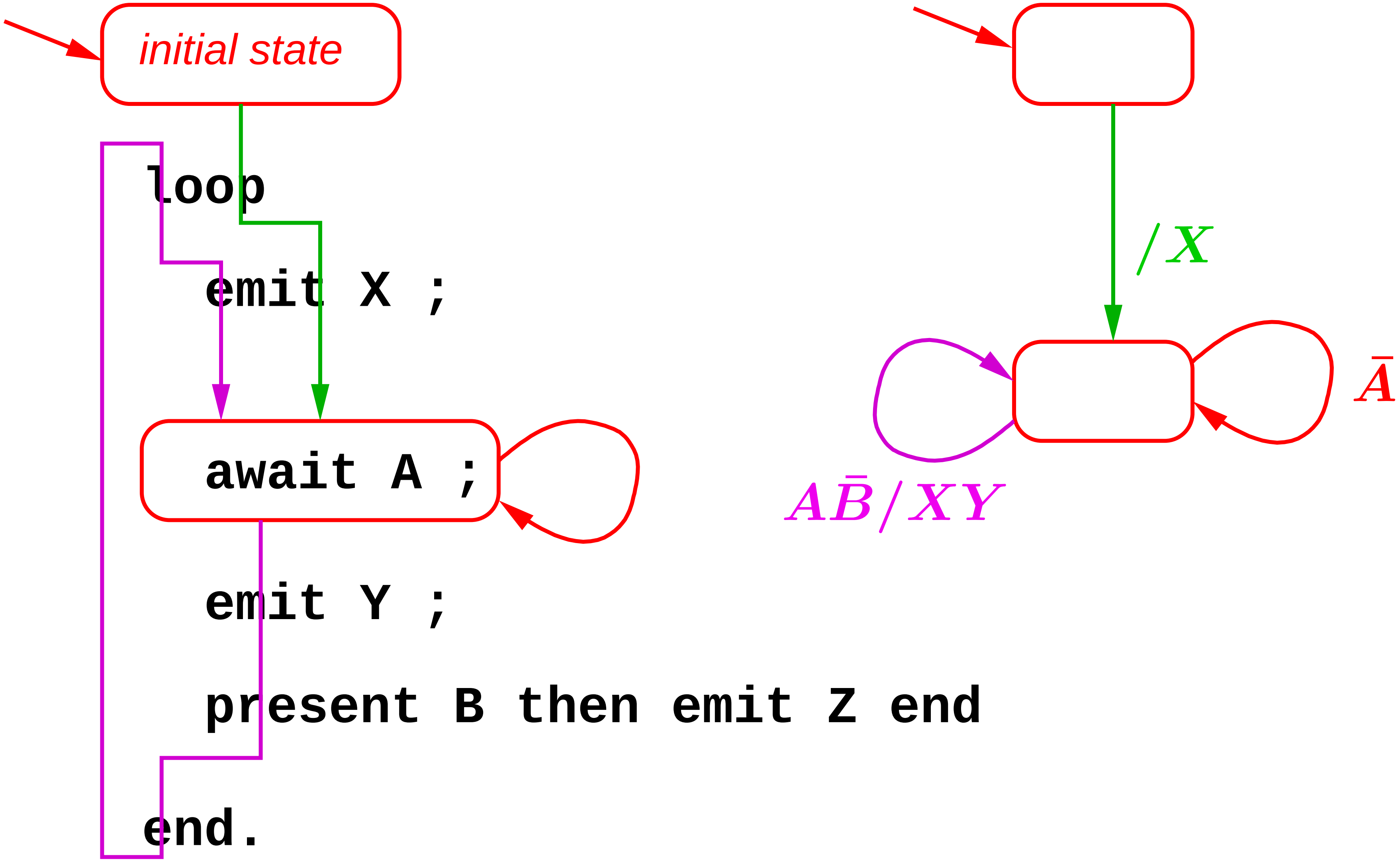
initial state

```
loop
  emit X ;
  await A ;
  emit Y ;
  present B then emit Z end
end.
```

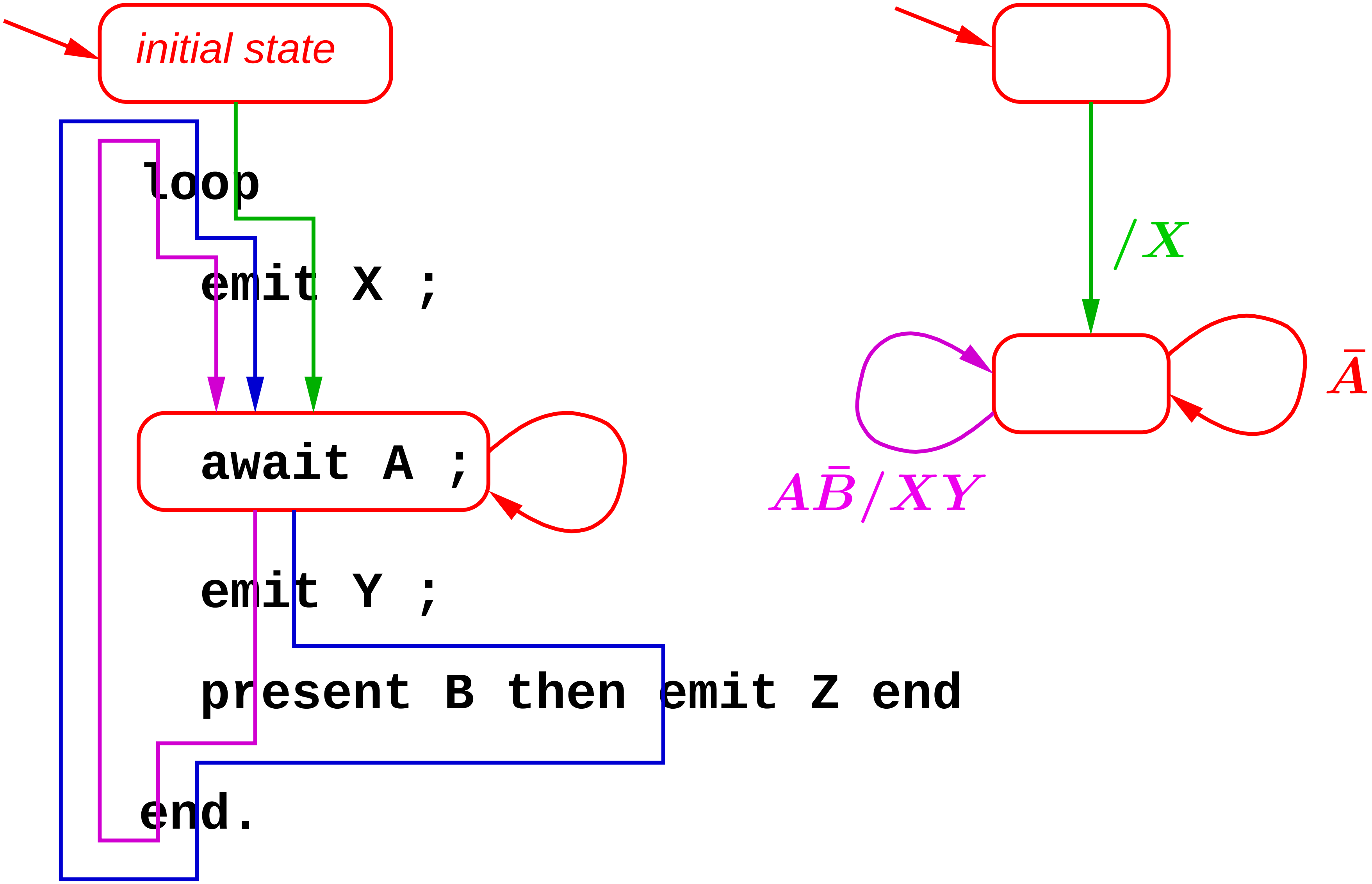




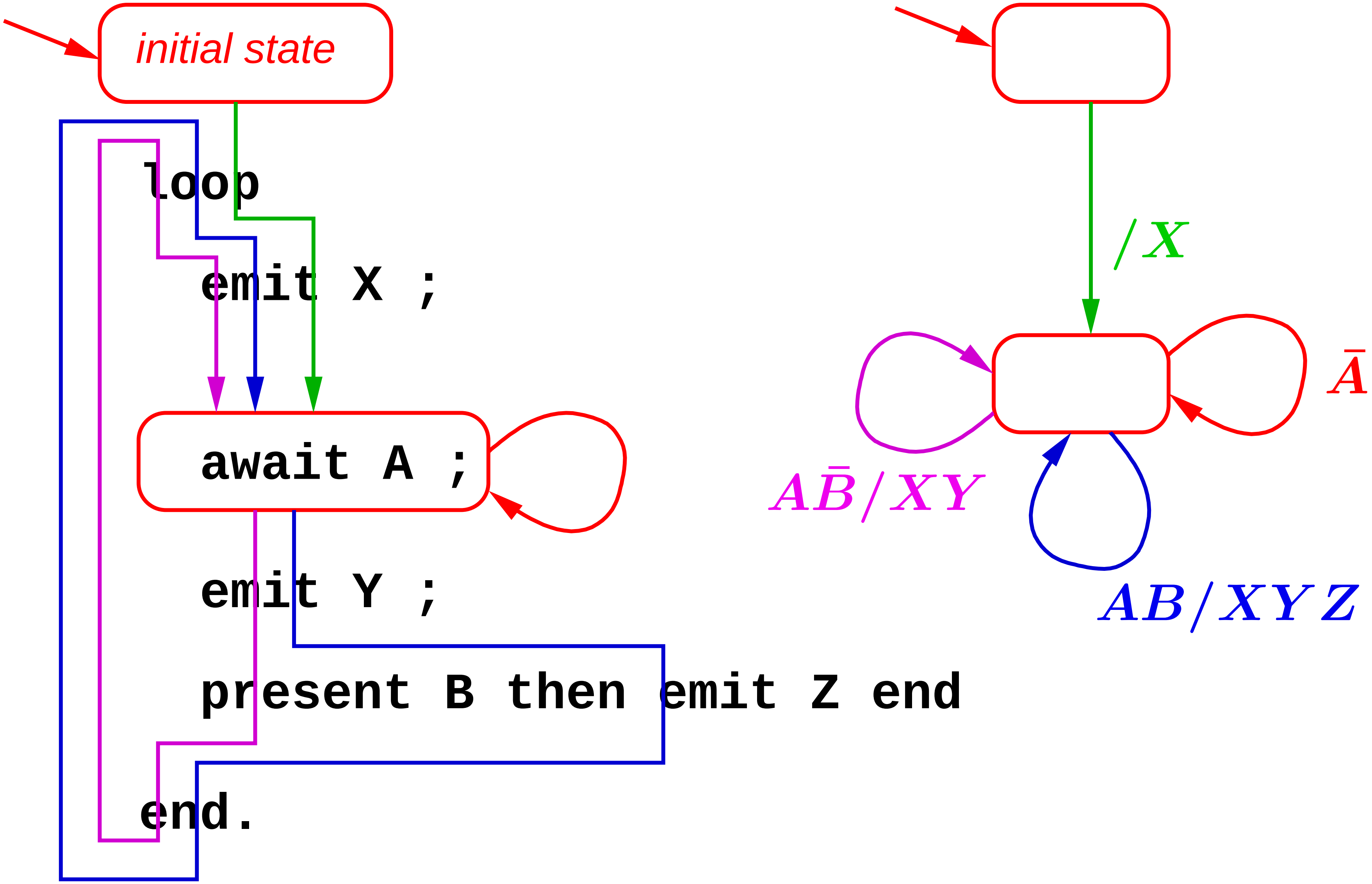
```
loop
  emit X ;
  await A ;
  emit Y ;
  present B then emit Z end
end.
```



```
loop
  emit X ;
  await A ;
  emit Y ;
  present B then emit Z end
end.
```



```
loop
   emit X ;
   await A ;
   emit Y ;
   present B then emit Z end
end.
```

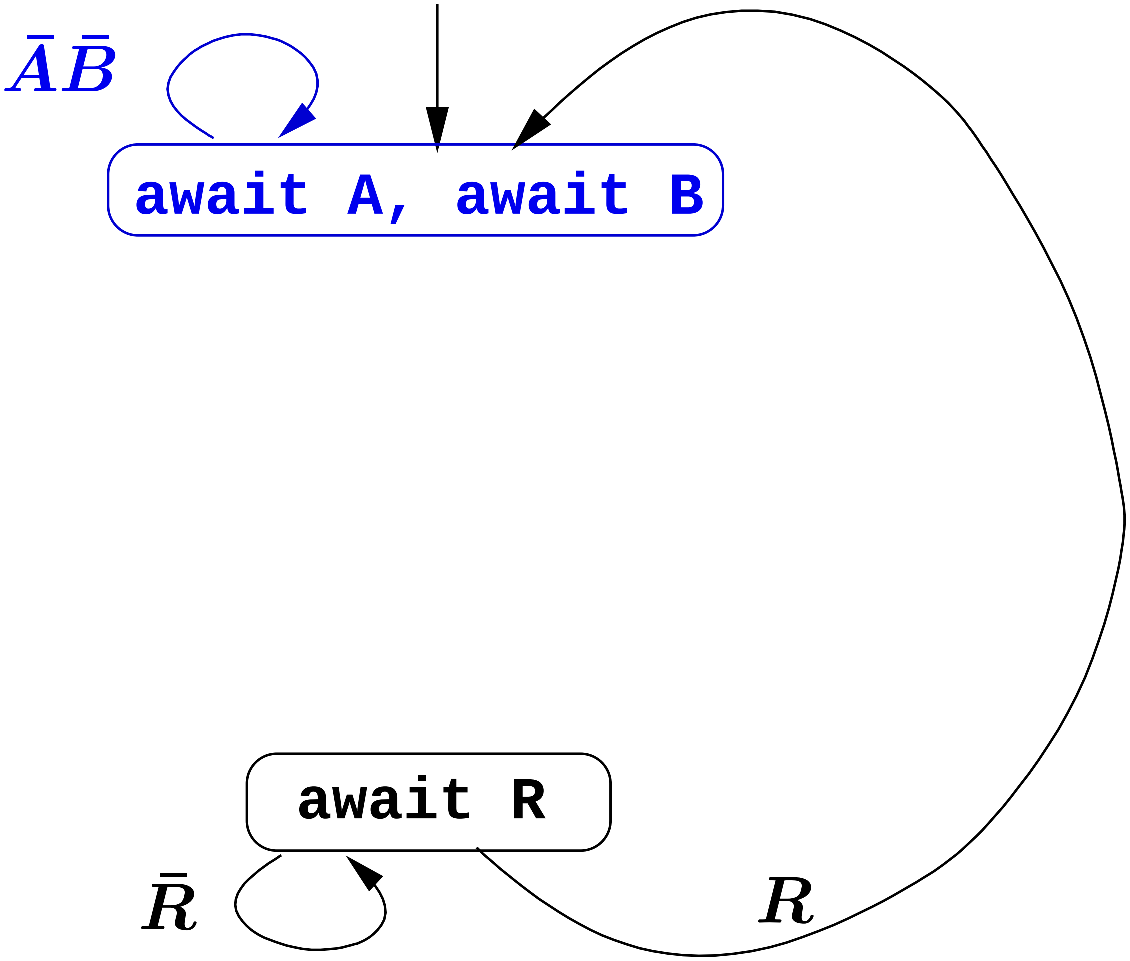


```
loop
   emit X ;
   await A ;
   emit Y ;
   present B then emit Z end
end.
```

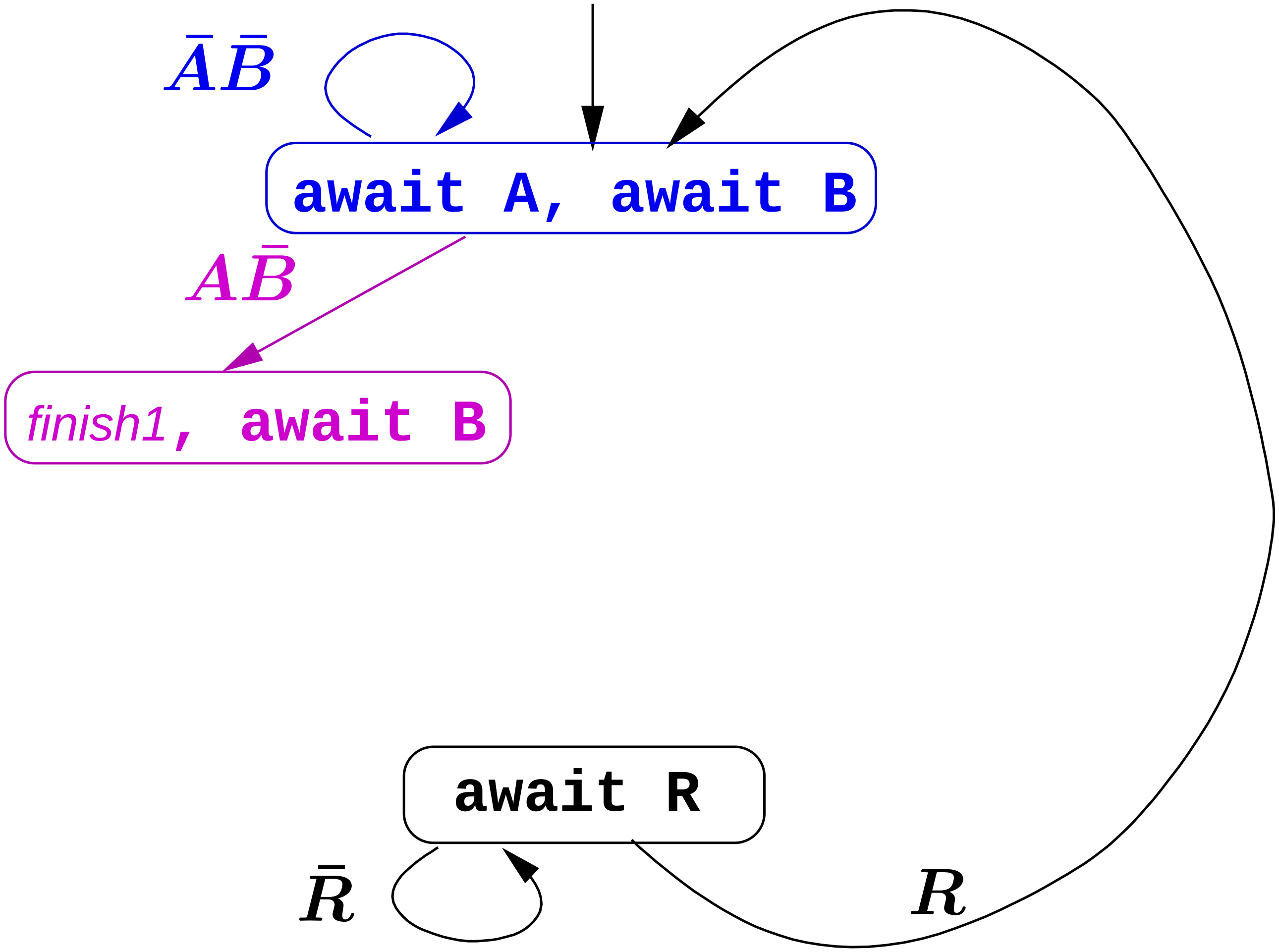
Parallel composition

- Synchronous product, with synchronization at the end
- Add a special state `finish1` to the states of `c1`
- Add a special state `finish2` to the states of `c2`
- Control points in `[ c1 || c2 ]`:
  - are couples (***c1*** *state*, ***c2*** *state*),
  - except `(finish1, finish2)` which is ***transient***
- Transitions:
  - Conjunction of conditions
  - Union of emissions

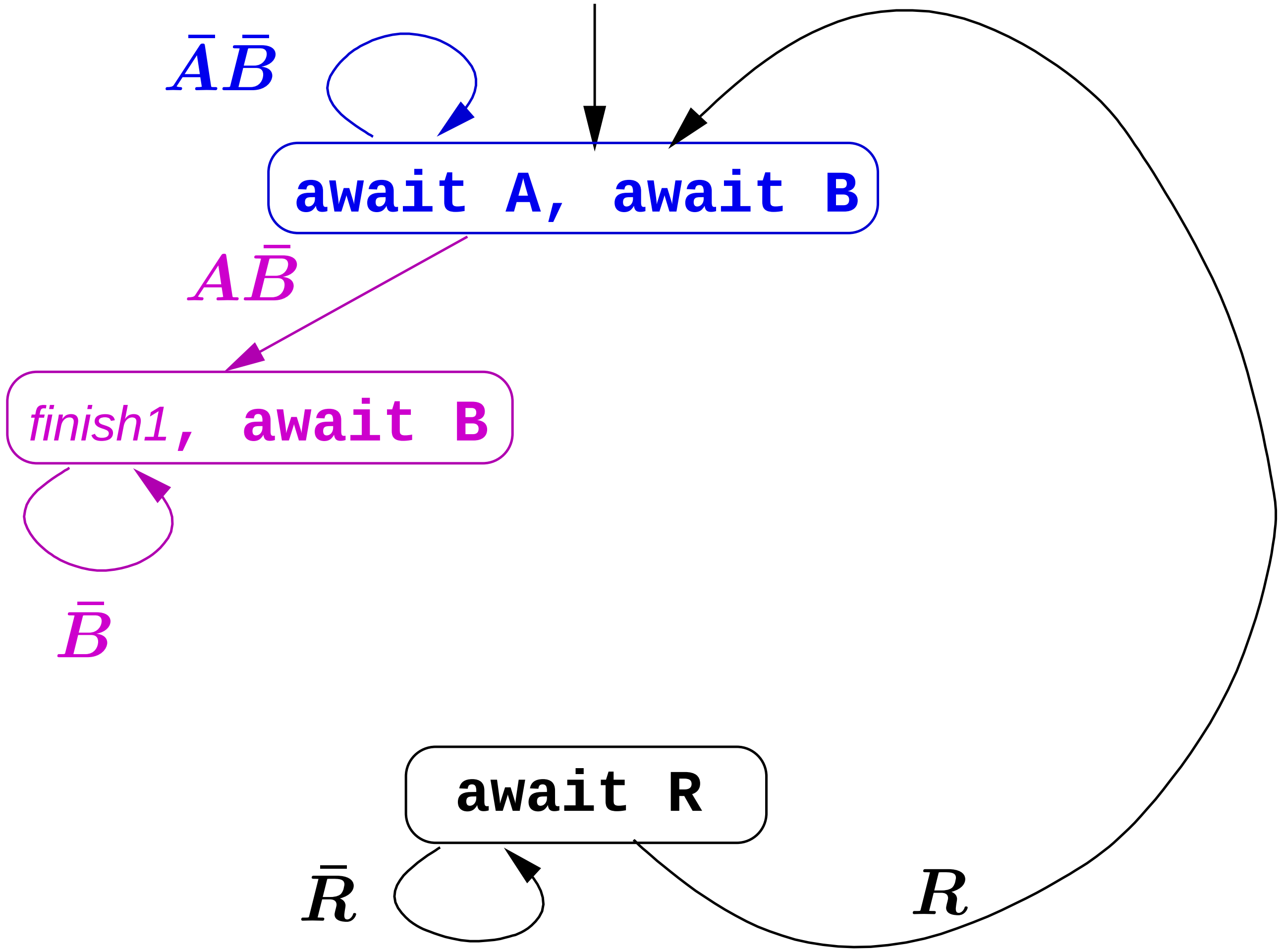
```
loop
  [
    await A
  ||
    await B
  ] ;
  emit O ;
  await R
end
```



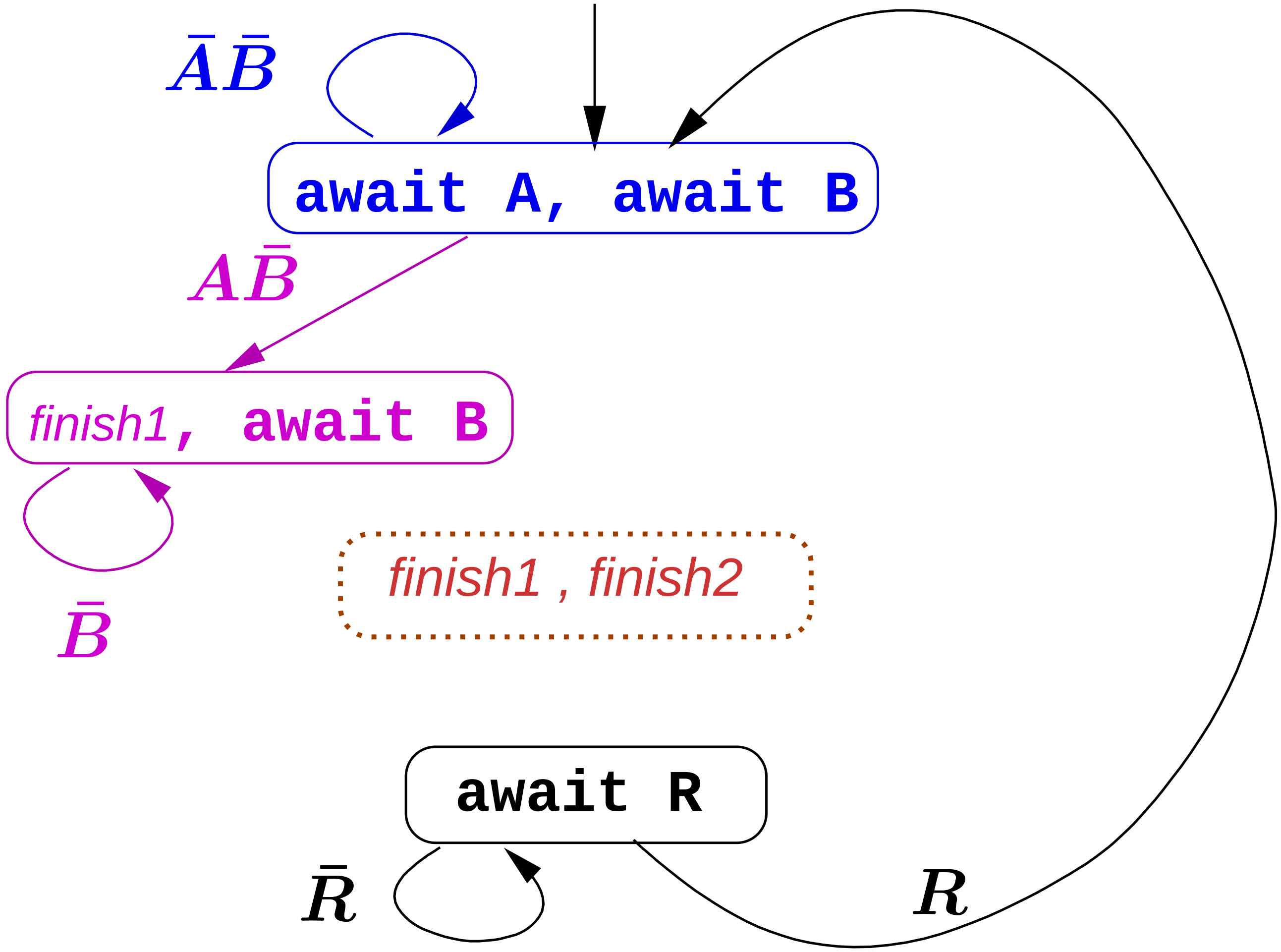
```
loop
  [
    await A
  ||
    await B
  ] ;
  emit O ;
  await R
end
```



```
loop
  [
    await A
  ||
    await B
  ] ;
  emit O ;
  await R
end
```



```
loop
  [
    await A
  ||
    await B
  ] ;
  emit O ;
  await R
end
```



$\bar{A}\bar{B}$

await A, await B

$A\bar{B}$

finish1, await B

$\bar{B}$

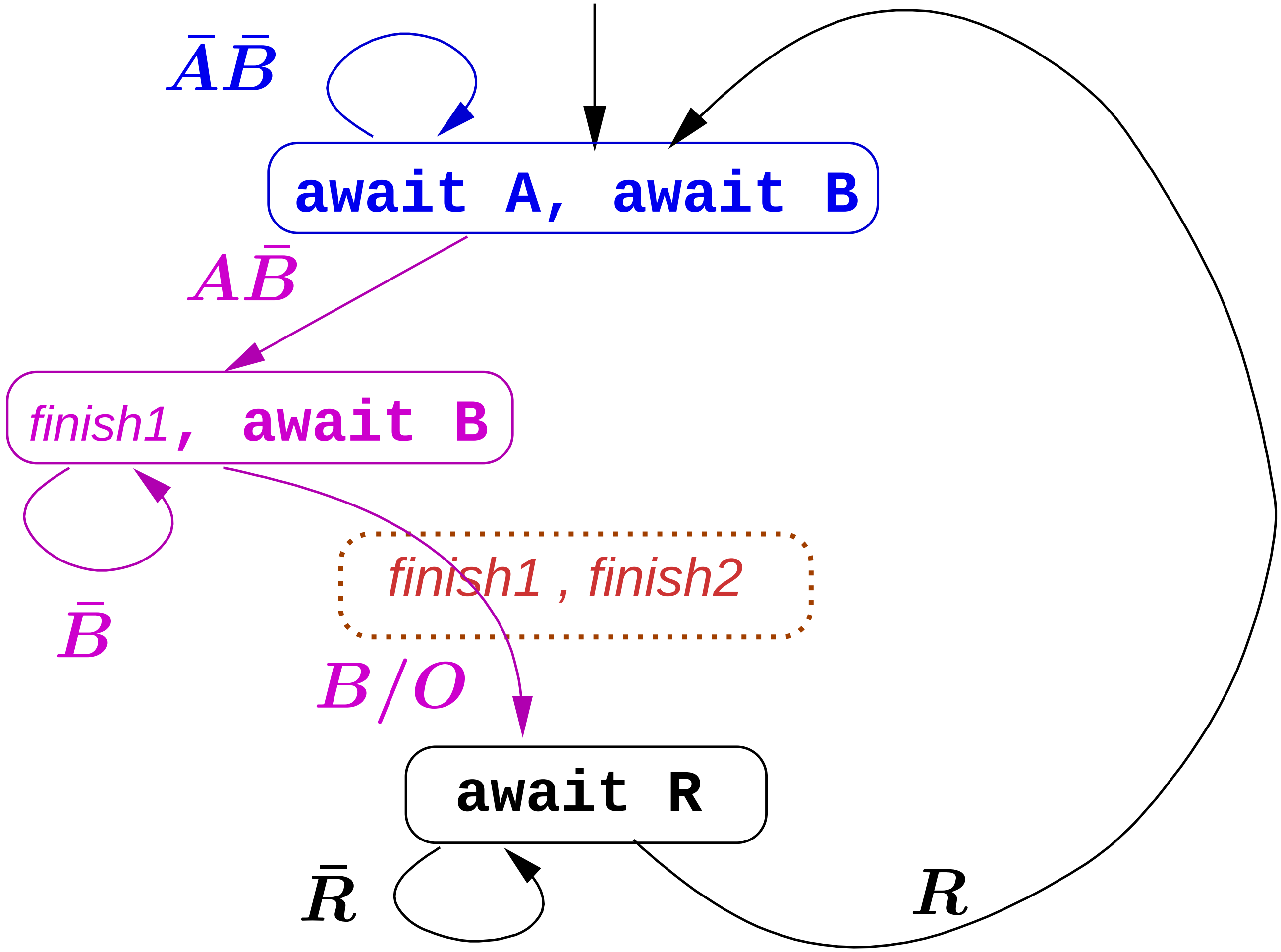
finish1 , *finish2*

await R

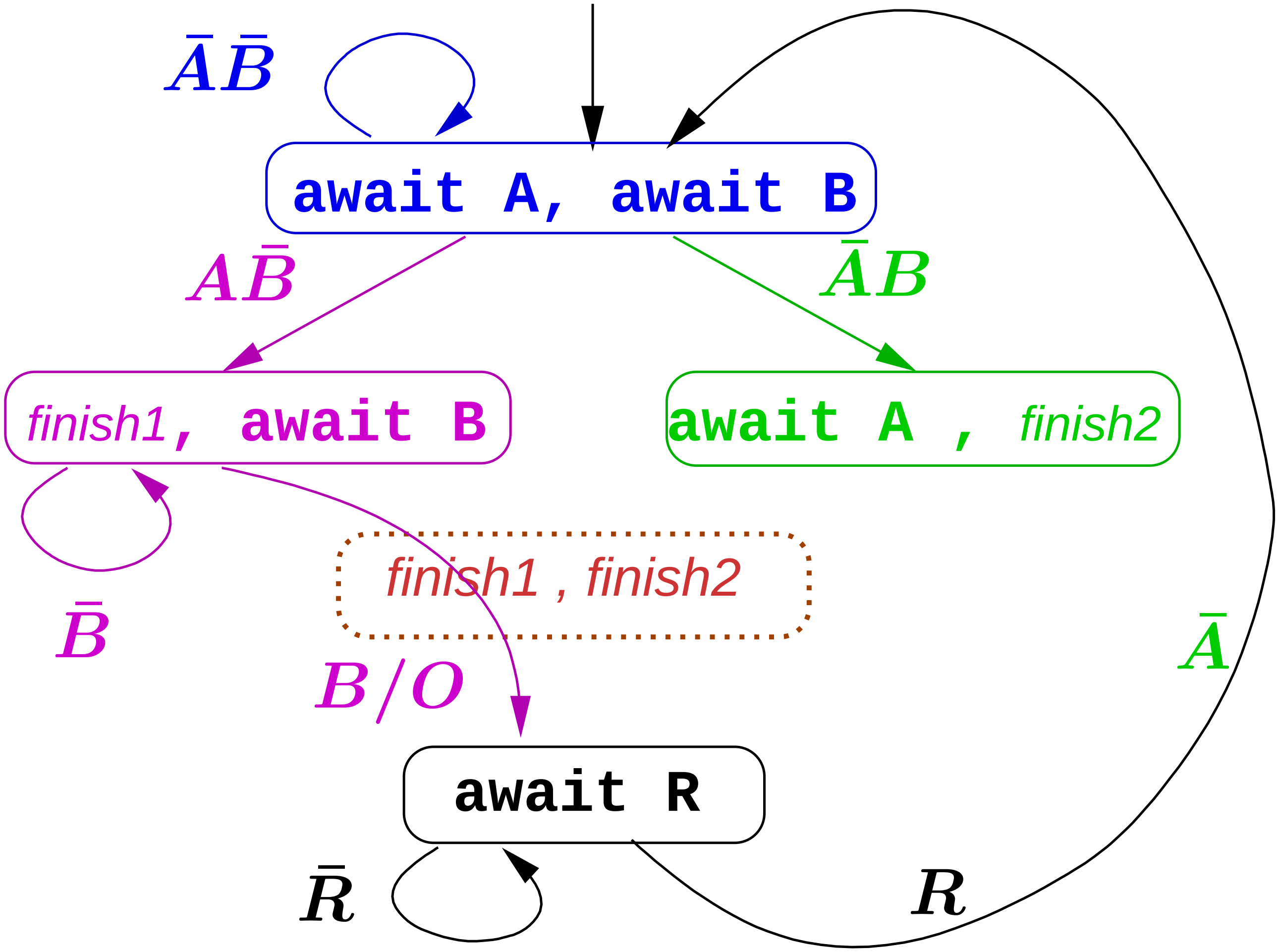
$\bar{R}$

R

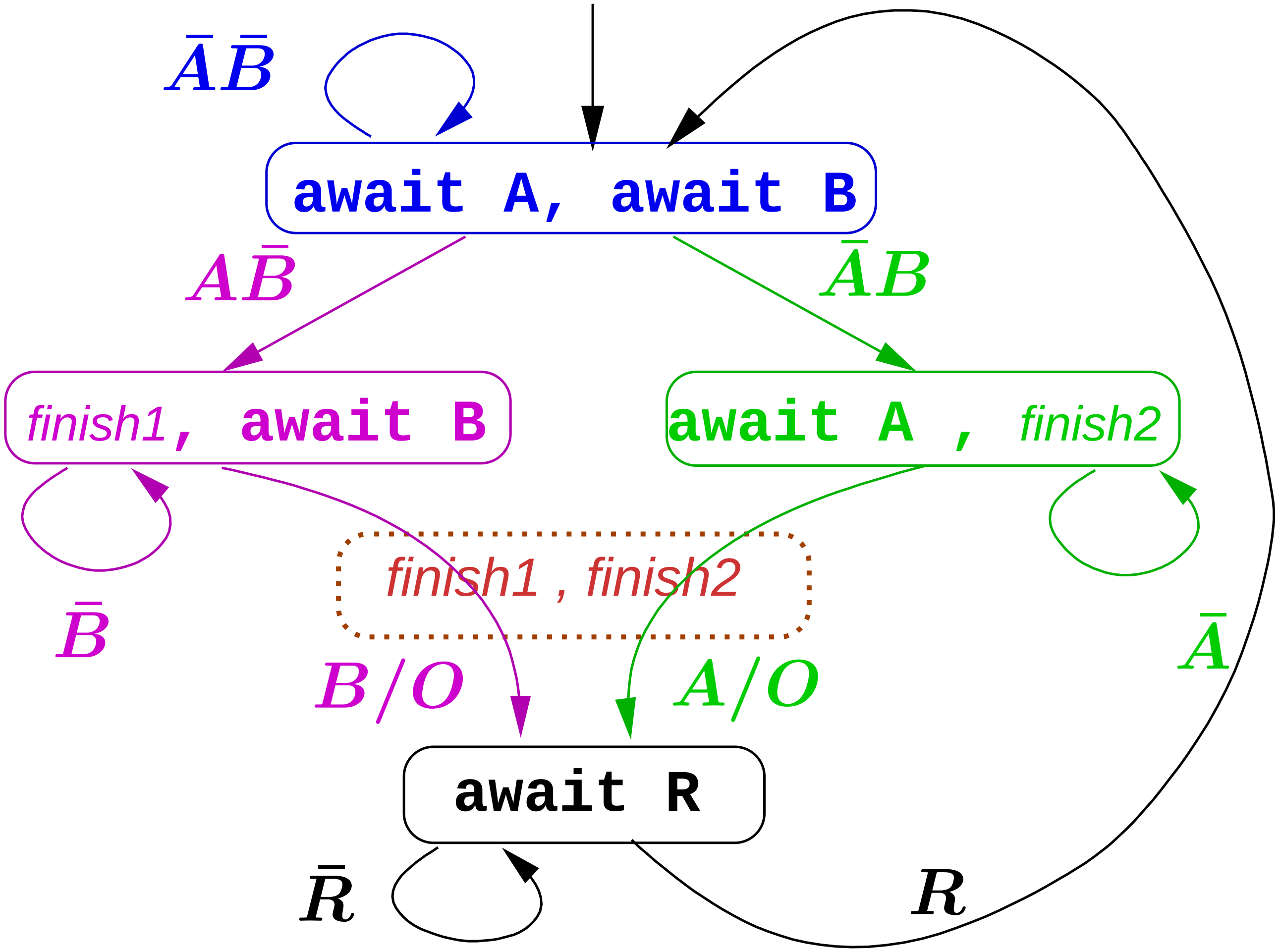
```
loop
  [
    await A
  ||
    await B
  ] ;
  emit O ;
  await R
end
```



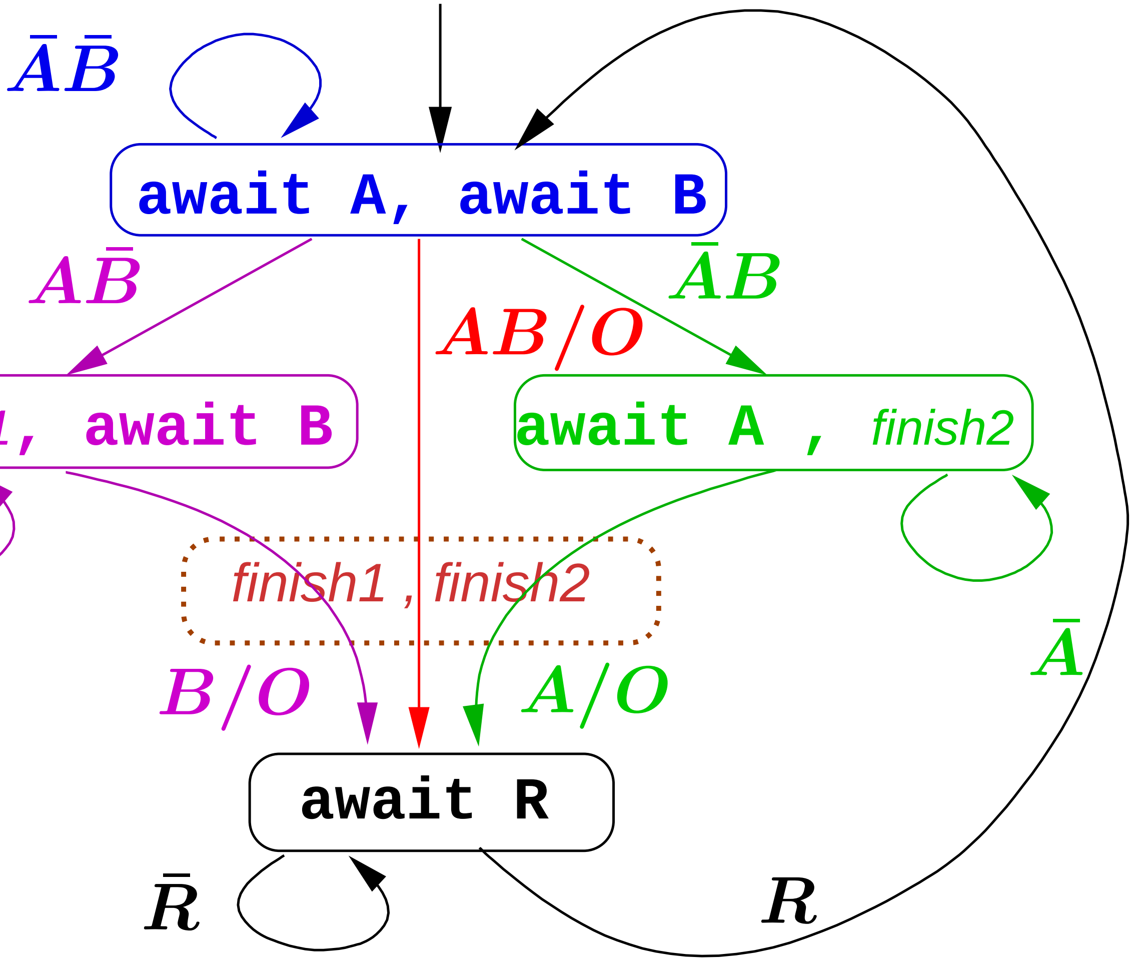
```
loop
  [
    await A
  ||
    await B
  ] ;
  emit O ;
  await R
end
```



```
   loop
     [
       await A
     ||
       await B
    ] ;
    emit O ;
    await R
  end
```



```
loop
  [
    await A
  ||
    await B
  ] ;
  emit O ;
  await R
end
```



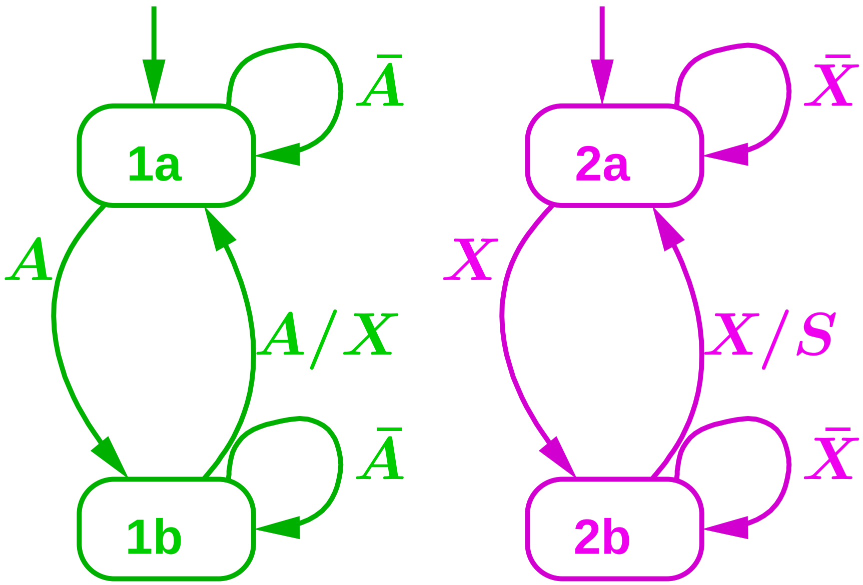
Local signals

Declaration

- `signal X in c end`
- Main use: communication between concurrent behaviours
- X can’t come from outside
- X can’t be received outside

```
signal X in [
  loop
    await A; % state 1a
    await A; % state 1b
    emit X
  end
  ||
  loop
    await X; % state 2a
    await X; % state 2b
    emit S
  end
]
```

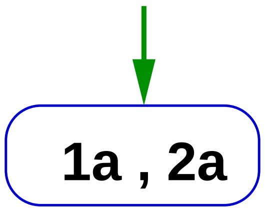
Automata product



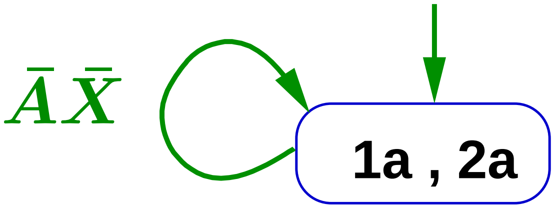
N.B. transient states finish1 et finish2 are useless

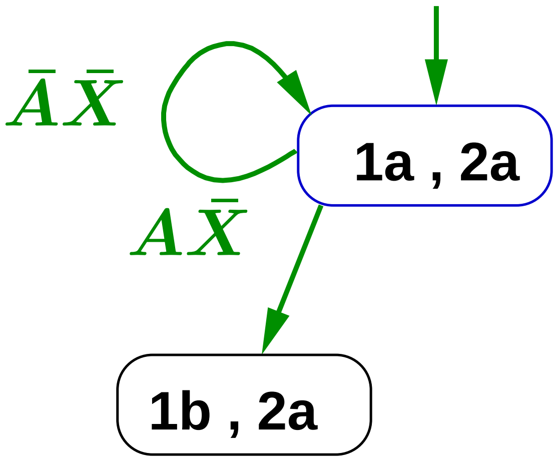
Local signal in product

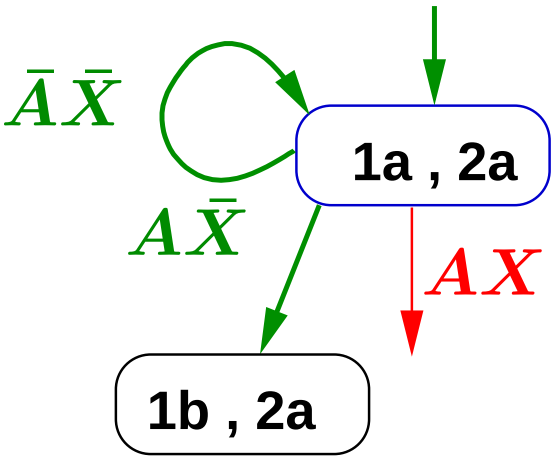
- $\xrightarrow{\bar{X}\,/\,X}$ **impossible (logic)**
- $\xrightarrow{X\,/}$ **impossible (local)**
- $\xrightarrow{X\,/\,X}$ **ok (logic)**
- $\xrightarrow{\bar{X}\,/}$ **ok (local)**

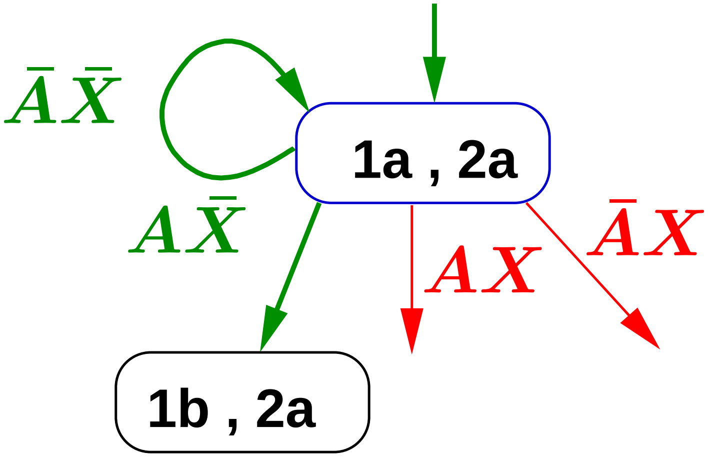


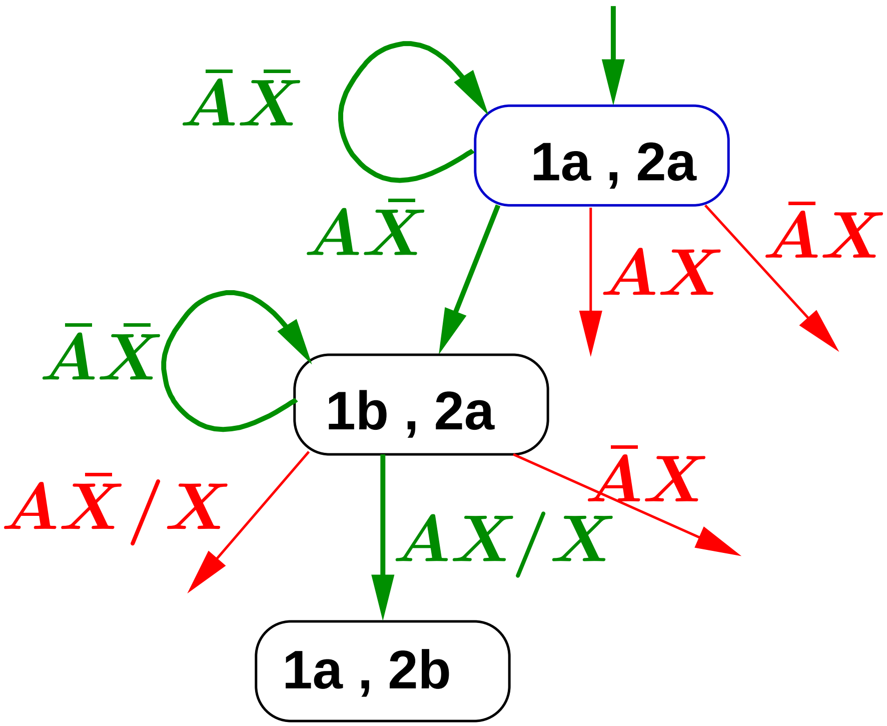
1a , 2a

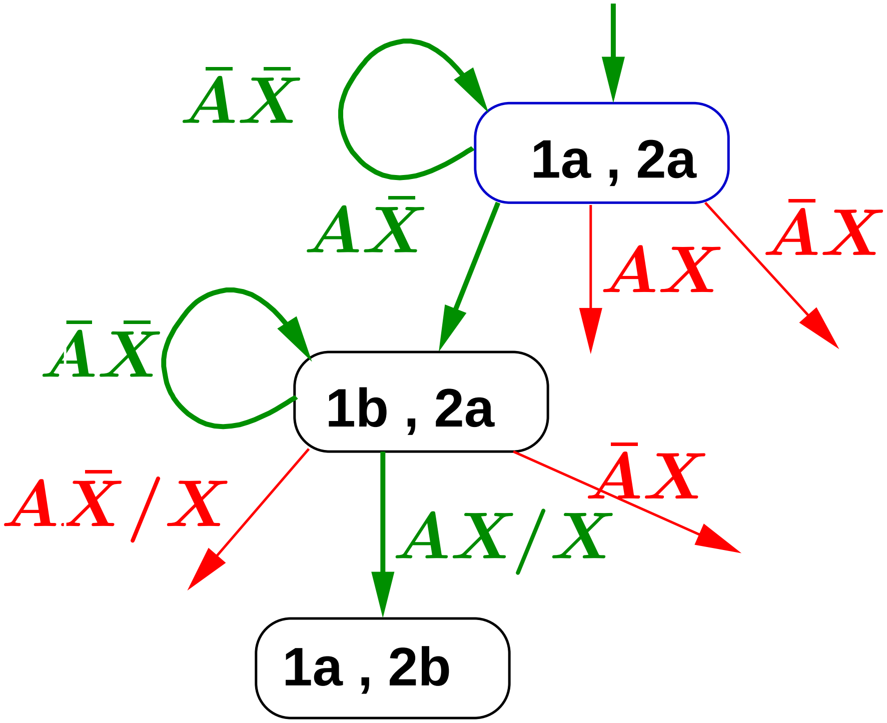

$\bar{A}\bar{X}$
1a , 2a


$\bar{A}\bar{X}$
1a , 2a
$A\bar{X}$
1b , 2a

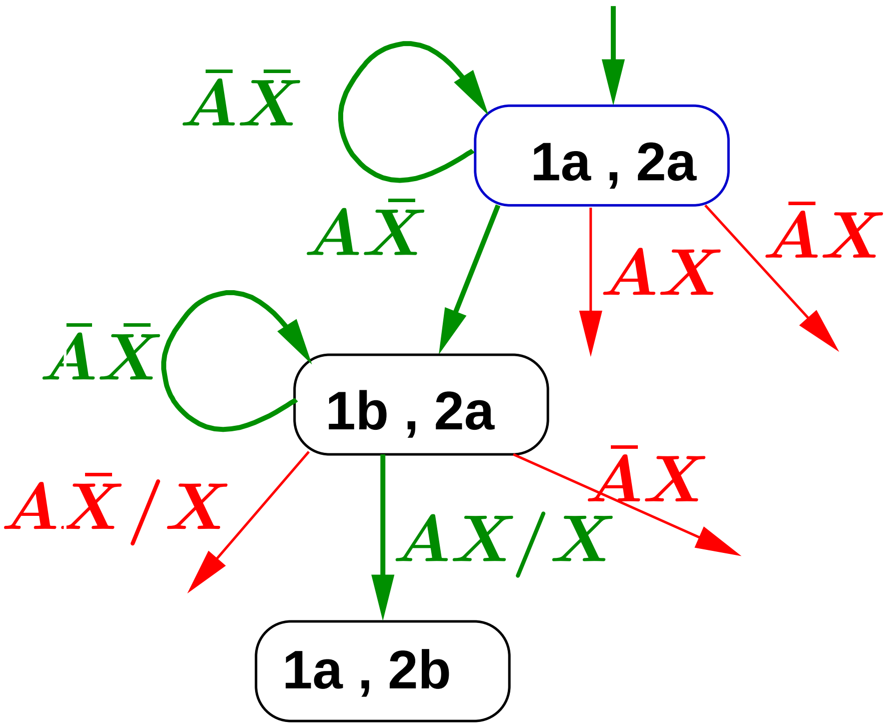

$\bar{A}\bar{X}$
1a , 2a
$A\bar{X}$
AX
1b , 2a


$\bar{A}\bar{X}$
1a , 2a
$A\bar{X}$
AX
$\bar{A}X$
1b , 2a

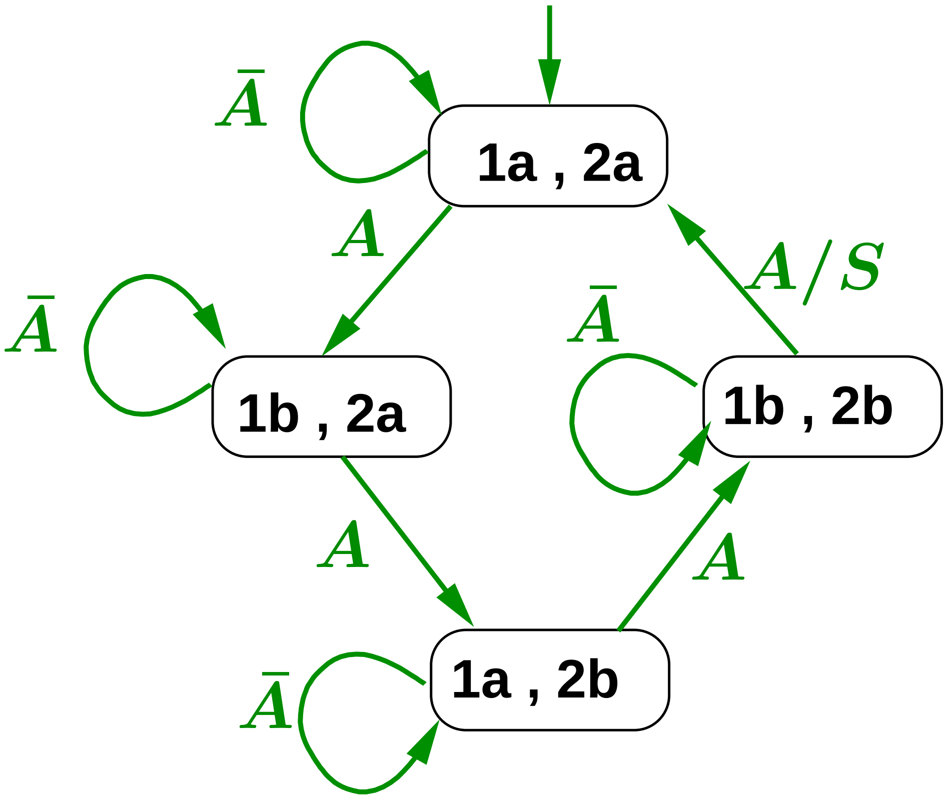

$\bar{A}\bar{X}$
1a , 2a
$A\bar{X}$
AX
$\bar{A}X$
$\bar{A}\bar{X}$
1b , 2a
$A\bar{X}/X$
$\bar{A}X$
AX/X
1a , 2b


$\bar{A}\bar{X}$
1a , 2a
$A\bar{X}$
AX
$\bar{A}X$
$\bar{A}\bar{X}$
1b , 2a
$A\bar{X}/X$
$\bar{A}X$
AX/X
1a , 2b

ETC ...


$\bar{A}\bar{X}$
1a , 2a
$A\bar{X}$
AX
$\bar{A}X$
$\bar{A}\bar{X}$
1b , 2a
$A\bar{X}/X$
$\bar{A}X$
AX/X
1a , 2b

ETC ...


$\bar{A}$
1a , 2a
A
A/S
$\bar{A}$
$\bar{A}$
1b , 2a
1b , 2b
A
A
$\bar{A}$
1a , 2b

Interruption structures

Strong pre-emption

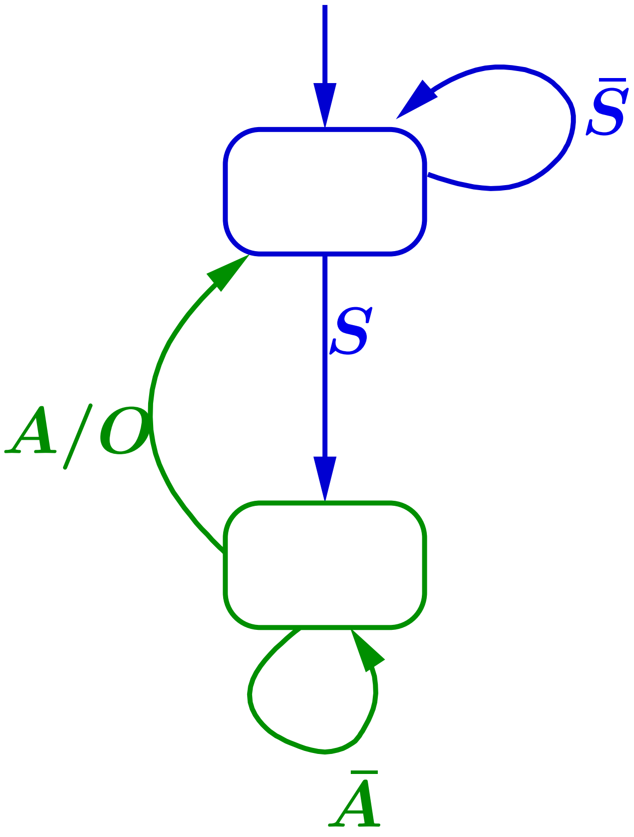
- `abort c when X`
- The next occurrence of X is a limit for the execution of c
- If X occurs c is immediately *killed*

Weak pre-emption

- `weak abort c when X`
- Similar, but if and when X occurs, c terminates its current reaction (last wishes)

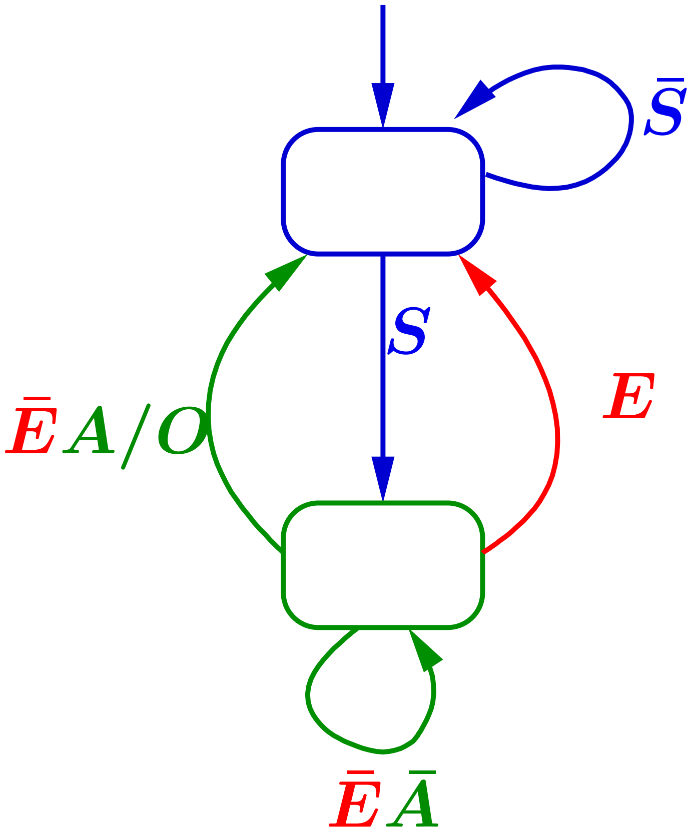
Strong vs weak pre-emption

```
loop
  await S ;

    await A ;
    emit O

end
```



Strong vs weak pre-emption

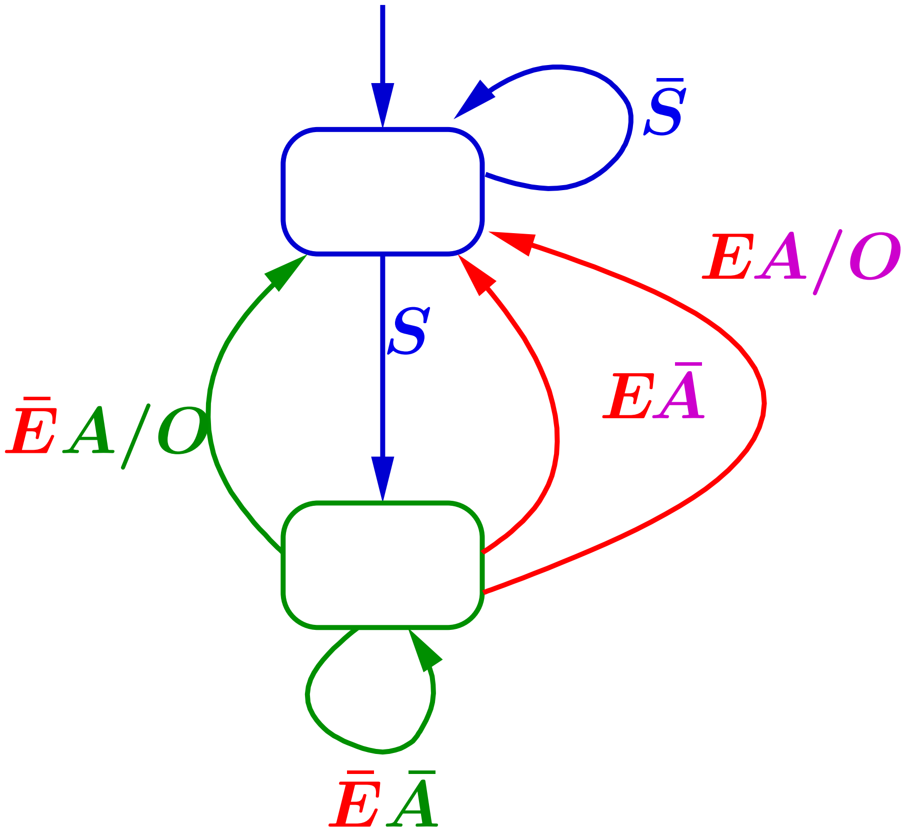
```
loop
  await S ;
  abort
    await A ;
    emit O
  when E
end
```



strong abort: no last wishes

Strong vs weak pre-emption

```
    loop
      await S ;
weak abort
        await A ;
        emit O
      when E
    end
```



strong abort: no last wishes

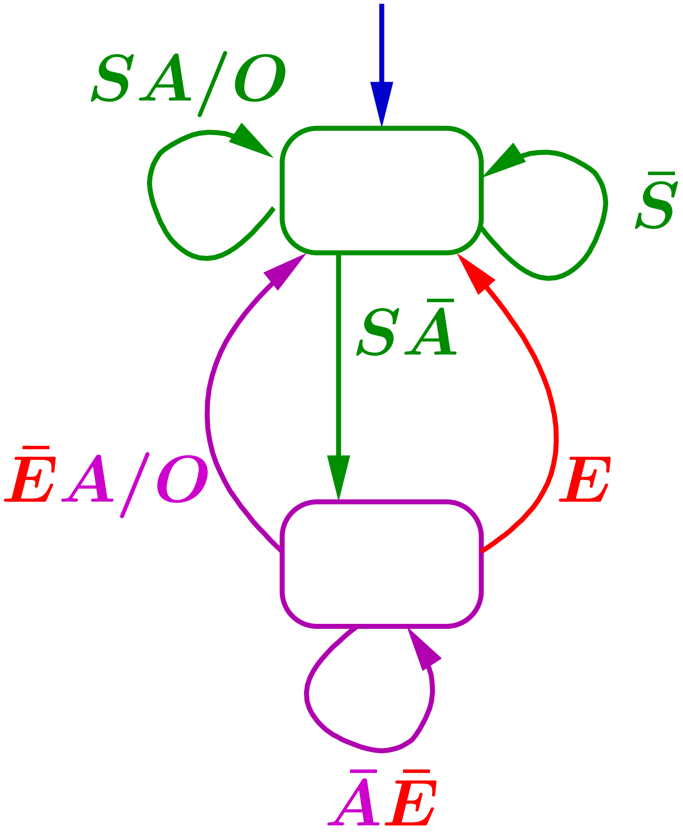
weak abort: last wishes

Example (exo)

```
loop
  await S ;
  abort
    present A else
      await A
    end ;
    emit O
  when E
end
```

Example (exo)

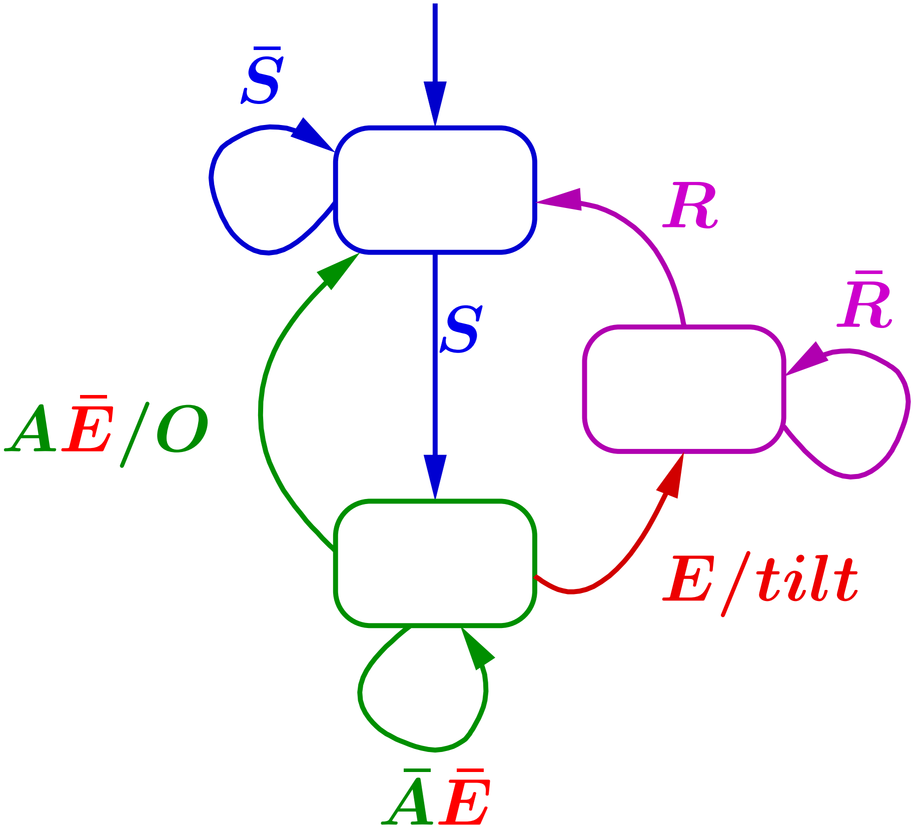
```
loop
  await S ;
  abort
    present A else
      await A
    end ;
    emit O
  when E
end
```



Catching exception

- `abort c1 when X do c2 end`
- In case of interruption, control is passed to `c2`

```
loop
  await S ;
  abort
    await A ;
    emit O
  when E do
    emit tilt ;
    await R
  end
end
```



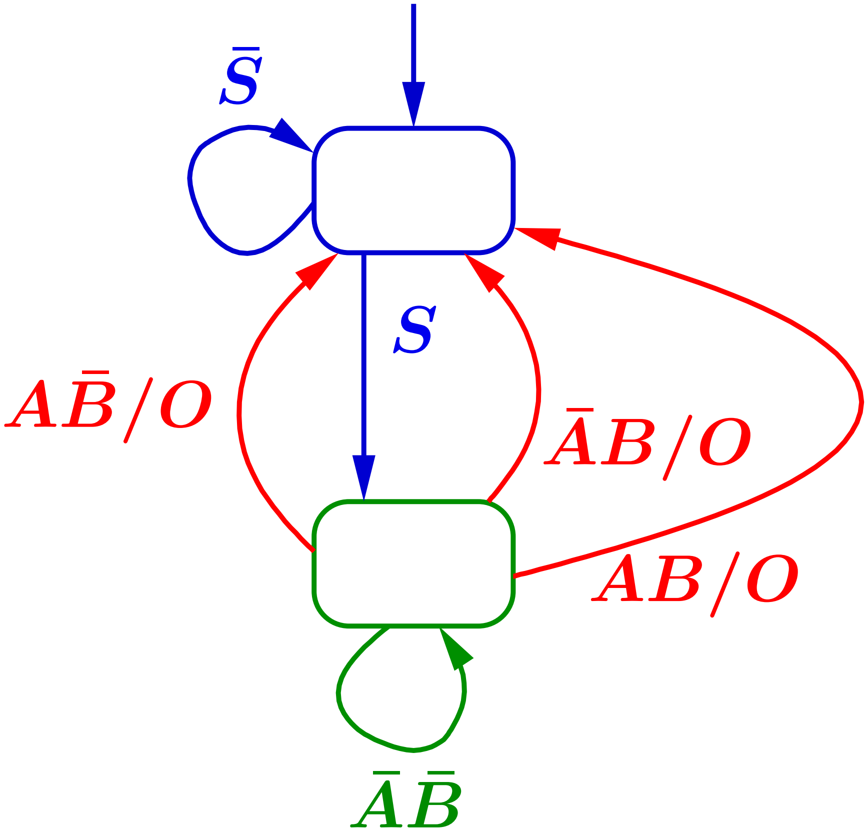
Trap/exit

- Termination from the “inside”
- Definition : `trap X in c end`
- Termination : c contains `exit X` statements
- When executed, `exit X` immediately passes the control to the end of the trap
- Ressembles both “goto” and “break” (in safer)

Example

Wait for next A or for next B

```
loop
   await S ;
   trap X in [
      await A ;
      exit X
   ||
      await B ;
      exit X
   ] end ;
   emit O
end
```



Trap/exit and parallel composition

- An `exit` statement in one branch of a parallel composition enforces all the branches to terminate
- The emiting branch stops immediately
- The other branches terminate their current reaction
- Example :

```
trap X in [
   emit A; exit X; emit B
|| emit C; await S; emit D
] end
```

is equivalent to: `emit A || emit C`

Concurrent trap/exit

General form:

```
trap X1, X2, X3 in
    c
handle X1 do c1
handle X2 do c2
handle X3 do c3
end
```

In case of simultaneous exit, all the corresponding handler are executed *in parallel*

More and more statements

Example

- `present X else await X end`
- **Very common, very useful ...**

More and more statements

Example

- `present X else await X end`
- Very common, very useful ... let’s define a new statement:

  `await immediate X`

More and more statements

Example

- `present X else await X end`
- Very common, very useful ... let’s define a new statement:

  `await immediate X`

Similarly for `abort`

- `present X else abort ...  when X end`
- becomes:

  `abort ...  when immediate X`

Notes of the evolution of the language

- More and more statements are introduced
- Simplify the writing, not increase the expression power
- Need for a (small) kernel

Esterel kernel

- `emit`, `loop`, `present`, `;`, `||`
- `signal/in`, `trap/exit`, `abort`
- `pause` (stops for a single instant), `halt` (stops forever)

Example: `await X` ⇔ `abort halt when X`

Some derived statements

- `sustain X`:

```
loop
   emit X ; pause
end
```

Some derived statements

- sustain X:

```
loop
   emit X ; pause
end
```

- do c upto X:

```
abort
   c ; halt
when X
```

Some derived statements

- **sustain X**:

```
loop
   emit X ; pause
end
```

- **do c upto X** :

```
abort
   c ; halt
when X
```

- **loop c each X**:

```
loop
   do c upto X
end
```

Some derived statements

- `sustain X`:

```
loop
   emit X ; pause
end
```

- `do c upto X` :

```
abort
   c ; halt
when X
```

- `loop c each X`:

```
loop
   do c upto X
end
```

- `every X do c end`:

```
await X ; loop c
each X
```

Conclusion

Dedicated language

- **Esterel (like Lustre) is dedicated to *reactive kernel***
- **Structured data types, complex functions, side effects are imported from the host language (tipycally C)**

Esterel and SynchCharts

- **SynchChart is a graphical language “à la StateCharts”, but with a clear synchronous semantics**
- **It can be viewed as a “graphical Esterel”**
  (automata are (just) more general than nested statements)