Imperative Synchronous Languages

Last Update: January 1, 2021

Outline

Introduction

- in this chapter, we consider the description of synchronous systems by imperative synchronous languages
- a synchronous system works in cycles, and in each cycle,
  - it reads all inputs $\mathcal{I}$
  - it computes all outputs $\mathcal{O}$ w.r.t. the internal state $\mathcal{S}$
  - it updates the internal state $\mathcal{S}$ for the next cycle

  each such computation step is called a reaction or macro step
- we must therefore describe the function $\mathcal{S} \times \mathcal{I} \to \mathcal{S} \times \mathcal{O}$
- it could be done by synchronous DPNs, where all nodes read one value from all inputs and produce one value on each output

Imperative Synchronous Languages

- in this chapter, we do not consider synchronous DPNs, and instead, consider imperative synchronous languages
- in general, these languages are obtained from any sequential imperative language by
  - introducing a new statement **`pause`**
  - and thereby the distinction between micro- and macro steps:
    - all actions between two **`pause`** statements belong to one reaction, i.e. one macro step
    - these actions are called micro steps
    - all micro steps refer to the same variable environment
      thus, one often says that micro steps are executed in zero time

Understanding Synchronous Languages

Example

```
x = 3;
w1: pause;
x = 5;
y = x;
w2: pause;
```

- in the first reaction, `x` is 3
- then, the control stops at `w1:pause`
- in the second reaction, `x` and `y` are both 5
- then, the control stops at `w2: pause`

Understanding Synchronous Languages

Example

```
x = 3;
w1: pause;
y = x;
x = 5;
w2: pause;
```

- this program is equivalent to the previous one, i.e.
  - in the first reaction, `x` is 3
  - in the second reaction, `x` and `y` are both 5
- the ordering of the micro steps in a macro step does not matter
  - macro steps are **sets** of micro steps
  - micro steps are executed by respecting data dependencies
  - i.e., first execute `x = 5` then `y = x`

synchronous programs dynamically reorder micro steps so that these are executed as if the complete variable environment would already be given at the beginning of the macro step

Dynamic Scheduling of Micro Steps

- the previous example might suggest that we could simply reorder the assignments in programs
- however, this does not work in general
- the execution order of micro steps depends on inputs, thus dynamic scheduling is required:

```
pause;
if(x) {
    y2 = y1 & !x; // = false due →
        to x==true
    y1 = y2 &  x; // = false due →
        to y2==false
} else {
    y1 = y2 &  x; // = false due →
        to x==false
```

Example

```
pause;
y1 = y2 & →
     x;
y2 = y1 & →
    !x;
pause;
```

Dynamic Scheduling of Micro Steps

Example

```
w1: pause;
if(y1) y1 = true;
if(y1)
    if(!y2) y2 = →
       true;
w2: pause;
```

Example

```
w1: pause;
y1 = y1;
y2 = y1 & !y2;
w2: pause;
```

- a second argument against static ordering of assignments is that not only their right hand sides have to be considered, but also their entire trigger conditions
- e.g., this involves conditions of if-statements

Delayed Assignments

Example

```
x = 3;
z = y;
next(y) = x;
w1: pause;
x = 5;
z = y;
w2: pause;
```

- there are also **delayed assignments**
  - their right hand sides are evaluated in the current macro step
  - but the assignment is done in the next macro step
- in the second macro step, y and z have therefore the value 3

Write Conflicts

Example

```
x = 3;
x = 5;
next(y) = 3;
w1: pause;
y = 5;
w2: pause;
```

- **each variable must have a unique value in each macro step**
- assigning different values to a variable in one macro step is therefore a write conflict
- this is not allowed
- the causality analysis of the compiler can also take care of this
- also, absence of typical runtime errors can be checked thereby
  - division by zero
  - out-of-bound access to array elements

Event and Memorized Variables

Example

```
x = 3;
w1: pause;
y = x;
w2: pause;
```

- in the first step, x is 3, and we do not know y's value
- in the second step, y should have the same value as x, but which value is that?
- since there is no action that assigns a value to x, the **reaction to absence** takes place
- it depends on the declaration of x
  - if x is an event variable, then it is reset in the second macro step to the default value 0
  - if x is a memorized variable, then it stores the value 3 of the previous macro step

Thread Interaction

Example

```
module P(event →
    a,b,c) {
    {
     b = true;
     p: pause;
     if(a) b = true;
     r: pause;
    }
  ||
    {
    q:pause;
    if(!b) c = true;
    a = true;
    s: pause;
```

- threads must interact with each other for a correct causal execution
- no problem for the compiler, since threads of the program do not necessarily mean threads in the compiled code
- in the synthesis part, we will explain the compilation of synchronous programs to guarded actions
- threads are no longer visible at that representation

Causality Analysis

Example

```
pause;
y1 = y2;
y2 = y1;
pause;
```

- there are programs where no execution ordering can be found
- compilers have to check at compile-time whether this can happen
- this phase is called causality analysis
- it assures that for all reachable states and all possible inputs, the micro steps can be executed in an ordering where all values are known when needed
- programs that are causally correct can, in principle, be rewritten to eliminate all cyclic dependencies
- however, this leads to an unavoidable blow-up [13, 19, 18, 17, 16]

Imperative Synchronous Languages

- the most popular imperative synchronous language is **Esterel** [7, 9, 2, 3, 4]
- we consider our descendent **Quartz**, which is very similar
- also, some **Statecharts** variants are synchronous
- and of course, all digital hardware circuits, and languages used to describe them (however, neither VHDL nor Verilog are synchronous languages!)
- in the following, we consider the Esterel variant Quartz developed at the TU Kaiserslautern in detail

Variants of Synchronous Languages

- some languages like Statemate Statecharts demand that all assignments are delayed assignments
- thus, there are no causality problems in these languages
- other variants differ in the way the reaction to absence is handled
  - some languages make explicit use of absence values ⊡
  - others, like Quartz define a default value for each type
    (but compiler optimizations may decide that values are not required and are therefore replaced by ⊡ [10])
- causality may also depend on particular definitions [8, 21, 24]

Our Programming Language Quartz

in the following, the language Quartz is briefly presented

- we start with the syntax of the language
  - declarations: using flows, data types, and storage classes
  - expressions
  - statements
- and proceed then with the (operational) semantics
  - SOS reaction rules
  - SOS transition rules
  - an interpreter based on SOS rules
- finally, we consider some specific issues like
  - causality problems
  - schizophrenia problems

Modules in Quartz

Example

```
module →
    NAME(<decls>) {
     <bodyStatement>
}
```

- Quartz programs are organized in modules
- a module can be viewed as a class of an object oriented language

- each module is defined by its name `NAME`, its interface, and its body statement
- modules can be instantiated several times by
  - inserting expressions for the inputs of the module
  - and inserting writable variables for the outputs of the module
- in Quartz, each module must be stored in a single file whose name is the name of the module (with file suffix `.qrz`)

Variable Declarations

each variable must be declared with the following information

- **information flow**
  - determines whether the program can read or write the variable
  - input, output, inout, local and label variables
- **data type**
  - determines the possible values a variable may have
  - typical types: booleans, nat, int, real, arrays, tuples
- **storage class**
  - determines the value if no assignment is executed on a variable
  - memorized variables store their previous value
  - event variables are reset to a default value

Information Flows in Module's Interface

- **input variables**
  - are declared by ? in the interface of a module
  - input variables can only be read by the module
  - their values are given by the environment (or a simulator)
- **output variables**
  - are declared by ! in the interface of a module
  - output variables can only be written by the module
  - their values are uniquely determined by the module itself
- **inout variables**
  - are default declarations (without ? or !) in the interface
  - inout variables can only be read and written by the module
  - values are determined by this module or other modules
  - after linking, inout becomes output

Further Information Flows

in addition to the modules' interfaces, there are further variable declarations

- **local variables**
  - are declared by local variable statements
  - the module can read and write these variables
  - but no other module can access these variables
  - local variables have a scope given by the syntax of the local variable declaration
- **labels**
  - denote control flow locations like **`pause`** statements
  - these variables are implicitly assigned by the control flow
  - no assignments are allowed to these variables
  - they can also not be read in the programs

Storage Classes

- **event variables**
  - are declared by keyword **`event`**
  - the reaction to absence resets these variables to a default value
  - these variables behave like wires in a hardware circuit
  - typical hardware-style variables
- **memorized variables**
  - are declared by keyword **`mem`** or nothing, since **`mem`** is the default
  - the reaction to absence stores the previous values in these variables
  - these variables behave like registers in a hardware circuit
  - typical software-style variables

Finite Scalar Data Types

- **bool**
  - has values **false** and **true**
  - has boolean operations !, &, |, ->, <->
- **nat**{m}
  - has values $0, \ldots, m-1$
  - has numeric operations +, -, *, /, %
  - and relations ==, !=, <, <=, >, >=
- **int**{m}
  - has values $-m, \ldots, m-1$
  - has numeric operations +, -, *, /, %, **abs**
  - and relations ==, !=, <, <=, >, >=

Infinite Scalar Data Types

in addition to the scalar finite types, there are also infinite types

- **`nat`**
  - has values $0, 1, 2, \ldots$
  - has numeric operations +, −, *, /, %,
  - and relations ==, !=, <, <=, >, >=
- **`int`**
  - has values $\ldots, -2, -1, 0, 1, 2, \ldots$
  - has numeric operations +, −, *, /, %,
  - and relations ==, !=, <, <=, >, >=

Further Scalar Data Types: Bitvectors

- **bv**{m}
  - has values 0...0b,...,1...1b (bitvectors of lengths m)
  - has bitwise boolean operations !, &, |, ->, <->
  - bitvector operations for bitvectors y and x $= [x_{k-1}, \ldots, x_0]$, we have:
    - x@y (concatenation)
    - b::m generates bitvector of length m consisting of bits b
    - x{m} extracts bit x_m from x
    - x{m:n} extracts segment $[x_m, \ldots, x_n]$ from x
    - **reverse**(x) is $[x_0, \ldots, x_{k-1}]$
    - **fromArray**(a) converts boolean array a to a bitvector
- **bv**
  - has values 0b, 1b, 00b, 01b,... (bitvectors of arbitrary lengths)
  - has same operations as **bv**{m}
- bitvectors are **not** boolean arrays!: in contrast to arrays, only one write operation is allowed per macro step on a bitvector

Compound Data Types

- arrays
  - for any type α, `[m]`α with some constant `m` is an array of base type α
  - access to array elements is written by square delimiters like `x[i+1]`
  - index expressions may be any expressions within the allowed range
- tuples
  - given $\alpha_1, \ldots, \alpha_n$, the type $\alpha_1 * \ldots * \alpha_n$ is a tuple type
  - one can access elements of a tuple by writing `x.0`, ..., `x.(n-1)`
  - index expressions must evaluate to constants at compile time

Example Declarations

- **event bool** ?x,y,!z
- **mem bool** ?x,y,!z
- **event** ?x,y,!z (means **event bool** ?x,y,!z)
- **event int**{5} ?x,y,!z
- **event** [7]**int**{5} ?x, **event int** y, **mem bool** !z
- **event** ([7]**int**{5} * **bool**)?x
- [6]([7]**int**{5} * **bv**{4})y
- **bool** ?x,y,!z (means **mem bool** ?x,y,!z)
- **int**{5} ?x,y,!z
- **mem bv**{5} ?x,y,!z
- **event** [7]**int**{5} ?x, **mem int** y, **bool** !z
- ([7]**int**{5} * **bool**)?x
- **event** [6]([7]**int**{5} * **bv**{4})y

Type System

- we have already mentioned typical operators on the slides of the types
- more operators are available like type converters
  - **arr2bv**(τ)
  - **tup2bv**(τ)
  - **nat2bv**(τ, n)
  - **int2bv**(τ, n)
  - **bv2nat**(τ)
  - **bv2int**(τ)

Type System

- Quartz is strongly typed,
  it contains neither polymorphic nor dynamic data types
- operations are bit-precise, e.g.

$$\frac{\tau : \mathbf{nat}\{\mathtt{m}\} \quad \pi : \mathbf{nat}\{\mathtt{n}\}}{\tau+\pi : \mathbf{nat}\{\mathtt{m+n-1}\}} \qquad \frac{\tau : \mathbf{int}\{\mathtt{m}\} \quad \pi : \mathbf{int}\{\mathtt{n}\}}{\tau+\pi : \mathbf{int}\{\mathtt{m+n}\}}$$

$$\frac{\tau : \mathbf{nat}\{\mathtt{m}\} \quad \pi : \mathbf{nat}\{\mathtt{n}\}}{\tau*\pi : \mathbf{nat}\{\mathtt{(m-1)*(n-1)+1}\}} \qquad \frac{\tau : \mathbf{int}\{\mathtt{m}\} \quad \pi : \mathbf{int}\{\mathtt{n}\}}{\tau*\pi : \mathbf{int}\{\mathtt{m*n+1}\}}$$

- for a complete list, see [20]
- this allows a compile-time estimation of bounds of expressions

Subtypes

- the type system respects subset inclusions of types, i.e.
  - **bool** is seen as equivalent to **bv**{1}
  - **nat**{m} is contained in **nat**{n} iff m<=n
  - **nat**{m} is contained in **nat**
  - **nat**{m} is contained in **int**{n} iff m<=n
  - **int**{m} is contained in **int**{n} iff m<=n
  - **int**{m} is contained in **int**
  - **nat** is contained in **int**

Quartz Statements

- we next consider the statements of the Quartz language
- most of these statements have been inherited from Esterel [7, 9, 2, 3, 4]
- we first consider a rather complete list of statements
- their behaviors are first informally explained after these lists
- and then presented formally by SOS rules

All Atomic Quartz Statements

assumptions and assertions	
assume(σ);	assumption
assert(σ);	assertion

actions	
x =τ;	immediate assignment
next(x)=τ;	delayed assignment
emit(x);	immediate boolean signal emission
emit next(x);	delayed boolean signal emission

wait statements	
nothing;	empty statement
ℓ:**pause**;	new macro step
ℓ:**halt**;	infinite loop doing nothing
ℓ:**await**(σ);	delayed wait on condition
ℓ:**immediate await**(σ);	immediate wait on condition

Sequential, Parallel, and Branching Control Flow

conditional statements	
if(σ) S_1	conditional statement
if(σ) S_1 **else** S_2	conditional statement
choose S_1 **else** S_2	nondeterministic choice
case (σ_1) **do** S_1 ... (σ_n) **do** S_n **default** S	case statement

sequential and parallel control flow	
$S_1;S_2$	sequential execution
$S_1 \mid\mid S_2$ and $S_1 \&\& S_2$	synchronous parallel execution
$S_1 \mid\mid\mid S_2$ and $S_1 \&\&\& S_2$	asynchronous parallel execution
$S_1 \mid S_2$ and $S_1 \& S_2$	interleaved parallel execution

Loop Statements

loops	
do S **while**(σ);	do-loop
while(σ) S	while-loop
loop S	infinite loop
ℓ:**loop** S **each**(σ);	triggered infinite loop
(ℓ_1, ℓ_2):**every**(σ) S	triggered infinite loop
(ℓ_1, ℓ_2):**immediate every**(σ) S	triggered infinite loop

Abortion and Suspension Statements

abortion	
weak immediate abort S **when**(σ);	weak immediate abortion
weak abort S **when**(σ);	weak delayed abortion
immediate abort S **when**(σ);	strong immediate abortion
abort S **when**(σ);	strong delayed abortion

suspension	
ℓ:**weak immediate suspend** S **when**(σ);	weak immediate suspension
weak suspend S **when**(σ);	weak delayed suspension
ℓ:**immediate suspend** S **when**(σ);	strong immediate suspension
suspend S **when**(σ);	strong delayed suspension

Miscellaneous Statements

generic sequential and parallel control flow	
choose(i=τ..π) S	generic choice
for(i=τ..π) S	generic sequence
for(i=τ..π) **do** η S	generic parallel statements
	where $\eta \in \{\texttt{|}, \texttt{||}, \texttt{|||}, \texttt{\&}, \texttt{\&\&}, \texttt{\&\&\&}\}$

miscellaneous	
$\{\alpha\ x_1, \ldots, x_n;\ S\}$	local declaration
let(x=τ) S	let-abbreviation
during S_1 **do** S_2	during statement
final during S_1 **do** S_2	final during statement
immediate during S_1 **do** S_2	immediate during statement
immediate final during S_1 **do** S_2	immediate final during statement
C : *name*$(\tau_1, \ldots, \tau_n)$;	module instantiation
$\{S\}$	statement block

General Remarks on Statements

- each statement S is started in some macro step $t \in \mathbb{N}$ and may terminate in a step $t + \delta$ $(0 \leq \delta)$
- if $\delta = 0$ holds, S is called instantaneous:
  - its execution does not take time
  - execution of S does only cover micro steps
- if S is not instantaneous, the control flow enters S and will stop somewhere inside S to wait for the next macro step
- due to concurrency,
  the control flow may rest at several locations
- it is possible, and often desirable, that statements do not terminate

assume(σ) and assert(σ)

- assumptions and assertions are instantaneously executed
- assume(σ)
  - assume(σ) tells the compiler that σ holds at this location
  - the compiler will not try to verify this, instead it believes the programmer
- assert(σ)
  - assert(σ) specifies that σ should be checked at this location
  - verification tools will check it, and programs will fail if an assertion is violated
- assumptions and assertions are a nice way to specify properties
- however, they do not work in a modular verification
  since they typically depend on the context

`x = τ` and `next(x) = τ`

- both `x = τ` and `next(x) = τ` are instantaneously executed
- `x` must be a writeable variable and τ must be readable
- the type of τ must be contained in the type of `x`
  - otherwise, an assertion is generated to ensure containment
  - in cases that are clearly unsatisfiable, the type-checking fails
- semantics
  - both statements evaluate the right hand side expression τ in the current macro step to a value v
  - `x = τ` immediately assigns v to the writeable variable `x`
  - `next(x) = τ` assigns v to the writeable variable `x` in the next macro step
- a typical error are assignments like $x = x + 1$

`emit(x)` and `emit next(x)`

- `emit(x)` is always instantaneous
- `x` must be a writeable event variable of boolean type
- `emit(x)` is an abbreviation for `x = true`
- `emit next(x)` is an abbreviation for `next(x) = true`
- emissions are added for historic reasons
  they were used as the assignments for 'event variables' in Esterel

Further Atomic Statements

- `nothing` does nothing and needs no time to do nothing
- `pause`
  - when executed, the control flow stops here (unless there is a surrounding abortion)
  - the current macro step will then end here
  - in the next macro step, the control is resumed from this place (unless there is a surrounding suspension)
  - `pause` is therefore never instantaneous
- `halt` waits for the rest of time, i.e., `halt` $\equiv$ `loop pause`
- the programmer can give the control flow locations defined by `pause` and `halt` names in that ℓ :`pause` and ℓ :`halt` is written

[immediate] await(σ)

- await(σ)
  - when executed, control moves to await(σ), and the macro step ends
  - when the execution resumes in the next macro step, condition σ is checked
  - if σ holds, await(σ) instantaneously terminates
  - otherwise, the control remains at await(σ)
- the variant immediate await(σ) differs in that σ is also checked at starting time, i.e., when started
  - and σ is true, immediate await(σ) behaves as nothing
  - if σ is false, immediate await(σ) behaves as await(σ)

Conditionals

- `if(`σ`)` S_1 `else` S_2
  - if started, evaluate expression σ
  - if σ holds, immediately execute S_1, otherwise execute S_2
- one may also write `if(`σ`)` S_1 as abbreviation for `if(`σ`)` S_1 `else nothing`
- more general form:

$$\left[\begin{array}{l}\texttt{case}\\ \quad (\sigma_1)\,\texttt{do}\ S_1\\ \quad (\sigma_2)\,\texttt{do}\ S_2\\ \quad \vdots\\ \quad (\sigma_n)\,\texttt{do}\ S_n\\ \texttt{default}\ S_{n+1}\end{array}\right] \equiv \left[\begin{array}{l}\texttt{if}(\sigma_1)\ S_1\\ \texttt{else if}(\sigma_2)\ S_2\\ \vdots\\ \texttt{else if}(\sigma_n)\ S_n\\ \texttt{else}\ S_{n+1}\end{array}\right]$$

Nondeterministic Choice

- `choose` S_1 `else` S_2
  - whenever started, a nondeterministic choice is made to decide whether S_1 or S_2 is executed
  - thus, it behaves like `if(x)` S_1 `else` S_2 with an oracle input `x`
- the statement is not intended for implementing deterministic controllers
- it is, however, useful for modeling the behavior of environments
- and also for writing test cases for simulation

$S_1 ; S_2$

- sequence $S_1 ; S_2$ is executed as follows
  - when started at time t, start S_1 immediately at time t
  - if S_1 terminates at time $t + \delta_1$, then S_2 is started at time $t + \delta_1$
  - note that $\delta_1 = 0$ may hold, which implies that S_1 and S_2 are then both started at time t
  - $S_1 ; S_2$ terminates if S_2 terminates
  - $S_1 ; S_2$ is instantaneous if both S_1 and S_2 are instantaneous
- moving the control from S_1 to S_2 does not take time

$\rightsquigarrow$ the sequence operation does not take time

$S_1 \,||\, S_2$

- synchronous parallel $S_1 \,||\, S_2$ is executed as follows:
  - if $S_1 \,||\, S_2$ is started at time t, S_1 and S_2 are started at time t
  - if S_1 and S_2 terminate at time $t + \delta_1$ and $t + \delta_2$, respectively, then $S_1 \,||\, S_2$ terminates at time $t + \max(\{\delta_1, \delta_2\})$
  - as long as the control is inside S_1 and S_2, both S_1 and S_2 execute their macro steps synchronously in lockstep
  - S_1 and S_2 may interact during concurrent execution
- curly braces $\{\ldots\}$ are used to determine priorities to avoid ambiguities due to the grammar:
  $P_1; P_2 || Q_1; Q_2$ is parsed as $P_1; \{P_2 || Q_1\}; Q_2$

$S_1 \,|\, S_2$

- interleaved parallel $S_1 \,|\, S_2$ is executed as follows:
  - if $S_1 \,|\, S_2$ is started at time t, S_1 and S_2 are started at time t
  - if S_1 and S_2 terminate at time $t + \delta_1$ and $t + \delta_2$, respectively, then $S_1 \,|\, S_2$ terminates at time $t + \max(\{\delta_1, \delta_2\})$
  - as long as the control is inside S_1 and S_2, a nondeterministic choice is made on whether the step of S_1 or the step of S_2 is executed
- similar to timesharing of tasks running on a single processor

$S_1 \,||\!|\, S_2$

- asynchronous parallel $S_1 \,||\!|\, S_2$ of S_1 and S_2 is executed as follows:
  - if $S_1 \,||\!|\, S_2$ is started at time t, S_1 and S_2 are started at time t
  - if S_1 and S_2 terminate at time $t + \delta_1$ and $t + \delta_2$, respectively, then $S_1 \,||\!|\, S_2$ terminates at time $t + \max(\{\delta_1, \delta_2\})$
  - as long as the control is inside S_1 and S_2, at least one of the steps of S_1 and S_2 is executed
- thus, $S_1 \,||\!|\, S_2$ is somehow the union of $S_1 \,|\, S_2$ and $S_1 \,||\, S_2$

$S_1 | S_2$, $S_1 || S_2$, and $S_1 ||| S_2$

```
module →
    Test(event →
    [4]bool a,b) {
    {emit(a[0]);
     p1: pause;
     emit(a[1]);
     p2: pause;
     emit(a[2]);
     p3: pause;
     emit(a[3]);
    }
  ||
    {emit(b[0]);
     q1: pause;
     emit(b[1]);
     q2: pause;
     emit(b[2]);
     q3: pause;
     emit(b[3]);
```

- using `||`, the only behavior is to emit `a[i]` and `b[i]` in step i
- using `|`, first `a[0]` and `b[0]`, and afterwards, exactly one of the `a[i]` and `b[j]` is emitted
- using `|||`, first `a[0]` and `b[0]`, and afterwards, either one or both of `a[i]` and `b[j]` is emitted
- in all cases, the emissions on `a` and those of `b` appear in the order `a[0]`,...,`a[3]` and `b[0]`,...,`b[3]`

$S_1 \& S_2$, $S_1 \&\& S_2$ and $S_1 \&\&\& S_2$

- $S_1 \& S_2$, $S_1 \&\& S_2$ and $S_1 \&\&\& S_2$ are variants of $S_1 \mid S_2$, $S_1 \mid\mid S_2$ and $S_1 \mid\mid\mid S_2$
  - at starting time, both S_1 and S_2 are started
  - if S_1 and S_2 terminate at time $t + \delta_1$ and $t + \delta_2$, respectively, then $S_1 \& S_2$, $S_1 \&\& S_2$ and $S_1 \&\&\& S_2$ terminate at time $t + \min(\{\delta_1, \delta_2\})$
  - recall that $S_1 \mid S_2$, $S_1 \mid\mid S_2$ and $S_1 \mid\mid\mid S_2$ terminate at time $t + \max(\{\delta_1, \delta_2\})$
- **the difference is only the termination**
- the first statement S_i that terminates aborts the execution of the other one

General Remarks on Loop Statements

- Quartz knows several loop statements
  - `loop` S
  - `do` S `while(`σ`);`
  - `while(`σ`)` S
  - `loop` S `each(`σ`);`
  - `every(`σ`)` S
- these are described on the following slides
- very important: body statement must not be instantaneous
- every macro step should consist of finitely many micro steps

`loop` S

- `loop` S is executed as follows:
  - first, S is executed
  - if S terminates at time $t + \delta$,
    then S is again started at time $t + \delta$
  - $\rightsquigarrow$ `loop` S is equivalent to S;`loop` S
- we will see later that abortion statements can abort such loops

do S while(σ)

- do S while(σ) is executed as follows:
  - if started at time t, S is started at time t without checking σ
  - if S terminates at time $t + \delta$, then
    - σ is evaluated
    - if σ is true, then do S while(σ) is executed again
    - if σ is false, then the loop terminates

$\rightsquigarrow$ loop S can be rewritten as do S while(true)

`while`(σ) S

- `while`(σ) S is executed as follows:
  - first, σ is evaluated
  - if σ is true, then `do` S `while`(σ) is executed
  - if σ is false, then the loop terminates instantaneously

$\rightsquigarrow$ `while`(σ) S can be rewritten as `if`(σ) `do` S `while`(σ)

`loop` S `each(`σ`)` and `every(`σ`)` S

- `loop` S `each(`σ`)` is executed as follows:
  - when started, S is started while ignoring σ
  - while S is running, condition σ is evaluated in each macro step
    - if σ is false, the execution of S is not disturbed
    - if σ is true, then the current execution of S is aborted, and `loop` S `each(`σ`)` is re-started
  - moreover, if S should terminate before σ became true, then the statement waits for the next macro step where σ holds, and re-starts then `loop` S `each(`σ`)`
- `[immediate] every(`σ`)` S can be replaced by

`[immediate] await(`σ`); loop` S `each(`σ`)`

Abortion Statements

- abort comes in four variants:
  - `abort` S `when(`σ`)`
  - `weak abort` S `when(`σ`)`
  - `immediate abort` S `when(`σ`)`
  - `weak immediate abort` S `when(`σ`)`
- `abort` S `when(`σ`)` is executed as follows
  - when started, S is started and σ is ignored
  - while S is running, σ is evaluated in each macro step
    - if σ is false, the execution of S is not disturbed
    - if σ is true, then the current execution of S is aborted, i.e.,
      the abortion statement instantaneously terminates,
      none of the micro steps to be executed by S in that step are executed
- strong abortion means: check abortion due to σ **before** executing the micro steps of S
- `immediate abort` S `when(`σ`)` means `if(!`σ`) abort` S `when(`σ`)`

Weak Abortion Statements

- `weak abort` S `when(`σ`)` is executed as follows
  - when started, S is started and σ is ignored
  - while S is running, σ is evaluated in each macro step
    - if σ is false, the execution of S is not disturbed
    - if σ is true, then
      the current execution of S is aborted, i.e.,
      the abortion statement instantaneously terminates,
      all of the micro steps to be executed by S in that step are nevertheless executed
- weak abortion means: check abortion due to σ **after** executing the micro steps of S
- `weak immediate abort` S `when(`σ`)` analogously checks also σ at starting time

Suspension Statements

- suspend statement comes also in four variants:
  - `suspend` S `when(`σ`)`
  - `weak suspend` S `when(`σ`)`
  - `immediate suspend` S `when(`σ`)`
  - `weak immediate suspend` S `when(`σ`)`
- `suspend` S `when(`σ`)` is executed as follows
  - when started, S is started and σ is ignored
  - while S is running, σ is evaluated in each macro step
    - if σ is false, the execution of S is not disturbed
    - if σ is true, then the current execution of S is suspended, i.e., the control flow remains at the current locations in S, and none of the micro steps to be executed by S in that step are executed
- strong suspension means: check suspension due to σ **before** executing the micro steps of S

Suspension Statements

- `weak suspend` S `when(`σ`)` is executed as follows
  - when started, S is started and σ is ignored
  - while S is running, σ is evaluated in each macro step
    - if σ is false, the execution of S is not disturbed
    - if σ is true, then the current execution of S is weakly suspended, i.e., the control flow remains at the current locations in S, and all of the micro steps to be executed by S in that step are executed
- weak suspension means: check suspension due to σ **after** executing the micro steps of S
- weak suspension can implement loops, e.g.

```
weak suspend
    pause;
    next(x) = x+1;
when(true);
```

Generic Statements

- Quartz has several generic statements:
  - `choose(i=`τ`..`π`)` S
  - `for(i=`τ`..`π`)` S (generic sequence)
  - `for(i=`τ`..`π`) do` η S where $\eta \in \{$`|`, `||`, `|||`, `&`, `&&`, `&&&`$\}$
- their meaning is as follows
  - expressions τ and π are evaluated
  - then, variable `i` is replaced in S by all numbers $\{\tau, \ldots, \pi\}$ to obtain instances S_i
  - these instances are then combined by `;`, `|`, `||`, `|||`, `&`, `&&`, `&&&` or `choose` to obtain non-generic statements
- note that all of these operations are associative!

Local Variable Declarations and Let-Abbreviations

- $\{\alpha\ x_1, \ldots, x_n;\ S\}$
  - new variables $x_1, \ldots, x_n$ are declared
  - they can be read and written in S
  - they are not known outside S
  - shadowing is currently not allowed (i.e., none of the x_i must already exists)
  - α must specify the storage class and the data type of the x_i
- `let(x=`τ`)` S
  - simply abbreviates τ in S by `x`
  - its implementation does not even require a new variable

[immediate] [final] during S_1 do S_2

- S_2 must always be instantaneous
- during S_1 do S_2 is executed as follows:
  - when started, start S_1 and and ignore S_2
  - while S_1 is running, but not terminating, extend the macro steps of S_1 with those of S_2
- immediate during S_1 do S_2
  adds S_2 also at starting time of S_1
- final during S_1 do S_2
  adds S_2 also at termination time of S_1
- immediate final during S_1 do S_2
  adds S_2 both at starting and at termination time of S_1

Example Programs: Button, ABRO, Speed

- to conclude the informal introduction, let's consider some example programs

A Simple Button

Example

```
module bt(event ?pressed,
          !stOff,!stOn) →
              {
  loop {
    abort
      loop {
        emit(stOff);
        pause;
      }
    when(pressed);
    abort
      loop {
        emit(stOn);
        pause;
      }
    when(pressed);
  }
```

- when started, `emit(stOff);` is executed
- this is repeated until `pressed` occurs
- then, `emit(stOn);` is executed until `pressed` occurs again
- and this repeats forever

Example: ABRO

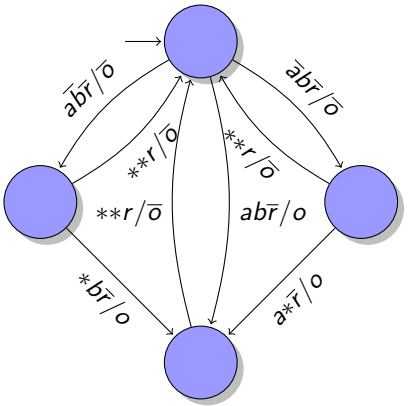
ABRO

The system has boolean inputs a, b, r, and an output o. Output o shall be true as soon as both inputs a and b have been true. This behavior should be restarted if r is true.

- problem: what if a, b and r are true at the same time?

$\rightarrow$ should we make o present?

Mealy Machine for ABRO



- circles are automaton states
- label $a\bar{b}\bar{r}/o$ means: if $a =$ true and $b = r =$ false is read, then output $o =$ true is generated
- default behavior: remain in state
- finite state machines are perfectly synchronous!

$\rightsquigarrow$ use finite state machines to explain the semantics

Quartz Program ABRO

Example (ABRO)

```
module ABRO(event ?a,?b,?r,!o) {
   loop
      abort {
         await(a); || await(b);
         emit(o);
         await(r);
      } when(r);
}
```

ABRO for More Inputs

Example (ABCRO)

```
module ABCRO(event →
    ?a,?b,?c,?r,!o) {
   loop
      abort {
         await(a); ||
         await(b); ||
         await(c);
         emit(o);
         await(r);
      } when(r);
}
```

- ABRO can be easily extended to more events
- only add new thread `await(c)`
- for n inputs, program has size $O(n)$
- but the finite state machine has $O(2^n)$ states

Program Speed

Example (Speed)

The system has inputs `cm` and `sec`. If `sec` holds, the number of macro steps where `cm` hold so far (not the current one) should be emitted via signal `speed`. The counter should be reset also at this point of time, and if `cm` should hold together with `sec`, this instance of `cm` should count for the next emission of the speed instead of the current one. This behavior should be repeated forever.

- problem: what if *cm* and *sec* hold at the same time?
- we first exclude this case and consider solutions for that later

Program Speed

Example (Speed (Incorrect))

```
module Speed(event ?cm,?sec,nat →
    !speed) {
    nat dist;
    loop {
        dist = 0;
        abort {
            every(cm)
                next(dist) = dist + 1;
        } when(sec);
        speed = dist;
    }
}
```

- the behavior is quite involved
- we learn later that the system always acts as follows:
  - if sec, assign `speed = 0` and `dist = 0`
  - if `!cm&!sec` do nothing
  - if `cm&!sec`, assign **next**`(dist)= dist+1`
- this yields a write conflict if at some time t, we have `cm&!sec` and at $t+1$, we have `sec` since then `dist` will be possibly assigned different values
- moreover, if `cm&sec` holds, `cm` is not counted!

Weak Abortion in Program Speed

Example (Speed (Incorrect))

```
module Speed(event ?cm,?sec,nat →
    !speed) {
    nat dist;
    loop {
        dist = 0;
        weak abort {
            every(cm)
                next(dist) = dist + 1;
        } when(sec);
        speed = dist;
    }
}
```

- we may use a weak abortion?
- after initialization, we have again two states `s0` and `s1` with the same dataflow
  - if `cm` assign **`next`**`(dist)= dist+1`
  - if `sec` assign `dist = 0`
  - if `sec` assign `speed = dist`

$\Longmapsto$ all `cm` are counted, but write conflict on `dist` remains

Immediate Abortion in Program Speed

Example (Speed (Incorrect))

```
module Speed(event ?cm,?sec,nat →
    !speed) {
    nat dist;
    loop {
        dist = 0;
        immediate abort {
            every(cm)
                next(dist) = dist + 1;
        } when(sec);
        speed = dist;
    }
}
```

- we may use an immediate abortion?
- after initialization, we have again two states s0 and s1 with the same dataflow
  - if cm&!sec assign **next**(dist)= dist+1
  - if sec assign dist = 0
  - if sec assign speed = dist

$\Longmapsto$ now the write conflict on dist is resolved, by in case of cm&!sec, the cm signal is not counted!

$\Longmapsto$ moreover, the loop body is now instantaneous

Fixing the Problem by Reincarnation of Local Variables

Example (Speed (Correct))

```
module Speed(event ?cm,?sec,nat →
    !speed) {
    loop {
        weak abort {
            nat dist;
            dist = 0;
            if(cm) next(dist) = dist+1;
            do {
                s0:pause;
                if(cm) next(dist) = →
                    dist+1;
            } while(!sec);
            speed = dist;
        } when(sec);
    }
}
```

- the program has then the following actions
  - if cm&!sec, assign **next**(dist)= dist+1
  - if cm& sec, assign **next**(dist)= dist@1+1
  - if sec, assign dist@1 = 0
  - if sec, assign speed = dist

$\Longrightarrow$ every cm is counted

$\Longrightarrow$ write conflict on dist is resolved by reincarnation

Yet another corrected version

Example (Speed (Correct))

```
module Speed(event ?cm,?sec,nat →
    !speed) {
  nat dist;
  loop {
    if(sec) speed = dist;
    case
      (!cm&!sec) do next(dist) = dist;
      (!cm& sec) do next(dist) = 0;
      ( cm&!sec) do next(dist) = →
          dist+1;
      ( cm& sec) do next(dist) = 1;
    default nothing;
    s0:pause;
  }
}
```

- the program has then the following actions
  - if !cm&!sec, assign **next**(dist)= dist
  - if !cm& sec, assign **next**(dist)= 0
  - if cm&!sec, assign **next**(dist)= dist+1
  - if cm& sec, assign **next**(dist)= 1
  - if sec, assign speed = dist

$\Longrightarrow$ every cm is counted; reset and increment is resolved by assigning 1

$\Longrightarrow$ no reincarnation and no additional variable is used

Core Statements

- the informal description of the behavior of statements is way too imprecise
- there is a tight interaction of concurrent control flows
- we have to formally specify the precise behavior of the statements
- in the following, we define an operational semantics
- to this end, we first reduce the set of statements to a smaller set of core statements

Core Statements

`nothing`	(empty statement)
ℓ : `pause`	(new macro step)
$x = \tau$, `next`$(x) = \tau$	(assignments)
`if`(σ) S_1 `else` S_2	(conditional)
S_1 `;` S_2	(sequence)
$S_1 \parallel S_2$	(synchronous concurrency)
`do` S `while`(σ)	(loop)
`[weak] [immediate] abort` S `when`(σ)	(abortion statements)
`[weak] [immediate] suspend` S `when`(σ)	(suspension statements)
$\{\alpha\ x; S\}$	(local variable declaration)

Simple Macro Definitions

- `emit(x);` $\equiv$ `x = true;`
- `emit next(x);` $\equiv$ `next(x) = true;`
- ℓ`:halt` $\equiv$ `do` ℓ`:pause; while(true);`
- ℓ`:await(`σ`)` $\equiv$ `do` ℓ`:pause; while(!`σ`);`
- ℓ`:immediate await(`σ`)` $\equiv$ `while(`σ`)` ℓ`:pause;`
- `while(`σ`)` S $\equiv$ `if(`σ`) do` S `while(`σ`);`
- `loop` S $\equiv$ `do` S `while(true);`
- `loop` S `each(`σ`)` $\equiv$ `loop abort {`S`; halt;} when(`σ`);`
- `every(`σ`)` S $\equiv$ `await(`σ`); loop` S `each(`σ`);`
- `immediate every(`σ`)` S $\equiv$ `immediate await(`σ`); loop` S `each(`σ`);`

Replacing Nondeterminism by Oracles

- the following statements are nondeterministic:
  - `choose` S_1 `else` S_2
  - $S_1 \,|\, S_2$ and $S_1 \,|\,|\, S_2$
  - $S_1 \& S_2$ and $S_1 \&\&\& S_2$
- we can replace them by asking new boolean input variables for the nondeterministic choices
- in case of `choose` S_1 `else` S_2, this is trivial
- for the other statements, we wrap each S_i into a suspend statement and ask oracles which of the possible executions should be performed

Eliminating $P\&\&Q$

Example (Eliminating $P\&\&Q$)

```
event tP,tQ;
  {weak abort
       P;
       emit(tP);
   when(tQ)
  }
||
  {weak abort
       Q;
       emit(tQ);
   when(tP)
  }
```

Eliminating during P do Q

Example (Eliminating during P do Q)

```
  {event t;
   P;
   emit(t);
  }
||
  immediate abort
    loop {
      pause;
      Q;
    }
  when(t)
```

Eliminating `immediate during` P `do` Q

Example (Eliminating `immediate during` P `do` Q)

```
  {event t;
   P;
   emit(t);
  }
||
  {Q;
   immediate abort
     loop {
       pause;
       Q;
     }
   when(t)}
```

Eliminating `final during` P `do` Q

Example (Eliminating `final during` P `do` Q)

```
  {event t;
   P;
   emit(t);
  }
||
  immediate weak abort
    loop {
      pause;
      Q;
    }
  when(t)
```

Eliminating `immediate final during` P `do` Q

Example (Eliminating `immediate final during` P `do` Q)

```
  {event t;
   P;
   emit(t);
  }
||
  {Q;
   immediate weak abort
     loop {
       pause;
       Q;
     }
   when(t)}
```

Uniqueness of Core Language?

- are there alternatives for defining a core language?
- of course, there are many alternatives, for example:
- `pause;` $\equiv$ `await(true);`
- `pause;` $\equiv$ `abort halt; when(true)`

Redundancy of Core Language

- some variants of abort and suspend could be eliminated as well
- `immediate abort` S `when(`σ`)` is equivalent to `if(`σ`) nothing else abort` S `when(`σ`)`
- however, this is not so simple with the weak version
- even pause can be eliminated

$$\texttt{pause} \equiv \left[\begin{array}{l}\texttt{abort} \\ \quad \texttt{immediate suspend} \\ \quad\quad \texttt{nothing} \\ \quad \texttt{when(true)} \\ \texttt{when(true)}\end{array}\right]$$

- nevertheless, the chosen subset is reasonable

Operational Semantics

- we now formally define the semantics
- it is an operational semantics, thus an interpreter can be implemented this way
- two steps are formalized
  - transition of the control flow for a full variable environment by SOS transition rules
  - computation of the reaction, i.e. the full variable environment of a macro step by SOS reaction rules

SOS Transition Rules

- SOS (structural operational semantics) is a way to describe semantics which goes back to Plotkin [14]
- SOS transition rules of Quartz describe the movement of the control flow
  - inputs are
    - statement S
    - environment $\mathcal{E}$ (knows the value $\mathcal{E}(x)$ of each variable x)
  - outputs are
    - statement S', which has to be executed next
    - actions $\mathcal{D}$ (assignments) performed by S in this macro step
    - termination flag $b \in \{\text{true}, \text{false}\}$

$\rightsquigarrow$ problem: to apply the rules, one must know the values of all variables, in particular the values of the output variables

SOS Transition Rules

Transition Rule

$$\langle \mathcal{E}, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, t \rangle$$

- $\mathcal{E}$: environment of the current macro step
- S: statement to be executed
- S': residual statement for the next micro or macro step (depending on t)
- $\mathcal{D}$: actions that are executed in the current step
- t: instantaneous flag; micro step if true, macro step otherwise

SOS Transition Rules with Assumptions

Transition Rule

$$\frac{\varphi_1 \dots \varphi_n}{\langle \mathcal{E}, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, t \rangle}$$

- some transition rules have assumptions
- if conditions $\varphi_1, \dots, \varphi_n$ are true, then we can conclude that also $\langle \mathcal{E}, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, t \rangle$ holds

Invariant for Instantaneous Executions

Transition Rule

$$\langle \mathcal{E}, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathit{true} \rangle$$

- the SOS transition rules maintain the following invariant:

  if the instantaneous execution flag is true, we know that S' is equivalent to `nothing;`
- S' could be something like `nothing;||nothing;`

Environments and Evaluation of Expressions

- the environment $\mathcal{E}$ is a function that maps variables to values
- this models the memory of the program
- we write $[\![\tau]\!]_{\mathcal{E}}$ to evaluate an expression τ in environment $\mathcal{E}$
- for example, $\mathcal{E}(x) = 3$ and $\mathcal{E}(y) = 5$ implies $[\![x + y]\!]_{\mathcal{E}} = 8$
- due to synchrony, environments are constant within a macro step

Atomic Statements

$$\langle \mathcal{E}, \texttt{nothing} \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \{\}, \text{true} \rangle$$

$$\langle \mathcal{E}, \ell : \texttt{pause} \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \{\}, \text{false} \rangle$$

$$\langle \mathcal{E}, \texttt{x=}\tau \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \{\texttt{x=}\tau\}, \text{true} \rangle$$

$$\langle \mathcal{E}, \texttt{next(x)=}\tau \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \{\texttt{next(x)=}\tau\}, \text{true} \rangle$$

Conditional

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \text{true and } \langle \mathcal{E}, S_1 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_1', \mathcal{D}_1, t_1 \rangle}{\langle \mathcal{E}, \texttt{if}(\sigma)\ \ S_1\ \ \texttt{else}\ \ S_2 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_1', \mathcal{D}_1, t_1 \rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \text{false and } \langle \mathcal{E}, S_2 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_2', \mathcal{D}_2, t_2 \rangle}{\langle \mathcal{E}, \texttt{if}(\sigma)\ \ S_1\ \ \texttt{else}\ \ S_2 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_2', \mathcal{D}_2, t_2 \rangle}$$

Sequence

$$\frac{\langle \mathcal{E}, S_1 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_1', \mathcal{D}_1, \mathsf{false} \rangle}{\langle \mathcal{E}, \{S_1; S_2\} \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \{S_1'; S_2\}, \mathcal{D}_1, \mathsf{false} \rangle}$$

$$\frac{\langle \mathcal{E}, S_1 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_1', \mathcal{D}_1, \mathsf{true} \rangle \quad \langle \mathcal{E}, S_2 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_2', \mathcal{D}_2, t_2 \rangle}{\langle \mathcal{E}, \{S_1; S_2\} \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_2', \mathcal{D}_1 \cup \mathcal{D}_2, t_2 \rangle}$$

Parallel Statement

$$\frac{\langle \mathcal{E}, S_1 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_1', \mathcal{D}_1, t_1 \rangle \quad \langle \mathcal{E}, S_2 \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S_2', \mathcal{D}_2, t_2 \rangle}{\langle \mathcal{E}, \{S_1 \parallel S_2\} \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \{S_1' \parallel S_2'\}, \mathcal{D}_1 \cup \mathcal{D}_2, t_1 \wedge t_2 \rangle}$$

do S while(σ) and while(σ) S

$$\frac{\langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{false}\rangle}{\langle \mathcal{E}, \texttt{do } S \texttt{ while}(\sigma)\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \{S'\texttt{; while}(\sigma)\ S\}, \mathcal{D}, \mathsf{false}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false}}{\langle \mathcal{E}, \texttt{while}(\sigma)\ S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \{\}, \mathsf{true}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true} \text{ and } \langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{false}\rangle}{\langle \mathcal{E}, \texttt{while}(\sigma)\ S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \{S'\texttt{; while}(\sigma)\ S\}, \mathcal{D}, \mathsf{false}\rangle}$$

Strong Delayed Abort

$$\frac{\langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{true}\rangle}{\langle \mathcal{E}, \texttt{abort}\ \ S\ \ \texttt{when}(\sigma)\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \mathcal{D}, \mathsf{true}\rangle}$$

$$\frac{\langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{false}\rangle}{\langle \mathcal{E}, \texttt{abort}\ \ S\ \ \texttt{when}(\sigma)\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \left[\begin{array}{l}\texttt{immediate abort}\ \ S'\\ \texttt{when}(\sigma)\end{array}\right], \mathcal{D}, \mathsf{false}\rangle}$$

Strong Immediate Abort

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true}}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{immediate abort } S \\ \texttt{when}(\sigma)\end{array}\right] \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \{\}, \mathsf{true} \rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \text{ and } \langle \mathcal{E}, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{true} \rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{immediate abort } S \\ \texttt{when}(\sigma)\end{array}\right] \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \mathcal{D}, \mathsf{true} \rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \text{ and } \langle \mathcal{E}, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{false} \rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{immediate abort } S \\ \texttt{when}(\sigma)\end{array}\right] \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \left[\begin{array}{l}\texttt{immediate abort } S' \\ \texttt{when}(\sigma)\end{array}\right], \mathcal{D}, \mathsf{false} \rangle}$$

Weak Delayed Abort

$$\frac{\langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{true}\rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{weak abort } S\\ \texttt{when}(\sigma)\end{array}\right]\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \mathcal{D}, \mathsf{true}\rangle}$$

$$\frac{\langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{false}\rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{weak abort } S\\ \texttt{when}(\sigma)\end{array}\right]\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \left[\begin{array}{l}\texttt{weak immediate abort } S'\\ \texttt{when}(\sigma)\end{array}\right], \mathcal{D}, \mathsf{false}\rangle}$$

Weak Immediate Abort

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \text{true} \text{ and } \langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, t\rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{weak immediate abort } S\\ \texttt{when}(\sigma)\end{array}\right]\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \mathcal{D}, \text{true}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \text{false} \text{ and } \langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \text{true}\rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{weak immediate abort } S\\ \texttt{when}(\sigma)\end{array}\right]\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \mathcal{D}, \text{true}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \text{false} \text{ and } \langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \text{false}\rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{weak immediate abort } S\\ \texttt{when}(\sigma)\end{array}\right]\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \left[\begin{array}{l}\texttt{weak immediate abort } S'\\ \texttt{when}(\sigma)\end{array}\right], \mathcal{D}, \text{false}\rangle}$$

Strong Delayed Suspend

$$\frac{\langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{true}\rangle}{\langle \mathcal{E}, \texttt{suspend}\ \ S\ \ \texttt{when}(\sigma)\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \mathcal{D}, \mathsf{true}\rangle}$$

$$\frac{\langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{false}\rangle}{\langle \mathcal{E}, \texttt{suspend}\ \ S\ \ \texttt{when}(\sigma)\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \begin{bmatrix} \texttt{immediate suspend}\ S' \\ \texttt{when}(\sigma) \end{bmatrix}, \mathcal{D}, \mathsf{false}\rangle}$$

Strong Immediate Suspend

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true}}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{immediate suspend } S \\ \texttt{when}(\sigma)\end{array}\right] \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \left[\begin{array}{l}\texttt{immediate suspend } S \\ \texttt{when}(\sigma)\end{array}\right], \{\}, \mathsf{false} \rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \text{ and } \langle \mathcal{E}, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{true} \rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{immediate suspend } S \\ \texttt{when}(\sigma)\end{array}\right] \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \mathcal{D}, \mathsf{true} \rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \text{ and } \langle \mathcal{E}, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{false} \rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{immediate suspend } S \\ \texttt{when}(\sigma)\end{array}\right] \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \left[\begin{array}{l}\texttt{immediate suspend } S' \\ \texttt{when}(\sigma)\end{array}\right], \mathcal{D}, \mathsf{false} \rangle}$$

Weak Delayed Suspend

$$\frac{\langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{true}\rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{weak suspend } S\\ \texttt{when}(\sigma)\end{array}\right]\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \mathcal{D}, \mathsf{true}\rangle}$$

$$\frac{\langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \mathsf{false}\rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{weak suspend } S\\ \texttt{when}(\sigma)\end{array}\right]\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \left[\begin{array}{l}\texttt{weak immediate suspend } S'\\ \texttt{when}(\sigma)\end{array}\right], \mathcal{D}, \mathsf{false}\rangle}$$

Weak Immediate Suspend

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \text{true and } \langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, t\rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{weak immediate suspend } S\\ \texttt{when}(\sigma)\end{array}\right]\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \left[\begin{array}{l}\texttt{weak immediate suspend } S\\ \texttt{when}(\sigma)\end{array}\right], \mathcal{D}, \text{false}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \text{false and } \langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \text{true}\rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{weak immediate suspend } S\\ \texttt{when}(\sigma)\end{array}\right]\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \mathcal{D}, \text{true}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \text{false and } \langle \mathcal{E}, S\rangle \twoheadrightarrow_{\mathbb{Q}} \langle S', \mathcal{D}, \text{false}\rangle}{\langle \mathcal{E}, \left[\begin{array}{l}\texttt{weak immediate suspend}\\ \quad S\\ \texttt{when}(\sigma)\end{array}\right]\rangle \twoheadrightarrow_{\mathbb{Q}} \langle \left[\begin{array}{l}\texttt{weak immediate suspend}\\ \quad S'\\ \texttt{when}(\sigma)\end{array}\right], \mathcal{D}, \text{false}\rangle}$$

Computing the Reaction

- using the SOS transition rules, we can compute the statement that has to be executed in the next macro step
- this is the update of the internal control flow state
- it remains to determine the outputs for given inputs and a given statement
- to this end, we make use of SOS reaction rules
- these rules work with an incomplete environment $\mathcal{E}$
- this environment $\mathcal{E}$ is then completed step by step
- to this end, we estimate the sets of assignments that must/can be executed

Incomplete Environments

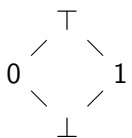
- to compute the reactions,
  we start with an incomplete environment $\mathcal{E}_0$
- $\mathcal{E}_0$ has known values for all inputs
- but has no values for the outputs
- to this end, we make use of the value $\perp$ that indicates that a value is not yet known
- initially, we thus have $\mathcal{E}_0(x) = \perp$ for all outputs
- we also introduce $\top$ meaning that the value cannot be computed due to a runtime error

Information Ordering

- on $\mathcal{D} \cup \{\bot, \top\}$, we introduce a partial order relation $\preceq$ as follows

$$x \preceq y :\Leftrightarrow (x = \bot) \vee (x = y) \vee (y = \top)$$

- $x \preceq y$ means that y contains more information than x
- $\preceq$ is not a total order (e.g. neither $0 \preceq 1$ nor $1 \preceq 0$ holds)
- $\preceq$ is a lattice, since for all elements x, y, we have $\sup(\{x, y\})$ and $\inf(\{x, y\})$
- e.g. for booleans, we have



sup()	⊥	0	1	⊤
⊥	⊥	0	1	⊤
0	0	0	⊤	⊤
1	1	⊤	1	⊤
⊤	⊤	⊤	⊤	⊤

inf()	⊥	0	1	⊤
⊥	⊥	⊥	⊥	⊥
0	⊥	0	⊥	0
1	⊥	⊥	1	1
⊤	⊥	0	1	⊤

Four-Valued Logic

- we extend all operations on $\mathcal{D} \cup \{\bot, \top\}$
- typically $\bot \circledast x = \bot$
- however, sometimes $\bot \circledast x \neq \bot$, since the result is already determined by one of the arguments, e.g.:

$\wedge$	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	$\bot$	$\top$
0	0	0	0	$\top$
1	$\bot$	0	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\vee$	$\bot$	0	1	$\top$
$\bot$	$\bot$	$\bot$	1	$\top$
0	$\bot$	0	1	$\top$
1	1	1	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

x	$\neg x$
$\bot$	$\bot$
0	1
1	0
$\top$	$\top$

- Why are these four-valued extensions of the boolean operations chosen?

Four-Valued Logic

- Are the definitions of the four-valued functions unique?
- we have to maintain the values for boolean operands, and we have to enforce monotonicity:

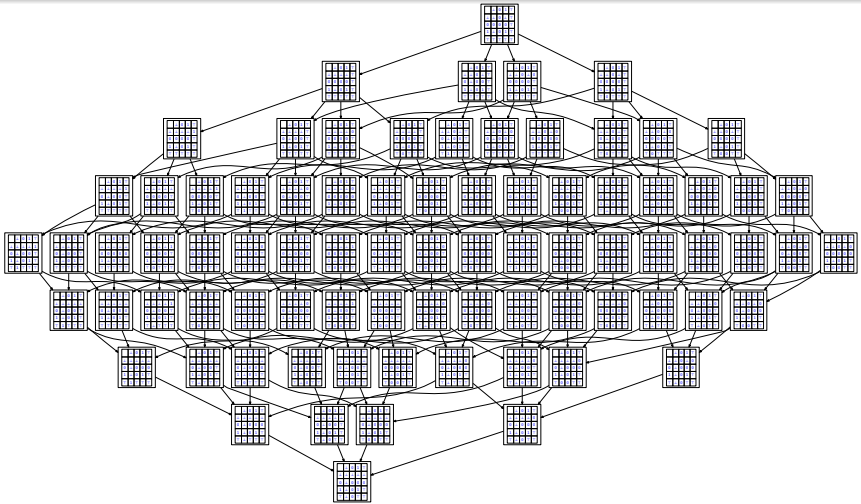
$$x_1 \preceq x_2 \wedge y_1 \preceq y_2 \rightarrow x_1 \circledast y_1 \preceq x_2 \circledast y_2$$

- to allow progress of information, we also want $\perp \wedge 0 = 0 \wedge \perp = 0$ and analogously $\perp \vee 1 = 1 \vee \perp = 1$
- based on these requirements, still many definitions are possible (see next slide)

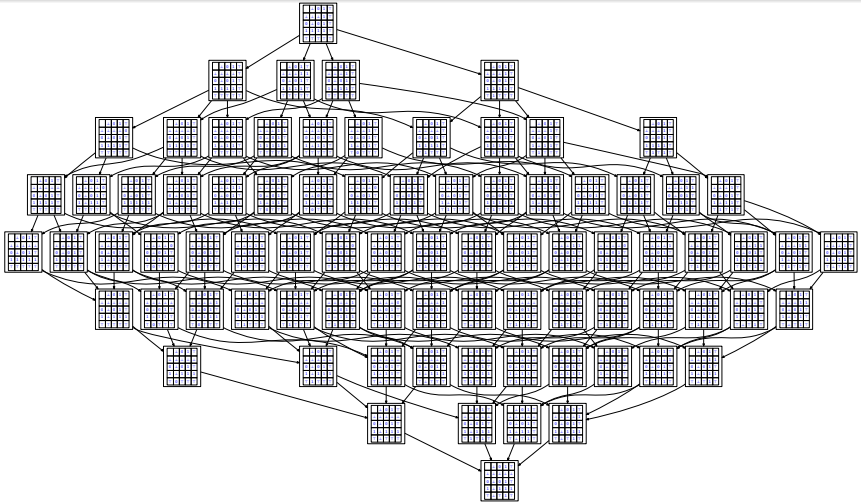
$\Longmapsto$ the greatest ones are chosen

- further enforcing commutativity for conjunction and disjunction reduces this to nine definitions (greatest ones are retained)

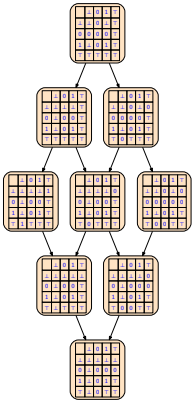
Four-Valued Logic (81 Solutions for Conjunction)

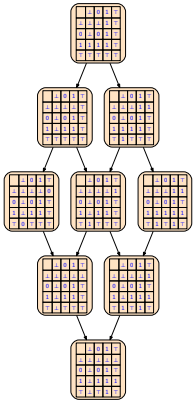


Four-Valued Logic (81 Solutions for Disjunction and Conjunction)



Four-Valued Logic (All Solutions for Commutative Con-/Disjunction)





- there are nine solutions
- these form a lattice
- hence, there is a greatest and a least four-valued extension
- one could prefer the one that contains most boolean values (as we did)
- or the greatest solution which is strict in the sense that one operand's failed evaluation is not overwritten by the other operand
- in what follows, the particular choice does not matter

Lattice of Incomplete Environments

- for a given program P, we consider the set $\mathfrak{E}_P$ of all incomplete environments $\mathcal{E}$ mapping the variables of P to values according to their types including $\bot$ and $\top$
- we also introduce a partial order relation on environments as follows

$$\mathcal{E}_1 \sqsubseteq \mathcal{E}_2 :\Leftrightarrow \forall x \in \mathcal{V}.\ \mathcal{E}_1(x) \preceq \mathcal{E}_2(x)$$

- $\mathcal{E}_1 \sqsubseteq \mathcal{E}_2$ means that $\mathcal{E}_2$ contains more information than $\mathcal{E}_1$
- $(\mathfrak{E}_P, \sqsubseteq)$ is a lattice:
  - $\mathcal{E}_{\mathsf{sup}}(x) := \mathsf{sup}(\{\mathcal{E}_1(x), \mathcal{E}_2(x)\})$
  - $\mathcal{E}_{\mathsf{inf}}(x) := \mathsf{inf}(\{\mathcal{E}_1(x), \mathcal{E}_2(x)\})$

Evaluation of Expressions using incomplete Environments

- given an incomplete environment $\mathcal{E}$, and a program expression σ, we define its evaluation $[\![\sigma]\!]_{\mathcal{E}}$ as expected, i.e.,
  - for variables x, we define $[\![x]\!]_{\mathcal{E}} := \mathcal{E}(x)$
  - for operators $\circledast$, we define $[\![\sigma_1 \circledast \sigma_2]\!]_{\mathcal{E}} := [\![\sigma_1]\!]_{\mathcal{E}} \circledast [\![\sigma_2]\!]_{\mathcal{E}}$
- to this end, we make use of the function tables
- and can thus sometimes evaluate expressions where one argument is $\bot$ to values different to $\bot$
- this way, we can evaluate expressions to known values even though the environment is incomplete
- this can be used to obtain a progress in information

Current Reactions as Fixpoints

- for a program P, we will define a function f_P that maps an environment $\mathcal{E} \in \mathfrak{E}_P$ to another environment $f_P(\mathcal{E}) \in \mathfrak{E}_P$
- f_P will be continuous w.r.t. $\sqsubseteq$,
  and thus, we can compute its least fixpoint
- the current reaction $\mathcal{E}$ of P is the least fixpoint of f_P
- P is causally correct iff all variables have known values in $\mathcal{E}$
- the definition of f_P is however not that easy
- we first have to consider SOS reaction rules and will then define f_P

SOS Reaction Rules

SOS Reaction Rule

$$\langle \mathcal{E}, S \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\text{must}}, \mathcal{D}_{\text{can}}, t_{\text{must}}, t_{\text{can}} \rangle$$

- $\mathcal{E}$: environment of the current macro step
- S: statement to be executed
- $\mathcal{D}_{\text{must}}$: immediate actions that must be executed
- $\mathcal{D}_{\text{can}}$: immediate actions that can be executed
- t_{must}: holds iff S must be instantaneously executed
- t_{can}: holds iff S can be instantaneously executed
- note $\mathcal{D}_{\text{must}} \subseteq \mathcal{D}_{\text{can}}$ and $t_{\text{must}} \rightarrow t_{\text{can}}$

SOS Reaction Rules

- SOS reaction rules are recursively defined for the statements
- analogously to the SOS transition rules, we have assumptions and conclusions
- we now consider the SOS reaction rules for all core statements
- and then continue with the definition of the current reaction as a least fixpoint
- t_{must} and t_{can} encode instantaneous execution as follows:

t_{must}	t_{can}	*instant*
false	false	false
false	true	$\bot$
true	false	$--$
true	true	true

Atomic Statements

$$\langle \mathcal{E}, \texttt{nothing} \rangle \not\hookrightarrow_{\mathbb{Q}} \langle \{\}, \{\}, \mathsf{true}, \mathsf{true} \rangle$$

$$\langle \mathcal{E}, \ell : \texttt{pause} \rangle \not\hookrightarrow_{\mathbb{Q}} \langle \{\}, \{\}, \mathsf{false}, \mathsf{false} \rangle$$

$$\langle \mathcal{E}, \texttt{x=}\tau \rangle \not\hookrightarrow_{\mathbb{Q}} \langle \{(x = \tau)\}, \{(x = \tau)\}, \mathsf{true}, \mathsf{true} \rangle$$

$$\langle \mathcal{E}, \texttt{next(x)=}\tau \rangle \not\hookrightarrow_{\mathbb{Q}} \langle \{\}, \{\}, \mathsf{true}, \mathsf{true} \rangle$$

Conditional

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true} \quad \langle \mathcal{E}, S_1 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, t^1_{\mathsf{must}}, t^1_{\mathsf{can}} \rangle}{\langle \mathcal{E}, \texttt{if}(\sigma)\ \ S_1\ \ \texttt{else}\ \ S_2 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, t^1_{\mathsf{must}}, t^1_{\mathsf{can}} \rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \quad \langle \mathcal{E}, S_2 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}} \rangle}{\langle \mathcal{E}, \texttt{if}(\sigma)\ \ S_1\ \ \texttt{else}\ \ S_2 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}} \rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \bot \quad \langle \mathcal{E}, S_1 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, t^1_{\mathsf{must}}, t^1_{\mathsf{can}} \rangle \quad \langle \mathcal{E}, S_2 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}} \rangle}{\langle \mathcal{E}, \texttt{if}(\sigma)\ \ S_1\ \ \texttt{else}\ \ S_2 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}} \cap \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}} \cup \mathcal{D}^2_{\mathsf{can}}, t^1_{\mathsf{must}} \wedge t^2_{\mathsf{must}}, t^1_{\mathsf{can}} \vee t^2_{\mathsf{can}} \rangle}$$

Sequence

$$\frac{\langle \mathcal{E}, S_1 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{false}, \mathsf{false} \rangle}{\langle \mathcal{E}, \{S_1 ; S_2\} \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{false}, \mathsf{false} \rangle}$$

$$\frac{\langle \mathcal{E}, S_1 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{false}, \mathsf{true} \rangle \quad \langle \mathcal{E}, S_2 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}} \rangle}{\langle \mathcal{E}, \{S_1 ; S_2\} \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}} \cup \mathcal{D}^2_{\mathsf{can}}, \mathsf{false}, t^2_{\mathsf{can}} \rangle}$$

$$\frac{\langle \mathcal{E}, S_1 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, \mathsf{true}, \mathsf{true} \rangle \quad \langle \mathcal{E}, S_2 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}} \rangle}{\langle \mathcal{E}, \{S_1 ; S_2\} \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}} \cup \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}} \cup \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}} \rangle}$$

Parallel Statement

$$\frac{\langle \mathcal{E}, S_1 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}}, t^1_{\mathsf{must}}, t^1_{\mathsf{can}} \rangle \qquad \langle \mathcal{E}, S_2 \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^2_{\mathsf{can}}, t^2_{\mathsf{must}}, t^2_{\mathsf{can}} \rangle}{\langle \mathcal{E}, \{S_1 \parallel S_2\} \rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}^1_{\mathsf{must}} \cup \mathcal{D}^2_{\mathsf{must}}, \mathcal{D}^1_{\mathsf{can}} \cup \mathcal{D}^2_{\mathsf{can}}, t^1_{\mathsf{must}} \wedge t^2_{\mathsf{must}}, t^1_{\mathsf{can}} \wedge t^2_{\mathsf{can}} \rangle}$$

Loops

$$\frac{\langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{do}\ \ S\ \ \texttt{while}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false}}{\langle \mathcal{E}, \texttt{while}(\sigma)\ \ S\rangle \not\mapsto_{\mathbb{Q}} \langle \{\}, \{\}, \mathsf{true}, \mathsf{true}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true} \quad \langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{while}(\sigma)\ \ S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \bot \quad \langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{while}(\sigma)\ \ S\rangle \not\mapsto_{\mathbb{Q}} \langle \{\}, \mathcal{D}_{\mathsf{can}}, \mathsf{false}, \mathsf{true}\rangle}$$

Abort (1/2)

$$\frac{\langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{[weak] abort } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \quad \langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{[weak] immediate abort } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true} \quad \langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{weak immediate abort } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, \mathsf{true}, \mathsf{true}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true}}{\langle \mathcal{E}, \texttt{immediate abort } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \{\}, \{\}, \mathsf{true}, \mathsf{true}\rangle}$$

Abort (2/2)

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \bot \quad \langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{weak immediate abort } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, \mathsf{true}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \bot \quad \langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{immediate abort } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \{\}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, \mathsf{true}\rangle}$$

Suspend (1/2)

$$\frac{\langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{[weak] suspend } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{false} \quad \langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{[weak] immediate suspend } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true} \quad \langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{weak immediate suspend } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, \mathsf{false}, \mathsf{false}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \mathsf{true}}{\langle \mathcal{E}, \texttt{immediate suspend } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \{\}, \{\}, \mathsf{false}, \mathsf{false}\rangle}$$

Suspend (2/2)

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \bot \quad \langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{immediate suspend } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \{\}, \mathcal{D}_{\mathsf{can}}, \mathsf{false}, t_{\mathsf{can}}\rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \bot \quad \langle \mathcal{E}, S\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, t_{\mathsf{must}}, t_{\mathsf{can}}\rangle}{\langle \mathcal{E}, \texttt{weak immediate suspend } S \texttt{ when}(\sigma)\rangle \not\mapsto_{\mathbb{Q}} \langle \mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}}, \mathsf{false}, t_{\mathsf{can}}\rangle}$$

Current Reactions as Fixpoints

- recall that we wanted to define a function f_P for a program P that maps environments to environments so that $\mathcal{E} \sqsubseteq f_P(\mathcal{E})$ holds
- to this end, we will make use of the SOS reaction rules
- we assume that the SOS reaction rules are implemented in a function `ReactSOS` that computes for inputs $(\mathcal{E}, S)$ the pair $(\mathcal{D}_{\mathsf{must}}, \mathcal{D}_{\mathsf{can}})$ (we do not need $t_{\mathsf{must}}, t_{\mathsf{can}}$)
- by $\mathcal{D}_{\mathsf{must}}$ and $\mathcal{D}_{\mathsf{can}}$, we then determine a new environment $\mathcal{E}'$

Completing Environments – Function f_P

- by $\mathcal{D}_{\text{must}}$ and $\mathcal{D}_{\text{can}}$, we then determine a new environment $\mathcal{E}'$ as follows:
- define $\mathcal{D}^{\mathtt{x}}_{\text{must}}$ as the assignments $\mathtt{x}{=}\tau$ of $\mathcal{D}_{\text{must}}$ writing to $\mathtt{x}$
  - let $\mathcal{D}^{\mathtt{x}}_{\text{must}} = \{\mathtt{x}{=}\tau_1, \ldots, \mathtt{x}{=}\tau_n\} \neq \{\}$
  - update $\mathcal{E}$ such that $\mathcal{E}(x) := \sup\{[\![x]\!]_{\mathcal{E}}, [\![\tau_1]\!]_{\mathcal{E}}, \ldots, [\![\tau_n]\!]_{\mathcal{E}}\}$
  - note that writing $\bot$ and a known value v yields $\mathcal{E}(x) = v$
  - note that writing value $v_1 \neq v_2$ with $v_i \neq \bot$ yields $\mathcal{E}(x) = \top$
- define $\mathcal{D}^{\mathtt{x}}_{\text{can}}$ as the assignments $\mathtt{x}{=}\tau$ of $\mathcal{D}_{\text{can}}$ writing to $\mathtt{x}$
  - assume $\mathcal{D}^{\mathtt{x}}_{\text{can}} = \{\} \Longrightarrow$ no immediate assignment can write to $\mathtt{x}$
  - if $[\![x]\!]_{\mathcal{E}} \neq \bot$, x already has a value due to delayed assignments of the previous step
  - thus, we apply the reaction to absence if $\mathcal{D}^{\mathtt{x}}_{\text{can}} = \{\} \wedge [\![x]\!]_{\mathcal{E}} = \bot$:
    - if $\mathtt{x}$ is an event variable, set $\mathcal{E}(x)$ to its default value
    - if $\mathtt{x}$ is a memorized variable, set $\mathcal{E}(x)$ to the previous value (or the default value when the initial reaction is computed)

Current Reaction as Least Fixpoint

- the updates described on the previous slide define a function UpdateEnv (which is the previously mentioned function f_P)
- recall our partial orders
  - $x \preceq y :\Leftrightarrow (x = \bot) \vee (x = y) \vee (y = \top)$
  - $\mathcal{E}_1 \sqsubseteq \mathcal{E}_2 :\Leftrightarrow \forall x \in \mathcal{V}.\ \mathcal{E}_1(x) \preceq \mathcal{E}_2(x)$
- it is easily seen that UpdateEnv is monotonous
- since P has only finitely many variables, we have a finite lattice
- and therefore UpdateEnv is continuous, and thus, we can compute its least fixpoint by the Tarski-Knaster iteration
- current reaction is the least fixpoint of UpdateEnv

Compute Current Reaction

function ComputeReaction($\mathcal{E}, S, \mathcal{E}_{\text{pre}}$)
  do
    $\mathcal{E}_{\text{old}} := \mathcal{E}$;
    $(\mathcal{D}_{\text{must}}, \mathcal{D}_{\text{can}}) :=$ ReactSOS($\mathcal{E}, S$);
    $\mathcal{E} :=$ UpdateEnv($\mathcal{D}_{\text{must}}, \mathcal{D}_{\text{can}}, \mathcal{E}_{\text{old}}, \mathcal{E}_{\text{pre}}$)
  while $\mathcal{E}_{\text{old}} \neq \mathcal{E}$
  return $\mathcal{E}$

- starting with an incomplete environment $\mathcal{E}$ and the previous environment $\mathcal{E}_{\text{pre}}$, we compute $\mathcal{D}_{\text{must}}$ and $\mathcal{D}_{\text{can}}$ by the SOS reaction rules
- then, we update the environment $\mathcal{E}$ by $\mathcal{D}_{\text{must}}$ and $\mathcal{D}_{\text{can}}$
- and repeat this until a fixpoint is obtained

Quartz Interpreter

function *InterpretQuartz*(S)
  $\mathcal{E}_{\text{pre}} := \mathcal{E}_{\text{def}}$; // default values for all variables
  $\mathcal{D}_{\text{del}} := \{\}$; // no delayed actions at the beginning
  do
    $\mathcal{E}_{\text{in}} :=$ ReadInputs();
    $\mathcal{E}_{\text{init}} :=$ UpdateDelayedActs$(\mathcal{D}_{\text{del}}, \mathcal{E}_{\text{pre}})$;
    $\mathcal{E} :=$ ComputeReaction$(\mathcal{E}_{\text{init}}, S, \mathcal{E}_{\text{pre}})$;
    if $\exists x \in \mathcal{V}.\, \mathcal{E}(x) \in \{\bot, \top\}$ then fail;
    $(S', \mathcal{D}, t) :=$ TransSOS$(S, \mathcal{E})$;
    $\mathcal{D}_{\text{del}} :=$ delayed actions of $\mathcal{D}$;
    $S := S'$;
    $\mathcal{E}_{\text{pre}} := \mathcal{E}$;
  while$(\neg t)$;

Constructive Programs

- the interpreter on the previous slide computes the reactions and control flow updates for any constructive program
  - after reading new inputs, we have $\mathcal{E}_{\text{in}}$
  - we then evaluate the delayed assignments $\mathcal{D}_{\text{del}}$ obtained by SOS transition rules of the previous reaction in the previous environment $\mathcal{E}_{\text{pre}}$ and update $\mathcal{E}_{\text{in}}$ thereby to $\mathcal{E}_{\text{init}}$
  - using $\mathcal{E}_{\text{init}}$, we compute $\mathcal{E}$ by ComputeReaction (i.e. SOS reaction rules)
  - using $\mathcal{E}$, we compute $(S', \mathcal{D}, t)$ by the SOS transition rules
  - we repeat this until the program terminates
- programs where the least fixpoint contains either $\bot$ or $\top$ are not causally correct
- therefore the interpreter fails in these cases

Relationship between 2-Value and 4-Valued Solutions

- consider any function $f(x_0, \ldots, x_{n-1}) = (\varphi_0, \ldots, \varphi_{n-1})$ with propositional logic formulas φ on the variables $x_0, \ldots, x_{n-1}$
- let $\check{x} = (\check{x}_0, \ldots, \check{x}_{n-1})$ be the least fixpoint, i.e., $\check{x} = f(\check{x})$
- Theorem: If $\check{x}_i \in \mathbb{B}$, then we have for all $y \in \mathbb{B}^n$ with $y = f(y)$ that $y_i = \check{x}_i$. The converse is however false.
- proof:
  - consider any boolean solution $y \in \mathbb{B}^n$ with $y = f(y)$
  - $\check{x} \preceq y$ since y is a fixpoint and $\check{x}$ is the least fixpoint
  - hence, $\check{x}_i \preceq y_i$ holds for $i = 0, \ldots, n-1$
  - if both $\check{x}_i \in \mathbb{B}$ and $y_i \in \mathbb{B}$ holds, this implies $\check{x}_i = y_i$
- Corollary: If $\check{x}$ is the least fixpoint of f, and $\check{x} \in \mathbb{B}^n$ holds, then this is the unique boolean solution of $x = f(x)$.

Relationship between 2-Value and 4-Valued Solutions

- we have proved that boolean values in the least fixpoint enforce that value also in all boolean solutions
- the converse does not hold: there are functions f where all solutions of $x = f(x)$ share the same boolean value x_i, but still the least fixpoint is not that value
- consider the example (left hand side) and its boolean solutions (right hand side)

$$\begin{cases} x_0 = \neg x_0 \wedge \neg x_1 \\ x_1 = x_2 \vee x_3 \\ x_2 = x_2 \\ x_3 = x_3 \end{cases}$$

x_0	x_1	x_2	x_3
0	1	0	1
0	1	1	0
0	1	1	1

- the least fixpoint is $\check{x} = (\bot, \bot, \bot, \bot)$
- but all boolean solutions (as shown above) agree on $x_0 = 0$

Relationship between 2-Value and 4-Valued Solutions

- so, we know that boolean values in $\check{x}$ are unique for all boolean solutions
- Can we say anything about boolean solutions in case $\check{x}_i = \bot$?
- the following examples show that we have no further information, in particular:
  - there may be no boolean solution
  - there may be a unique boolean solution
  - there may be many boolean solutions

Module P01

Example

```
module P01(event ?i,o1,o2,o3){
  if(i)   emit(o1);
  if(!o1) emit(o2);
  if(o2)  emit(o3);
}
```

if i is 0, then

	i	o1	o2	o3	$\mathcal{D}_{\text{must}}$	$\mathcal{D}_{\text{can}}$
$\mathcal{E}_0$	0	$\bot$	$\bot$	$\bot$	$\{\}$	$\{\texttt{emit(o2)}, \texttt{emit(o3)}\}$
$\mathcal{E}_1$	0	0	$\bot$	$\bot$	$\{\texttt{emit(o2)}\}$	$\{\texttt{emit(o2)}, \texttt{emit(o3)}\}$
$\mathcal{E}_2$	0	0	1	$\bot$	$\{\texttt{emit(o2)}, \texttt{emit(o3)}\}$	$\{\texttt{emit(o2)}, \texttt{emit(o3)}\}$
$\mathcal{E}_3$	0	0	1	1	$\{\texttt{emit(o2)}, \texttt{emit(o3)}\}$	$\{\texttt{emit(o2)}, \texttt{emit(o3)}\}$

with transition $\langle \mathcal{E}_3, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \{\texttt{emit(o2)}, \texttt{emit(o3)}\}, \text{true} \rangle$

Module P01

Example

```
module P01(event ?i,o1,o2,o3){
  if(i)   emit(o1);
  if(!o1) emit(o2);
  if(o2)  emit(o3);
}
```

if i is 1, then

	i	o1	o2	o3	$\mathcal{D}_{\text{must}}$	$\mathcal{D}_{\text{can}}$
$\mathcal{E}_0$	1	$\bot$	$\bot$	$\bot$	{emit(o1)}	{emit(o1), emit(o2), emit(o3)}
$\mathcal{E}_1$	1	1	$\bot$	$\bot$	{emit(o1)}	{emit(o1), emit(o3)}
$\mathcal{E}_2$	1	1	0	$\bot$	{emit(o1)}	{emit(o1)}
$\mathcal{E}_3$	1	1	0	0	{emit(o1)}	{emit(o1)}

with transition $\langle \mathcal{E}_3, S \rangle \twoheadrightarrow_{\mathbb{Q}} \langle \texttt{nothing}, \{\texttt{emit(o1)}\}, \text{true} \rangle$

Module P02

Example

```
module P02(event →
    o1,o2) {
    emit(o2);
    if(!o1) {
        if(o2) w: pause;
        emit(o1);
    }
}
```

initial reaction:

	o1	o2	$\mathcal{D}_{\text{must}}$	$\mathcal{D}_{\text{can}}$
$\mathcal{E}_0$	$\bot$	$\bot$	{emit(o2)}	{emit(o1), emit(o2)}
$\mathcal{E}_1$	$\bot$	1	{emit(o2)}	{emit(o2)}
$\mathcal{E}_2$	0	1	{emit(o2)}	{emit(o2)}

Module P02

Example

```
module P02(event →
    o1,o2) {
    emit(o2);
    if(!o1) {
        if(o2) w: pause;
        emit(o1);
    }
}
```

second reaction (executing $S' = \texttt{emit(o1)}$):

	o1	o2	$\mathcal{D}_{\text{must}}$	$\mathcal{D}_{\text{can}}$
$\mathcal{E}_0$	$\bot$	$\bot$	{emit(o1)}	{emit(o1)}
$\mathcal{E}_1$	1	0	{emit(o1)}	{emit(o1)}

Modules P03 and P04

Example

```
module P03(event o) {
    if(!o) emit(o);
}

module P04(event o) {
    if(o) emit(o);
}
```

- there is no way to run these programs!
- P03 does not even have satisfying behaviors
- P04 could be satisfied by both `o=true` and `o=false`
- none of the programs is causally correct

	o	$\mathcal{D}_{\text{must}}$	$\mathcal{D}_{\text{can}}$
$\mathcal{E}_0$	$\bot$	{}	{emit(o)}
$\mathcal{E}_1$	$\bot$	{}	{emit(o)}

Modules P05 and P06

Example

```
module P05(event →
    o1,o2) {
    if(o1)  emit(o1);
    if(!o2) emit(o2);
}

module P06(event →
    o1,o2) {
    if(o1) emit(o2);
    if(o2) emit(o1);
}
```

- there is no way to run these programs!
- P05 does not even have satisfying behaviors
- P06 could be satisfied by both `o1=o2=true` and `o1=o2=false`
- none of the programs is causally correct

	o1	o2	$\mathcal{D}_{\text{must}}$	$\mathcal{D}_{\text{can}}$

Modules P07a and P07b

Example

```
module P07a(event o) {
    if(o) w: pause;
    emit(o);
}

module P07b(event o) {
    if(!o) w: pause;
    emit(o);
}
```

- P07a has no behavior
- assume o would be `false`, then
  - the control would not stop at label `w`
  - and thus would execute `emit(o)`
  - but then, o would be `true`
- assume o would be `true`, then
  - the control would stop at label `w`
  - and thus would not execute `emit(o)`
  - but then, o would be `false`

$\Longmapsto$ there is no behavior (same with P07b)

	o	$\mathcal{D}_{\text{must}}$	$\mathcal{D}_{\text{can}}$
$\mathcal{E}_0$	$\bot$	$\{\}$	$\{\texttt{emit(o)}\}$
$\mathcal{E}_1$	$\bot$	$\{\}$	$\{\texttt{emit(o)}\}$

Module P08

Example

```
module P08(event →
    ?i,o1,o2) {
    weak immediate abort {
            {
            if(!i) w: →
                pause;
            emit(o1);
            }
        ||
            if(o1) →
                emit(o2);
    } when(o2);
    emit(o1);
}
```

- assume `i` would be `false`, then
  - the control would stop at label `w`
  - and thus would not execute the first `emit(o1)`
  - at this stage, we have to speculate about the value of `o1`
  - assume `o1` would be `false`, then
    - we would not execute `emit(o2)`
    - no abortion takes place, and the control will remain at label `w`
  - assume `o1` would be `true`, then
    - we would execute `emit(o2)`
    - the abortion takes place, and the control will leave label `w`
    - thus, the second `emit(o1)` is

Module P08

Example

```
module P08(event →
    ?i,o1,o2) {
    weak immediate abort {
            {
            if(!i) w: →
                pause;
            emit(o1);
            }
        ||
            if(o1) →
                emit(o2);
    } when(o2);
    emit(o1);
}
```

- assume `i` would be `true`, then
  - the control would not stop at label `w` and instead executes `emit(o1)`
  - thus, also `emit(o2)` is executed
  - the weak abortion takes place, but has no effect
  - we then execute the second `emit(o1)`
  - thus, `o1=o2=true` and nothing is left for further macro steps

Module P09

Example

```
module P09(event →
    o1,o2) {
      if(o1) emit(o1);
    ||
      if(o1)
          if(!o2) →
              emit(o2);
}
```

- P09 is not constructive
- note that `if(!o2) emit(o2);` is a contradiction
- thus, `o1=true` would lead to a contradiction
- thus logically only `o1=o2=false` makes sense
- however, this is not constructively found

Module P10

Example

```
module P10(event o) {
    if(o) nothing;
    emit(o);
}
```

- in Quartz, P10 is constructive, while in Esterel, it is not
-

Module P11

Example

```
module P11(event →
    o1,o2) {
    if(o1) {
        emit(o2);
        if(o2) w: pause;
        emit(o1);
    }
}
```

- use the simulator!

Module P12

Example

```
module P12(event o) {
    if(o) emit(o);
    else emit(o);
}
```

- use the simulator!

Module P13

Example

```
module P13(event →
    ?i,o1,o2) {
    if(i) {
        if(o1) emit(o2);
    } else {
        if(o2) emit(o1);
    }
}
```

- use the simulator!

Module P14

Example

```
module P14(event →
    o1,o2) {
    if(o1) emit(o2);
    w: pause;
    if(!o2) emit(o1);
}
```

- use the simulator!

Module P15

Example

```
module P15(event →
    o1,o2) {
    emit(o2);
    if(o1)
        if(!o2) emit(o1);
}
```

- use the simulator!

Module P16

Example

```
module P16(event o) {
    if(o)
        if(!o) emit(o);
}
```

- use the simulator!

Module P17

Example

```
module P17(event →
    o1,o2) {
    if(o1) {
        emit(o2);
        if(!o2) emit(o1);
    }
}
```

- use the simulator!

Module P18

Example

```
module P18(event →
    o1,o2) {
    if(o1) {
        emit(o2);
        ||
        if(!o2) emit(o1);
    }
}
```

- use the simulator!

Module P19

Example

```
module P19(event →
    o1,o2) {
    if(o1) {
        emit(o2);
        ||
        if(o2) emit(o1);
    }
}
```

- use the simulator!

Module P20

Example

```
module P20(event →
    o1,o2,o3,o4) {
      if(o2) emit(o1);
    ||
      if(o1 & o3) →
          emit(o2);
    ||
      if(!o1 & o4) →
          emit(o2);
    ||
      emit(o3);
    ||
      emit(o4);
}
```

- use the simulator!

Module P21

Example

```
module P21(event →
    ?i1,?i2,o1,o2) {
    {
        if(o1 & i1) →
            emit(o2);
    ||
        if(o2 & i2) →
            emit(o1);
    }
}
```

- use the simulator!

References and Further Reading I

[1] A. Benveniste, P. Caspi, S. Edwards, et al. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[2] G. Berry. The Esterel v5 language primer. http://www.inria.fr/meije/esterel/, 1997.

[3] G. Berry. A quick guide to Esterel, 1997.

[4] G. Berry. The constructive semantics of pure Esterel, 1999.

[5] G. Berry. The Esterel v5 language primer, 2000.

[6] G. Berry. The Esterel v5.91 system manual, 2000.

[7] G. Berry, L. Cosserat. The Esterel synchronous programming language and its mathematical semantics. *Seminar on Concurrency (CONCUR)*, vol. 197 of *LNCS*, 389–448, 1985.

[8] F. Boussinot. SugarCubes implementation of causality. Research Report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), 1998.

[9] F. Boussinot, R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[10] J. Brandt, K. Schneider. Static data-flow analysis of synchronous programs. *Formal Methods and Models for Codesign (MEMOCODE)*, 161–170, 2009.

[11] A. Gamatie. *Designing Embedded Systems with the SIGNAL Programming Language*, 2010.

[12] N. Halbwachs. *Synchronous programming of reactive systems*, 1993.

[13] D. Huffman. Combinational circuits with feedback. *Recent Developments in Switching Theory*, 27–55, 1971.

[14] G. Plotkin. A structural approach to operational semantics. Tech. Rep. FN-19, DAIMI, 1981.

[15] D. Potop-Butucaru, S. Edwards, G. Berry. *Compiling Esterel*, 2007.

[16] M. Riedel. *Cyclic Combinational Circuits*. Ph.D. thesis, California Institute of Technology, 2004.

[17] M. Riedel, J. Bruck. Cyclic combinational circuits: Analysis for synthesis. *International Workshop on Logic and Synthesis (IWLS)*, 2003.

[18] M. Riedel, J. Bruck. The synthesis of cyclic combinational circuits. *Design Automation Conference (DAC)*, 163–168, 2003.

[19] R. Rivest. The necessity of feedback in minimal monotone combinational circuits. *IEEE Transactions on Computers (T-C)*, C-26(6):606–607, 1977.

[20] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, 2009.

[21] K. Schneider, J. Brandt, T. Schüle. Causality analysis of synchronous programs with delayed actions. *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 179–189, 2004.

References and Further Reading II

[22] K. Schneider, J. Brandt, T. Schüle. A verified compiler for synchronous programs with local declarations (proceedings version). *Synchronous Languages, Applications, and Programming (SLAP)*, 2004.

[23] K. Schneider, J. Brandt, T. Schüle, et al. Improving constructiveness in code generators. *Synchronous Languages, Applications, and Programming (SLAP)*, 1–19, 2005.

[24] K. Schneider, J. Brandt, T. Schüle, et al. Maximal causality analysis. *Application of Concurrency to System Design (ACSD)*, 106–115, 2005.

[25] T. Shiple, G. Berry, H. Touati. Constructive analysis of cyclic circuits. *European Design Automation Conference (EDAC)*, 328–333, 1996.