Synchronous Languages—Lecture 02

Prof. Dr. Reinhard von Hanxleden

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

24 October 2016

Last compiled: November 7, 2016, 8:51 hrs

Esterel I—Overview

Overview

Introduction to Esterel

- Imperative, textual language
- Concurrent
- Based on synchronous model of time
  - Program execution synchronized to an external clock
  - Like synchronous digital logic
  - Suits the cyclic executive approach

Thanks to Stephen Edwards (Columbia U), Klaus Schneider (U Kaiserslautern) and Gerald Luettgen (U Bamberg) for providing part of the following material

History

- Developed at Centre de Mathématiques Appliquées (CMA), Ecole des Mines de Paris
- J.-P. Marmorat and J.-P. Rigault built an autonomous vehicle
- They were not satisfied by traditional programming languages (no adequate support for reactive control flow, non-determinism due to language and/or OS)

$\leadsto$ and developed a first version of Esterel

- Estérel is a mountain area between Cannes and St. Raphaël, the name sounds like "real-time" in french (*temps-réel*)
- G. Berry developed a formal semantics for Esterel

Esterel Dialects

- Esterel v5: Has been stable since late 1990s
- Esterel v7: same principles as in v5, several extensions (e. g., multi-clock designs, refined type system). There is an IEEE standardization draft.

Esterel Dialects

- Esterel v5: Has been stable since late 1990s
- Esterel v7: same principles as in v5, several extensions (e. g., multi-clock designs, refined type system). There is an IEEE standardization draft.
- Sequentially Constructive Esterel (SCEst): Extension of Esterel, based on Sequentially Constructive Model of Computation (SC MoC)

Graphical Variants

There are several graphical languages following a similar MoC as Esterel, using a Statechart-like syntax:

- Argos: first graphical language
- SyncCharts : successor of Argos
- Safe State Machines (SSMs): equivalent to SyncCharts, the name of the modeling language supported by the commercial tool Esterel Studio, which uses Esterel as intermediate step in code generation

Graphical Variants

There are several graphical languages following a similar MoC as Esterel, using a Statechart-like syntax:

- Argos: first graphical language
- SyncCharts : successor of Argos
- Safe State Machines (SSMs): equivalent to SyncCharts, the name of the modeling language supported by the commercial tool Esterel Studio, which uses Esterel as intermediate step in code generation
- Sequentially Constructive Statecharts (SCCharts): Extension of SyncCharts/SSMs based on SC MoC
- In this class, we will mainly consider Esterel v5, SCEst and SCCharts

Signals

- Esterel programs/SSMs communicate through signals

Signals

- Esterel programs/SSMs communicate through signals
- These are like wires
  - Each signal is either present or absent in each tick
  - Can't take multiple values within a tick

Signals

- Esterel programs/SSMs communicate through signals
- These are like wires
  - Each signal is either present or absent in each tick
  - Can't take multiple values within a tick
- Presence/absence not held between ticks

Signals

- Esterel programs/SSMs communicate through signals
- These are like wires
  - Each signal is either present or absent in each tick
  - Can't take multiple values within a tick
- Presence/absence not held between ticks
- Broadcast across the program
  - Any process can read or write a signal

Signals

- Status of an input signal is determined by input event, and by local emissions

Signals

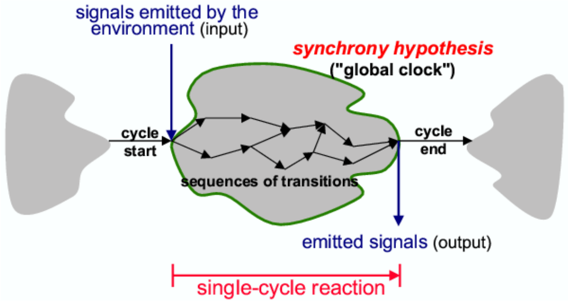
- Status of an input signal is determined by input event, and by local emissions
- Status of local or output signal is determined per tick
  - Default status: absent
  - Must execute an "`emit S`" statement to set signal `S` present

Signals

- Status of an input signal is determined by input event, and by local emissions
- Status of local or output signal is determined per tick
  - Default status: absent
  - Must execute an "`emit S`" statement to set signal `S` present
- `await A`:
  - Waits for A and terminates when A occurs

Synchrony Hypothesis

- Computations are considered to
  - take no time
  - be atomic



G. Luettgen 2001

Perfect Synchrony

Definition [Perfect Synchrony]
A system works in perfect synchrony, if all reactions of the system are executed in zero time. Hence, outputs are generated at the same time, when the inputs are read.

- Of course, this is only an idealized programmer's model
- In practice, 'zero time' means before the next interaction
- Physical time between interactions may not always be the same
- Synchronous programs use natural numbers for *logical time*, where only interactions, *i. e.*, macro steps, are counted

Synchronous Model of Computation

To summarize: the synchronous model of computation of SSMs/Esterel is characterized by:

1. Computations considered to take no time (synchrony hypothesis)
2. Time is divided into discrete ticks
3. Signals are either present or absent in each tick

Synchronous Model of Computation

To summarize: the synchronous model of computation of SSMs/Esterel is characterized by:

1. Computations considered to take no time (synchrony hypothesis)
2. Time is divided into discrete ticks
3. Signals are either present or absent in each tick

Sometimes, "synchrony" refers to just the first two points (*e.g.*, in the original Statecharts as implemented in Statemate); to explicitly include the third requirement as well, we also speak of the strict synchrony

Perfect Synchrony and Worst-Case Execution Time

- *When are real-time constraints considered?*
- Macro steps consist of *only finitely many micro steps*, *i. e.*, there are no data dependent loops in a macro step
- Hence, the runtime of a single macro step can be easily checked (at least compared to non-synchronous languages) for a specific platform (processor)

⇝ Low-level worst case execution time analysis (WCET), also called worst case reaction time analysis (WCRT)

- Additionally, one can check how many macro steps are required from one system state to another (high-level WCET analysis)

The Multiform Notion of Time

- Some "classical" programming languages already include a concept of real-time
- Consider the following Ada code fragment, which signals minutes to a task B:

```
loop
  delay 60;
  B.Minute
end
```

- This works in principle
- However, it is not deterministic!

The Multiform Notion of Time

- A design goal of synchronous languages:
  - Fully deterministic behavior
  - Applies to functionality and (logical) timing

The Multiform Notion of Time

- A design goal of synchronous languages:
  - Fully deterministic behavior
  - Applies to functionality and (logical) timing
- Approach:
  - Replace notion of *physical time* with notion of *order*
  - Only consider *simultaneity* and *precedence* of events

The Multiform Notion of Time

- A design goal of synchronous languages:
  - Fully deterministic behavior
  - Applies to functionality and (logical) timing
- Approach:
  - Replace notion of *physical time* with notion of *order*
  - Only consider *simultaneity* and *precedence* of events
- Hence, physical time does not play any special role
  - Is handled like any other event from program environment
  - This is called multiform notion of time

The Multiform Notion of Time

- Consider following requirements:
  - "The train must stop within 10 seconds"
  - "The train must stop within 100 meters"

The Multiform Notion of Time

- Consider following requirements:
  - "The train must stop within 10 seconds"
  - "The train must stop within 100 meters"
- These are conceptually of the same nature!

The Multiform Notion of Time

- Consider following requirements:
  - "The train must stop within 10 seconds"
  - "The train must stop within 100 meters"
- These are conceptually of the same nature!
- In languages where physical time plays particular role, these requirements are typically expressed completely differently
- In synchronous model, use similar precedence constraints:
  - "The event stop must precede the 10th (respectively, 100th) next occurrence of the event second (respectively, meter)"

The Multiform Notion of Time

- History of system is a totally ordered sequence of logical ticks

The Multiform Notion of Time

- History of system is a totally ordered sequence of logical ticks
- At each tick, an arbitrary number of events (including 0) occurs

The Multiform Notion of Time

- History of system is a totally ordered sequence of logical ticks
- At each tick, an arbitrary number of events (including 0) occurs
- Event occurrences that happen at the same logical tick are considered simultaneous
- Other events are ordered as their instances of occurrences

Basic Esterel Statements

`emit S`

- Make signal `S` present in the current instant
- A signal is absent unless it is emitted

Basic Esterel Statements

`emit S`

- Make signal S present in the current instant
- A signal is absent unless it is emitted

`pause`

- Stop and resume after the next cycle after the pause

Basic Esterel Statements

`emit S`

- Make signal `S` present in the current instant
- A signal is absent unless it is emitted

`pause`

- Stop and resume after the next cycle after the pause

`present S then` *stmt1* `else` *stmt2* `end`

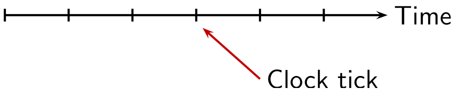
- If signal `S` is present in the current instant, immediately run *stmt1*, otherwise run *stmt2*

Esterel's Model of Time

- The standard CS model (e.g., Java's) is asynchronous
  - Threads run at their own rate
  - Synchronization is done (for example) through calls to `wait()` and `notify()`

Esterel's Model of Time

- The standard CS model (e.g., Java's) is asynchronous
  - Threads run at their own rate
  - Synchronization is done (for example) through calls to `wait()` and `notify()`
- Esterel's model of time is synchronous like that used in hardware. Threads march in lockstep to a global clock.

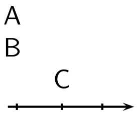


Basic Esterel Statements

```
module EXAMPLE1:
output A, B, C;

emit A;
present A then emit B end;
pause;
emit C

end module
```

Basic Esterel Statements

```
module EXAMPLE1:
output A, B, C;

emit A;
present A then emit B end;
pause;
emit C

end module
```



EXAMPLE1 makes signals A & B present the first instant, C present the second

Signal Coherence Rules

- Each signal is only present or absent in a cycle, never both
- All writers run before any readers do
- Thus

```
present A else
   emit A
end
```

is an erroneous program

Advantage of Synchrony

- Easy to control time
- Synchronization comes for free

Advantage of Synchrony

- Easy to control time
- Synchronization comes for free
- Speed of actual computation nearly uncontrollable
- Allows function and timing to be specified independently

Advantage of Synchrony

- Easy to control time
- Synchronization comes for free
- Speed of actual computation nearly uncontrollable
- Allows function and timing to be specified independently
- Makes for deterministic concurrency
- Explicit control of "before" "after" "at the same time"

Time Can Be Controlled Precisely

This guarantees every 60th S an M is emitted:

```
every 60 S do
  emit M
end
```

`every` invokes its body every 60th S

`emit` takes no time (cycles)

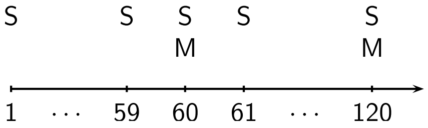
Time Can Be Controlled Precisely

This guarantees every 60th S an M is emitted:

```
every 60 S do
  emit M
end
```

`every` invokes its body every 60th `S`

`emit` takes no time (cycles)



The || Operator

Groups of statements separated by || run concurrently and terminate when all groups have terminated

The || Operator

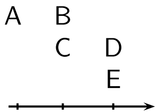
Groups of statements separated by || run concurrently and terminate when all groups have terminated

```
[
  emit A;
  pause; emit B;
||
  pause; emit C;
  pause; emit D
];
emit E
```

The || Operator

Groups of statements separated by || run concurrently and terminate when all groups have terminated

```
[
  emit A;
  pause; emit B;
||
  pause; emit C;
  pause; emit D
];
emit E
```



Communication Is Instantaneous

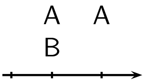
A signal emitted in a cycle is visible immediately

```
[
  pause; emit A;
  pause; emit A
||
  pause;
  present A then
    emit B end
]
```

Communication Is Instantaneous

A signal emitted in a cycle is visible immediately

```
[
  pause; emit A;
  pause; emit A
||
  pause;
  present A then
    emit B end
]
```



Bidirectional Communication

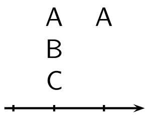
Processes can communicate back and forth in the same cycle

```
[
  pause; emit A;
  present B then
    emit C end;
  pause; emit A
||
  pause;
  present A then
    emit B end
]
```

Bidirectional Communication

Processes can communicate back and forth in the same cycle

```
[
  pause; emit A;
  present B then
    emit C end;
  pause; emit A
||
  pause;
  present A then
    emit B end
]
```



Concurrency and Determinism

- Signals are the only way for concurrent processes to communicate

Concurrency and Determinism

- Signals are the only way for concurrent processes to communicate
- Esterel does have variables, which (unlike signals) can be sequentially modified within a tick, but they cannot be shared

Concurrency and Determinism

- Signals are the only way for concurrent processes to communicate
- Esterel does have variables, which (unlike signals) can be sequentially modified within a tick, but they cannot be shared
- **Signal coherence rules ensure deterministic behavior**
- Language semantics clearly defines who must communicate with whom when

The Await Statement

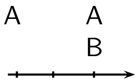
- The `await` statement waits for a particular cycle
- `await S` waits for the **next** cycle in which `S` is present

```
[
  emit A;
  pause;
  pause; emit A
||
  await A; emit B
]
```

The Await Statement

- The `await` statement waits for a particular cycle
- `await S` waits for the **next** cycle in which `S` is present

```
[
  emit A;
  pause;
  pause; emit A
||
  await A; emit B
]
```



The Await Statement

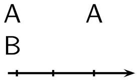
- `await` normally waits for a cycle before beginning to check
- `await immediate` also checks the **initial** cycle

```
[
  emit A;
  pause;
  pause; emit A
||
  await immediate A;
  emit B
]
```

The Await Statement

- await normally waits for a cycle before beginning to check
- await immediate also checks the **initial** cycle

```
[
  emit A;
  pause;
  pause; emit A
||
  await immediate A;
  emit B
]
```



Loops

- Esterel has an infinite loop statement

Loops

- Esterel has an infinite loop statement
- Rule: loop body cannot terminate instantly
  - Needs at least one `pause`, `await`, etc.

Loops

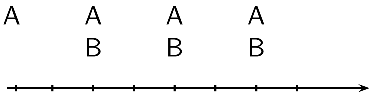
- Esterel has an infinite loop statement
- Rule: loop body cannot terminate instantly
  - Needs at least one `pause`, `await`, etc.
  - Can't do an infinite amount of work in a single cycle

```
loop
  emit A;
  pause;
  pause;
  emit B
end
```

Loops

- Esterel has an infinite loop statement
- Rule: loop body cannot terminate instantly
  - Needs at least one `pause`, `await`, etc.
  - Can't do an infinite amount of work in a single cycle

```
loop
  emit A;
  pause;
  pause;
  emit B
end
```



Loops and Synchronization

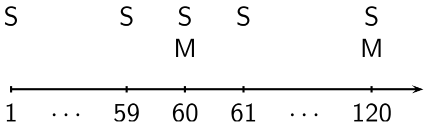
Instantaneous nature of loops plus `await` provide very powerful synchronization mechanisms

```
loop
  await 60 S;
  emit M
end
```

Loops and Synchronization

Instantaneous nature of loops plus `await` provide very powerful synchronization mechanisms

```
loop
  await 60 S;
  emit M
end
```



Preemption

- Often want to stop doing something and start doing something else

Preemption

- Often want to stop doing something and start doing something else
- E.g., Ctrl-C in Unix: stop the currently-running program

Preemption

- Often want to stop doing something and start doing something else
- E.g., Ctrl-C in Unix: stop the currently-running program
- Esterel has many constructs for handling preemption

The Abort Statement

- Basic preemption mechanism
- General form:

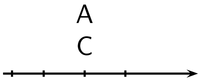
```
abort
  statement
when condition
```

- Runs *statement* to completion
- If *condition* ever holds, abort terminates immediately.

The Abort Statement

```
abort
  pause;
  pause;
  emit A
when B;
emit C
```

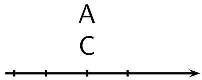
The Abort Statement



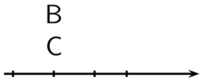
Normal termination

```
abort
  pause;
  pause;
  emit A
when B;
emit C
```

The Abort Statement



Normal termination

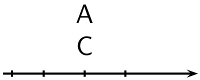


Aborted termination

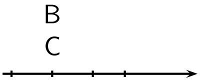
```
abort
  pause;
  pause;
  emit A
when B;
emit C
```

The Abort Statement

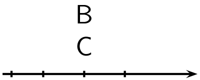
```
abort
  pause;
  pause;
  emit A
when B;
emit C
```



Normal termination



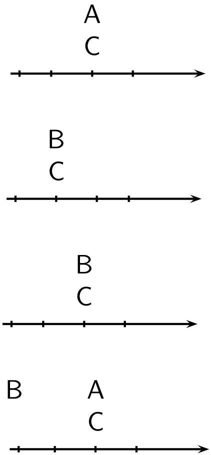
Aborted termination



Aborted termination;
`emit A` preempted

The Abort Statement

```
abort
  pause;
  pause;
  emit A
when B;
emit C
```



Normal termination

Aborted termination

Aborted termination; `emit A` preempted

Normal termination
`B` not checked in first cycle
(like `await`)

Strong vs. Weak Preemption

- Strong preemption:
  - The body does not run when the preemption condition holds
  - The previous example illustrated strong preemption

Strong vs. Weak Preemption

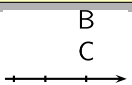
- Strong preemption:
  - The body does not run when the preemption condition holds
  - The previous example illustrated strong preemption
- Weak preemption:
  - The body is allowed to run even when the preemption condition holds, but is terminated thereafter
  - `weak abort` implements this in Esterel

Strong vs. Weak Abort

```
abort
  pause;
  pause;
  emit A;
  pause
when B;
emit C
```

Strong vs. Weak Abort

```
abort
  pause;
  pause;
  emit A;
  pause
when B;
emit C
```

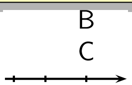


`emit A` not allowed to run

Strong vs. Weak Abort

```
abort
  pause;
  pause;
  emit A;
  pause
when B;
emit C
```

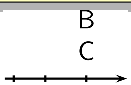
```
weak abort
  pause;
  pause;
  emit A;
  pause
when B;
emit C
```



emit A not allowed to run

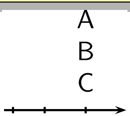
Strong vs. Weak Abort

```
abort
  pause;
  pause;
  emit A;
  pause
when B;
emit C
```



emit A not allowed to run

```
weak abort
  pause;
  pause;
  emit A;
  pause
when B;
emit C
```



emit A does run, body terminated afterwards

Strong vs. Weak Preemption

- Important distinction
- Something cannot cause its own strong preemption

Strong vs. Weak Preemption

- Important distinction
- Something cannot cause its own strong preemption

```
abort
  pause;
  emit A
when A
```

Strong vs. Weak Preemption

- Important distinction
- Something cannot cause its own strong preemption

```
abort
  pause;
  emit A
when A
```

Erroneous!

Strong vs. Weak Preemption

- Important distinction
- Something cannot cause its own strong preemption

```
abort
  pause;
  emit A
when A
```

Erroneous!

```
weak abort
  pause;
  emit A
when A
```

Strong vs. Weak Preemption

- Important distinction
- Something cannot cause its own strong preemption

```
abort
  pause;
  emit A
when A
```

Erroneous!

```
weak abort
  pause;
  emit A
when A
```

Ok!

Nested Preemption

```
module RUNNER
input LAP, METER, MORNING, SECOND, STEP;
output ... ;

every MORNING do
  abort
    loop
      abort run RUNSLOWLY when 15 SECOND;
      abort
        every STEP do
          run JUMP || run BREATHE
        end every
      when 100 METER;
      run FULLSPEED
    each LAP
  when 2 LAP
end every
end module
```

Exceptions—The Trap Statement

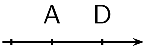
- Esterel provides an exception facility for *weak* preemption
- Interacts nicely with concurrency
- Rule: outermost trap takes precedence

The Trap Statement

```
trap T in
[
  pause;
  emit A;
  pause;
  exit T
||
  await B;
  emit C
]
end trap;
emit D
```

The Trap Statement

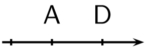
```
trap T in
[
  pause;
  emit A;
  pause;
  exit T
||
  await B;
  emit C
]
end trap;
emit D
```



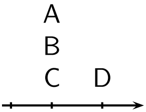
Normal termination from first process

The Trap Statement

```
trap T in
[
  pause;
  emit A;
  pause;
  exit T
||
  await B;
  emit C
]
end trap;
emit D
```



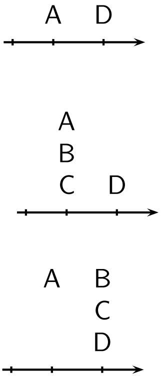
Normal termination from first process



emit C also runs

The Trap Statement

```
trap T in
[
  pause;
  emit A;
  pause;
  exit T
||
  await B;
  emit C
]
end trap;
emit D
```



Normal termination from first process

`emit C` also runs

Second process allowed to run even though first process has exited

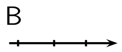
Nested Traps

```
trap T1 in
  trap T2 in
  [
    exit T1
    ||
    exit T2
  ]
  end;
  emit A
end;
emit B
```

Nested Traps

```
trap T1 in
  trap T2 in
  [
    exit T1
    ||
    exit T2
  ]
  end;
  emit A
end;
emit B
```

- Outer trap takes precedence; control transferred directly to the outer trap statement.
- `emit A` not allowed to run.



Combining Abortion and Exceptions

```
trap HEARTATTACK in
  abort
    loop
      abort RUNSLOWLY when 15 SECOND;
      abort
        every STEP do
          JUMP || BREATHE || CHECKHEART
        end every
      when 100 METER;
      FULLSPEED
    each LAP
  when 2 LAP
handle HEARTATTACK do
  GOTOHOSPITAL
end trap
```

The Suspend Statement

- Preemption (`abort`, `trap`) terminate something, but what if you want to pause it?

The Suspend Statement

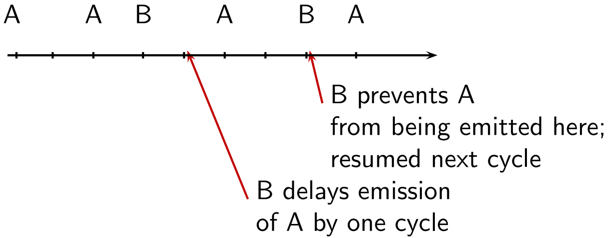
- Preemption (`abort`, `trap`) terminate something, but what if you want to pause it?
- Like the POSIX Ctrl-Z

The Suspend Statement

- Preemption (`abort`, `trap`) terminate something, but what if you want to pause it?
- Like the POSIX Ctrl-Z
- Esterel's `suspend` statement pauses the execution of a group of statements
- Only strong preemption: statement does not run when condition holds

The Suspend Statement

```
suspend
  loop
    emit A;
    pause;
    pause
  end
when B
```



Summary

- Esterel assumes perfect synchrony, with reactions discretized into *ticks*
- Information in Esterel is passed via broadcast of *signals*, which (unlike in SCEst) cannot be sequentially updated within a tick
- Esterel includes various preemption mechanisms
- Distinguish *strong* and *weak* preemption
- Orthogonally distinguish *delayed* (default) and *immediate* preemption

To Go Further

- Gérard Berry, The Foundations of Esterel, Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte, editors, MIT Press, *Foundations of Computing Series*, 2000, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.6221`
- Gérard Berry, The Esterel v5 Language Primer, Version v5_91, 2000 `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.8212`