Synchronous Languages—Lecture 05

Prof. Dr. Reinhard von Hanxleden

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

12 Nov 2018

Last compiled: November 18, 2018, 16:53 hrs

Esterel III—The Logical Semantics
The 5-Minute Review Session
The 5-Minute Review Session

1. How do concurrent threads in Esterel communicate?
The 5-Minute Review Session

1. How do concurrent threads in Esterel communicate?
2. What is the difference between weak and strong abortion?
The 5-Minute Review Session

1. How do concurrent threads in Esterel communicate?
2. What is the difference between *weak* and *strong* abortion?
3. What is the difference between *aborts* and *traps*?
The 5-Minute Review Session

1. How do concurrent threads in Esterel communicate?
2. What is the difference between weak and strong abortion?
3. What is the difference between aborts and traps?
4. What is syntactic sugar, and what is it good for?
The 5-Minute Review Session

1. How do concurrent threads in Esterel communicate?
2. What is the difference between weak and strong abortion?
3. What is the difference between aborts and traps?
4. What is syntactic sugar, and what is it good for?
5. What is the multiform notion of time?
Overview

**Logical Correctness**
- Causality issues
- The logical coherence law
- Logical reactivity and determinism
- Instantaneous Feedback

**The Logical Behavioral Semantics**
Causality Problems

```plaintext
present A
  else emit A
end

It's easy to write contradictory programs. Unfortunate side-effect of instantaneous communication coupled with the single valued signal rule. These sorts of programs are erroneous and flagged by the Esterel compiler as incorrect. Note: the first and third example are considered valid in SCEst, see later.
```
Causality Problems

```plaintext
present A
  else emit A
end

abort
  pause;
  emit A
when A
```

It's easy to write contradictory programs. Unfortunate side-effect of instantaneous communication coupled with the single valued signal rule. These sorts of programs are erroneous and flagged by the Esterel compiler as incorrect. Note: the first and third example are considered valid in SCEst, see later.
Causality Problems

```plaintext
present A
else emit A
end

abort
    pause;
    emit A
when A

present A
    then pause
end;
emit A
```
Causality Problems

- It’s easy to write contradictory programs
- Unfortunate side-effect of instantaneous communication coupled with the single valued signal rule
- These sorts of programs are erroneous and flagged by the Esterel compiler as incorrect
- Note: the first and third example are considered valid in SCEst, see later...
Causality Problems

Can be very complicated because of instantaneous communication

```
[abort
  emit A
  when immediate B
]
||
[
present A
  then emit B
  end;
]
```
Causality

Definition has evolved since first version of the language
Causality

- Definition has evolved since first version of the language
- Original compiler had concept of “potentials”
  - Static concept: at a particular program point, which signals could be emitted along any path from that point
Causality

- Definition has evolved since first version of the language
- Original compiler had concept of “potentials”
  - Static concept: at a particular program point, which signals could be emitted along any path from that point
- Current definition based on “constructive causality”
  - Dynamic concept: whether there’s a “guess-free proof” that concludes a signal is absent
Causality Example

```
emit A;
present B then emit C end;
present A else emit B end;
```

Red statements reachable

Analysis done by original compiler:

- After emit A runs, there's a static path to emit B
- Therefore, the value of B cannot be decided yet
- Execution procedure deadlocks: Program is bad
Causality Example

```plaintext
emit A;
present B then emit C end;
present A else emit B end;
```

Red statements reachable

Analysis done by later compilers:

- After `emit A` runs, it is clear that B cannot be emitted because A’s presence runs the “then” branch of the second `present`.
- B declared absent, both `present` statements run
- Program is OK
Logical Correctness

- The intuitive semantics:
  - Specifies what should happen when executing a program
Logical Correctness

- The intuitive semantics:
  - Specifies what should happen when executing a program
- However, also want to guarantee that
  - Execution actually exists (at least one possible execution)
  - Execution is unique (at most one possible execution)
Logical Correctness

- The intuitive semantics:
  - Specifies what should happen when executing a program
  - However, also want to guarantee that
    - Execution actually exists (at least one possible execution)
    - Execution is unique (at most one possible execution)
  - Need extra criteria for this!
- The apparently simplest possible criterion: logical correctness
Logical Correctness

Recall:

- Signal $S$ is absent by default
- Signal $S$ is present if an `emit S` statement is executed

The Logical Coherence Law:

A signal $S$ is present in a tick if and only if an `emit S` statement is executed in this tick.

Logical Correctness requires:

- There exists exactly one status for each signal that respects the coherence law
Logical Correctness

Recall:

- Signal $S$ is absent by default
- Signal $S$ is present if an $\text{emit } S$ statement is executed

The **Logical Coherence Law**:

A signal $S$ is present in a tick if and only if an $\text{emit } S$ statement is executed in this tick.
Logical Correctness

Recall:

- Signal $S$ is absent by default
- Signal $S$ is present if an $\text{emit } S$ statement is executed

The Logical Coherence Law:

A signal $S$ is present in a tick if and only if an $\text{emit } S$ statement is executed in this tick.

Logical Correctness requires:

- There exists exactly one status for each signal that respects the coherence law
Logical Correctness

Given:

- Program $P$ and input event $I$

Is logical correctness decidable?

- Yes!
Logical Correctness

Given:

- Program $P$ and input event $I$

$P$ is logically reactive w. r. t. $I$:

- There is at least one logically coherent global status

Is logical correctness decidable?

- Yes!
Logical Correctness

Given:
- Program $P$ and input event $I$

$P$ is **logically reactive** w.r.t. $I$:
- There is at least one logically coherent global status

$P$ is **logically deterministic** w.r.t. $I$:
- There is at most one logically coherent global status

Is logical correctness decidable?
- Yes!
Logical Correctness

Given:

- Program $P$ and input event $I$

$P$ is **logically reactive** w. r. t. $I$:
  - There is at least one logically coherent global status

$P$ is **logically deterministic** w. r. t. $I$:
  - There is at most one logically coherent global status

$P$ is **logically correct** w. r. t. $I$:
  - $P$ is both logically reactive and deterministic

Is logical correctness decidable?
- **Yes!**
Logical Correctness

Given:

- Program $P$ and input event $I$

$P$ is **logically reactive w. r. t. $I$**:

- There is at least one logically coherent global status

$P$ is **logically deterministic w. r. t. $I$**:

- There is at most one logically coherent global status

$P$ is **logically correct w. r. t. $I$**:

- $P$ is both logically reactive and deterministic

$P$ is **logically correct**:

- $P$ is logically correct w. r. t. *all* possible input events

Is logical correctness decidable? Yes!
Logical Correctness

Given:

- Program $P$ and input event $I$

$P$ is logically reactive w. r. t. $I$:
- There is at least one logically coherent global status

$P$ is logically deterministic w. r. t. $I$:
- There is at most one logically coherent global status

$P$ is logically correct w. r. t. $I$:
- $P$ is both logically reactive and deterministic

$P$ is logically correct:
- $P$ is logically correct w. r. t. all possible input events

Is logical correctness decidable?
Logical Correctness

Given:

- Program $P$ and input event $I$
  
  $P$ is logically reactive w. r. t. $I$:
  - There is at least one logically coherent global status
  
  $P$ is logically deterministic w. r. t. $I$:
  - There is at most one logically coherent global status

$P$ is logically correct w. r. t. $I$:

- $P$ is both logically reactive and deterministic

$P$ is logically correct:

- $P$ is logically correct w. r. t. all possible input events

Is logical correctness decidable?

- Yes!
module P1:
input I;
output 0;
signal S1, S2 in
  present I then emit S1 end
||
  present S1 else emit S2 end
||
  present S2 then emit 0 end
end signal
end module
Logical Correctness

module P1:
input I;
output O;
signal S1, S2 in
  present I then emit S1 end
||
  present S1 else emit S2 end
||
  present S2 then emit O end
end signal
end module

Is P1 logically correct?
module P1:
input I;
output O;
signal S1, S2 in
  present I then emit S1 end
||
  present S1 else emit S2 end
||
  present S2 then emit O end
end signal
end module

Is P1 logically correct?
▶ Yes!
Logical Correctness

module P2:
signal S in
  emit S;
  present O then
  present S then
    pause
  end;
  emit O
end
end signal
Logical Correctness

module P2:
signal S in
   emit S;
   present O then
      present S then
         pause
      end;
   emit 0
end
end signal

Is P2 logically correct?
Logical Correctness

module P2:
signal S in
  emit S;
  present 0 then
    present S then
      pause
    end;
  emit 0
end
end signal

Is P2 logically correct?
▶ Yes!
Is P2 logically correct?

▶ Yes!

▶ Notice that P2 is inputless

▶ Inputless programs react on empty input events, i.e., on clock ticks
Logical Correctness

module P3:
    present 0 else emit 0 end
end module

Is P3 logically correct?

▶ No!
▶ This is non-reactive

module P4:
    present O emit O end
end module

Is P4 logically correct?

▶ No!
▶ This is nondeterministic

module P5:
    present O1 then emit O2 end
    ||
    present O2 else emit O1 end
end module

Is P5 logically correct?

▶ No!
▶ This is non-reactive
Logical Correctness

module P3:
    present 0 else emit 0 end
end module

Is P3 logically correct?

▶ No!
Logical Correctness

module P3:
  present 0 else emit 0 end
end module

Is P3 logically correct?
▶ No!
▶ This is non-reactive
Logical Correctness

module P3:
present 0 else emit 0 end
end module

Is P3 logically correct?

▶ No!
▶ This is non-reactive

module P4:
present 0 emit 0 end
end module

Is P4 logically correct?
Logical Correctness

module P3:
present 0 else emit 0 end
end module

Is P3 logically correct?
▶ No!
▶ This is non-reactive

module P4:
present 0 emit 0 end
end module

Is P4 logically correct?
▶ No!
Logical Correctness

module P3:
  present 0 else emit 0 end
end module

Is P3 logically correct?

▶ No!
▶ This is non-reactive

module P4:
  present 0 emit 0 end
end module

Is P4 logically correct?

▶ No!
▶ This is nondeterministic
Logical Correctness

module P3:
  present 0 else emit 0 end
end module

Is P3 logically correct?
► No!
► This is non-reactive

module P4:
  present 0 emit 0 end
end module

Is P4 logically correct?
► No!
► This is nondeterministic

module P5:
  present 01 then emit 02 end
  ||
  present 02 else emit 01 end

Is P5 logically correct?
Logical Correctness

module P3:
present 0 else emit 0 end
end module

Is P3 logically correct?

▶ No!
▶ This is non-reactive

module P4:
present 0 emit 0 end
end module

Is P4 logically correct?

▶ No!
▶ This is nondeterministic

module P5:
present 01 then emit 02 end
||
present 02 else emit 01 end

Is P5 logically correct?

▶ No!
Logical Correctness

Is P3 logically correct?
- No!
- This is non-reactive

Is P4 logically correct?
- No!
- This is nondeterministic

Is P5 logically correct?
- No!
- This is non-reactive
Logical Correctness

module P6:
  present 01 then emit 02 end
||
  present 02 then emit 01 end

Is P6 logically correct?
Logical Correctness

module P6:
  present 01 then emit 02 end
 ||
  present 02 then emit 01 end

Is P6 logically correct?
➤ No!
Logical Correctness

module P6:
present 01 then emit 02 end
||
present 02 then emit 01 end

Is P6 logically correct?

▶ No!
▶ This is nondeterministic
Logical Correctness

Is P6 logically correct?

► No!
► This is nondeterministic

Is P7 logically correct?

module P6:
present 01 then emit 02 end
||
present 02 then emit 01 end

module P7:
present 0 then pause end;
emit 0
Logical Correctness

module P6:
present 01 then emit 02 end
||
present 02 then emit 01 end

Is P6 logically correct?
▶ No!
▶ This is nondeterministic

module P7:
present 0 then pause end;
emit 0

Is P7 logically correct?
▶ No!
Logical Correctness

Is P6 logically correct?
- No!
  - This is nondeterministic

Is P7 logically correct?
- No!
  - This is non-reactive
Logical Correctness

module P8:
trap T in
    present I else pause end;
    emit 0
||
    present 0 then exit T end
end trap;
emit 0

Is this logically correct?
Logical Correctness

module P8:
trap T in
  present I else pause end;
  emit 0
||
  present 0 then exit T end
end trap;
emit 0

Is this logically correct?
▶ Yes for I present
Logical Correctness

module P8:
trap T in
    present I else pause end;
    emit 0
||
    present 0 then exit T end
end trap;
emit 0

Is this logically correct?

▷ Yes for I present
▷ Nondeterministic for I absent
Logical Correctness

module P9:
[
    present 01 then emit 01 end
    ||
    present 01 then
        present 02 else emit 02 end
    end
]
Logical Correctness

module P9:
[
    present 01 then emit 01 end
|
    present 01 then
    present 02 else emit 02 end
end
]
Logical Correctness

Is P9 logically correct?

▶ Yes

▶ Note that this contains the nondeterministic program P4 and the non-reactive program P3!
Instantaneous Feedback

▶ Want to reject logically incorrect programs at compile time
Instantaneous Feedback

- Want to reject logically incorrect programs at compile time
- One option:
  - Forbid static self-dependency of signals
  - Similar to acyclicity requirement for electrical circuits
  - This is what the Esterel v4 compiler did
Instantaneous Feedback

► Want to reject logically incorrect programs at compile time
► One option:
  ► Forbid static self-dependency of signals
  ► Similar to acyclicity requirement for electrical circuits
  ► This is what the Esterel v4 compiler did

\[
\equiv 0 = \text{not } 0
\]
Instantaneous Feedback

- Want to reject logically incorrect programs at compile time
- One option:
  - Forbid static self-dependency of signals
  - Similar to acyclicity requirement for electrical circuits
  - This is what the Esterel v4 compiler did

```plaintext
module P3:
  present 0 else emit 0 end
end module
≡ 0 = not 0
```

```plaintext
module P4:
  present 0 emit 0 end
end module
≡ 0 = 0
```
Instantaneous Feedback

- However, forbidding cycles would also reject the following:
Instantaneous Feedback

- However, forbidding cycles would also reject the following:

```plaintext
module GoodCycle1:
  present I then
    present 01 then emit 02 end
  else
    present 02 then emit 01 end
end present
```

- 01 and 02 cyclically depend on each other
- However, any given status of I breaks the cycle
Instantaneous Feedback

module GoodCycle2:
  present 01 then emit 02 end;
pause;
present 02 then emit 01 end
Instantaneous Feedback

module GoodCycle2:
present 01 then emit 02 end;
pause;
present 02 then emit 01 end

▶ Here the cycle is neutralized with a delay

▶ In general, requiring acyclicity turns out to be inadequate to Esterel practice
Logical Correctness—Assessment

- We now introduced logical correctness
- But do we want to use it as basis for the language?
Logical Correctness—Assessment

- We now introduced logical correctness
- But do we want to use it as basis for the language?
  - 😊 sound
Logical Correctness—Assessment

- We now introduced logical correctness
- But do we want to use it as basis for the language?
  - 🌟 sound
  - 😞 sometimes unintuitive (consider P9)
Logical Correctness—Assessment

- We now introduced logical correctness
- But do we want to use it as basis for the language?
  - ☮️ sound
  - ☹️ sometimes unintuitive (consider P9)
  - ☹️ computationally complex
Logical Correctness—Assessment

► We now introduced logical correctness
► But do we want to use it as basis for the language?
  ◎ sound
  ☹️ sometimes unintuitive (consider P9)
  ☹️ computationally complex
► Alternative 1: allow only programs that are acyclic
  ◎ simple
  ☹️ too restrictive (consider GoodCycle1/2)
Logical Correctness—Assessment

- We now introduced logical correctness
- But do we want to use it as basis for the language?
  - ☺ sound
  - ☹ sometimes unintuitive (consider P9)
  - ☹ computationally complex
- Alternative 1: allow only programs that are acyclic
  - ☺ simple
  - ☹ too restrictive (consider GoodCycle1/2)
- Alternative 2: accept everything for which the compiler finds a static execution schedule
  - ☺ relatively simple for the compiler
  - ☹ definition not precise, depends on abilities of compiler (different compilers accept different programs)
Logical Correctness—Assessment

➤ We now introduced logical correctness
➤ But do we want to use it as basis for the language?
   - ☺ sound
   - ☹ sometimes unintuitive (consider P9)
   - ☹ computationally complex
➤ Alternative 1: allow only programs that are acyclic
   - ☺ simple
   - ☹ too restrictive (consider GoodCycle1/2)
➤ Alternative 2: accept everything for which the compiler finds a static execution schedule
   - ☺ relatively simple for the compiler
   - ☹ definition not precise, depends on abilities of compiler (different compilers accept different programs)
➤ Alternative 3: the constructive semantics
   - ☹ analysis not trivial
   - ☺ clear semantics
Overview

Logical Correctness

The Logical Behavioral Semantics
  Notation and Definitions
  The Basic Broadcasting Calculus
  Transition Rules
  Reactivity and Determinism
The Semantics of Esterel

1. **Logical Behavioral Semantics**
   - Rewriting rules defining reactivity, determinism, and logical correctness
   - Signal coherence law embedded in rules for local signals
The Semantics of Esterel

1. **Logical** Behavioral Semantics
   - Rewriting rules defining reactivity, determinism, and logical correctness
   - Signal coherence law embedded in rules for local signals

2. **Constructive** Behavioral Semantics
   - Refines logical behavioral semantics
   - Based on *must* and *cannot* analysis
The Semantics of Esterel

1. **Logical** Behavioral Semantics
   - Rewriting rules defining reactivity, determinism, and logical correctness
   - Signal coherence law embedded in rules for local signals

2. **Constructive** Behavioral Semantics
   - Refines logical behavioral semantics
   - Based on *must* and *cannot* analysis

3. Logical/Constructive **State** Behavioral Semantics
   - Replaces rewriting with marking of active delays (v5 debugger)
The Semantics of Esterel

1. **Logical Behavioral Semantics**
   - Rewriting rules defining reactivity, determinism, and logical correctness
   - Signal coherence law embedded in rules for local signals

2. **Constructive Behavioral Semantics**
   - Refines logical behavioral semantics
   - Based on *must* and *cannot* analysis

3. **Logical/Constructive State Behavioral Semantics**
   - Replaces rewriting with marking of active delays (v5 debugger)

4. **Constructive State Operational Semantics**
   - Defines reaction as sequence of microsteps (v3 compiler)
The Semantics of Esterel

1. **Logical** Behavioral Semantics
   - Rewriting rules defining reactivity, determinism, and logical correctness
   - Signal coherence law embedded in rules for local signals

2. **Constructive** Behavioral Semantics
   - Refines logical behavioral semantics
   - Based on *must* and *cannot* analysis

3. Logical/Constructive **State** Behavioral Semantics
   - Replaces rewriting with marking of active delays (v5 debugger)

4. **Constructive State Operational** Semantics
   - Defines reaction as sequence of microsteps (v3 compiler)

5. **Constructive Circuit** Semantics
   - Translates Esterel programs into Boolean digital circuits (v5 compiler)
Notation and Definitions

- **Sort** $S$: A set of signals
Notation and Definitions

- **Sort** $S$: A set of signals
- **Signal statuses**: $B = \{+, -\}$
Notation and Definitions

- **Sort** $S$: A set of signals
- **Signal statuses**: $B = \{+, -\}$
- **Event** $E$:
  - Given sort $S$, defines **status** $E(s) \in B$ for each $s \in S$
  - Obtain sort of $E$ as $S(E) = S$
Notation and Definitions

- **Sort** $S$: A set of signals
- **Signal statuses**: $B = \{+, -\}$
- **Event** $E$:
  - Given sort $S$, defines status $E(s) \in B$ for each $s \in S$
  - Obtain sort of $E$ as $S(E) = S$
- **Two equivalent representations for $E$**:
  - As subset of $S$: $E = \{s \in S \mid E(s) = +\}$
  - As a mapping from $S$ to $B$: $E = \{(s, b) \mid b = E(s)\}$
Notation and Definitions

- Write $s^+ \in E$ iff $E(s) = +$
- Write $s^- \in E$ iff $E(s) = -$.
- Write $E' \subset E$ iff $\forall s \in S(E') : s^+ \in E' \implies s^+ \in E$. 

---

Logical Correctness
The Logical Behavioral Semantics
Notation and Definitions

- Write $s^+ \in E$ iff $E(s) = +$
- Write $s^- \in E$ iff $E(s) = -$ 
- Write $E' \subset E$ iff $\forall s \in S(E') : s^+ \in E' \implies s^+ \in E$
- Given signal $s$, define singleton event $\{s^+\}$:
  - $\{s^+\}(s) = +$
  - $\forall s' \neq s : \{s^+\}(s') = -$
Notation and Definitions

- Write $s^+ \in E$ iff $E(s) = +$
- Write $s^- \in E$ iff $E(s) = -$
- Write $E' \subset E$ iff $\forall s \in S(E') : s^+ \in E' \implies s^+ \in E$
- Given signal $s$, define singleton event $\{s^+\}$:
  - $\{s^+\}(s) = +$
  - $\forall s' \neq s : \{s^+\}(s') = -$  
- Given signal set $S$ and signal $s \in S$, write $S \setminus s = S - \{s\}$
- Given $E$ and $s \in S(E)$, write $E \setminus s$ to denote event of sort $S(E) \setminus s$, which coincides with $E$ on all signals but $s$
Notation and Definitions

- Write $s^+ \in E$ iff $E(s) = +$
- Write $s^- \in E$ iff $E(s) = -$
- Write $E' \subseteq E$ iff $\forall s \in S(E') : s^+ \in E' \implies s^+ \in E$
- Given signal $s$, define singleton event $\{s^+\}$:
  - $\{s^+\}(s) = +$
  - $\forall s' \neq s : \{s^+\}(s') = -$
- Given signal set $S$ and signal $s \in S$, write $S \setminus s = S - \{s\}$
- Given $E$ and $s \in S(E)$, write $E \setminus s$ to denote event of sort $S(E) \setminus s$, which coincides with $E$ on all signals but $s$
- Define $E \star s^b$ as event $E'$ of sort $S(E) \cup \{s\}$ with
  - $E'(s) = b$
  - $E'(s') = E(s')$ for $s' \neq s$
Notation and Definitions

- Will present formal semantics as Plotkin’s Structural Operational Semantics (SOS) inference rules
Notation and Definitions

- Will present formal semantics as Plotkin’s Structural Operational Semantics (SOS) inference rules
- Behavioral Semantics formalizes reaction of program $P$ as behavioral transition

\[ P \xrightarrow{O} P' \]

- $I$: input event
- $O$: output event
- $P'$: derivative of $P$—the program for the next instance
Notation and Definitions

- Auxiliary statement transition relation:

\[
\begin{array}{c}
p \xrightarrow{E',k} p' \\
E \end{array}
\]

- \(p\): program body (of \(P\))
- \(E\): event defining status of all signals declared in scope of \(p\)
- \(E'\): event composed of all signals emitted by \(p\) in the reaction
- \(k\): completion code returned by \(p\) (0 iff \(p\) terminates)
- \(p'\): derivative of \(p\)
Notation and Definitions

- Auxiliary statement transition relation:
  \[ p \xrightarrow{E',k,E,p'} \]

- \( p \): program body (of \( P \))
- \( E \): event defining status of all signals declared in scope of \( p \)
- \( E' \): event composed of all signals emitted by \( p \) in the reaction
- \( k \): completion code returned by \( p \) (0 iff \( p \) terminates)
- \( p' \): derivative of \( p \)
- Logical coherence (or broadcasting invariant):
Notation and Definitions

- Auxiliary statement transition relation:

\[ p \xrightarrow{E',k} p' \]

- \( p \): program body (of \( P \))
- \( E \): event defining status of all signals declared in scope of \( p \)
- \( E' \): event composed of all signals emitted by \( p \) in the reaction
- \( k \): completion code returned by \( p \) (0 iff \( p \) terminates)
- \( p' \): derivative of \( p \)
- Logical coherence (or broadcasting invariant):

\[ E' \subset E \]
Notation and Definitions

Given:
- Program $P$ with body $p$
- Input event $I$

Define program transition of $P$ by statement transition of $p$:

$$P \xrightarrow{O \cup I} P' \text{ iff } p \xrightarrow{O, k \cup I \cup O} p' \text{ for some } k$$

These program transitions, yielding an output reaction $O$ and a derivative $P'$, determine the logical behavioral semantics of $P$. 
For concise presentation of rules: Replace Esterel syntax with terser process-calculus syntax

Use parenthesis for grouping statements

- nothing
- pause
- emit s
- present s then p else q end
- p; q
- loop p end
- p || q
- signal s in p end
- suspend p when s end
- trap T in p end
- exit T
- [no concrete syntax]

(nothing) 0
(pause) 1
(emit s) !s
(present s then p else q end) s?p, q
(p; q) p; q
(loop p end) p*
(p || q) p|q
(signal s in p end) p \ s
(suspend p when s end) s \ p
(trap T in p end) {p}
(exit T) k with k ≥ 2
([no concrete syntax]) ↑p
Example

```
pause;
emit 01;
loop
  pause;
  [  
    present I1 then
      emit 02
    end present 
    ||
    present I3 else
      emit 03
    end present
  ]
end loop
```
Example

```
pause;
emit O1;
loop
  pause;
  [ present I1 then emit 02 end present ||
    present I3 else emit 03 end present ]
end loop
```
Example

pause;
emit O1;
loop
    pause;
    [
        present I1 then
            emit O2
        end present
    ]
    []
    present I3 else
        emit O3
    end present
] end loop

≡

1; !O1; (1; ((I1 ? !O2, 0) | (I3 ? 0, !O3)))*
Basic Transition Rules

The null process 0:

\[
0 : \emptyset, 0 \vdash E \quad (null)
\]

The unit delay process 1:

\[
1 : \emptyset, 1 \vdash E \quad (unit \ delay)
\]

Signal emission:

\[
!s : \{s\}, 0 \vdash E \quad (emit)
\]
Basic Transition Rules

The null process 0:

\[
0 \xrightarrow{\emptyset,0} 0 \quad \text{(null)}
\]
Basic Transition Rules

The null process 0:

\[ 0 \xrightarrow{\emptyset,0} 0 \]

The unit delay process 1:


Basic Transition Rules

The null process 0:

\[
0 \xrightarrow{} 0
\]

\[
E
\]

The unit delay process 1:

\[
1 \xrightarrow{} 0
\]

\[
E
\]

(null)

(unit delay)
Basic Transition Rules

The null process 0:

\[ 0 \xrightarrow{\emptyset,0} 0 \]

(null)

The unit delay process 1:

\[ 1 \xrightarrow{\emptyset,1} 0 \]

(unit delay)

Signal emission:

\[ !s \xrightarrow{} 0 \]
Basic Transition Rules

The null process 0:

\[ 0 \xrightarrow{\emptyset,0} E 0 \]  

\( \text{null} \)

The unit delay process 1:

\[ 1 \xrightarrow{\emptyset,1} E 0 \]  

\( \text{unit delay} \)

Signal emission:

\[ !s \xrightarrow{\{s\},0} E 0 \]  

\( \text{emit} \)
Deduction Rules

- In addition to simple transition rules, will also use deduction rules
Deduction Rules

- In addition to simple transition rules, will also use deduction rules.

- Hypothesis: If sub-instructions behave like this . . .

\[
\begin{align*}
p_1 & \xrightarrow{E,k_1} p'_1 \\
p_2 & \xrightarrow{E,k_2} p'_2 \\
\text{Other hypotheses}
\end{align*}
\]

\[
\text{Instruction}(p_1, p_2) \xrightarrow{E(E'_1,E'_2), K(k_1,k_2)} \text{Instruction}'(p'_1, p'_2)
\]

- Conclusion: . . . then the compound instruction behaves like that.
Deduction Rules—Sequencing
Deduction Rules—Sequencing

\[
\begin{align*}
  \frac{p \xrightarrow{E',k} p' \quad k \neq 0}{E} \\
  \frac{p; q \xrightarrow{E',k} p'; q}{E}
\end{align*}
\]
Deduction Rules—Sequencing

(seq1)

(seq2)
Deduction Rules—Looping and Parallel
Deduction Rules—Looping and Parallel

\[
\begin{align*}
    p & \xrightarrow{E',k} p' \quad k \neq 0 \\
    p^* & \xrightarrow{E',k} p'; (p^*)
\end{align*}
\]
Deduction Rules—Looping and Parallel

\[
p \xrightarrow{E',k} p' \quad k \neq 0
\]
\[
p^* \xrightarrow{E',k} p'; (p^*)
\] (loop)

\[
p \xrightarrow{E_p,k} p' \quad q \xrightarrow{E_q,l} q'
\]
\[
p|q \xrightarrow{E_p \cup E_q, \max(k,l)} p'|q'
\] (parallel)
Deduction Rules—Conditional
Deduction Rules—Conditional

\[
\begin{array}{c}
s^+ \in E \\
p \xrightarrow{E',k} p' \\
\hline \\
s?p, q \xrightarrow{E',k} p'
\end{array}
\]

(present +)
Deduction Rules—Conditional

\[ s^+ \in E \quad p \xrightarrow{E',k} p' \]
\[ \frac{}{s?p, q \xrightarrow{E',k} p'} \]

(present +)

\[ s^- \in E \quad q \xrightarrow{E',k} q' \]
\[ \frac{}{s?p, q \xrightarrow{E',k} q'} \]

(present −)
Deduction Rules—Conditional

Zero delay: can use decision trees to test for arbitrary Boolean conditions:

1. \((s_1 \land s_2)\) ? \(p, q\) is \(s_1 ? (s_2 ? p, q), q\)
2. \((s_1 \lor s_2)\) ? \(p, q\) is \(s_1 ? p, (s_2 ? p, q)\)
3. \(\neg s ? p, q\) is \(s ? q, p\)
Deduction Rules—Restriction

\[
p \xrightarrow{E's^+,k} p' \quad S(E') = S(E)\setminus s
\]

\[
p \setminus s \xrightarrow{E',k} p' \setminus s
\]

Note: This also properly handles nested restrictions of the same signal.

\[
(sig +)
\]
Deduction Rules—Restriction

\[ p \xrightarrow{E' *_{s^+},k} p' \quad S(E') = S(E) \setminus s \]

\[ p \setminus s \xrightarrow{E',k} p' \setminus s \]

\[ (\text{sig } +) \]

\[ p \xrightarrow{E' *_{s^-},k} p' \quad S(E') = S(E) \setminus s \]

\[ p \setminus s \xrightarrow{E',k} p' \setminus s \]

\[ (\text{sig } -) \]
Deduction Rules—Restriction

\[
\begin{align*}
p \xrightarrow{E' * s^+, k} p' \quad S(E') &= S(E) \setminus s \\
E * s^+ &
\hline
p \setminus s \xrightarrow{E', k} p' \setminus s
\end{align*}
\]

\(\text{(sig +)}\)

\[
\begin{align*}
p \xrightarrow{E' * s^-, k} p' \quad S(E') &= S(E) \setminus s \\
E * s^- &
\hline
p \setminus s \xrightarrow{E', k} p' \setminus s
\end{align*}
\]

\(\text{(sig \(-\))}\)

Note: This also properly handles nested restrictions of the same signal
Traps—Example

- The trap exit encoding is
  - $k = 2$ if the closest enclosing trap is exited, and
  - $k = n + 2$ if $n$ trap declarations have to be traversed
Traps—Example

- The trap exit encoding is
  - $k = 2$ if the closest enclosing trap is exited, and
  - $k = n + 2$ if $n$ trap declarations have to be traversed

```plaintext
trap U in
  trap T in
    nothing
    ||
    pause
    ||
    exit T
  ||
  exit U
end
||
exit U
end
≡
```
Traps—Example

The trap exit encoding is
- \( k = 2 \) if the closest enclosing trap is exited, and
- \( k = n + 2 \) if \( n \) trap declarations have to be traversed

```
trap U in
  trap T in
    nothing
  ||
    pause
  ||
    exit T
||
  ||
    exit U
end
||
exit U
end
```

\[ \equiv \{0 \mid 1 \mid 2 \mid 3\} | 2 \]
Two Operators on Completion Codes

- The $\downarrow k$ operator computes completion code of $\{p\}$ from that of $p$:

  $$\downarrow k = \begin{cases} 0 & \text{if } k = 0 \text{ or } k = 2 \\ 1 & \text{if } k = 1 \\ k - 1 & \text{if } k > 2 \end{cases}$$
Two Operators on Completion Codes

- The $\downarrow k$ operator computes completion code of $\{p\}$ from that of $p$:
  
  \[
  \begin{align*}
  \downarrow k &= 0 & \text{if } k &= 0 \quad \text{or} \quad k &= 2 \\
  \downarrow k &= 1 & \text{if } k &= 1 \\
  \downarrow k &= k - 1 & \text{if } k &> 2 
  \end{align*}
  \]

- The $\uparrow k$ operator computes completion code of $\uparrow p$ from that of $p$; want $\{\uparrow p\} \equiv p$
  
  \[
  \begin{align*}
  \uparrow k &= k & \text{if } k &= 0 \quad \text{or} \quad k &= 1 \\
  \uparrow k &= k + 1 & \text{if } k &> 1 
  \end{align*}
  \]
The Shift Operator

▶ \( \uparrow \) ("shift") shifts exit numbers of \( p \) by 1 when placing \( p \) in a trap block
The Shift Operator

- $\uparrow$ ("shift") shifts exit numbers of $p$ by 1 when placing $p$ in a trap block
- May use $\uparrow$ in definitions of derived operators

\[
\text{suspend } p \text{ when immediate } s \quad s \supset p \quad \equiv
\]
The Shift Operator

- $\uparrow$ ("shift") shifts exit numbers of $p$ by 1 when placing $p$ in a trap block
- May use $\uparrow$ in definitions of derived operators

suspend $p$ when immediate $s$

$s \cdot \triangleright p \equiv \{(s, 1, 2)^*; s \triangleright p\}$
The Shift Operator

- $\uparrow$ ("shift") shifts exit numbers of $p$ by 1 when placing $p$ in a trap block
- May use $\uparrow$ in definitions of derived operators

\[
\begin{align*}
\text{suspend } p \text{ when immediate } s & \quad s \cdot \supset p \equiv \{(s\uparrow 1, 2)^*\}; s \supset p \\
\text{await immediate } s; p & \quad s \cdot \Rightarrow p \equiv \end{align*}
\]
The Shift Operator

- \(\uparrow\) ("shift") shifts exit numbers of \(p\) by 1 when placing \(p\) in a trap block
- May use \(\uparrow\) in definitions of derived operators

\[
suspend p \text{ when immediate } s \\
\text{await immediate } s; p \\
\text{await } s; p
\]

\[
s \cdot \supset p \equiv \{ (s?1, 2)^* ; s \supset p \} \\
s \Rightarrow p \equiv \{ (s(?\uparrow p; 2), 1)^* \} \\
s \Rightarrow p \equiv \quad
\]
The Shift Operator

- $\uparrow$ ("shift") shifts exit numbers of $p$ by 1 when placing $p$ in a trap block
- May use $\uparrow$ in definitions of derived operators

\begin{align*}
\text{suspend } p \text{ when immediate } s & \quad s \cdot \sqsubset p \quad \equiv \{(s?1, 2)^*\}; \; s \supset p \\
\text{await immediate } s; \; p & \quad s \cdot \Rightarrow p \quad \equiv \{(s?(\uparrow p; 2), 1)^*\} \\
\text{await } s; \; p & \quad s \Rightarrow p \quad \equiv 1; \; s \cdot \Rightarrow p
\end{align*}
The Shift Operator

- $\uparrow$ ("shift") shifts exit numbers of $p$ by 1 when placing $p$ in a trap block
- May use $\uparrow$ in definitions of derived operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>suspend $p$ when immediate $s$</td>
<td>$s \cdot \supset p \equiv {(s?1, 2)^*}; s \supset p$</td>
</tr>
<tr>
<td>await immediate $s; p$</td>
<td>$s \cdot \Rightarrow p \equiv {(s? (\uparrow p; 2), 1)^*}$</td>
</tr>
<tr>
<td>await $s; p$</td>
<td>$s \Rightarrow p \equiv 1; s \cdot \Rightarrow p$</td>
</tr>
<tr>
<td>weak abort $p$ when immediate $s$</td>
<td>$s \cdot \succ p \equiv {(\uparrow p; 2) \mid s \cdot \Rightarrow 2}$</td>
</tr>
<tr>
<td>weak abort $p$ when $s$</td>
<td>$s \succ p \equiv {(\uparrow p; 2) \mid s \Rightarrow 2}$</td>
</tr>
<tr>
<td>abort $p$ when immediate $s$</td>
<td>$s \cdot \gg p \equiv s \cdot \succ (s \cdot \supset p)$</td>
</tr>
<tr>
<td>abort $p$ when $s$</td>
<td>$s \gg p \equiv s \succ (s \supset p)$</td>
</tr>
</tbody>
</table>
Traps—The Rules

\[
k \xrightarrow{\emptyset, k} 0
\]

(exit)
Traps—The Rules

\[ k \xrightarrow{\emptyset, k} E \xrightarrow{0} 0 \quad \text{(exit)} \]

\[ p \xrightarrow{E', k} p' \quad k = 0 \text{ or } k = 2 \]

\[ \{p\} \xrightarrow{E', 0} 0 \quad \text{(trap1)} \]
Traps—The Rules

\[ k \overset{\emptyset, k}{E} 0 \]

\[ p \overset{E', k}{E} p' \quad k = 0 \text{ or } k = 2 \]

\[ \{p\} \overset{E', 0}{E} 0 \]

\[ p \overset{E', k}{E} p' \quad k = 1 \text{ or } k > 2 \]

\[ \{p\} \overset{E', \downarrow k}{E} \{p'\} \]

(exit)

(trap1)

(trap2)
Traps—The Rules

```
\begin{align*}
&k \xrightarrow{\emptyset,k} E 0 \\
&\text{(exit)} \\
&p \xrightarrow{E',k} p' \quad k = 0 \text{ or } k = 2 \\
&\quad \quad \{p\} \xrightarrow{E',0} 0 \\
&\text{(trap1)} \\
&p \xrightarrow{E',k} p' \quad k = 1 \text{ or } k > 2 \\
&\quad \quad \{p\} \xrightarrow{E',\downarrow k} \{p'\} \\
&\text{(trap2)} \\
&p \xrightarrow{E',k} p' \\
&\quad \quad \uparrow p \xrightarrow{E',\uparrow k} \uparrow p' \\
&\text{(shift)}
\end{align*}
```
Deduction Rules—Suspension

\[
p \xrightarrow{E',0} p' \\
E \\
\overline{\exists} p \xrightarrow{E',0} 0
\]  
\[ (\text{suspend1}) \]
Deduction Rules—Suspension

\[ p \xrightarrow{E',0} p' \]
\[ \implies s \supset p \xrightarrow{E',0} 0 \]  
\(\text{(suspend1)}\)

\[ p \xrightarrow{E',k} p' \quad k \neq 0 \]
\[ \implies s \supset p \xrightarrow{E',k} s \cdot \supset p' \]  
\(\text{(suspend2)}\)
Reactivity and Determinism

Definition: Program $P$ is logically reactive (resp. logically deterministic) w.r.t. an input event $I$ if
Reactivity and Determinism

Definition: Program $P$ is logically reactive (resp. logically deterministic) w.r.t. an input event $I$ if there exists at least (resp. at most) one program transition $P \xrightarrow{O} P'$ for some output event $O$ and program derivative $P'$. 
Reactivity and Determinism

Definition: Program $P$ is logically reactive (resp. logically deterministic) w.r.t. an input event $I$ if there exists at least (resp. at most) one program transition $P \xrightarrow{O} P'$ for some output event $O$ and program derivative $P'$. 

Definition: Program $P$ is logically correct if it is logically reactive and logically deterministic.
Reactivity and Determinism

- **Definition:** Program $P$ is logically reactive (resp. logically deterministic) w.r.t. an input event $I$ if there exists at least (resp. at most) one program transition $P \xrightarrow{O} P'$ for some output event $O$ and program derivative $P'$

- **Definition:** Program $P$ is logically correct if it is logically reactive and logically deterministic

- How about $(s?!s, 0)$?
- And how about $(s?0, !s)$?
Reactivity and Determinism

- I/O determinism still leaves room for internal non-determinism
  - Consider $(s?!s, 0) \setminus s$
  - Forbidden in constructive semantics
Reactivity and Determinism

- I/O determinism still leaves room for internal non-determinism
  - Consider \((s?!s, 0) \setminus s\)
  - Forbidden in constructive semantics

- **Definition**: Program \(P\) is **strongly deterministic** for an input event \(I\) iff
  - \(P\) is reactive and deterministic for this event, and
  - there exists a unique proof of the unique transition \(P \xrightarrow{O} P'\).
Summary (1/3)

- The intuitive semantics specifies what should happen when executing a program.
- However, also want to guarantee that exactly one possible execution exists that satisfies the intuitive semantics.
- The Logical Coherence Law specifies that a signal $S$ is present in a tick if and only if an “emit $S$” statement is executed in this tick.
- Logical Correctness requires that there exists exactly one status for each signal that respects the coherence law.
Summary (2/3)

- $P$ is logically reactive w.r.t. input $I$ if there is at least one logically coherent global status.
- $P$ is logically deterministic w.r.t. $I$ if there is at most one logically coherent global status.
- $P$ is logically correct w.r.t. $I$ if $P$ is both logically reactive and deterministic.
- $P$ is logically correct if $P$ is logically correct w.r.t. all possible input events.
Summary (3/3)

- There exist several semantics for the Esterel language—one important distinction is between *logical* and *constructive* semantics, the latter being a refinement of the former.
- We started discussing the logical behavioral semantics, expressed in Plotkin’s Structural Operational Semantics, with basic transition rules and deduction rules.
- We formally defined reactivity, determinism, logical correctness, and strong determinism.
To Go Further


- Gérard Berry, Preemption in Concurrent Systems, In Proceedings FSTTCS 93, Lecture Notes in Computer Science 761, pages 72-93, Springer-Verlag, 1993,
  http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.1557