Synchronous data-flow programming

Benoit Caillaud Julien Forget
benoit.caillaud@inria.fr
julien.forget@univ-lille.fr

École Temps Réel 2026

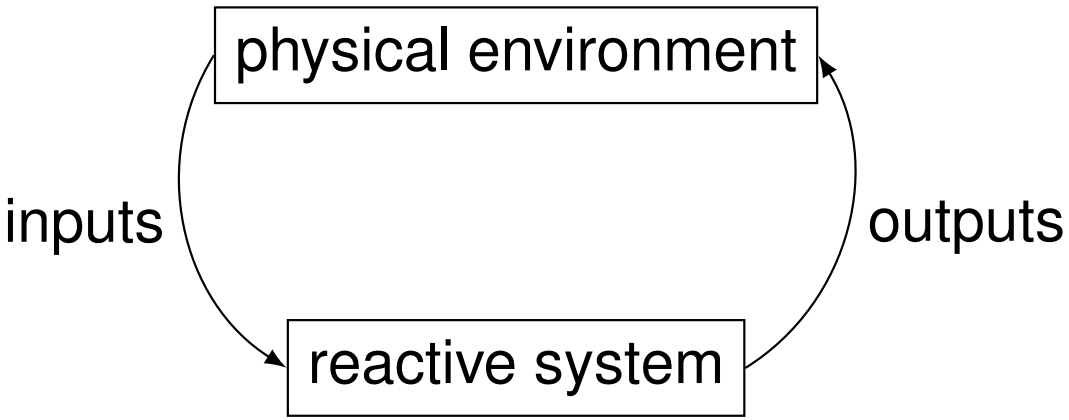
Introduction

Objectives for this session:

- Understand the specifics of synchronous data-flow programming
- Learn to write simple synchronous data-flow programs

Outline

Reactive system



- React to inputs:
  1. Acquire inputs on sensors
  2. Compute
  3. Produce outputs on actuators
- Outputs impact the environment, thus subsequent inputs
- Response time from input to output must be **bounded**

A reactive system: aircraft

Aircraft control-command system

Mission	Stabilize an aircraft
Inputs	Attitude of the aircraft + pilot orders
Outputs	Order for control surfaces + alarms
Reactivity	Surfaces must be controlled every 1ms

Programming

Functional correction

Compute the correct output values

Temporal correction

Compute faster than the reactivity constraint

Programming: the functional part

Constraints:

- The sequence of outputs S_i depends *only* on the sequence of inputs E_i
- Must compute with *bounded memory* M_i

How to program:

1. Identify: inputs and outputs
2. Define:
   - The output function $S_i = f(E_i, M_i)$
   - The transition function $M_{i+1} = g(E_i + M_i)$

Programming: the temporal part

Difficulties:

- Input datation:
  - Inputs may not be acquired simultaneous on sensors
  - Inputs may arrive non-simultaneously at the processor
- Non simultaneous inputs:
  - When do we start computing?
  - How to interpret a delayed input: absent? delayed?
- Difficult to predict when computations complete

$\Rightarrow$ How to program with such uncertainties?

The Asynchronous approach

Asynchronous programming

Execute tasks independently, enabling concurrence for improved performance.

Difficulties:

- Scheduling: inaccurate WCET, jitter, ...
- Inter-task communications: synchronizations

$\Rightarrow$ Possibly variable task inter-leaving

How to ensure deterministic execution?

The Synchronous approach

Real-time is replaced by a simplified abstract logical time

- **Instant**: one reaction of the system
- **Logical time**: sequence of instants
- The program describes what happens during each instant
- **Synchronous hypothesis**: computations complete before the next instant. If so:
  - $\Rightarrow$ We can ignore time inside an instant, only order matters
  - $\Rightarrow$ We only care about how instants are chained together

Synchronous flavours

- *Control-flow synchronous languages*:
  - ESTEREL
  - Computation is driven by the **occurrence of events**
  - The program describes simultaneity and precedence between events
- *Data-flow synchronous languages*:
  - LUSTRE/SCADE, HEPTAGON, SIGNAL, ...
  - Computation is driven by the **arrival of data**
  - The program describes causality relations

Outline

Flows and clocks

Example

x	3	4	5	2	6	...
y	True			False	True	...

- HEPTAGON is a data-flow language, inspired by LUSTRE
- Computations are triggered by incoming data
- Every expression or variable is a flow
- **Flow**: infinite sequence of values + clock
- **Clock**: defines when a flow is present (has a value)

Data-flow

Example

x	x_1	x_2	x_3	x_4	...
y	y_1	y_2	y_3	y_4	...
x+y	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	...
f(x)	$f(x_1)$	$f(x_2)$	$f(x_3)$	$f(x_4)$	...

- Arithmetic operators, functions, etc, are applied whenever they receive a new input

$\Rightarrow$ That's what **data-flow** stands for

Sampling

Example

c	True	False	False	True	...
x	x_1	x_2	x_3	x_4	...
y=x **when** c	x_1			x_4	...

- The **when** operator under-samples a flow
- x **when** c is present only when c is true, in which case its value is x
- The clock of x **when** c is a sub-clock of the clock of x

Merge

Example

c	True	False	False	True	...
xc=M(x **when** c)	m_1			m_2	...
xnc=N(x **when** not c)		n_1	n_2		...
merge c xc xnc	m_1	n_1	n_2	m_2	...

- Combine flows with complementary clocks
- Produce a more frequent flow

Synchronous

- Under-sampling (**when**) introduces undefined values. The flow is said to be **absent**
- Only flows with the same clock can be combined:

  Ex: x+x **when** c is forbidden!

  ⇒ That's the **synchronous** part
- The compiler verifies synchronization constraints during the clock calculus

Delay operator

Example

x	x_1	x_2	x_3	x_4	...
pre x		x_1	x_2	x_3	...
0–>**pre** x	0	x_1	x_2	x_3	...

- The **pre** operator denotes the previous value of a flow
- Initially undefined
- Usually combined with the initialisation operator ->.
  x–>y is x on the first instant, y otherwise
- That's how we introduce **memory** of previous instants

Structuring: nodes

- A program consists of a set of **nodes**
- The main node is specified on the compilation line
- **Equations** define output flows from input flows
- Equations are unordered
- Nodes can be instantiated in flow expressions

Example: a resettable counter

Example

```
node counter(rst:bool) returns (count:int);
let
  count=0->if rst then 0 else pre(count)+1;
tel
```

reset	False	False	False	True	False	...
count						...

Compiling

1. Write the synchronous program
2. Compile $\Rightarrow$ generates C code
3. Write the **integration program**:
   1. Read inputs on sensors
   2. Call the synchronous program
   3. Apply outputs to actuators
4. Compile synchronous code + integration code $\Rightarrow$ executable

And...

And...

That's all folks !

And...

That's all folks !

But the programming style requires some getting used to.

Outline

Causality

Do not write

```
node error(i:bool) returns (o:int)
var x,y: int;
let
   x=y+1; y=x+2; o=y+i
tel
```

No immediate loop!

- Verified by a **causality analysis**
- A **pre** is missing somewhere

Initialisation

Code

```
node f(x: int) returns (o: int)
let
  o=pre(x)+1;
tel
```

Semantics

i	0	0	1	1	...
o					...

Double initialisation

Semantics

osc	true	true	false	true	false	true	...

Code

if then else

Code

```
node count_decount(rst, c:bool) returns (count:int)
let
  count=if c then counter(rst)
        else decounter(rst);
tel
```

Semantics

c	True	True	True	False	True	...
count						...

NB: counter was defined previously, decounter is a decreasing counter, i.e. decounter(false): 0,-1,-2,-3,...

when and **merge**

Code

```
node count_alt(rst, c:bool) returns (count:int)
  var c1: int when c;
      c2: int when not c;
let
  c1=counter(rst when c);
  c2=decounter(rst when not c);
  count=merge c c1 c2;
tel
```

Semantics

c	True	True	True	False	True	...
count						...

pre and **when**

Code

```
node p_when(i:int; c:bool) returns
    (o1: int when c; o2:int when c);
  let
    o1=0->pre(i when c);
    o2=0->pre(i) when c;
  tel
```

Semantics

c	T	T	F	T	F	F	T	...
i	1	2	3	4	5	6	7	...
o1								...
o2								...

Outline

Synchronous languages vs others

Advantages:

- High abstraction level $\Rightarrow$ less work for the programmer
- Bounded memory and execution time
- Barely needs an OS
- Formal semantics $\Rightarrow$ formal proofs and provable compilation

Disadvantages:

- Executable less efficient than hand-written C code?
- Synchronous hypothesis can be hard to *ensure*
- No direct support for multiple real-time constraints (periods, deadlines)

My sources

Some inspirations for this course:

- *Abdoulaye GAMATIE (LIRMM)*, Synchronous Programming of Real-Time Systems with the Signal language, course at Telecom Lille
- *Pascal RAYMOND (Verimag)*, various teaching notes