

Synchronous Languages—Lecture 06

Prof. Dr. Reinhard von Hanxleden

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

14 Nov. 2016

Last compiled: November 17, 2016, 9:13 hrs



*Esterel IV—The
Constructive Semantics*

The 5-Minute Review Session

1. What is the *state* of an Esterel program?

The 5-Minute Review Session

1. What is the *state* of an Esterel program? Which implementation alternatives are there to memorize state?

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?

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*?

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*?

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?

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*?

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*?

Overview

The Constructive Semantics

External Justification vs. Self-Justification

The Constructive Behavioral Semantics

The Constructive Operational Semantics

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

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*

```
present I then  
  emit 0  
end
```

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 O end
  end signal
end module
```

- ▶ Is this logically correct?

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 O end
  end signal
end module
```

- ▶ Is this logically correct?
 - ▶ Yes!

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 O end
  end signal
end module
```

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

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 O end
  end signal
end module
```

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

External Justification vs. Self-Justification

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

- ▶ Is this logically correct?

External Justification vs. Self-Justification

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

- ▶ Is this logically correct?
 - ▶ Yes!

External Justification vs. Self-Justification

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

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

External Justification vs. Self-Justification

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

- ▶ Is this logically correct?
 - ▶ Yes!
- ▶ Is this well-behaved wrt information propagation?
 - ▶ No!

External Justification vs. Self-Justification

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

- ▶ 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

External Justification vs. Self-Justification

- ▶ “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

External Justification vs. Self-Justification

- ▶ “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**

External Justification vs. Self-Justification

- ▶ “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”

External Justification vs. Self-Justification

- ▶ “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”

External Justification vs. Self-Justification

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

External Justification vs. Self-Justification

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

```
module P10:  
  present 0 then  
    nothing;  
  end;  
  emit 0
```

External Justification vs. Self-Justification

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

```
module P10:  
  present 0 then  
    nothing;  
  end;  
  emit 0
```

- ▶ This is logically correct

External Justification vs. Self-Justification

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

```
module P10:  
  present 0 then  
    nothing;  
  end;  
  emit 0
```

- ▶ 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

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 Constructive 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

The Constructive 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

The Constructive 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

The Constructive Behavioral Semantics

- ▶ ... retains the spirit of the logical coherence semantics
- ▶ ... adds reasoning about what a program must or cannot do

The Constructive Behavioral Semantics

- ▶ ... 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 Constructive Behavioral Semantics

- ▶ ... 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.

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

The Constructive Behavioral Semantics

- ▶ Define *Must* and *Cannot* predicates by structural induction on statements
- ▶ A signal can have three statuses:
 - ▶ “+”: known to be present
 - ▶ “-”: known to be absent
 - ▶ “⊥”: yet unknown

The Constructive Behavioral Semantics

- ▶ Define *Must* and *Cannot* predicates by structural induction on statements
- ▶ A signal can have three statuses:
 - ▶ “+”: known to be **present**
 - ▶ “-”: known to be **absent**
 - ▶ “⊥”: 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

The Constructive Behavioral Semantics

$p; q$ (Sequence)

- ▶ Must (resp. can) execute q if p must (resp. can) terminate

The Constructive Behavioral Semantics

$p; q$ (Sequence)

▶ Must (resp. can) execute q if p must (resp. can) terminate
present S then p else q end (Test)

The Constructive Behavioral Semantics

$p; q$ (Sequence)

- ▶ Must (resp. can) execute q if p must (resp. can) terminate

present S then p else q end (Test)

- ▶ If S is known to be present:
 - ▶ Test behaves as p

The Constructive Behavioral Semantics

$p; q$ (Sequence)

- ▶ Must (resp. can) execute q if p must (resp. can) terminate

present S then p else q end (Test)

- ▶ If S is known to be present:
 - ▶ Test behaves as p
- ▶ If S is known to be absent:
 - ▶ Test behaves as q

The Constructive Behavioral Semantics

$p; q$ (Sequence)

- ▶ Must (resp. can) execute q if p must (resp. can) terminate

present S then p else q end (Test)

- ▶ If S is known to be present:
 - ▶ Test behaves as p
- ▶ If S is known to be absent:
 - ▶ Test behaves as q
- ▶ If S is yet unknown:

The Constructive Behavioral Semantics

$p; q$ (Sequence)

- ▶ Must (resp. can) execute q if p must (resp. can) terminate

present S then p else q end (Test)

- ▶ If S is known to be present:
 - ▶ Test behaves as p
- ▶ If S is known to be absent:
 - ▶ Test behaves as q
- ▶ If S is yet unknown:
 - ▶ Test can do whatever p or q can do

The Constructive Behavioral Semantics

$p; q$ (Sequence)

- ▶ Must (resp. can) execute q if p must (resp. can) terminate

present S then p else q end (Test)

- ▶ If S is known to be present:
 - ▶ Test behaves as p
- ▶ If S is known to be absent:
 - ▶ Test behaves as q
- ▶ If S is yet unknown:
 - ▶ Test can do whatever p or q can do
 - ▶ **There is nothing the test must do.** In particular, it does not even have to do what both p and q have to do—this is the essence of disallowing speculative execution.

The Constructive Behavioral Semantics

- ▶ 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)

The Constructive Behavioral Semantics

- ▶ 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

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 \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 \perp 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

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 \perp 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
```

Accepting Programs

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


Accepting Programs

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

- ▶ 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

Accepting Programs

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

- ▶ 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!**

Rejecting Programs

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

Rejecting Programs

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

```
module P3:  
  output 0;  
  present 0 else emit 0 end  
end module
```

Rejecting Programs

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

```
module P3:  
  output 0;  
  present 0 else emit 0 end  
end module
```

```
module P4:  
  output 0;  
  present 0 then emit 0 end  
end module
```

Rejecting Programs

- ▶ Constructiveness \implies logical correctness
- ▶ **But not vice versa!**

Rejecting Programs

- ▶ Constructiveness \implies logical correctness
- ▶ **But not vice versa!**

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

Rejecting Programs

- ▶ Constructiveness \implies logical correctness
- ▶ **But not vice versa!**

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

- ▶ Both 01 and 02 can be emitted

Rejecting Programs

- ▶ Constructiveness \implies logical correctness
- ▶ **But not vice versa!**

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

- ▶ Both 01 and 02 can be emitted
- ▶ No signal must be emitted

Rejecting Programs

- ▶ Constructiveness \implies logical correctness
- ▶ **But not vice versa!**

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

- ▶ Both 01 and 02 can be emitted
- ▶ No signal must be emitted
- ▶ No progress—**reject P9!**

Rejecting Programs

Consider variant of P2:

```
module P11:
  signal S
    present 0 then
      emit S;
    present S then
      pause
    end;
    emit 0
  end
end signal
```

Rejecting Programs

Consider variant of P2:

```
module P11:
signal S
  present 0 then
    emit S;
  present S then
    pause
  end;
  emit 0
end
end signal
```

- ▶ Are not allowed to speculatively execute branches

Rejecting Programs

Consider variant of P2:

```
module P11:
  signal S
  present 0 then
    emit S;
  present S then
    pause
  end;
  emit 0
end
end signal
```

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

Rejecting Programs

```
module P12:  
  present 0 then  
    emit 0;  
  else  
    emit 0  
  end
```

Rejecting Programs

```
module P12:  
  present 0 then  
    emit 0;  
  else  
    emit 0  
  end
```

- ▶ Must reject P12 as well!

Rejecting Programs

```
module P12:  
  present 0 then  
    emit 0;  
  else  
    emit 0  
  end
```

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

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

- ▶ *Must* function determines what must be done in a reaction

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

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

- ▶ *Must* function determines what must be done in a reaction

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

- ▶ Has the form $\text{Must}(p, E) = \langle S, K \rangle$
 - ▶ E : partial event, associating status in $B_{\perp} = \{+, -, \perp\}$ 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

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

- ▶ *Must* function determines what must be done in a reaction

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

- ▶ Has the form $\mathit{Must}(p, E) = \langle S, K \rangle$
 - ▶ E : partial event, associating status in $B_{\perp} = \{+, -, \perp\}$ 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:
 - ▶ $\mathit{Must}(p, E) = \langle \mathit{Must}_s(p, E), \mathit{Must}_k(p, E) \rangle$

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

- ▶ *Cannot*^{*m*} function prunes out false paths
- ▶ $\text{Cannot}^m(p, E) = \langle \text{Cannot}_s^m(p, E), \text{Cannot}_k^m(p, E) \rangle = \langle S, K \rangle$
- ▶ Extra argument $m \in \{+, \perp\}$ indicates whether it is known that p must be executed in event E
- ▶ *Can*^{*m*}(p, E) is component-wise complement

Definitions of *Must* and *Can*

- ▶ Completion and signal emission:

$$\text{Must}(k, E) = \text{Can}^m(k, E) = \langle \emptyset, \{k\} \rangle$$

$$\text{Must}(!s, E) = \text{Can}^m(!s, E) = \langle \{s\}, \{0\} \rangle$$

Definitions of *Must* and *Can*

- ▶ Completion and signal emission:

$$\text{Must}(k, E) = \text{Can}^m(k, E) = \langle \emptyset, \{k\} \rangle$$

$$\text{Must}(!s, E) = \text{Can}^m(!s, E) = \langle \{s\}, \{0\} \rangle$$

- ▶ 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:

Definitions of *Must* and *Can*

- ▶ Signal test:

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

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

Definitions of *Must* and *Can*

- ▶ Sequencing:

Definitions of *Must* and *Can*

- ▶ Sequencing:

$$Must(p; q, E) = \begin{cases} Must(p, E) & \\ \text{if } 0 \notin Must_k(p, E) & \\ \langle Must_S(p, E) \cup Must_S(q, E), Must_k(q, E) \rangle & \\ \text{if } 0 \in Must_k(p, E) & \end{cases}$$

$$Can^m(p; q, E) = \begin{cases} Can^m(p, E) & \\ \text{if } 0 \notin Can_k^m(p, E) & \\ \langle Can_S^m(p, E) \cup Can_S^{m'}(q, E), Can_k^m(p, E) \setminus 0 \cup Can_k^{m'}(q, E) \rangle & \\ \text{if } 0 \in Can_k^m(p, E) & \\ \text{with } m' = \begin{cases} + & \text{if } m = + \wedge 0 \in Must_k(p, E) \\ \perp & \text{otherwise} \end{cases} & \end{cases}$$

Definitions of *Must* and *Can*

- ▶ Local signal declaration:

Definitions of *Must* and *Can*

- ▶ Local signal declaration:

$$Must(p \setminus s, E) = \begin{cases} Must(p, E * s^+) \setminus s & \text{if } s \in Must_S(p, E * s^\perp) \\ Must(p, E * s^-) \setminus s & \text{if } s \notin Can_S^+(p, E * s^\perp) \\ Must(p, E * s^\perp) \setminus s & \text{otherwise} \end{cases}$$

$$Can^m(p \setminus s, E) = \begin{cases} Can^+(p, E * s^+) \setminus s & \text{if } m = + \text{ and } s \in Must_S(p, E * s^\perp) \\ Can^m(p, E * s^-) \setminus s & \text{if } s \notin Can_S^+(p, E * s^\perp) \\ Can^m(p, E * s^\perp) \setminus s & \text{otherwise} \end{cases}$$

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

```
present 0 then
  signal S in
    emit S
  ||
  present S else emit 0 end
end
end
```

Definitions of *Must* and *Can*

- ▶ Loop:

Definitions of *Must* and *Can*

- ▶ Loop:

$$\text{Must}(p^*, E) = \text{Must}(p, E)$$

$$\text{Can}^m(p^*, E) = \text{Can}^m(p, E)$$

Definitions of *Must* and *Can*

- ▶ Loop:

$$\text{Must}(p^*, E) = \text{Must}(p, E)$$

$$\text{Can}^m(p^*, E) = \text{Can}^m(p, E)$$

- ▶ Parallel:

Definitions of *Must* and *Can*

- ▶ Loop:

$$\text{Must}(p^*, E) = \text{Must}(p, E)$$

$$\text{Can}^m(p^*, E) = \text{Can}^m(p, E)$$

- ▶ Parallel:

$$\text{Must}(p|q, E) = \langle \text{Must}_S(p, E) \cup \text{Must}_S(q, E), \\ \text{Max}(\text{Must}_k(p, E), \text{Must}_k(q, E)) \rangle$$

$$\text{Can}^m(p|q, E) = \langle \text{Can}_S^m(p, E) \cup \text{Can}_S^m(q, E), \\ \text{Max}(\text{Can}_k^m(p, E), \text{Can}_k^m(q, E)) \rangle$$

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

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

Definitions of *Must* and *Can*

- ▶ Trap:

Definitions of *Must* and *Can*

- ▶ Trap:

$$\text{Must}(\{p\}, E) = \langle \text{Must}_S(p, E), \downarrow \text{Must}_k(p, E) \rangle$$

$$\text{Can}^m(\{p\}, E) = \langle \text{Can}_S^m(p, E), \downarrow \text{Can}_k^m(p, E) \rangle$$

- ▶ Shift:

Definitions of *Must* and *Can*

▶ Trap:

$$\text{Must}(\{p\}, E) = \langle \text{Must}_S(p, E), \downarrow \text{Must}_k(p, E) \rangle$$

$$\text{Can}^m(\{p\}, E) = \langle \text{Can}_S^m(p, E), \downarrow \text{Can}_k^m(p, E) \rangle$$

▶ Shift:

$$\text{Must}(\uparrow p, E) = \langle \text{Must}_S(p, E), \uparrow \text{Must}_k(p, E) \rangle$$

$$\text{Can}^m(\uparrow p, E) = \langle \text{Can}_S^m(p, E), \uparrow \text{Can}_k^m(p, E) \rangle$$

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 *Must* and *Cannot* predicates
 - ▶ This fails if status of some output signal cannot be determined to be $+$ or $-$

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 *Must* and *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

Definition of the Constructive Semantics

Step 1: Compute output event O

Approach:

- ▶ Start with undefined O (all output signal statuses = \perp)
- ▶ Iteratively enrich O using *Must* and *Can* information
- ▶ Terminate when this stabilizes (guaranteed by monotonicity)

Formalize this as computation of a least fixed point (see draft book)

Algorithm to Compute Outputs

```
function computeOut(P, I)
  E = I ∪ {s⊥ | s ∈ Out(P)}
  do
    E' = E
    can = CanS+(P, E)
    must = MustS(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
```


Example for *Can* analysis

Consider the program $p = !S; S? !O, 1$ and environment $\{S^\perp, O^\perp\}$.

Example for *Can* analysis

Consider the program $p = !S; S? !O, 1$ and environment $\{S^\perp, O^\perp\}$.

Example for *Can* analysis

Consider the program $p = !S; S? !O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$\text{Can}^+(!S, \{S^\perp, O^\perp\}) =$$

Example for *Can* analysis

Consider the program $p = !S; S? !O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$\text{Can}^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

Example for *Can* analysis

Consider the program $p = !S; S? !O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$Can^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$Must_k(!S, \{S^\perp, O^\perp\}) =$$

Example for *Can* analysis

Consider the program $p = !S; S? !O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$\text{Can}^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}_k(!S, \{S^\perp, O^\perp\}) = \{0\}$$

Example for *Can* analysis

Consider the program $p = !S; S? !O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$Can^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$Must_k(!S, \{S^\perp, O^\perp\}) = \{0\}$$

$$Can^\perp(!O, \{S^\perp, O^\perp\}) =$$

Example for *Can* analysis

Consider the program $p = !S; S? !O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$Can^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$Must_k(!S, \{S^\perp, O^\perp\}) = \{0\}$$

$$Can^+(!O, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

Example for *Can* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$Can^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$Must_k(!S, \{S^\perp, O^\perp\}) = \{0\}$$

$$Can^+(!O, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$Can^+(1, \{S^\perp, O^\perp\}) =$$

Example for *Can* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$Can^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$Must_k(!S, \{S^\perp, O^\perp\}) = \{0\}$$

$$Can^+(!O, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$Can^+(1, \{S^\perp, O^\perp\}) = \langle \emptyset, \{1\} \rangle$$

Example for *Can* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$Can^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$Must_k(!S, \{S^\perp, O^\perp\}) = \{0\}$$

$$Can^+(!O, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$Can^+(1, \{S^\perp, O^\perp\}) = \langle \emptyset, \{1\} \rangle$$

$$Can^+(S?!O, 1, \{S^\perp, O^\perp\}) =$$

Example for *Can* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$Can^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$Must_k(!S, \{S^\perp, O^\perp\}) = \{0\}$$

$$Can^+(!O, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$Can^+(1, \{S^\perp, O^\perp\}) = \langle \emptyset, \{1\} \rangle$$

$$Can^+(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0, 1\} \rangle$$

Example for *Can* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$Can^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$Must_k(!S, \{S^\perp, O^\perp\}) = \{0\}$$

$$Can^+(!O, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$Can^+(1, \{S^\perp, O^\perp\}) = \langle \emptyset, \{1\} \rangle$$

$$Can^+(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0, 1\} \rangle$$

$$Can^+(!S; S?!O, 1, \{S^\perp, O^\perp\}) =$$

Example for *Can* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$Can^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$Must_k(!S, \{S^\perp, O^\perp\}) = \{0\}$$

$$Can^+(!O, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$Can^+(1, \{S^\perp, O^\perp\}) = \langle \emptyset, \{1\} \rangle$$

$$Can^+(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0, 1\} \rangle$$

$$Can^+(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S, O\}, \{0, 1\} \rangle$$

Example for *Can* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$.

$$Can^+(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$Must_k(!S, \{S^\perp, O^\perp\}) = \{0\}$$

$$Can^+(!O, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$Can^+(1, \{S^\perp, O^\perp\}) = \langle \emptyset, \{1\} \rangle$$

$$Can^+(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{O\}, \{0, 1\} \rangle$$

$$Can^+(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S, O\}, \{0, 1\} \rangle$$

Gives no new information on signal status

Example for *Must* analysis

Consider the program $p = !S; S? !O, 1$ and environment $\{S^\perp, O^\perp\}$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

1. $Must(!S, \{S^\perp, O^\perp\}) =$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

1. $Must(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

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

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

1. $Must(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$
 $Must(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

- $Must(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$
 $Must(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$
 $Must(!S; S?!O, 1, \{S^\perp, O^\perp\}) =$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

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

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

- $Must(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$
 $Must(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$
 $Must(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$
- Update environment to $\{S^+, O^\perp\}$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

1. $Must(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$

$$Must(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$$

$$Must(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$$

2. Update environment to $\{S^+, O^\perp\}$

3. $Must(!S, \{S^+, O^\perp\}) =$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

- $Must(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$
 $Must(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$
 $Must(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$
- Update environment to $\{S^+, O^\perp\}$
- $Must(!S, \{S^+, O^\perp\}) = \langle \{S\}, \{0\} \rangle$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

- $Must(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$
 $Must(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$
 $Must(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$
- Update environment to $\{S^+, O^\perp\}$
- $Must(!S, \{S^+, O^\perp\}) = \langle \{S\}, \{0\} \rangle$
 $Must(!O, \{S^+, O^\perp\}) =$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

- $$\text{Must}(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$$

$$\text{Must}(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$$
- Update environment to $\{S^+, O^\perp\}$
- $$\text{Must}(!S, \{S^+, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(!O, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

- $Must(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$
 $Must(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$
 $Must(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$
- Update environment to $\{S^+, O^\perp\}$
- $Must(!S, \{S^+, O^\perp\}) = \langle \{S\}, \{0\} \rangle$
 $Must(!O, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$
 $Must(S?!O, 1, \{S^+, O^\perp\}) =$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

- $$\text{Must}(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$$

$$\text{Must}(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$$
- Update environment to $\{S^+, O^\perp\}$
- $$\text{Must}(!S, \{S^+, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(!O, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$\text{Must}(S?!O, 1, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

- $$\text{Must}(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$$

$$\text{Must}(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$$
- Update environment to $\{S^+, O^\perp\}$
- $$\text{Must}(!S, \{S^+, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(!O, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$\text{Must}(S?!O, 1, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$\text{Must}(!S; S?!O, 1, \{S^+, O^\perp\}) =$$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

- $$\text{Must}(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$$

$$\text{Must}(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$$
- Update environment to $\{S^+, O^\perp\}$
- $$\text{Must}(!S, \{S^+, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(!O, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$\text{Must}(S?!O, 1, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$\text{Must}(!S; S?!O, 1, \{S^+, O^\perp\}) = \langle \{S, O\}, \{0\} \rangle$$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

- $$\text{Must}(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$$

$$\text{Must}(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$$
- Update environment to $\{S^+, O^\perp\}$
- $$\text{Must}(!S, \{S^+, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(!O, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$\text{Must}(S?!O, 1, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$\text{Must}(!S; S?!O, 1, \{S^+, O^\perp\}) = \langle \{S, O\}, \{0\} \rangle$$
- Update environment to $\{S^+, O^+\}$

Example for *Must* analysis

Consider the program $p = !S; S?!O, 1$ and environment $\{S^\perp, O^\perp\}$

- $$\text{Must}(!S, \{S^\perp, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(S?!O, 1, \{S^\perp, O^\perp\}) = \langle \emptyset, \emptyset \rangle$$

$$\text{Must}(!S; S?!O, 1, \{S^\perp, O^\perp\}) = \langle \{S\}, \emptyset \rangle$$
- Update environment to $\{S^+, O^\perp\}$
- $$\text{Must}(!S, \{S^+, O^\perp\}) = \langle \{S\}, \{0\} \rangle$$

$$\text{Must}(!O, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$\text{Must}(S?!O, 1, \{S^+, O^\perp\}) = \langle \{O\}, \{0\} \rangle$$

$$\text{Must}(!S; S?!O, 1, \{S^+, O^\perp\}) = \langle \{S, O\}, \{0\} \rangle$$
- Update environment to $\{S^+, O^+\}$
- All signals have a defined status \rightarrow 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

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

(sig +)

Definition of the Constructive Semantics

Step 2: Compute transition

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

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

is replaced with

$$\boxed{\frac{s \in \text{Must}_s(p, E * s^\perp) \quad p \xrightarrow[E * s^+]{E' * s^+, k} p' \quad S(E') = S(E) \setminus s}{p \setminus s \xrightarrow[E]{E', k} p' \setminus s}} \quad (\text{csig } +)$$

Definition of the Constructive Semantics

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

is replaced with

$$\frac{s \in \text{Cannot}_s^+(p, E * s^\perp) \quad p \xrightarrow[E*s^-]{E'*s^-,k} p' \quad S(E') = S(E) \setminus s}{p \setminus s \xrightarrow[E]{E',k} p' \setminus s} \quad (\text{csig } -)$$

The Constructive Operational Semantics

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

The Constructive Operational Semantics

- ▶ ... is defined by a rewriting-based interpretation scheme
 - 😊 Instead of reasoning about what we must do, just do it

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

The Constructive Operational Semantics

- ▶ Decorate signal declarations with status \vdash , \dashv , \perp
- ▶ Initially, all signals except inputs unknown
- ▶ Constructive operational semantics is a **micro-step semantics**
 - ▶ Current state indicated by \bullet

The Constructive Operational Semantics

Consider P1 with I present:

```
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

Fork of the parallel statement:

```
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:

```
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 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

Now the 2nd 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 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 3rd 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
```


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 O end
end signal●
end module
```

The Constructive Operational Semantics

Now consider P2:

```
module P2:
  output 0⊥;
  ●signal S⊥ in
    emit S;
    present 0 then
      present S then
        pause
      end;
    emit 0
  end
end signal
end module
```

The Constructive Operational Semantics

After 3 microsteps:

```
module P2:
  output 0+;
  signal S+ in
    emit S;
    •present 0 then
      present S then
        pause
      end;
      emit 0
    end
  end signal
end module
```

The Constructive Operational Semantics

Perform cannot analysis (as in constructive behavioral semantics)—and set 0 absent:

```
module P2:
  output 0-;
  signal S+ in
    emit S;
    •present 0 then
      present S then
        pause
      end;
    emit 0
  end
end signal
end module
```

The Constructive Operational Semantics

Take implicit else branch of test:

```
module P2:
  output 0-;
  signal S+ in
    emit S;
    present 0 then
      present S then
        pause
      end;
    emit 0
  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

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

```
module P3:  
  output 0;  
  present 0 else emit 0 end  
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

```
module P3:  
  output 0;  
  present 0 else emit 0 end  
end module
```

- ▶ No possible initial microstep \implies cannot set O^+

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

```
module P3:  
  output 0;  
  present 0 else emit 0 end  
end module
```

- ▶ No possible initial microstep \implies cannot set O^+
- ▶ Potential path to emit 0 \implies cannot set O^-

Summary of Constructive Interpretation

Signals:

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

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

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

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

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

Summary of Constructive Interpretation

Control:

- ▶ Threads are stopped by termination or by executing `pause` or `exit` statements
- ▶ Parallel statements synchronize stopped threads, as explained in the intuitive semantics
- ▶ Finally, the false path analysis explores all possible instantaneous paths towards emit statements
 - ▶ Takes into account all facts established so far
 - ▶ No speculative reasoning

Summary of Constructive Interpretation

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

- ▶ Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, Robert De Simone, The synchronous languages 12 years later, *Proceedings of the IEEE*, Jan. 2003 vol. 91, issue 1, pages 64–83, <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.1117>