Synchronous Languages—Lecture 05

Prof. Dr. Reinhard von Hanxleden

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

23 April 2020

Last compiled: April 28, 2020, 13:48 hrs

Esterel III—The Logical Semantics

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

Causality Problems

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

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

```
present A
  then pause
end;
emit 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

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

Can be very complicated because of instantaneous communication

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

```
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
- 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

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!

- Pure Esterel programs can be analyzed for logical correctness by performing exhaustive case analysis
- Given the status of each input signal, one can make all possible assumptions about the global status and check them individually
- Therefore, logical correctness is decidable
- We here generally consider just a single reaction. However, in general one also has to consider all possible sequences of reactions and all possible program states. As there is a finite number of program states, this is still decidable.

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?

- Yes!

Logical Correctness

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

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 O else emit O 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
```

Is P5 logically correct?

- No!
- This is non-reactive

- To make examples shorter, we omit input-output declarations from now on
- Inputs will be written `I`, `I1`, etc., and outputs will be written `O`, `O1`, etc.

Logical Correctness

```
module P6:
present O1 then emit O2 end
||
present O2 then emit O1 end
```

Is P6 logically correct?

- No!
- This is nondeterministic

```
module P7:
present O then pause end;
emit O
```

Is P7 logically correct?

- No!
- This is non-reactive

Logical Correctness

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

Is this logically correct?

- Yes for I present
- Nondeterministic for I absent

Logical Correctness

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

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
- One option:
  - Forbid static self-dependency of signals
  - Similar to acyclicity requirement for electrical circuits
  - This is what the Esterel v4 compiler did

```
module P3:
present O else emit O end
end module
```

$\equiv O = \text{not } O$

```
module P4:
present O emit O end
end module
```

$\equiv O = O$

Instantaneous Feedback

- However, forbidding cycles would also reject the following:

```
module GoodCycle1:
present I then
  present O1 then emit O2 end
else
  present O2 then emit O1 end
end present
```

- O1 and O2 cyclically depend on each other
- However, any given status of I breaks the cycle

Instantaneous Feedback

```
module GoodCycle2:
present O1 then emit O2 end;
pause;
present O2 then emit O1 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?
  - ☺ 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

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)

- The logical behavioral semantics accepts more programs than we would like (for example, program P9 presented in Lecture 03)
- However, the logical behavioral semantics is important in that all other semantics should be a refinement of it, and it is also a natural starting point
- The constructive semantics are equivalent; the constructive behavioral semantics is the most intuitive, and can be derived fairly directly from the logical behavioral semantics, so we will focus on these two semantics here
- Note that the terminology (and categorization) used in different references (and sometimes within the same reference—e.g., in Berry's draft book) is a bit in flux; the keywords to look out for to distinguish which is which are "logical" vs. "constructive", "state" and "behavioral" vs. "operational"
- In this class, will focus on semantics 1, 2, and 5

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: $E = \{(s, b) \mid b = E(s)\}$

- Allowing to represent events in alternate ways somewhat simplifies the subsequent presentation of the rewriting rules

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
- Define $E * s^{b}$ as event E' of sort $S(E) \cup \{s\}$ with
  - $E'(s) = b$, $E'(s') = E(s')$ for $s' \neq s$

- Note that in the definition of $E * s^{b}$, s may or may not be in $S(E)$; in the former case, the status of s in E is lost in $E * s^{b}$

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

$$\boxed{P \xrightarrow[I]{O} P'}$$

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

Notation and Definitions

- Auxiliary statement transition relation:

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

- 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):

$$E' \subset E$$

- Here, we consider an Esterel program to consist of an input/output signal interface and an executable body
- Note that the event E is an assumption in the sense of the logical semantics

Notation and Definitions

- Given:
  - Program P with body p
  - Input event I
- Define program transition of P by statement transition of p:

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

- These program transitions, yielding an output reaction O and a derivative P', determine the logical behavioral semantics of P

- Note how the definition of the program transition reflects the logical coherence

The Basic Broadcasting Calculus

- For concise presentation of rules: Replace Esterel syntax with terser process-calculus syntax
- Use parenthesis for grouping statements

`nothing`	0
`pause`	1
`emit` s	$!s$
`present` s `then` p `else` q `end`	$s?p, q$
$p; q$	$p; q$
`loop` p `end`	$p*$
$p \parallel q$	$p \vert q$
`signal` s `in` p `end`	$p \setminus s$
`suspend` p `when` s `end`	$s \supset p$
`trap` T `in` p `end`	$\{p\}$
`exit` T	k with $k \geq 2$
[no concrete syntax]	$\uparrow p$

Recall: `trap` T `in` p `end`

- Defines a lexically scoped exit point T for p
- Immediately starts its body p and behaves as p until termination or exit
- If p terminates, so does the trap statement
- If p exits T, then the trap statement terminates instantaneously
- If p exits an enclosing trap U, this exit is propagated by the trap statement
- Is part of pure Esterel

Example

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

$$\equiv$$

$$1; !O1; (1; ((I1 ? !O2, 0) \mid (I3 ? 0, !O3)))*$$

Basic Transition Rules

The null process 0:	$0 \xrightarrow[E]{\emptyset,0} 0$	(null)
The unit delay process 1:	$1 \xrightarrow[E]{\emptyset,1} 0$	(unit delay)
Signal emission:	$!s \xrightarrow[E]{\{s\},0} 0$	(emit)

Deduction Rules

- In addition to simple transition rules, will also use deduction rules
- Hypothesis: If sub-instructions behave like this . . .

$$\frac{p_1 \xrightarrow[E]{E_1',k_1} p_1' \qquad p_2 \xrightarrow[E]{E_2',k_2} p_2' \qquad \text{Other hypotheses}}{\text{Instruction}(p_1, p_2) \xrightarrow[E]{E'(E_1',E_2') \quad K(k_1,k_2)} \text{Instruction}'(p_1', p_2')}$$

- Conclusion: . . . then the compound instruction behaves like that

- The null process 0 terminates instantaneously and rewrites into itself
- The unit delay process 1 waits in the current reaction and rewrites itself into 0 for the next reaction

Deduction Rules—Sequencing

$$\frac{p \xrightarrow[E]{E',k} p' \quad k \neq 0}{p;q \xrightarrow[E]{E',k} p';q} \quad \text{(seq1)}$$

$$\frac{p \xrightarrow[E]{E'_p,0} p' \quad q \xrightarrow[E]{E'_q,k} q'}{p;q \xrightarrow[E]{E'_p \cup E'_q,k} q'} \quad \text{(seq2)}$$

- If the first component of a sequence waits, the sequence also waits
  - For reasons that will become clear later, write waiting as $k \neq 0$ instead of $k = 1$
- If the first component of a sequence terminates, the second is started (in zero delay), in the same environment E, and the emitted signals are merged
  - Using same E for both premises implements forward broadcasting from p to q, as broadcasting invariant of first premise implies $E'_p \subset E$
  - However, with the same reasoning we have backward broadcasting from q to p, conflicting with our requirement for causality—will rule this out later

Deduction Rules—Looping and Parallel

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

$$\frac{p \xrightarrow[E]{E'_p,k} p' \quad q \xrightarrow[E]{E'_q,l} q'}{p|q \xrightarrow[E]{E'_p \cup E'_q,max(k,l)} p'|q'} \quad \text{(parallel)}$$

- Note how the global broadcasting invariant expresses that signals are broadcast between parallel branches: $E'_p \cup E'_q \subset E$ holds iff both $E'_p \subset E$ and $E'_q \subset E$ hold
- Note that parallel constructs where all threads have terminated get cleaned up by the (seq2) rule or (trap1)

Deduction Rules—Conditional

$$\frac{s^+ \in E \quad p \xrightarrow[E]{E',k} p'}{s?p,q \xrightarrow[E]{E',k} p'} \qquad \text{(present } +\text{)}$$

$$\frac{s^- \in E \quad q \xrightarrow[E]{E',k} q'}{s?p,q \xrightarrow[E]{E',k} q'} \qquad \text{(present } -\text{)}$$

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

- $(s_1 \wedge s_2)?p,q$ is $s_1?(s_2?p,q),q$
- $(s_1 \vee s_2)?p,q$ is $s_1?p,(s_2?p,q)$
- $\neg s?p,q$ is $s?q,p$

Example: `loop emit S; pause; emit T end.`

In the process calculus: $(!S;1;!T)*$

Calculating initial reaction, as a derivative tree (*Ableitungsbaum*):

$$\dfrac{\dfrac{\dfrac{!S \xrightarrow[\{S\}]{\{S\},0} 0, \quad 1 \xrightarrow[\{S\}]{\emptyset,1} 0}{!S;1 \xrightarrow[\{S\}]{\{S\},1} 0}\text{(seq2)}}{!S;1;!T \xrightarrow[\{S\}]{\{S\},1} 0;!T}\text{(seq1)}}{(!S;1;!T)* \xrightarrow[\{S\}]{\{S\},1} 0;!T;(!S;1;!T)*}\text{(loop)}$$

See next note for an alternative notation.
Similarly, for next reaction (and all following):

$$0;!T;(!S;1;!T)* \xrightarrow[\{S,T\}]{\{S,T\},1} 0;!T;(!S;1;!T)*$$

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

Deduction Rules—Restriction

$$\frac{p \xrightarrow[E*s^+]{E'*s^+,k} p' \quad S(E') = S(E)\backslash s}{p \setminus s \xrightarrow[E]{E',k} p' \setminus s} \qquad \text{(sig } +\text{)}$$

$$\frac{p \xrightarrow[E*s^-]{E'*s^-,k} p' \quad S(E') = S(E)\backslash s}{p \setminus s \xrightarrow[E]{E',k} p' \setminus s} \qquad \text{(sig } -\text{)}$$

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

- The additional sort condition expresses that the sort of E' does not contain s—this avoids propagating the local status of s outside the $p\backslash s$ statement

Another notation for initial reaction of example from previous note:

$$!S \xrightarrow[\{S\}]{\{S\},0} 0, \quad 1 \xrightarrow[\{S\}]{\emptyset,1} 0 \quad \overset{\text{(seq2)}}{\Longrightarrow} \quad !S;1 \xrightarrow[\{S\}]{\{S\},1} 0$$

$$\overset{\text{(seq1)}}{\Longrightarrow} \quad !S;1;!T \xrightarrow[\{S\}]{\{S\},1} 0;!T$$

$$\overset{\text{(loop)}}{\Longrightarrow} \quad (!S;1;!T)* \xrightarrow[\{S\}]{\{S\},1} 0;!T;(!S;1;!T)*$$

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
```

$$\equiv \quad \{\{0 \mid 1 \mid 2 \mid 3\} \mid 2\}$$

Two Operators on Completion Codes

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

$$\begin{array}{ll} \downarrow k = 0 & \text{if } \; k = 0 \quad \text{or} \quad k = 2 \\ \downarrow k = 1 & \text{if } \; k = 1 \\ \downarrow k = k - 1 & \text{if } \; k > 2 \end{array}$$

- The $\uparrow k$ operator computes completion code of $\uparrow p$ from that of p; want $\{\uparrow p\} \equiv p$

$$\begin{array}{ll} \uparrow k = k & \text{if } \; k = 0 \quad \text{or} \quad k = 1 \\ \uparrow k = k + 1 & \text{if } \; k > 1 \end{array}$$

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	$s \cdot\!\supset p$	$\equiv \{(s?1, 2)^*\}; s \supset p$
`await immediate` s; p	$s \cdot\!\Rightarrow p$	$\equiv \{(s?(\uparrow p; 2), 1)^*\}$
`await` s; p	$s \Rightarrow p$	$\equiv 1; s \cdot\!\Rightarrow p$
`weak abort` p `when immediate` s	$s \cdot\!> p$	$\equiv \{(\uparrow p; 2) \mid s \cdot\!\Rightarrow 2\}$
`weak abort` p `when` s	$s > p$	$\equiv \{(\uparrow p; 2) \mid s \Rightarrow 2\}$
`abort` p `when immediate` s	$s \cdot\!\gg p$	$\equiv s \cdot\!> (s \cdot\!\supset p)$
`abort` p `when` s	$s \gg p$	$\equiv s > (s \supset p)$

Traps—The Rules

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

$$\frac{p \xrightarrow[E]{E',k} p' \quad k = 0 \text{ or } k = 2}{\{p\} \xrightarrow[E]{E',0} 0} \qquad (\text{trap1})$$

$$\frac{p \xrightarrow[E]{E',k} p' \quad k = 1 \text{ or } k > 2}{\{p\} \xrightarrow[E]{E',\downarrow k} \{p'\}} \qquad (\text{trap2})$$

$$\frac{p \xrightarrow[E]{E',k} p'}{\uparrow p \xrightarrow[E]{E',\uparrow k} \uparrow p'} \qquad (\text{shift})$$

Deduction Rules—Suspension

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

$$\frac{p \xrightarrow[E]{E',k} p' \quad k \neq 0}{s \supset p \xrightarrow[E]{E',k} s \cdot\!\supset p'} \qquad (\text{suspend2})$$

Note: It might be a bit surprising that in (trap2), the braces (trap scope) remain in the program derivative when an internal exception is propagated up. However, this works fine: the $\downarrow k$ operator keeps lowering the trap completion code, and as soon as we reach the trap scope corresponding to the exception, everything reduces to nothing. See for example { { !S1; 3; !S2 }; !S3 }:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{!S1 \xrightarrow[\{S1\}]{\{S1\},0} 0, \quad 3 \xrightarrow[\{S1\}]{\emptyset,3} 0}{!S1;3 \xrightarrow[\{S1\}]{\{S1\},3} 0}\,(\text{seq1})}{!S1;3;!S2 \xrightarrow[\{S1\}]{\{S1\},3} 0;!S2}\,(\text{seq2})}{\{!S1;3;!S2\} \xrightarrow[\{S1\}]{\{S1\},2} \{0;!S2\}}\,(\text{trap2})}{\{!S1;3;!S2\};!S3 \xrightarrow[\{S1\}]{\{S1\},2} \{0;!S2\};!S3}\,(\text{seq1})}{\{\{!S1;3;!S2\};!S3\} \xrightarrow[\{S1\}]{\{S1\},0} 0}\,(\text{trap1})$$

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 $P \xrightarrow[I]{O} P'$ 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
- 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 $P \xrightarrow[I]{O} P'$.

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, The Constructive Semantics of Pure Esterel, Draft book, current version 3.0, Dec. 2002 `http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.zip`
- 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`