The 5-Minute Review Session

1. What is the state of an Esterel program? Which implementation alternatives are there to memorize state?
2. What are implementation alternatives to interface with the environment, e.g., a device that can be on or off?
3. What is the relationship between events and states?
4. What are possible examples for causality problems? What is the reason for these problems?
5. When is an Esterel program logically reactive? ... correct?

External Justification vs. Self-Justification

- Programming in Esterel:
  - Analyze input events to generate appropriate output signals
  - Use concurrent statements and intermediate local signals to create modular, well-structured programs
- Natural way of thinking:
  - Information propagation by cause and effect
External Justification vs. Self-Justification

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 0 end end signal
end module

▸ Is this logically correct?
▸ Yes!
▸ Is this well-behaved wrt information propagation?
▸ Yes!

▸ “present S then p end”:
▸ First test the status of S, then execute p if S is present
▸ Status of S should not depend on what p might do

▸ Synchrony hypothesis:
▸ Ordering implicit in the then word is not that of time, but that of sequential causality
▸ Want actual computation:
▸ “Since S is present, we take the then branch”
▸ Don’t want speculative computation:
▸ “If we assume S present, then we take the then branch”

Aside from the explicit concurrency “||”, all Esterel statements are sequential
Want to preserve this in the semantics

▸ Is this logically correct?
▸ Yes!
▸ Is this well-behaved wrt information propagation?
▸ No!
▸ Accepting P9 as correct is
▸ Logically possible
▸ But against (imperative) intention of the language

▸ This is logically correct
▸ But still want to reject it:
▸ In the logical semantics, the information that 0 is present flows backwards across the sequencing operator
Contradicts basic intuition about sequential execution
The Constructive Semantics

- Constructive semantics:
  - Does not check assumptions about signal statuses
  - Instead, propagates facts about control flow and signal statuses

- Three equivalent presentations:
  1. Constructive behavioral semantics
  2. Constructive operational semantics
  3. Circuit semantics

The name constructive semantics is borrowed from constructive logic, in which one handles fact-propagating proofs, instead of handling values as in classical logic.

The Constructive Behavioral Semantics

1. Constructive behavioral semantics:
   - Derived from the logical behavioral semantics
   - Adds constructive restrictions to logical coherence rule
   - Is the simplest way of defining the language

2. Constructive operational semantics:
   - Based on an interpretation scheme expressed by term rewriting rules defining microstep sequences
   - Is the simplest way of defining an efficient interpreter

3. Circuit semantics:
   - Translation of programs into constructive circuits
   - Is the core of the Esterel v5 compiler

... retains the spirit of the logical coherence semantics
... adds reasoning about what a program must or cannot do
Define disjoint predicates to express
- “A statement must terminate, must pause, must exit a trap \( T \), or must emit a signal \( S \)”
- “A statement cannot terminate, cannot pause, cannot exit a trap \( T \), or cannot emit a signal \( S \)”

The Must (Cannot) predicate determines
- Which signals are present (absent)
- Which statements are (cannot be) executed
The Constructive Behavioral Semantics

Recall: Logical Coherence Law

A signal $S$ is present in an instant iff an "emit $S$" statement is executed in this instant.

Replace with disjoint Constructive Coherence Laws:

A signal $S$ is present iff an "emit $S$" statement must be executed.
A signal $S$ is absent iff an "emit $S$" statement cannot be executed.

Define Must and Cannot predicates by structural induction on statements

A signal can have three statuses:
- "$+$": known to be present
- "$-$": known to be absent
- "$\perp$": yet unknown

Is technically easier to define the Cannot predicate as the negation of a Can predicate
- No constructiveness problem here as we only deal with finite sets

Main novelty is in analysis of output and local signals

Consider local signal here; output signal is similar

signal $S$ in $p$ end (Local signal)

Can predicate:
- Recursively analyze $p$ with status $\perp$ for $S$
The Constructive Behavioral Semantics

signal \( S \) in \( p \) end (Local signal)

Must predicate:

- Assume we already know that we must execute signal \( S \) in \( p \) end in some signal context \( E \)
- Must compute final status of \( S \) to determine signal context of \( p \)
- First analyze \( p \) in \( E \) augmented by setting the unknown status \( \bot \) for \( S \)
- If \( S \) must be emitted:
  - Propagate this information by reanalyzing \( p \) in \( E \) with \( S \) present
  - This may generate more information about the other signals
- Similarly, if we find that \( S \) cannot be emitted:
  - Reanalyze \( p \) in \( E \) with \( S \) absent

Accepting Programs

In the constructive behavioral semantics, a program is accepted as constructive iff fact propagation using the Must and Can (or Cannot) predicates suffices in establishing presence or absence of all output signals (and we can also compute a derivative—see later)

```
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
```

- If \( I \) is present, then the first present statement must take its first branch, emit \( S1 \), and terminate. From this, we deduce that \( S1 \) is present. Then, the second present statement must take its (empty) then branch and cannot take its else branch. Since the “emit \( S2 \)” statement cannot be executed, \( S2 \) cannot be emitted, which implies that \( S2 \) is absent. Finally, the third present statement cannot take its then branch, which implies that \( O \) cannot be emitted and is absent.
- If \( I \) is absent, then the first present statement cannot take its first branch, and the “emit \( S1 \)” statement cannot be executed, which implies that \( S1 \) is absent. Therefore, the second present statement must take its then branch, the “emit \( S2 \)” statement must be executed, which implies that \( S2 \) is present. Finally, the third present statement must take its then branch, and the “emit \( O \)” statement must be executed, setting \( O \) present.
Accepting Programs

- Can analyze this with just propagating facts
  - No need for speculative computation based on assumptions
  - Our analysis still "looks ahead" to see what must/cannot be done, but always builds on facts established so far, not on speculations

- However, analysis involves recomputations
  - Avoiding this is goal of operational and circuit semantics!

- We first start analyzing what the "signal S" statement must do with status ⊥ for O. For this, we analyze its body with status ⊥ for 0 and S.
- We immediately find that S must be emitted since we must execute the "emit S" statement. Therefore, we redo the analysis with status ⊥ for 0 and ⊤ for S.
- We reach the test for 0. Since the status of 0 is unknown, there is nothing we must do and we can make progress only by analyzing what we cannot do in the branches of the test.
- In the then branch, there is a present test for 0. Since S is known to be present, we cannot take the implicit else branch that would terminate. Since the then branch is a pause statement, it cannot terminate.
- Summing up things, the "present S" test cannot terminate. Therefore, the "emit 0" statement cannot be executed and 0 cannot be emitted. As a consequence we must set 0 absent and redo the analysis of the program with status — for 0.
- We now find that we must take the implicit else branch of the "present 0" test that terminates execution.
- The program is constructive since we have fully determined the signal statuses.

Rejecting Programs

- If the must and cannot predicates bring no information about the status of some signal:
  - Programs is rejected

Note:

- The constructive semantics rejected P3 and P4 for the very same reason
- The logical behavioral semantics rejected P3 and P4 for two different reasons, namely non-reactivity and non-determinism
Rejecting Programs

- Constructiveness $\implies$ logical correctness
- But not vice versa!

```plaintext
module P9:
    [ present O1 then emit O1 end
    ||
      present O1 then
      present O2 else emit O2 end
    end
    ]
```

- Both O1 and O2 can be emitted
- No signal must be emitted
- No progress—reject P9!

```
module P11:
    signal S
    present O then
    emit S;
    present S then
    pause
    end;
    emit O
    end
end signal
```

- Are not allowed to speculatively execute branches
- Again no progress—reject P11!

- The "emit S" statement is now inside the then branch of the "present 0" statement.
- The must analysis with status $\bot$ for 0 and S finds nothing we must do since we are not allowed to speculatively compute within the branches of the test for 0.
- In the same status, the cannot analysis finds that both S and O can be emitted since it finds potentially reachable emitters.
- Therefore, we can make no progress and we reject P11.
The Constructive Semantics
External Justification vs. Self-Justification
The Constructive Behavioral Semantics
The Constructive Operational Semantics

**Rejecting Programs**

- Must reject P12 as well!
- Does an equivalent HW-circuit always stabilize? 
  *(Will come back to this later . . .)*

```module P12: present O then emit O; else emit O end```

**The Must, Cannot, and Can Functions**

- **Must** function determines what must be done in a reaction
  \[ P \overset{O}{\rightarrow} P' \]
- Has the form \( \text{Must}(p, E) = (S, K) \)
  - \( E \): partial event, associating status in \( B_\bot = \{+, -, \bot\} \) with each signal
  - \( S \): set of signals that \( p \) must emit
  - \( K \): set of completion codes that \( p \) must return
  - Is either empty or a singleton
- Use subscripts to access elements of result pair:
  - \( \text{Must}(p, E) = (\text{Must}_s(p, E), \text{Must}_k(p, E)) \)

**Definitions of Must and Can**

- Completion and signal emission:
  \[
  \text{Must}(k, E) = \text{Can}^{m}(k, E) = (\emptyset, \{k\})
  \]
  \[
  \text{Must}(s, E) = \text{Can}^{m}(s, E) = (\{s\}, \{0\})
  \]
- Suspension:
  \[
  \text{Must}(s \supset p, E) = \text{Must}(p, E)
  \]
  \[
  \text{Can}^{m}(s \supset p, E) = \text{Can}^{m}(p, E)
  \]
Definitions of *Must* and *Can*

- Signal test:

\[
\text{Must}(s? p, q, E) = \begin{cases} 
\text{Must}(p, E) & \text{if } s^+ \in E \\
\text{Must}(q, E) & \text{if } s^- \in E \\
(\emptyset, \emptyset) & \text{if } s^\perp \in E
\end{cases}
\]

\[
\text{Can}^m(s? p, q, E) = \begin{cases} 
\text{Can}^m(p, E) & \text{if } s^+ \in E \\
\text{Can}^m(q, E) & \text{if } s^- \in E \\
\text{Can}^+(p, E) \cup \text{Can}^-(q, E) & \text{if } s^\perp \in E
\end{cases}
\]

- Note that replacing \((\emptyset, \emptyset)\) by \(\text{Must}(p, E) \cap \text{Must}(q, E)\) would permit speculative execution.
- This would accept for example present \(0\) then emit \(0\) else emit \(0\) end, where information flows backwards wrt. control.
- The physical analogy is that this would allow reasoning based on the law of excluded middle \((x \lor \neg x = 1)\)—and electrons cannot perform this kind of reasoning (cf. unstable Hamlet circuit).

Definitions of *Must* and *Can*

- Sequencing:

\[
\text{Must}(p \setminus s, E) = \begin{cases} 
\text{Must}(p, E \ast s^+) \setminus s & \text{if } s \in \text{Must}_5(p, E \ast s^+) \\
\text{Must}(p, E \ast s^-) \setminus s & \text{if } s \notin \text{Can}^+_5(p, E \ast s^-) \\
\text{Must}(p, E \ast s^\perp) \setminus s & \text{otherwise}
\end{cases}
\]

\[
\text{Can}^m(p \setminus s, E) = \begin{cases} 
\text{Can}^+(p, E \ast s^+) \setminus s & \text{if } m = + \text{ and } s \in \text{Must}_5(p, E \ast s^+) \\
\text{Can}^m(p, E \ast s^-) \setminus s & \text{if } s \notin \text{Can}^+_5(p, E \ast s^-) \\
\text{Can}^m(p, E \ast s^\perp) \setminus s & \text{otherwise}
\end{cases}
\]
Definitions of Must and Can

- Note the Can/Must asymmetry: in the Can-predicate of the local signal declaration, check for $m = +$ before calling Must to avoid speculative computation.
- Otherwise, would accept program.

Would accept program on the following grounds: since the body of the signal statement must emit $S$, the output $O$ cannot be emitted and can be set absent.
- This reasoning speculatively executes the $emit\ S$ statement.

The Max-operator on sets of completion codes is defined as

\[
Max(K, L) = \begin{cases} 
\emptyset & \text{if } K = \emptyset \text{ or } L = \emptyset \\
\{\max(k, l) \mid k \in K, l \in L\} & \text{if } K, L \neq \emptyset 
\end{cases}
\]

Definitions of Must and Can

- Loop:
  \[
  Must(p*, E) = Must(p, E) \\
  Can^m(p*, E) = Can^m(p, E)
  \]

- Parallel:
  \[
  Must(p|q, E) = \langle Must_S(p, E) \cup Must_S(q, E), \\
  Max(Must_k(p, E), Must_k(q, E)) \rangle \\
  Can^m(p|q, E) = \langle Can^m_S(p, E) \cup Can^m_S(q, E), \\
  Max(Can^m_k(p, E), Can^m_k(q, E)) \rangle
  \]

The Trap:

\[
Must(\{p\}, E) = \langle Must_S(p, E), \downarrow Must_k(p, E) \rangle \\
Can^m(\{p\}, E) = \langle Can^m_S(p, E), \downarrow Can^m_k(p, E) \rangle
\]

The Shift:

\[
Must(\uparrow p, E) = \langle Must_S(p, E), \uparrow Must_k(p, E) \rangle \\
Can^m(\uparrow p, E) = \langle Can^m_S(p, E), \uparrow Can^m_k(p, E) \rangle
\]
Definition of the Constructive Behavioral Semantics

The constructive behavioral semantics of a given program is defined by a two-step procedure, yielding the current reaction and the derivative:

1. Compute output event $O$ using $\text{Must}$ and $\text{Cannot}$ predicates
   - This fails if status of some output signal cannot be determined to be $+$ or $-$
2. Compute behavioral transition yielding program derivative
   - This fails if body of some loop is found to terminate instantaneously
   - This also fails if we cannot establish the presence/absence of a local signal

Algorithm to Compute Outputs

```plaintext
function computeOut(P, I)
  E = I ∪ \{s^+ | s ∈ Out(P)\}
  do
    E' = E
    can = Can^+(P, E)
    must = Must^+(P, E)
    E = I ∪ \{s^+ | s ∈ must\}
    ∪ \{s^- | s ∈ Out(P) \can\}
    ∪ \{s^⊥ | s ∈ can \must\}
    while (E' ≠ E)
    if ∃s : s^⊥ ∈ E then error ("not constructive")
  return E
```

$\text{I}$ is an input event, which maps $+$, $-$ to all input signals

$\text{In}(P)$ and $\text{Out}(P)$ give the input and output signals of an program $P$. 

Step 1: Compute output event $O$

Approach:
- Start with undefined $O$ (all output signal statuses = $\bot$)
- Iteratively enrich $O$ using $\text{Must}$ and $\text{Can}$ information
- Terminate when this stabilizes (guaranteed by monotonicity)

Formalize this as computation of a least fixed point (see draft book)
Example for *Can* analysis

Consider the program \( p = !S; S ?! O, 1 \) and environment \( \{ S^\bot, O^\bot \} \).

\[
\begin{align*}
Can^+(!S, \{ S^\bot, O^\bot \}) &= \langle \{ S \}, \{ 0 \} \rangle \\
Must_k(!S, \{ S^\bot, O^\bot \}) &= \{ 0 \} \\
Can^+(O, \{ S^\bot, O^\bot \}) &= \langle \{ O \}, \{ 0 \} \rangle \\
Can^+(1, \{ S^\bot, O^\bot \}) &= \langle 0, \{ 1 \} \rangle \\
Can^+(S ?! O, 1, \{ S^\bot, O^\bot \}) &= \langle \{ O \}, \{ 0, 1 \} \rangle \\
Can^+(S ? S ?! O, 1, \{ S^\bot, O^\bot \}) &= \langle \{ S, O \}, \{ 0, 1 \} \rangle \\
\end{align*}
\]

Gives no new information on signal status

The *Must* analysis in the third line is needed to determine the correct status for the *Can* analysis.

Example for *Must* analysis

Consider the program \( p = !S; S ?! O, 1 \) and environment \( \{ S^\bot, O^\bot \} \)

1. \( Must(!S, \{ S^\bot, O^\bot \}) = \langle \{ S \}, \{ 0 \} \rangle \)
   \( Must(S ?! O, 1, \{ S^\bot, O^\bot \}) = \langle 0, 0 \rangle \)
   \( Must(!S; S ?! O, 1, \{ S^\bot, O^\bot \}) = \langle \{ S \}, 0 \rangle \)

2. Update environment to \( \{ S^+, O^\bot \} \)
3. \( Must(!S, \{ S^+, O^\bot \}) = \langle \{ S \}, \{ 0 \} \rangle \)
   \( Must(O, \{ S^+, O^\bot \}) = \langle \{ O \}, \{ 0 \} \rangle \)
   \( Must(S ?! O, 1, \{ S^+, O^\bot \}) = \langle \{ O \}, \{ 0 \} \rangle \)
   \( Must(!S; S ?! O, 1, \{ S^+, O^\bot \}) = \langle \{ S, O \}, \{ 0 \} \rangle \)

4. Update environment to \( \{ S^+, O^+ \} \)
5. All signals have a defined status \( \to \) done
Definition of the Constructive Semantics

Step 2: Compute transition

Rules are exactly as for logical behavioral semantics—except for changed rules for local signals

\[ \frac{E \cdot s \cdot k \rightarrow E' \cdot s' \cdot k}{p \cdot E \cdot s \rightarrow p' \cdot E' \cdot s'} \quad \text{(sig +)} \]

is replaced with

\[ \frac{s \in \text{Must}_s(p, E \cdot s^\perp)}{p \cdot E \cdot s \rightarrow p' \cdot E' \cdot s'} \quad \text{(csig +)} \]

\[ \frac{E \cdot s \cdot k \rightarrow E' \cdot s' \cdot k}{p \cdot E \cdot s \rightarrow p' \cdot E' \cdot s'} \quad \text{(sig −)} \]

is replaced with

\[ \frac{s \in \text{Cannot}_s(p, E \cdot s^\perp)}{p \cdot E \cdot s \rightarrow p' \cdot E' \cdot s'} \quad \text{(csig −)} \]

The Constructive Operational Semantics

►... is defined by a rewriting-based interpretation scheme

◎ Instead of reasoning about what we must do, just do it

◎ Formal definition and technical treatment of the constructive operational semantics is much heavier than that of the constructive behavioral semantics

► Will still take constructive behavioral semantics as the primary semantics

► Decorate signal declarations with status +, −, ⊥

► Initially, all signals except inputs unknown

► Constructive operational semantics is a micro-step semantics

► Current state indicated by •
The Constructive Operational Semantics

Consider P1 with I present:

```plaintext
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
```

The Constructive Operational Semantics

Only first thread can continue:

```plaintext
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
```

When encountering a “present S” statement in a thread:
- If S is annotated with +, we transfer control to the then branch.
- If S is annotated with −, we transfer control to the else branch.
- If S is annotated by ⊥, we block until the status of S becomes + or −
The Constructive Operational Semantics

Now emit S1:

```
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
```

The Constructive Operational Semantics

Cannot emit S2 any more:

```
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
```

The Constructive Operational Semantics

Now the 2\textsuperscript{nd} branch can continue:

```
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
```

The Constructive Operational Semantics

Now 3\textsuperscript{rd} branch can continue:

```
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
```
The Constructive Operational Semantics

Cannot emit 0 any more:

```
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 0 end
end signal
end module
```

Synchronous Languages Lecture 06 Slide 48

The Constructive Operational Semantics

Now consider P2:

```
module P2:
  output O ;
  signal S in
  • emit S;
  • present O then
    • present S then
      • pause
    end;
  • emit O
end signal
end module
```

Synchronous Languages Lecture 06 Slide 50

The Constructive Operational Semantics

Synchronize the terminated threads:

```
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 0 end
end signal
end module
```

Synchronous Languages Lecture 06 Slide 49

The Constructive Operational Semantics

After 3 microsteps:

```
module P2:
  output O ;
  signal S in
  • present O then
    • present S then
      • pause
    end;
  • emit O
end signal
end module
```

Synchronous Languages Lecture 06 Slide 51
The Constructive Operational Semantics

Perform cannot analysis (as in constructive behavioral semantics)—and set $O$ absent:

```plaintext
module P2:
  output O ;
  signal S in
  emit S;
  •present O then
  present S then
  pause
  end;
  emit O
end
end signal
end module
```

The Constructive Operational Semantics

Statuses evolve monotonically

- Hence avoid most of the recomputations that take place in the constructive behavioral semantics

- Rejecting programs is similar to constructive behavioral semantics

No possible initial microstep $\Rightarrow$ cannot set $O^+$

Potential path to $\text{emit } O \Rightarrow$ cannot set $O^-$

Summary of Constructive Interpretation

Signals:

- Signals are shared objects with status $\{+,-,\perp\}$

- Signal status initialization:
  - Input signals are initialized according to the input event
  - Other signals initialized to $\perp$

- Signal status changes:
  - Status of a signal $S$ changes from $\perp$ to $+$ as soon as an "emit $S$" statement is executed
  - Status of a signal $S$ changes from $\perp$ to $-$ as soon as all the "emit $S$" statements have been found unreachable by the cannot false path analysis
Summary of Constructive Interpretation

Control:
▶ Sequential threads of control forked by parallel statements
▶ When a thread reaches a “present $S$” statement:
  ▶ As long as the status of $S$ is $\perp$:
    ▶ Control remains there, frozen,
  ▶ As soon as $S$ has a non-$\perp$ status:
    ▶ Control can resume
▶ If several threads are enabled, any one of them can be chosen

Program Acceptance:
▶ Given an input, a program is accepted if the analysis succeeds in setting each signal status to a defined value $+$ or $-$
▶ Logical correctness is guaranteed for accepted programs

To Go Further