

Synchronous Languages—Lecture 03

Prof. Dr. Reinhard von Hanxleden

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

27 October 2016

Last compiled: November 7, 2016, 15:52 hrs



*Esterel II—The Full
Language*

Overview

A Tour through Esterel

The ABRO Example

The SPEED Example, Signals and Variables

Weak and Strong Abortion

Modules

Further Esterel Statements

The Kernel Language

The Hello World of Synchronous Programming: ABRO

The system has boolean valued inputs A , B , R , and an output O . Output O shall be true as soon as both inputs A and B have been true. This behavior should be restarted if R is true.

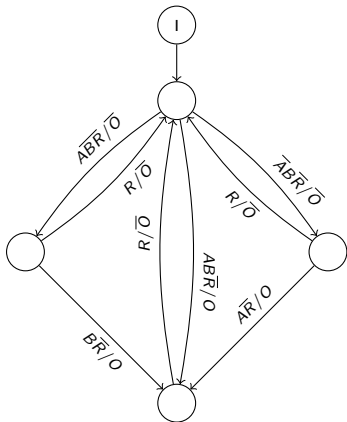
- ▶ **Question:** what if A , B and R are true at the same time?

The Hello World of Synchronous Programming: ABRO

The system has boolean valued inputs A , B , R , and an output O . Output O shall be true as soon as both inputs A and B have been true. This behavior should be restarted if R is true.

- ▶ **Question:** what if A , B and R are true at the same time?
- ▶ Should we make O present? —we consider both possibilities
- ▶ Nondeterminism? Not possible in Esterel!

Mealy Machine for ABRO



- ▶ Circles are automaton states
- ▶ Label $\overline{A}B\overline{R}/\overline{O}$ means: if $A = true$ and $B = R = false$ is read, then output $O = true$ is generated
- ▶ *Default behavior*: remain in state
- ▶ *Finite state machines (FSMs) are perfectly synchronous!*
- ~ use FSMs to explain the semantics

Write Things Once

- ▶ The disadvantage of this (flat) notation:
 - ▶ Size grows exponentially
 - ▶ A little change to the specification may incur a major change to the automaton (often ends with full rewriting)

Write Things Once

- ▶ The disadvantage of this (flat) notation:
 - ▶ Size grows exponentially
 - ▶ A little change to the specification may incur a major change to the automaton (often ends with full rewriting)
- ▶ The answer:
 - ▶ Add hierarchy
 - ▶ More generally: **Write Things Once** (WTO)

Write Things Once

- ▶ The disadvantage of this (flat) notation:
 - ▶ Size grows exponentially
 - ▶ A little change to the specification may incur a major change to the automaton (often ends with full rewriting)
- ▶ The answer:
 - ▶ Add hierarchy
 - ▶ More generally: **Write Things Once** (WTO)
- ▶ Analogy from language theory:
 - ▶ Use regular expressions to represent large (possibly infinite) sets of strings

Esterel Program ABRO

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

- ▶ Declarations of inputs and outputs
- ▶ Module body contains a statement
- ▶ Modules have names
- ▶ Esterel programs are a list of modules

Remarks on Signal Declarations

- ▶ **Signals** are special data types with a **presence status** $\in \{true, false\}$
- ▶ If $S = true$ holds, S is said to be **present**, otherwise **absent**
- ▶ Signals describe **events**, thus they **do not store the status** when control flow proceeds to the next macro step
- ▶ Status of **input signals** is generated by the **environment**
- ▶ Status of **output signals** is made present by executing `emit S`
- ▶ Output signals are present iff they are currently emitted
- ▶ `emit S` does not take time

Remarks on Signal Declarations

- ▶ Signal status is uniquely determined per macro step
- ▶ This may lead to the fact that “information flows backwards”:

```
present R then emit S end;  
emit R
```

- ▶ In the above program, the emission of R is also seen by the conditional statement (present R checks the status of R)
- ▶ This may lead to **causality problems**, but implements the perfect synchrony

General Remarks on Statements

- ▶ Statements p are started at step $t \in \mathbf{N}$ and terminate in a (not necessarily strictly) later step $t + \delta$ ($0 \leq \delta$)
- ▶ If $\delta = 0$ holds, p is called **instantaneous**:
 - ▶ Its execution does not take time
 - ▶ p does only execute micro steps
- ▶ Whether p is instantaneous or not depends on current inputs
- ▶ If p is not instantaneous, the control flow enters p and will stop somewhere inside p to wait for the next macro step
- ▶ Due to concurrency, the control flow may rest at several locations

Remarks on emit

- ▶ emit S is always instantaneous
- ▶ Executing emit S makes S immediately present for the current macro step

Remarks on emit

- ▶ `emit S` is always instantaneous
- ▶ Executing `emit S` makes `S` immediately present for the current macro step
- ▶ There are also **delayed emissions** (since Esterel version 7):
 - ▶ `emit next S` makes `S` present in the next macro step
 - ▶ Executing `emit next S` is also instantaneous
- ▶ Input signals may also be emitted

Remarks on await

- ▶ When started, control remains at `await S`
- ▶ At the **next** macro step, `S` is tested:
 - ▶ if `S` holds, `await S` terminates
 - ▶ otherwise, the behavior is repeated at the next macro step

Remarks on await

- ▶ When started, control remains at `await S`
- ▶ At the **next** macro step, `S` is tested:
 - ▶ if `S` holds, `await S` terminates
 - ▶ otherwise, the behavior is repeated at the next macro step
- ▶ `await S` *always* consumes time (*i. e.*, is *never* instantaneous)
- ▶ The variant `await immediate S` tests `S` also at starting time, and therefore may also be instantaneous
- ▶ `S` can either be a signal or a signal expression

Remarks on Parallel Statements

$p \parallel q$ means parallel execution of p and q

Remarks on Parallel Statements

- $p \parallel q$ means parallel execution of p and q
- ▶ if $p \parallel q$ is started at time t , both p and q are started at time t
 - ▶ if p and q terminate at time $t + \delta_p$ and $t + \delta_q$, respectively, then $p \parallel q$ terminates at time $t + \max\{\delta_p, \delta_q\}$
- ~> as long as the control is inside p and q , both p and q *execute their macro steps synchronously*
- ▶ p and q may interact during concurrent execution

Remarks on Parallel Statements

- $p \parallel q$ means parallel execution of p and q
- ▶ if $p \parallel q$ is started at time t , both p and q are started at time t
 - ▶ if p and q terminate at time $t + \delta_p$ and $t + \delta_q$, respectively, then $p \parallel q$ terminates at time $t + \max\{\delta_p, \delta_q\}$
 - ↪ as long as the control is inside p and q , both p and q *execute their macro steps synchronously*
 - ▶ p and q may interact during concurrent execution

Brackets $[\dots]$ are used to control statement scoping to avoid ambiguities due to the grammar

Remarks on Sequences

- ▶ `p;q` is a **sequence**

Remarks on Sequences

- ▶ $p; q$ is a **sequence**
 - ▶ if $p; q$ is started at time t , at least p is started at time t
 - ▶ if p terminates at time $t + \delta_p$, then q is started at time $t + \delta_p$
 - ▶ note that $\delta_p = 0$ may hold, which implies that p and q are both started at time t
 - ▶ $p; q$ terminates when q terminates
 - ▶ Moving the control from p to q does not take time
- ↪ the sequence operator $;$ does not take time

Remarks on Loops

- ▶ Esterel knows several loop constructs
- ▶ `loop p each S` behaves as follows:

Remarks on Loops

- ▶ Esterel knows several loop constructs
- ▶ `loop p each S` behaves as follows:
 - ▶ if `loop p each S` is started at time t , then p is started at time t
 - ▶ in subsequent instants, p is restarted whenever $S = true$ holds (S is present)
 - ▶ if p terminates, then the program waits for the next step where $S = true$ holds
 - ▶ note that p is aborted when it is currently active and S holds
- ~> no dynamic thread generation
- ~> this guarantees finitely many control states

Generic ABRO Program

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

- ▶ ABRO can be easily extended for more events
- ▶ To this end, only a new thread with an `await` statement has to be added
- ▶ For n inputs, the program has size $O(n)$

Generic ABRO Program

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

- ▶ ABRO can be easily extended for more events
 - ▶ To this end, only a new thread with an `await` statement has to be added
 - ▶ For n inputs, the program has size $O(n)$
 - ▶ But the finite state machine has $O(2^n)$ states
- ~> **Esterel programs can be exponentially more compact than finite state machines**

Program SPEED

The system has inputs cm and sec . If sec holds, the number of macro steps where cm holds should be counted. If sec holds again, the number of so far seen cm signals should be reported, reset to zero, and the behavior should be repeated.

- ▶ **Question:** what if cm and sec hold at the same time?
- ▶ We first exclude this case, and consider solutions for that later

Program SPEED

```
module SPEED:
input cm, sec;
output Speed:integer;
relation cm # sec;
loop
  var distance := 0 : integer in
    abort
      every cm do
        distance := distance + 1
      end every
    when sec do
      emit Speed(distance)
    end abort
  end var
end loop
end module
```

New constructs:

- ▶ Valued signals
- ▶ Input relations
- ▶ Local variables
- ▶ Process preemption (abortion)

Remarks on Valued Signals

- ▶ **Input restriction** 'R#S'
tells the compiler that R and S cannot be both present
- ▶ $S:\alpha$ **declares a valued signal** of type α
 - ▶ such a signal has a present/absent **status**
 - ▶ and a **value** of type α that is denoted as ?S
 - ▶ the value is stored, unless changed by an emission `emit S(v)` that immediately changes the value to v
 - ▶ as the status, the value is uniquely defined per macro step
- ▶ **Note: Emissions immediately change the values, hence, `emit S(?S+1)` makes no sense!**
- ▶ For that, use delayed emissions: `emit next(S(v))`
 - ▶ v is immediately evaluated
 - ▶ But the value of S is changed in the next macro step

Remarks on Local Variables

- ▶ `var x := τ : α in p end var` declares a local variable `x` of type α which is initialized by τ and is visible in statement `p`.
- ▶ Differences between variables and signals:
 - ▶ variables do not have a status, but only a value
 - ▶ variables store values unless these are changed by assignments
 $x := \tau$
 - ▶ variables can be *changed by micro steps*, hence, they may have several values in a macro step
 - ▶ for this reason, there are restrictions on the use of variables in parallel threads: if a local variable declaration contains parallel threads and the variable is written to within a thread, none of the concurrent threads may access (read or write) that variable
- ~> assignments to a variable never have write conflicts

Remarks on Local Declarations

- ▶ There are also local signals: `signal S: α in p end signal`
- ▶ These are treated like output signals inside S
- ▶ Like output signals, local signals may have a value or not
- ▶ Status and value of a local signal is uniquely determined per macro step
- ▶ This may result in write conflicts (as with valued signals in general), e.g.: `emit S(2); emit S(3)`
- ▶ In contrast to local variables, threads may interact via local signals

Remarks on Loops

- ▶ `loop p end` is the basic loop
 - ▶ if `loop p end` is started at time t , then p is started at time t
 - ▶ **execution of p must always take time**, *i. e.*, there must not be inputs such that p becomes instantaneous
 - ▶ if S terminates at time $t + \delta > t$, then p is started at time $t + \delta > t$
- ↷ `loop p end` is equivalent to p ; `loop p end`
 - ▶ however, such statements can be terminated by surrounding process abortion

Remarks on Loops

- ▶ `loop p end` is the basic loop
 - ▶ if `loop p end` is started at time t , then p is started at time t
 - ▶ **execution of p must always take time**, *i. e.*, there must not be inputs such that p becomes instantaneous
 - ▶ if S terminates at time $t + \delta > t$, then p is started at time $t + \delta > t$
- ↷ `loop p end` is equivalent to `p; loop p end`
 - ▶ however, such statements can be terminated by surrounding process abortion
- ▶ `every S do p end every`
 - ▶ is equivalent to `await S; loop p each S`
 - ▶ hence, every time S holds, p is started (and possibly aborted)

Remarks on abort

- ▶ abort p when S do q end abort
 - ▶ if started at time t , p is started at time t without checking S

Remarks on abort

- ▶ `abort p when S do q end abort`
 - ▶ if started at time t , p is started at time t without checking S
 - ▶ if p terminates at time t , then the entire statement terminates

Remarks on abort

- ▶ `abort p when S do q end abort`
 - ▶ if started at time t , p is started at time t without checking S
 - ▶ if p terminates at time t , then the entire statement terminates
 - ▶ otherwise, the execution of p takes time:
 - ▶ in all macro steps that start inside p , S is checked

Remarks on abort

- ▶ `abort p when S do q end abort`
 - ▶ if started at time t , p is started at time t without checking S
 - ▶ if p terminates at time t , then the entire statement terminates
 - ▶ otherwise, the execution of p takes time:
 - ▶ in all macro steps that start inside p , S is checked
 - ▶ if S does not hold, p is executed for this macro step

Remarks on abort

- ▶ `abort p when S do q end abort`
 - ▶ if started at time t , p is started at time t without checking S
 - ▶ if p terminates at time t , then the entire statement terminates
 - ▶ otherwise, the execution of p takes time:
 - ▶ in all macro steps that start inside p , S is checked
 - ▶ if S does not hold, p is executed for this macro step
 - ▶ if S holds, **no action of p is executed**, instead, q is started
 - ▶ if the latter happens, q is executed without checking S
- ↪ Abortion is also called **process preemption**
- ▶ **Note:** the abort handler (`do q`) is optional

Variants of Process Abortion

- ▶ abort comes in four variants:
 - ▶ abort p when S do q end abort
 - ▶ **weak** abort p when S do q end abort
 - ▶ abort p when **immediate** S do q end abort
 - ▶ **weak** abort p when **immediate** S do q end abort

Variants of Process Abortion

- ▶ abort comes in four variants:
 - ▶ abort p when S do q end abort
 - ▶ **weak** abort p when S do q end abort
 - ▶ abort p when **immediate** S do q end abort
 - ▶ **weak** abort p when **immediate** S do q end abort
- ▶ **weak abortion** differs in macro steps where abortion takes place:
 - ▶ weak abort executes all micro steps of p at abortion time (*i. e.*, p **may execute a “last wish”** even when it is aborted)

Variants of Process Abortion

- ▶ abort comes in four variants:
 - ▶ abort p when S do q end abort
 - ▶ **weak** abort p when S do q end abort
 - ▶ abort p when **immediate** S do q end abort
 - ▶ **weak** abort p when **immediate** S do q end abort
- ▶ **weak abortion** differs in macro steps where abortion takes place:
 - ▶ weak abort executes all micro steps of p at abortion time (*i. e.*, p **may execute a “last wish”** even when it is aborted)
- ▶ **immediate abortions** consider S **also at starting time**
 - ▶ if S holds at starting time, strong abort immediately starts q
 - ▶ weak abort additionally executes all micro steps of p that were executed if abortion would not take place

Other immediate Statements

- ▶ Many other statements have immediate variants
 - ▶ `await immediate S`
 - ▶ `every immediate S do p end`
- ▶ We will see later that this is because these statements contain in some sense abortion statements

Other immediate Statements

- ▶ Many other statements have immediate variants
 - ▶ `await immediate S`
 - ▶ `every immediate S do p end`
- ▶ We will see later that this is because these statements contain in some sense abortion statements
- ▶ **Note:** There is no immediate variant of `loop p each S`. Why?

Other immediate Statements

- ▶ Many other statements have immediate variants
 - ▶ `await immediate S`
 - ▶ `every immediate S do p end`
- ▶ We will see later that this is because these statements contain in some sense abortion statements
- ▶ **Note:** There is no immediate variant of `loop p each S`. Why? Because otherwise this would lead to an instantaneous loop.

Other immediate Statements

- ▶ Many other statements have immediate variants
 - ▶ `await immediate S`
 - ▶ `every immediate S do p end`
- ▶ We will see later that this is because these statements contain in some sense abortion statements
- ▶ **Note:** There is no immediate variant of `loop p each S`. Why? Because otherwise this would lead to an instantaneous loop.
- ▶ **Note:** `every immediate S do p end` expands to

Other immediate Statements

- ▶ Many other statements have immediate variants
 - ▶ `await immediate S`
 - ▶ `every immediate S do p end`
- ▶ We will see later that this is because these statements contain in some sense abortion statements
- ▶ **Note:** There is no immediate variant of `loop p each S`. Why? Because otherwise this would lead to an instantaneous loop.
- ▶ **Note:** `every immediate S do p end` expands to `await immediate S; loop p each S end`

Weak Abortion in Program SPEED

```
module SPEED:
input cm, sec;
output Speed:integer;
loop
  var distance1 := 0 : integer in
    weak abort
      every cm do
        distance1 := distance1 + 1
      end every
    when sec do
      emit Speed(distance1)
    end abort
  end var
end loop
end module
```

Changes by weak abortion:

Weak Abortion in Program SPEED

```
module SPEED:
input cm, sec;
output Speed:integer;
loop
  var distance1 := 0 : integer in
    weak abort
      every cm do
        distance1 := distance1 + 1
      end every
    when sec do
      emit Speed(distance1)
    end abort
  end var
end loop
end module
```

Changes by weak abortion:

- ▶ if sec holds, the abortion takes place
- ▶ if additionally cm holds, distance is once more incremented
- ▶ and thus, this cm is added to the current interval

Using 'immediate' in Program SPEED

Changes by 'immediate':

```
module SPEED:
input cm, sec;
output Speed:integer;
loop
  var distance2 := 0 : integer in
    abort
      every immediate cm do
        distance2 := distance2 + 1
      end every
    when sec do
      emit Speed(distance2)
    end abort
  end var
end loop
end module
```


Using 'immediate' in Program SPEED

```
module SPEED:
input cm, sec;
output Speed:integer;
loop
  var distance2 := 0 : integer in
    abort
      every immediate cm do
        distance2 := distance2 + 1
      end every
    when sec do
      emit Speed(distance2)
    end abort
  end var
end loop
end module
```

Changes by 'immediate':

- ▶ if sec holds, the abortion takes place
- ▶ if additionally cm holds, distance is not incremented (strong abort)
- ▶ after emission of Speed, every immediately executes its body statement
- ▶ thus, this cm is added to the next interval

Using Modules

```
module TwoStates :  
  input Pressed;  
  output StateOff, StateOn;  
  loop  
    abort  
      sustain StateOff;  
    when Pressed;  
    abort  
      sustain StateOn;  
    when Pressed;  
  end loop  
end module
```

- ▶ Starting sustain S immediately emits S
- ▶ Control flow rests inside sustain S
- ▶ and repeats emit S for all macro steps, unless abortion by Pressed takes place

Using Modules

```
module TwoStates :  
  input Pressed;  
  output StateOff, StateOn;  
  loop  
    abort  
      sustain StateOff;  
    when Pressed;  
    abort  
      sustain StateOn;  
    when Pressed;  
  end loop  
end module
```

- ▶ Starting sustain S immediately emits S
- ▶ Control flow rests inside sustain S
- ▶ and repeats emit S for all macro steps, unless abortion by Pressed takes place
- ▶ Hence, each time Pressed is present, the control flow toggles between the two sustain statements

Using Modules

```
module TwoStates:  
  input Pressed;  
  output StateOff, StateOn;  
  loop  
    abort  
      sustain StateOff;  
    when Pressed;  
    abort  
      sustain StateOn;  
    when Pressed;  
  end loop  
end module
```

```
module NoName:  
  input Button;  
  output inactive;  
  
  run TwoStates  
    [signal  
      Button/Pressed,  
      inactive/StateOff  
    ]  
  ||  
  ...  
end module
```

Using Modules

- ▶ If module `m` has already been defined, then `m` can be instantiated in other module bodies
 - ▶ This is done by executing the statement `'run m'`
- ↪ compiler replaces `run m` with the body of `m`

Using Modules

- ▶ If module m has already been defined, then m can be instantiated in other module bodies
- ▶ This is done by executing the statement 'run m '
- ↪ compiler replaces run m with the body of m
- ▶ Additionally, declared objects in m can be renamed:
$$\text{run } m \ [t_1 \ y_1/x_1, \dots, t_n \ y_n/x_n], \text{ where}$$
$$t_j \ x_j \text{ is a declaration of module } m$$

Using Modules

- ▶ If module m has already been defined, then m can be instantiated in other module bodies
- ▶ This is done by executing the statement 'run m '
~> compiler replaces run m with the body of m
- ▶ Additionally, declared objects in m can be renamed:
$$\text{run } m \ [t_1 \ y_1/x_1, \dots, t_n \ y_n/x_n], \text{ where}$$
$$t_i \ x_i \text{ is a declaration of module } m$$
- ▶ **no recursive module calls allowed** (possibly infinite recursion)
- ▶ Primitive recursion (which always terminates) could be allowed

Overview

A Tour through Esterel

Further Esterel Statements

- Further Basic Statements

- Process Suspension

- Variants of Discussed Statements, Trap vs. Abort

- Host Language

The Kernel Language

Esterel Statements Discussed So Far

- ▶ `emit S` and `emit S(v)`
- ▶ `sustain S` and `sustain S(v)`

Esterel Statements Discussed So Far

- ▶ `emit S` and `emit S(v)`
- ▶ `sustain S` and `sustain S(v)`
- ▶ sequence: `p; q`
- ▶ parallel: `p || q`

Esterel Statements Discussed So Far

- ▶ `emit S` and `emit S(v)`
- ▶ `sustain S` and `sustain S(v)`
- ▶ sequence: `p; q`
- ▶ parallel: `p || q`
- ▶ loops
 - ▶ `loop p end`
 - ▶ `loop p each S`
 - ▶ `every [immediate] S do p end`

Esterel Statements Discussed So Far

- ▶ `emit S` and `emit S(v)`
- ▶ `sustain S` and `sustain S(v)`
- ▶ sequence: `p; q`
- ▶ parallel: `p || q`
- ▶ loops
 - ▶ `loop p end`
 - ▶ `loop p each S`
 - ▶ `every [immediate] S do p end`
- ▶ `await [immediate] S`

Esterel Statements Discussed So Far

- ▶ `emit S` and `emit S(v)`
- ▶ `sustain S` and `sustain S(v)`
- ▶ sequence: `p; q`
- ▶ parallel: `p || q`
- ▶ loops
 - ▶ `loop p end`
 - ▶ `loop p each S`
 - ▶ `every [immediate] S do p end`
- ▶ `await [immediate] S`
- ▶ `[weak] abort p when [immediate] S do q end abort`

Esterel Statements Discussed So Far

- ▶ `emit S` and `emit S(v)`
- ▶ `sustain S` and `sustain S(v)`
- ▶ sequence: `p; q`
- ▶ parallel: `p || q`
- ▶ loops
 - ▶ `loop p end`
 - ▶ `loop p each S`
 - ▶ `every [immediate] S do p end`
- ▶ `await [immediate] S`
- ▶ `[weak] abort p when [immediate] S do q end abort`
- ▶ local declarations
 - ▶ `var x: α in p end var`
 - ▶ `signal S: α in p end signal`

Further Esterel Statements

- ▶ nothing
- ▶ pause
- ▶ halt
- ▶ present S then p else q end
- ▶ if E then p else q end
- ▶ repeat n times p end repeat
- ▶ suspend p when [immediate] S
- ▶ trap T in p end trap with exit T
- ▶ call $P(x_1, \dots, x_n)(v_1, \dots, v_m)$
- ▶ exec $P(x_1, \dots, x_n)(v_1, \dots, v_m)$ return R

Further Basic Statements

- ▶ `nothing` does nothing and needs no time to do nothing

Further Basic Statements

- ▶ `nothing` does nothing and needs no time to do nothing
- ▶ `pause` waits for the next macro step

Further Basic Statements

- ▶ `nothing` does nothing and needs no time to do nothing
- ▶ `pause` waits for the next macro step
- ▶ `halt` waits for all the time, *i. e.*, `halt` \equiv

Further Basic Statements

- ▶ `nothing` does nothing and needs no time to do nothing
- ▶ `pause` waits for the next macro step
- ▶ `halt` waits for all the time, *i. e.*, `halt` \equiv `loop pause end`

Conditionals

```
present S then p else q end present
```

- ▶ if started, evaluate expression S
- ▶ if S holds, immediately execute p, otherwise q
- ▶ both the then and the else branches are optional

Conditionals

`present S then p else q end present`

- ▶ if started, evaluate expression S
- ▶ if S holds, immediately execute p, otherwise q
- ▶ both the `then` and the `else` branches are optional

More general form:

```
present  
case S_1 do p_1  
  ...  
case S_n do p_n  
else q  
end present
```

\equiv

Conditionals

```
present S then p else q end present
```

- ▶ if started, evaluate expression S
- ▶ if S holds, immediately execute p, otherwise q
- ▶ both the then and the else branches are optional

More general form:

```
present  
case S_1 do p_1  
  ...  
case S_n do p_n  
else q  
end present
```

≡

```
present S_1 then p_1  
else present S_2 then p_2  
  ...  
else present S_n then p_n  
else S_q  
end present  
...  
end present
```

Conditionals

- ▶ `if E then p else q end if`
 - ▶ if started, evaluate expression E
 - ▶ if E holds, immediately execute p, otherwise execute q

Conditionals

- ▶ `if E then p else q end if`
 - ▶ if started, evaluate expression E
 - ▶ if E holds, immediately execute p, otherwise execute q
- ▶ `present S` is restricted for signal expressions
- ▶ `if` instead checks variable values.
- ▶ **Note:** In Esterel v7, `if` may also be used as a synonym for `present`.

Process Suspension

suspend p when S

- ▶ If started at time t , p is started at time t without checking S

Process Suspension

suspend p when S

- ▶ If started at time t , p is started at time t without checking S
- ▶ If p terminates at time t , then the entire statement terminates

Process Suspension

suspend p when S

- ▶ If started at time t , p is started at time t without checking S
- ▶ If p terminates at time t , then the entire statement terminates
- ▶ Otherwise, the execution of p takes time. In all macro steps that start inside p :

Process Suspension

suspend p when S

- ▶ If started at time t , p is started at time t without checking S
- ▶ If p terminates at time t , then the entire statement terminates
- ▶ Otherwise, the execution of p takes time. In all macro steps that start inside p :
 - ▶ S is checked first

Process Suspension

suspend p when S

- ▶ If started at time t , p is started at time t without checking S
- ▶ If p terminates at time t , then the entire statement terminates
- ▶ Otherwise, the execution of p takes time. In all macro steps that start inside p :
 - ▶ S is checked first
 - ▶ If S does not hold, p is executed for this macro step

Process Suspension

suspend p when S

- ▶ If started at time t , p is started at time t without checking S
- ▶ If p terminates at time t , then the entire statement terminates
- ▶ Otherwise, the execution of p takes time. In all macro steps that start inside p :
 - ▶ S is checked first
 - ▶ If S does not hold, p is executed for this macro step
 - ▶ If S holds, the control flow rests at the current locations, and no action of p is executed
 - ▶ Hence, the **control flow is frozen** whenever S holds

Process Suspension

suspend p when S

- ▶ If started at time t , p is started at time t without checking S
- ▶ If p terminates at time t , then the entire statement terminates
- ▶ Otherwise, the execution of p takes time. In all macro steps that start inside p :
 - ▶ S is checked first
 - ▶ If S does not hold, p is executed for this macro step
 - ▶ If S holds, the control flow rests at the current locations, and no action of p is executed
 - ▶ Hence, the **control flow is frozen** whenever S holds

For comparison: in Unix, a process is aborted with

Process Suspension

suspend p when S

- ▶ If started at time t , p is started at time t without checking S
- ▶ If p terminates at time t , then the entire statement terminates
- ▶ Otherwise, the execution of p takes time. In all macro steps that start inside p :
 - ▶ S is checked first
 - ▶ If S does not hold, p is executed for this macro step
 - ▶ If S holds, the control flow rests at the current locations, and no action of p is executed
 - ▶ Hence, the **control flow is frozen** whenever S holds

For comparison: in Unix, a process is aborted with \hat{C} ,
suspended with

Process Suspension

suspend p when S

- ▶ If started at time t , p is started at time t without checking S
- ▶ If p terminates at time t , then the entire statement terminates
- ▶ Otherwise, the execution of p takes time. In all macro steps that start inside p :
 - ▶ S is checked first
 - ▶ If S does not hold, p is executed for this macro step
 - ▶ If S holds, the control flow rests at the current locations, and no action of p is executed
 - ▶ Hence, the **control flow is frozen** whenever S holds

For comparison: in Unix, a process is aborted with \hat{C} , suspended with \hat{Z} , and released again with

Process Suspension

suspend p when S

- ▶ If started at time t , p is started at time t without checking S
- ▶ If p terminates at time t , then the entire statement terminates
- ▶ Otherwise, the execution of p takes time. In all macro steps that start inside p :
 - ▶ S is checked first
 - ▶ If S does not hold, p is executed for this macro step
 - ▶ If S holds, the control flow rests at the current locations, and no action of p is executed
 - ▶ Hence, the **control flow is frozen** whenever S holds

For comparison: in Unix, a process is aborted with \hat{C} , suspended with \hat{Z} , and released again with fg

Process Suspension

Similar to abort, there are 2×2 variants:

- ▶ suspend p when S
- ▶ weak suspend p when S
- ▶ suspend p when immediate S
- ▶ weak suspend p when immediate S

Process Suspension

Immediate suspend can be transformed into non-immediate suspend:

```
suspend
```

```
  P
```

```
when immediate S
```

≡

Process Suspension

Immediate suspend can be transformed into non-immediate suspend:

```
suspend  
  P  
when immediate S
```

≡

```
suspend  
  present S then  
    pause  
  end;  
  P  
when S
```

Process Suspension

Immediate suspend can be transformed into non-immediate suspend:

```
suspend  
  P  
when immediate S
```

≡

```
suspend  
  present S then  
    pause  
  end;  
  P  
when S
```

Note: the immediate variant implies an additional control point (behaving like a pause statement) where control may rest between ticks.

```
suspend  
  nothing  
when immediate tick
```

≡

Process Suspension

Immediate suspend can be transformed into non-immediate suspend:

```
suspend  
  P  
when immediate S
```

≡

```
suspend  
  present S then  
    pause  
  end;  
  P  
when S
```

Note: the immediate variant implies an additional control point (behaving like a pause statement) where control may rest between ticks.

```
suspend  
  nothing  
when immediate tick
```

≡

```
loop  
  pause  
end loop
```

Weak Process Suspension

weak suspend p when S

- ▶ Behaves like (strong) suspend at initial tick.

Weak Process Suspension

weak suspend p when S

- ▶ Behaves like (strong) suspend at initial tick.
- ▶ In all macro steps that start inside p , S is again checked first

Weak Process Suspension

weak suspend p when S

- ▶ Behaves like (strong) suspend at initial tick.
- ▶ In all macro steps that start inside p , S is again checked first
 - ▶ If S does not hold, p is executed for this macro step

Weak Process Suspension

weak suspend p when S

- ▶ Behaves like (strong) suspend at initial tick.
- ▶ In all macro steps that start inside p , S is again checked first
 - ▶ If S does not hold, p is executed for this macro step
 - ▶ If S holds, the control flow rests at the current locations—**but the actions of p for the current tick are still executed**
 - ▶ **Note:** if S holds, the execution is still limited to p , *i. e.*, no actions following the suspend statement get executed

Weak Process Suspension

weak suspend p when S

- ▶ Behaves like (strong) suspend at initial tick.
- ▶ In all macro steps that start inside p , S is again checked first
 - ▶ If S does not hold, p is executed for this macro step
 - ▶ If S holds, the control flow rests at the current locations—**but the actions of p for the current tick are still executed**
 - ▶ **Note:** if S holds, the execution is still limited to p , *i. e.*, no actions following the suspend statement get executed

weak suspend p when immediate S

- ▶ Similar to non-immediate variant, except that S is also checked in initial tick
- ▶ Again, an additional control point gets introduced at the beginning of p where control may resume at the next tick

Weak Process Suspension

Weak suspend may hide a loop:

```
weak suspend  
  pause;  
  emit next(S(?S+1))  
when true
```

≡

Weak Process Suspension

Weak suspend may hide a loop:

```
weak suspend  
  pause;  
  emit next(S(?S+1))  
when true
```

\equiv

```
loop  
  pause;  
  emit next(S(?S+1))  
end loop
```

Resolution Functions

Signals can be emitted in one macro step with different values

~> write conflicts

Solving write conflicts by **resolution functions**

- ▶ **output 0: combine α with f**
- ▶ f is used to compute the final value by applying f to the emitted values

Resolution Functions

Signals can be emitted in one macro step with different values
→ write conflicts

Solving write conflicts by **resolution functions**

- ▶ **output 0: combine α with f**
- ▶ f is used to compute the final value by applying f to the emitted values
- ▶ **Example: output votes: combine integer with + resolves emit votes(2); emit votes(3) so that votes has value $2 + 3 = 5$**

Resolution Functions

Signals can be emitted in one macro step with different values
→ write conflicts

Solving write conflicts by **resolution functions**

- ▶ **output 0: combine α with f**
- ▶ f is used to compute the final value by applying f to the emitted values
- ▶ **Example: output votes: combine integer with $+$ resolves emit votes(2); emit votes(3) so that votes has value $2 + 3 = 5$**
- ▶ $f : \alpha \times \alpha \rightarrow \alpha$ must be commutative and associative
- ▶ Commutativity and associativity of f makes the value independent of the ordering of the values

Input Restrictions

- ▶ Compilers for synchronous languages have to analyze the program
 - ▶ Most problems are undecidable, so (conservative) heuristics have to be used
 - ▶ Known information about inputs should be given to compiler
- ↪ **input restrictions**
- ▶ **inclusion**: relation $R \rightarrow S$ means that presence of R implies presence of S
 - ▶ **exclusion**: relation $S_1 \# S_2 \# \dots \# S_n$ means that at most one of the signals S_i can be present per macro step

Input Restrictions

- ▶ Compilers for synchronous languages have to analyze the program
 - ▶ Most problems are undecidable, so (conservative) heuristics have to be used
 - ▶ Known information about inputs should be given to compiler
- ~> **input restrictions**
- ▶ **inclusion**: relation $R \rightarrow S$ means that presence of R implies presence of S
 - ▶ **exclusion**: relation $S_1 \# S_2 \# \dots \# S_n$ means that at most one of the signals S_i can be present per macro step
- ▶ **Examples**
- ▶ relation `minute -> second`
 - ▶ relation `liftup # liftdown`

Further Loops

repeat n times p end repeat

- ▶ n , an integer expression, is immediately evaluated
- ▶ then execute n times p
- ▶ p must not be instantaneous

Equivalent:

```
var i,j: integer in
  i := 0; j := n;
  signal stop in
    weak abort
      loop
        if i<j then p; i := i+1
        else emit stop
        end if
      end loop
    when stop
  end signal
end var
```

Further Loops

repeat n times p end repeat

- ▶ n , an integer expression, is immediately evaluated
- ▶ then execute n times p
- ▶ p must not be instantaneous

Equivalent:

```
var i,j: integer in
  i := 0; j := n;
  signal stop in
    weak abort
      loop
        if i<j then p; i := i+1
        else emit stop
        end if
      end loop
    when stop
  end signal
end var
```

Wait ... does this work?

Further Loops

repeat n times p end repeat

- ▶ n , an integer expression, is immediately evaluated
- ▶ then execute n times p
- ▶ p must not be instantaneous

Equivalent:

```
var i,j: integer in
  i := 0; j := n;
  signal stop in
    weak abort
      loop
        if i<j then p; i := i+1
        else emit stop
        end if
      end loop
    when stop
  end signal
end var
```

Wait ... does this work?
No—this is a (potentially)
instantaneous loop.

Further Loops

repeat n times p end repeat

- ▶ n , an integer expression, is immediately evaluated
- ▶ then execute n times p
- ▶ p must not be instantaneous

Equivalent:

```
var i,j: integer in
  i := 0; j := n;
  signal stop in
    weak abort
      loop
        if i<j then p; i := i+1
        else emit stop
        end if
      end loop
    when stop
  end signal
end var
```

Wait ... does this work?
No—this is a (potentially)
instantaneous loop.
How would you fix it?

Further Loops

repeat n times p end repeat

- ▶ n , an integer expression, is immediately evaluated
- ▶ then execute n times p
- ▶ p must not be instantaneous

Equivalent:

```
var i,j: integer in
  i := 0; j := n;
  signal stop in
    weak abort
      loop
        if i<j then p; i := i+1
        else emit stop
        end if
      end loop
    when stop
  end signal
end var
```

Wait ... does this work?
No—this is a (potentially)
instantaneous loop.
How would you fix it?
Add a pause statement
after emit stop

Further Await Statements

`await [immediate] S` can be generalized as follows:

```
await [immediate]  
case S_1 do p_1  
...  
case S_n do p_n  
end await
```

$::=$

Further Await Statements

`await [immediate] S` can be generalized as follows:

```
await [immediate]
case S_1 do p_1
...
case S_n do p_n
end await
```

$:\equiv$

```
await [immediate] S_1 or ... or S_n;
present
case S_1 do p_1
...
case S_n do p_n
end present
```

Further Abort Statements

[weak] abort p when S do q can be generalized as follows:

```
[weak] abort p when  
case S_1 do p_1  
...  
case S_n do p_n  
end abort
```

≡

Further Abort Statements

[weak] abort p when S do q can be generalized as follows:

```
[weak] abort p when  
case S_1 do p_1  
...  
case S_n do p_n  
end abort
```

≡

```
[weak] abort p when S_1 or ... or S_n  
do  
  present  
  case S_1 do p_1  
  ...  
  case S_n do p_n  
  end present  
end abort
```

Priorities of Nested Aborts

- ▶ Nested aborts have different priorities
- ▶ Example:

```
abort
  abort
    p
  when S_1 do
    e
  end abort
when S_2
end abort
```

- ▶ If control is inside `p`, and both `S_1` and `S_2` hold, then `e` is not executed, since **the outer abortion has priority**
- ▶ Question: what happens if one or the other is weak? Try it!

Trap Statements

```
trap T in p end trap with exit T
```

- ▶ `exit T` is similar to `emit T`, but refers to the trap `T`
- ▶ when the statement is started, `p` starts immediately
- ▶ if `exit T` is executed inside `p`, `p` is immediately aborted

Trap Statements

`trap T in p end trap with exit T`

- ▶ `exit T` is similar to `emit T`, but refers to the trap `T`
- ▶ when the statement is started, `p` starts immediately
- ▶ if `exit T` is executed inside `p`, `p` is immediately aborted

Differences to abort:

- ▶ `exit T` can only be executed within `p` (due to scope of `T`)
 - ▶ abortion due to trap is neither really weak nor really strong
 - ▶ instead: 'asynchronous abortion'
 - ▶ `exit T` works like a `goto` in that those micro steps are executed up to the micro step where `exit T` is executed, but no further ones
- ~> `exit T` terminates the trap statement

Trap vs. Abort

| P_1 | P_2 | P_3 | P_4 |
|---|--|---|--|
| <pre>trap T in emit A; exit T; emit B; end trap</pre> | <pre>signal T in weak abort emit A; emit T; emit B; when T end</pre> | <pre>signal T in abort emit A; emit T; emit B; when immediate T end</pre> | <pre>signal T in weak abort emit A; emit T; emit B; when immediate T end</pre> |
| Emitted Signals: | | | |
| | | | |

Trap vs. Abort

| P_1 | P_2 | P_3 | P_4 |
|---|--|---|--|
| <pre> trap T in emit A; exit T; emit B; end trap </pre> | <pre> signal T in weak abort emit A; emit T; emit B; when T end </pre> | <pre> signal T in abort emit A; emit T; emit B; when immediate T end </pre> | <pre> signal T in weak abort emit A; emit T; emit B; when immediate T end </pre> |
| Emitted Signals: | | | |
| {A} | | | |

Trap vs. Abort

| P_1 | P_2 | P_3 | P_4 |
|---|--|---|--|
| <pre>trap T in emit A; exit T; emit B; end trap</pre> | <pre>signal T in weak abort emit A; emit T; emit B; when T end</pre> | <pre>signal T in abort emit A; emit T; emit B; when immediate T end</pre> | <pre>signal T in weak abort emit A; emit T; emit B; when immediate T end</pre> |
| Emitted Signals: | | | |
| {A} | {A,B} | | |

Trap vs. Abort

| P_1 | P_2 | P_3 | P_4 |
|---|--|---|--|
| <pre> trap T in emit A; exit T; emit B; end trap </pre> | <pre> signal T in weak abort emit A; emit T; emit B; when T end </pre> | <pre> signal T in abort emit A; emit T; emit B; when immediate T end </pre> | <pre> signal T in weak abort emit A; emit T; emit B; when immediate T end </pre> |
| Emitted Signals: | | | |
| {A} | {A,B} | ⊥ | |

Trap vs. Abort

| P_1 | P_2 | P_3 | P_4 |
|---|--|---|--|
| <pre> trap T in emit A; exit T; emit B; end trap </pre> | <pre> signal T in weak abort emit A; emit T; emit B; when T end </pre> | <pre> signal T in abort emit A; emit T; emit B; when immediate T end </pre> | <pre> signal T in weak abort emit A; emit T; emit B; when immediate T end </pre> |
| Emitted Signals: | | | |
| {A} | {A,B} | \perp | {A,B} |

P_3 is inconsistent:

Trap vs. Abort

| P_1 | P_2 | P_3 | P_4 |
|---|--|---|--|
| <pre> trap T in emit A; exit T; emit B; end trap </pre> | <pre> signal T in weak abort emit A; emit T; emit B; when T end </pre> | <pre> signal T in abort emit A; emit T; emit B; when immediate T end </pre> | <pre> signal T in weak abort emit A; emit T; emit B; when immediate T end </pre> |
| Emitted Signals: | | | |
| {A} | {A,B} | \perp | {A,B} |

P_3 is inconsistent:

it is aborted due to the emission of T, thus, T can not be emitted

Trap vs. Abort

- ▶ Is this a solution?

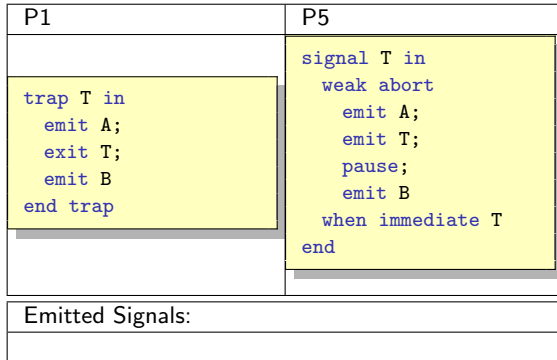
```
trap T in  
  P  
end
```



```
signal T in  
  weak abort  
    p[exit T / emit T; pause]  
  when immediate T  
end
```

- ▶ `p[exit T / emit T; pause]` means: `exit T` is replaced by `emit T; pause`
- ▶ The control flow will never rest on this pause statement, since the abort will instantaneously take place

Trap vs. Abort



Trap vs. Abort

| P1 | P5 |
|---|---|
| <pre>trap T in emit A; exit T; emit B; end trap</pre> | <pre>signal T in weak abort emit A; emit T; pause; emit B; when immediate T end</pre> |
| Emitted Signals: | |
| {A} | {A} |

that works!, however, ...

Trap vs. Abort

| P_problem | P_problem' |
|--|---|
| <pre>trap T_1 in trap T_2 in exit T_1 exit T_2 end trap emit A end trap</pre> | <pre>signal T_1 in weak abort signal T_2 in weak abort emit T_1; pause emit T_2; pause when immediate T_2 end signal; emit A when immediate T_1 end signal</pre> |

Emitted Signals:

Trap vs. Abort

| P_problem | P_problem' |
|--|---|
| <pre> trap T_1 in trap T_2 in exit T_1 exit T_2 end trap emit A end trap </pre> | <pre> signal T_1 in weak abort signal T_2 in weak abort emit T_1; pause emit T_2; pause when immediate T_2 end signal; emit A when immediate T_1 end signal </pre> |

Emitted Signals:

{ }

Trap vs. Abort

| P_problem | P_problem' |
|--|---|
| <pre> trap T_1 in trap T_2 in exit T_1 exit T_2 end trap emit A end trap </pre> | <pre> signal T_1 in weak abort signal T_2 in weak abort emit T_1; pause emit T_2; pause when immediate T_2 end signal; emit A when immediate T_1 end signal </pre> |
| Emitted Signals: | |
| { } | { A } |

Trap vs. Abort

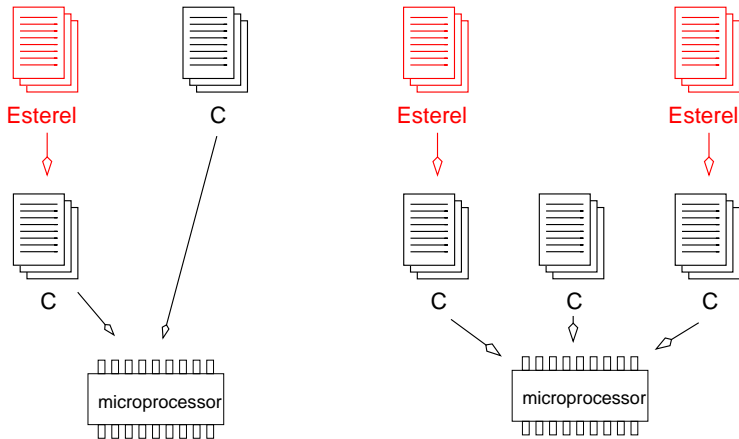
| P_problem | P_problem' |
|--|---|
| <pre> trap T_1 in trap T_2 in exit T_1 exit T_2 end trap emit A end trap </pre> | <pre> signal T_1 in weak abort signal T_2 in weak abort emit T_1; pause emit T_2; pause when immediate T_2 end signal; emit A when immediate T_1 end signal </pre> |
| Emitted Signals: | |
| { } | { A } |

- ▶ If started, P_problem exits both T_1 and T_2
- ▶ The trap with the highest (outermost) priority (T_1) is raised
- ▶ Hence, A is not emitted by P_problem, but is emitted by P_problem'
- ▶ Trap and abort have different priority schemes
- ▶ How can this be repaired?

Esterel and the Host Language

- ▶ Esterel has only a few data types
- ▶ Data types and functions can be imported from host languages
- ▶ Esterel programs are translated to the host language
- ▶ Esterel mainly cares about compiling multi-threaded programs to a single thread
- ▶ To this end, all thread interaction is handled at compile time
- ▶ After successful compilation, the programs are free of runtime errors due to concurrency like write conflicts and deadlocks
- ▶ **The result is a deterministic system**
(rather unusual for multi-threaded systems)

Esterel and the Host Language (Software)



Host Language

- ▶ Esterel (v5) does not implement many data types
has only boolean, integer, float, and string
- ▶ There are no means to define new data types
- ▶ or simple (instantaneous) functions on user-defined data types
- ▶ **However:**
 - ▶ Esterel programs are translated to program of a **host language**
 - ▶ for software, often C is used
 - ▶ obtained C program can be linked with other C programs
- ▶ **Esterel can import data types, functions and procedures from the host language**

Imported Data Types and Functions

- ▶ **type** α imports a data type from host language
- ▶ This type must be implemented in the host language
- ▶ **function** $f(\alpha_1, \dots, \alpha_n) : \alpha$ imports a function
- ▶ Esterel is able to perform type checking, but knows nothing else of f
- ▶ Arguments are passed-by-value
- ▶ Functions f must not have side effects
- ▶ Functions are used to generate expressions
- ▶ Therefore, **function calls are instantaneous**

Imported Procedures

- ▶ **procedure** $P(\alpha_1, \dots, \alpha_n)(\beta_1, \dots, \beta_m)$ imports a procedure from host language with types α_i and β_i
- ▶ Arguments of first argument list are given with call-by-reference
- ▶ Arguments of second argument list are given with call-by-value
- ▶ Procedures have no return value, but can change the variables that were given in the first argument list
- ▶ Procedure calls **call** $P(x_1, \dots, x_n)(\tau_1, \dots, \tau_m)$ **are instantaneous**

Imported Tasks

- ▶ `task P($\alpha_1, \dots, \alpha_n$)(β_1, \dots, β_m)` imports a task from host language with types α_j and β_j
 - ▶ Arguments are the same as with procedures
 - ▶ `exec P(x_1, \dots, x_n)(τ_1, \dots, τ_m) return R` executes task p , which may not be instantaneous
 - ▶ The `exec` statement terminates when the task terminates;
- Tasks are not instantaneous**

Imported Tasks

- ▶ **task** $P(\alpha_1, \dots, \alpha_n)(\beta_1, \dots, \beta_m)$ imports a task from host language with types α_i and β_j
 - ▶ Arguments are the same as with procedures
 - ▶ **exec** $P(x_1, \dots, x_n)(\tau_1, \dots, \tau_m)$ **return** R executes task p , which may not be instantaneous
 - ▶ The **exec** statement terminates when the task terminates;
- Tasks are not instantaneous**
- ▶ P runs in parallel with Esterel threads
 - ▶ P may correspond to a C-program, or also to a physical process (“Robot drives distance X ”)
 - ▶ No interaction with Esterel threads, except for termination of P
 - ▶ Termination of p is signaled by R
 - ▶ R is a **return signal**, declared at module interface analogous to input/output signals

Abortion of Tasks

```
abort  
  exec P(X) (23) return R  
when S
```

- ▶ If R holds before S, then X is updated and the abort terminates
- ▶ If S holds before R, then task P is aborted and X is not updated
- ▶ If R and S both hold, then the abort terminates and X is not updated
- ▶ Using weak abort allows to update X

Multiple Task Execution

```
exec
  case T_1 ... return R_1 do p_1
  ...
  case T_n ... return R_n do p_n
end exec
```

- ▶ When started, all tasks T_1, \dots, T_n are concurrently started
- ▶ When at least one return signal occurs:
 - ▶ Let R_i be the first return signal in the case-list that is present
 - ▶ Update only reference arguments corresponding to R_i
 - ▶ Abort all non-terminated tasks

Overview

A Tour through Esterel

Further Esterel Statements

The Kernel Language

Kernel Language

- ▶ Many Esterel statements p can be viewed as macros
- ▶ Important: write-things-once-principle (WTO)
- ~> guarantees expanded statements of size $O(\|p\|)$
- ▶ For programming, redundant statements (called **syntactic sugar**) are important to directly express what is meant
- ▶ However, compilation should be based on few constructs
- ~> *using small kernel language*

Kernel Language: Esterel

| | | |
|--|-------------------------------|----------------------------|
| | <code>nothing</code> | (empty statement) |
| | <code>pause</code> | (separation of macro step) |
| | <code>emit S</code> | (signal emission) |
| <code>present S then p else q end</code> | | (conditional) |
| | <code>suspend p when S</code> | (process suspension) |
| | <code>p;q</code> | (sequence) |
| | <code>p q</code> | (synchronous concurrency) |
| | <code>loop p end</code> | (infinite loop) |
| | <code>trap T in p end</code> | (exception handling) |
| | <code>exit T</code> | (exception raising) |
| <code>signal S in p end</code> | | (local declarations) |

Summary

- ▶ The ABR0 example, the “hello world” of Esterel, illustrates reactive control flow
- ▶ Traps are similar to weak aborts, but there are subtle differences
- ▶ Esterel can be thought of as a “coordination language” that allows deterministic concurrency and preemption, while much of the computational details is left to a host language (typically C)
- ▶ All Esterel statements can be derived from a few kernel statements

To Go Further

- ▶ Gérard Berry, The Esterel v5 Language Primer, Version v5_91, 2000
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.8212>