Expressions

- Data expressions:
  - references to constants or variables
  - ?S yields the current value of signal S
  - pre(?S) yields the value of signal S at the previous instant
- Signal expressions:
  - S: current status of signal *S*
  - pre(S): status of signal *S* at previous instant
  - Boolean expressions over signal statuses (using the logical and, or, not operators, the pre operator and the predefined tick signal). present is considered true, absent false.
  - First instant of a signal *S*:
    - interface signal: first instant of program execution
    - local signal: any instant where the corresponding local signal declaration is entered.

Expressions

- Delay expressions:
  - Used in temporal statements like await or abort.
  - Standard delays:
    - Defined by a signal expression.
    - Never elapse instantaneously.
    - Example: meter and not second
  - Immediate delays
    - Defined as immediate s, where s is a signal expression
    - Can elapse instantaneously.
    - Example: immediate [meter and not second]

Expressions

- Count delays
  - Defined by an integer count expression e followed by a signal expression s.
  - The expression is evaluated only once when the delay is initiated. If the value is 0 or less, it is set to 1. Thus a count delay never elapses instantaneously.
  - There is no immediate count delay, and counts cannot be combined with Boolean signal operators.
  - Example: 3 [second and not meter]

Example (1)

```
module System1:
input A, B, R;
output O;
loop
  [await A || await B]
  emit O
each R
end module
```

Example (2)

- every S do p end awaits the first future occurence (ie not at initialization time) of S to start p.
- every immediate S do p end immediately starts p if I is present at the first instant.

```
module Count1:
input I;
output COUNT:=0:integer;
every I do
  emit COUNT(pre(?COUNT)+1);
end every
end module
```

```
module Count2:
input I;
output COUNT;
var Count:=0:integer in
  every I do
    Count:=Count+1;
    emit(COUNT(Count)
  end every
end var
end module
```

Abortion

- Behavior of abort p when S:
  - In the starting instant, p is immediately started, the initial presence or absence of S being ignored (delayed abort).
  - If p terminates before S occurs, then the whole abort statement terminates.
  - If S occurs while p is not yet terminated, the abort statement immediately terminates and p does not receive control in the current instant (strong abort).
- To make abort sensitive to S in the first instant:

  abort p when immediate S
- To give p control a last time when S occurs:

  weak abort p when S

```
module Speedometer:
input Second, Meter;
output Speed: integer in
  loop
    var Distance:=0: integer in
      abort
        every Meter do
          Distance:=Distance+1
        end every
      when Second do
        emit Speed(Distance)
      end abort
    end var
  end loop
end module
```

Generic Behaviors and Modules

- Each data object used by a module must be declared in that module.
- A data object defined in different submodules must be identically declared.
- Calling modules: run statement. Explicit renaming by '/'.
- Renaming arguments are passed by name and not by position!
- If a name is kept unchanged in a substitution, it need not be passed as a parameter.

```
module TWO_STATES:
input On, Off;
output IsOn, IsOff;
loop
  abort
    sustain IsOff
  when On
  abort
    sustain IsOn
  when Off
end loop
end module

...
signal IsOff in

  run TWO_STATES [signal RadioOn / On,
                  RadioOff / Off,
                  Playing / IsOn]

end
```

Correctness Issues

- Easy to write syntactically correct but semantically nonsensical programs.
- Esterel programs are required to be reactive and deterministic.
- Reactive:
  - A well-defined output for each input
- Deterministic:
  - Only one output for each input.
- Logically correct: reactive and deterministic

Logical Correctness

- Logical coherence law: A signal S is present in an instant if and only if an emit S statement is executed in this instant.
- Logical correctness requires: there exists exactly one status for each signal that respects the coherence law.
- Let a program P and an input I be given:
  - P is logically reactive wrt I: at least one logically coherent global status.
  - P is logically deterministic wrt I: at most one logically coherent global status.
  - P is logically correct wrt I: logically reactive and deterministic.
  - P is logically correct: logically correct wrt all possible input events.

Logical Correctness

- Pure Esterel programs can be analyzed for logical correctness by exhaustive case analysis.
- Given the status of each input signal, one can make all possible assumptions about the global status and checke them individually.
- Logical correctness is decidable ☺ – but NP complete ☹
- Logical correctness can be counter-intuitive – other basis for language semantics needed.

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
```

logically correct

- I present: Assumption S1 present, S2 not present, O not present
  - Justification: The emit S1 statement is executed justifying the assumption S1 present, no emit S2 and emit O statements are executed, justifying the assumption S2 absent and O absent.
- I absent: Assumption S1 absent, S2 present, O present.
  - Justification: The emit S1 statement is not executed justifying the assumption S1 absent, the emit S2 statement is executed justifying the assumption S2 present and the emit O statement is executed justifying the assumption O present.
- All other assumptions can be shown to be logically incoherent.

Logical Correctness

```
module P2:
output O;
  present O
    else emit O
  end present
end module
```

non-reactive

```
module P3:
output O;
  present O
    then emit O
  end present
end module
```

reactive,
but non-deterministic

Logical Correctness

```
module P4:
present O1 then emit O1 end
||
present O1 then
  present O2 else emit O2 end
end
```

logically correct

Acyclicity and Constructiveness

- Esterel programs can be required to be acyclic:
  - No dependency cycles
  - Can be defined precisely and checked at compile time.
  - BUT: good programs will be rejected.

- Weaker property called constructiveness:
  - Cyclic programs can be constructive
  - Can be checked at compile time
  - More programs will be accepted, but constructiveness is harder to check than acyclicity.

Examples

```
module P5:
output O;
  present O
    else emit O
  end present
end module
```

non-reactive

```
module P6:
output O;
  present O
    then emit O
  end present
end module
```

reactive, but non-deterministic

Both are rejected by cyclicity test.

Examples

```
module P7:
input I;
output O1,O2;
present I then
  present O2 then emit O1 end
else
  present O1 then emit O2 end
end present
end module
```

rejected by acyclicity test
reactive and deterministic

```
module P8:
output O1,O2;
present O1 then emit O1 end
||
present [O1 and not O2] then emit O2 end
end module
```

Logically correct by self justification:
unique behavior
O1 and O2 absent

Problem: self justification does not fit with the standard intuition of imperative languages

Which Semantics to Adopt?

- Esterel has been designed as an imperative language.
- Thus, e.g., in present S then p end
  the status of S should not depend on what p might do.
- In other words: things may happen in the same instant, but have to happen in order.
  The ordering implicit in then is not that of time but that of sequential causality.

The Constructive Semantics

- Idea: do not check assumptions about signal statuses, but propagate facts about control flow and signal statuses. Self-justification is replaced by fact-to-fact propagation.
- Three-valued logic for signals: present, absent, unknown.
- In each instant the statuses of the input signals are given by the environment and the statuses of the other signals are initially set to unknown.

The Constructive Semantics

- Three equivalent presentations:
  - Constructive behavioral semantics
    - Derived from the logical behavioral semantics
    - Constructive restrictions are added to the logical coherence rule
  - Constructive operational semantics
    - Based on term rewriting rules defining microstep sequences
    - Simplest way of defining an efficient interpreter
  - Circuit semantics
    - Translation of program into constructive circuits
    - Core of the Esterel v5 compiler.

Constructive Behavioral Semantics

- Logical coherence semantics augmented by reasoning about what a program must or cannot do, both predicates being disjoint and defined in a constructive way.

- The must predicate determines which signals are present and which statements are executed.
- The cannot predicate determines when signals are absent and it serves in pruning out false execution paths.

- A program is accepted as constructive if and only if fact propagation using the must and cannot predicates suffices in establishing presence or absence of all signals.

Constructive Behavioral Semantics

- Logical Coherence Law:
  - A signal S is present in an instant iff an emit S statement is executed in this instant.
- Constructive Coherence Law:
  - 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.

Constructive Behavioral Semantics

- A signal can have three statuses:
  - $+$: known to be present
  - $-$: known to be absent
  - $\perp$: yet unknown
- must and cannot predicates are defined by structural induction on statements.

- *p* ; *q*
  - Must (resp. can) execute *q* if *p* must (resp. can) terminate
- present *S* then *p* else *q* end
  - *S* known to be present -> Test behaves as *p*
  - *S* known to be absent -> Test behaves as *q*
  - *S* yet unknown -> Test can do whatever *p* or *q* can do; there is nothing the test must do.

Example

```
module P1:
input I;
output O;
signal S1, S2 in
  present I then emit S1 end (i1)
  ||
  present S1 else emit S2 end (i2)
  ||
  present S2 then emit O end (i3)
end signal
end module
```

- If I is present:
  - i1 must take its then branch, emit S1 and terminate → S1 present
  - i2 must take its (empty) then branch and cannot take its else branch → emit S2 cannot be executed, S2 cannot be emitted → S2 absent
  - i3 cannot take its then branch → O cannot be emitted and is absent.
- If I is absent:
  - i1 cannot take its then branch → emit S1 cannot be executed → S1 absent
  - i2 must take its then branch → emit S2 must be executed → S2 present.
  - i3 must take its then branch → emit O must be executed → O present.

Constructive Behavioral Semantics

- signal *S* in *p* end
  - Can: recursively analyze *p* with status ⊥ for *S*
  - Must:
    - Assume we already know that we must execute the declaration 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 ⊥ 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
    - If S cannot be emitted:
      - reanalyze *p* in *E* with *S* absent

Constructive Behavioral Semantics – Formal Definition

- Let S be a set of signals.
  An event E is a mapping $E : S \rightarrow \mathrm{B}_\perp = \{+, -, \perp\}$ which assigns a status from $\mathrm{B}_\perp$ to all signals in S.

- Notation:
  - s^+ : $E(s) = +$
  - s^- : $E(s) = -$
  - $E \subseteq E'$: s^+ in $E \Rightarrow s^+$ in E'

- Singleton event $\{s^+\}$:
  $\{s^+\}(s) = +$ and $\{s^+\}(s') = -$ forall $s' \neq s$

- Let an event E for a set S be given, a signal s possibly not in S and a status b in $B_\perp$. Then $E * s^b$ is an event for the set $S \cup \{s\}$ where
  $E*s^b(s) = b$ and $E*s^b(s') = E(s')\ \forall\ s' \neq s$.

Constructive Behavioral Semantics – Formal Definition

- The statements nothing, pause and exit are represented by completion codes $k \geq 0$:
  - nothing is encoded by 0
  - pause is encoded by 1
  - exit T is encoded by 2, if the directly enclosing trap declaration is that of T and n +2 if n trap declarations have to be traversed before reaching that of T.

- Each control thread returns a completion code k≥0 when it has completed its execution in that instant. The completion code is generated by executing a k statement, ie a nothing, pause or exit T kernel statement.

Constructive Behavioral Semantics – Formal Definition

- To handle trap propagation we define two operators

$$\downarrow k = \begin{cases} 0, & \text{if } k = 0 \text{ or } k = 2 \\ 1, & \text{if } k = 1 \\ k-1, & \text{if } k > 2 \end{cases}$$

$$\uparrow k = \begin{cases} k, & \text{if } k = 0 \text{ or } k = 1 \\ k+1, & \text{if } k > 1 \end{cases}$$

Constructive Behavioral Semantics – Formal Definition

- Given a program P with body p and an input event I. A reaction of the program is given by a behavioral transition of the form

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

  where O is an output event and the resulting program P' is the new state reached by P after the reaction. P' is called the derivative of P by the reaction.

- The statement transition relation has the form

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

  where
  - E is an event that defines the status of all signals in the scope of p
  - E' is an event composed of all signals emitted by p in the reaction, k is the completion code returned.

The statement p' is called the derivative of p by the reaction.

Constructive Behavioral Semantics – Formal Definition

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

$$Max(K,L) = \begin{cases} \emptyset & \text{if } K = \emptyset \text{ or } L = \emptyset \\ \{\max\{k,l\}\}, & \text{for } k \in K, l \in L \end{cases}$$

Constructive Behavioral Semantics – Formal Definition

- The Must function determines what must be done in a reaction

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

  $$\text{Must}(p, E) = \langle S, K \rangle$$

  where
  - E is an event,
  - S is the set of signals that p must emit
  - K is the set of completion codes that p must return.

- We write
  $\text{Must}(p, E) = \langle S, K \rangle =: \langle \text{Must}_s(p,E), \text{Must}_k(p,E) \rangle$.

Constructive Behavioral Semantics – Formal Definition

- The function $\text{Cannot}^m(p,E)$ is used to prune out false paths.

  $\text{Cannot}^m(p,E) = \langle \text{Cannot}_s^m(p,E), \text{Cannot}_k^m(p,E) \rangle = \langle S,K \rangle$

  – S is the set of signals that p cannot emit
  – K is the set of completion codes that p cannot exit with when the input event is E.

- $m \in \{+, \perp\}$ indicates whether it is known that the statement p must be executed in the event E. The case $m = -$ will never occur since Cannot will only be called for potentially executable statements.

- In the following, we will use $\text{Can}^m(p,E)$ since it is easier to be defined formally; from this, $\text{Cannot}^m(p,E)$ can be determined by componentwise complementation.