Code generation from synchronous languages

a short survey

Nicolas Halbwachs

Verimag/CNRS
Grenoble, France

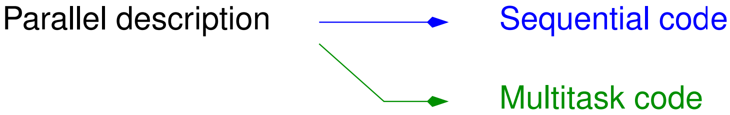
Synthesis or compilation?

Parallel description

Synthesis or compilation?

Parallel description → Sequential code

Synthesis or compilation?

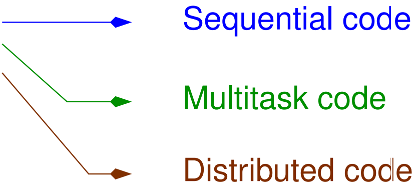


Parallel description

Sequential code

Multitask code

Synthesis or compilation?



Parallel description

Sequential code

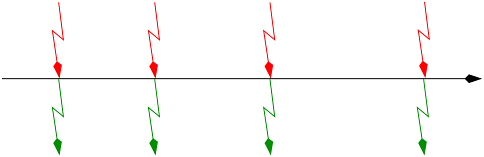
Multitask code

Distributed code

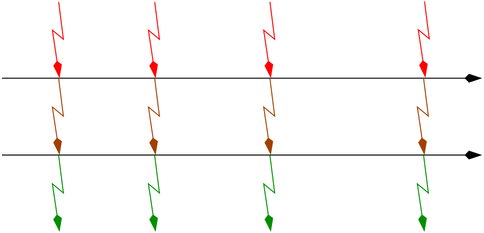
Synchronous programming

Abstract synchronous behavior

sequence of reactions to input events:

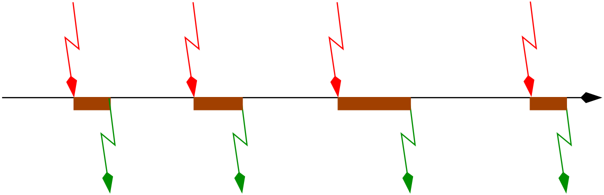


Composition of behaviors:



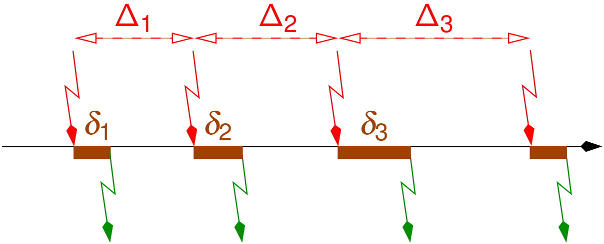
Synchronous programming

Concrete behavior



Synchronous programming

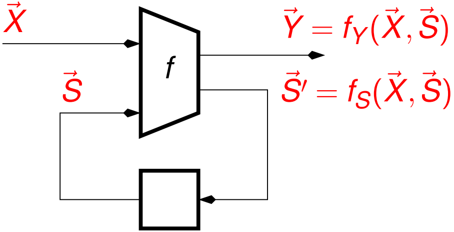
Concrete behavior



Valid abstraction as long as $\delta_i < \Delta_i$

Synchronous data-flow

Generalized Mealy machines



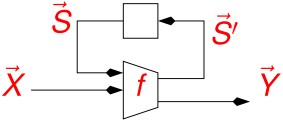
Behaviour: $(\vec{S}_0, \vec{X}_0, \vec{Y}_0), (\vec{S}_1, \vec{X}_1, \vec{Y}_1), \ldots, (\vec{S}_n, \vec{X}_n, \vec{Y}_n), \ldots,$

with $Y_n = f_Y(\vec{X}_n, \vec{S}_n)$ and $\vec{S}_{n+1} = f_S(\vec{X}_n, \vec{S}_n)$

Deterministic!

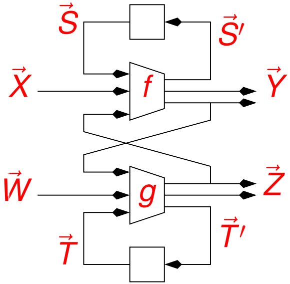
Synchronous data-flow

Parallel composition:



Synchronous data-flow

Parallel composition:

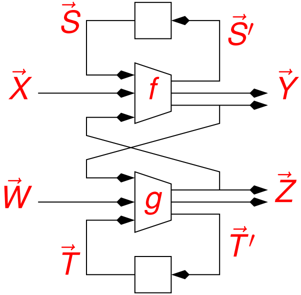


$$(\vec{S}', \vec{Y}) = f(\vec{X}, \vec{S}, \vec{Z})$$
$$(\vec{T}', \vec{Z}) = g(\vec{W}, \vec{T}, \vec{Y})$$

(deterministic, provided there is no combinational loop)

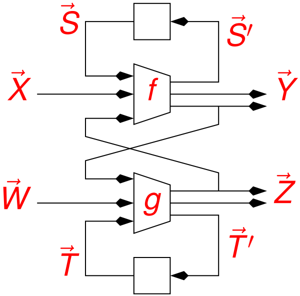
Sequential code generation

The absence of combinational loop ensures that there exists a sequential order (topological order)



Sequential code generation

The absence of combinational loop ensures that there exists a sequential order (topological order)



```
$S = S_0\,;\; T = T_0\,;$
forever do
    read (X, W);
    $Z = g_Z(W, T)\,;$
    $Y = f_Y(X, Z, S)\,;$
    $T = g_T(W, Y, T)\,;$
    $S = f_S(X, Z, S)\,;$
done
```

Standard way of compiling Lustre and Scade

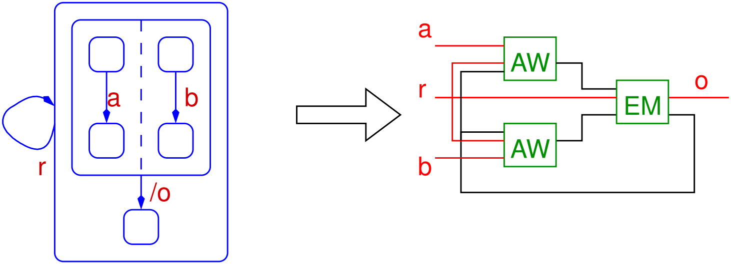
Sequential code generation

What about imperative languages (Esterel, Synccharts)?

Sequential code generation

What about imperative languages (Esterel, Synccharts)?

By translation into data-flow [Berry 1992, Esterel on Hardware]

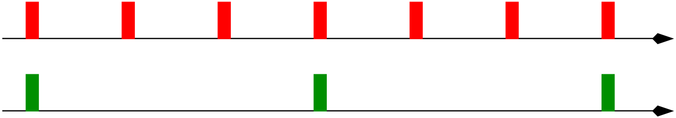


Deterministic multi-tasking (1/3)

[Scaife& Caspi, ECRTS 2004]

Periodic tasks, with different periods

Purely synchronous behavior:

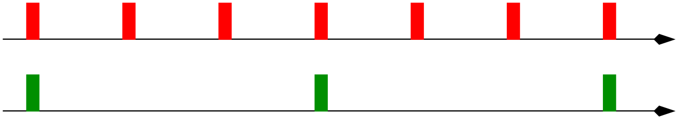


Deterministic multi-tasking (1/3)

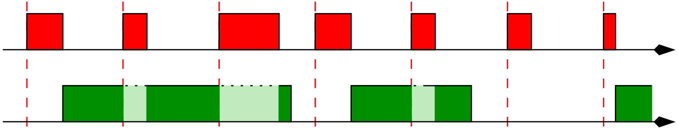
[Scaife& Caspi, ECRTS 2004]

Periodic tasks, with different periods

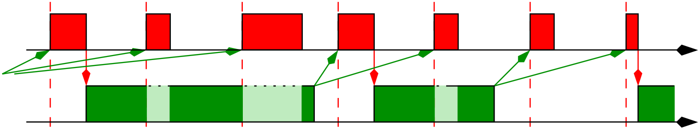
Purely synchronous behavior:



The desired behavior:

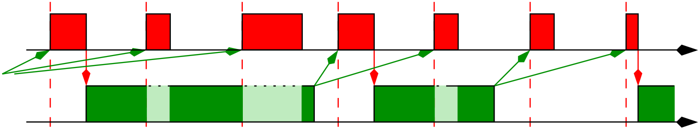


Deterministic multi-tasking (2/3)



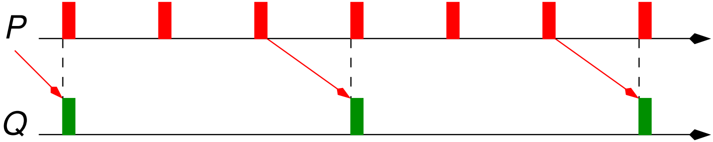
Communication can be non deterministic!

Deterministic multi-tasking (2/3)

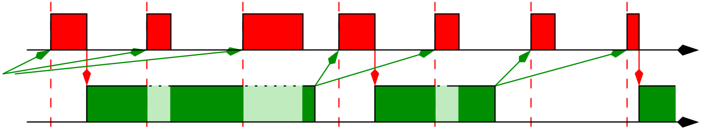


Communication can be non deterministic!

Determinism can be preserved if the communication is delayed:

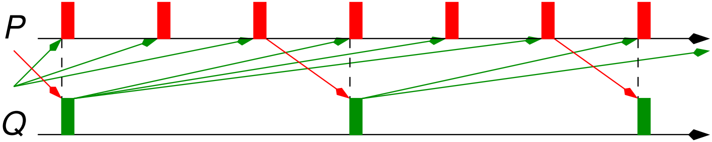


Deterministic multi-tasking (2/3)

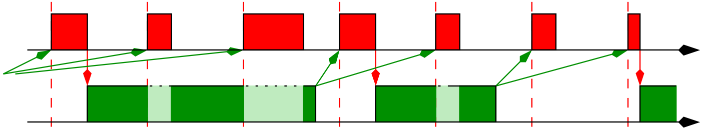


Communication can be non deterministic!

Determinism can be preserved if the communication is delayed:

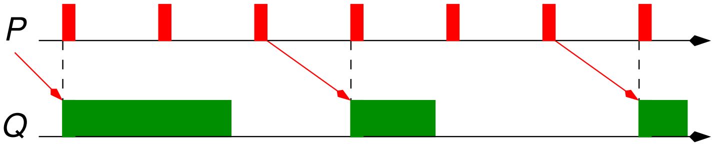


Deterministic multi-tasking (2/3)

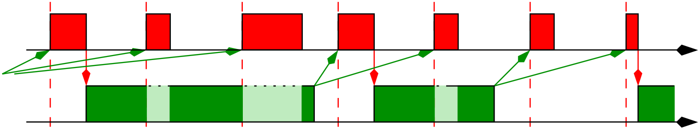


Communication can be non deterministic!

Determinism can be preserved if the communication is delayed:

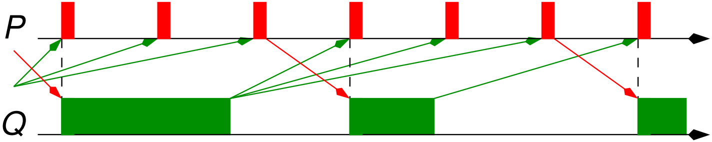


Deterministic multi-tasking (2/3)



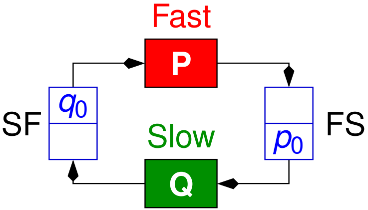
Communication can be non deterministic!

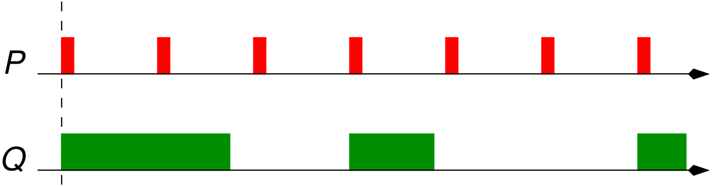
Determinism can be preserved if the communication is delayed:



Deterministic multi-tasking (3/3)

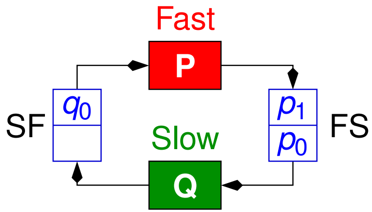
Generating correct code: One fast, high priority task, one slow, low priority task, communicating through double-buffers

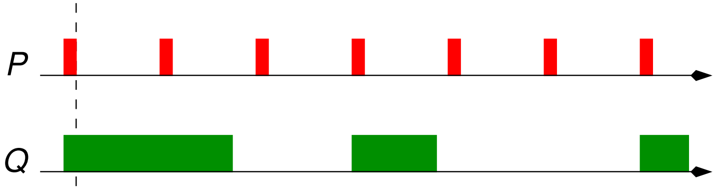




Deterministic multi-tasking (3/3)

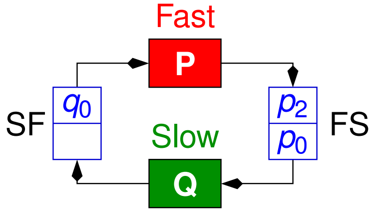
Generating correct code: One fast, high priority task, one slow, low priority task, communicating through double-buffers

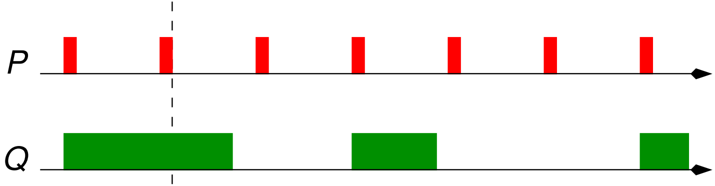




Deterministic multi-tasking (3/3)

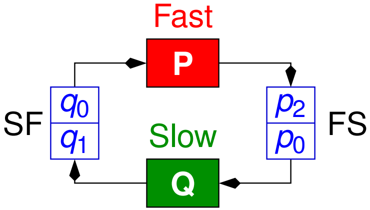
Generating correct code: One fast, high priority task, one slow, low priority task, communicating through double-buffers

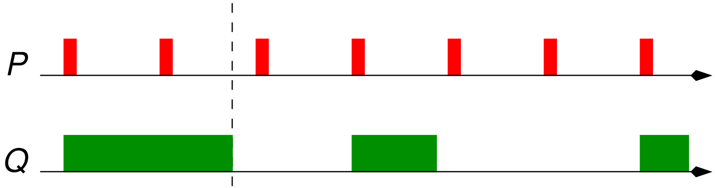




Deterministic multi-tasking (3/3)

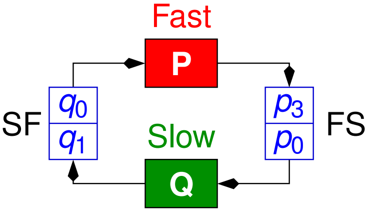
Generating correct code: One fast, high priority task, one slow, low priority task, communicating through double-buffers

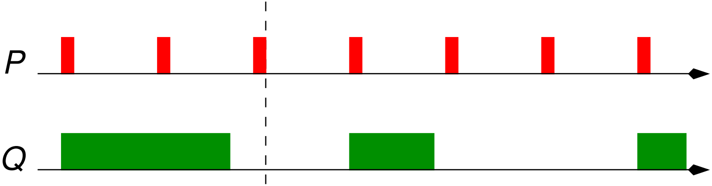




Deterministic multi-tasking (3/3)

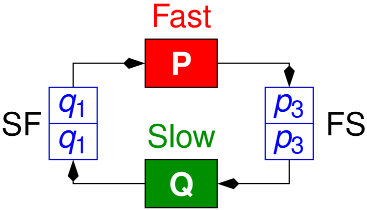
Generating correct code: One fast, high priority task, one slow, low priority task, communicating through double-buffers

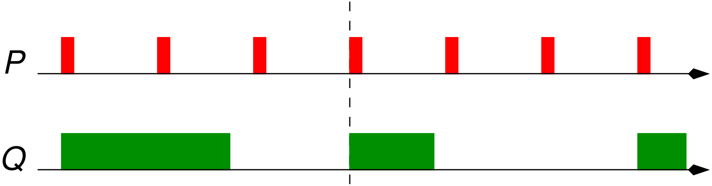




Deterministic multi-tasking (3/3)

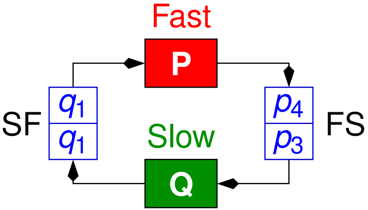
Generating correct code: One fast, high priority task, one slow, low priority task, communicating through double-buffers

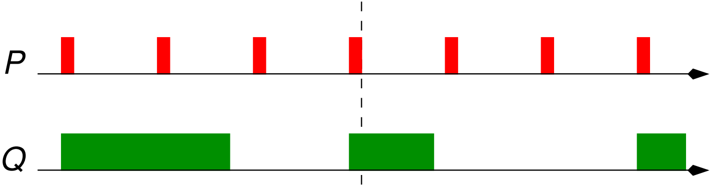




Deterministic multi-tasking (3/3)

Generating correct code: One fast, high priority task, one slow, low priority task, communicating through double-buffers





Generation of distributed code

- for fault-tolerance (redundancy)
- to improve the performances (??)
- because of physical constraints
  (position of sensors and actuators)

Generation of distributed code

- for fault-tolerance (redundancy)
- to improve the performances (??)
- because of physical constraints
  (position of sensors and actuators)

Solutions preserving functional semantics

- Implementation of Lustre on top of TTA
  [Caspi-Curic-Maignan-Sofronis-Tripakis-Niebert, LCTES03]
- Distributing sequential code
  [Caspi-Girault-Pilaud, TSE 1999]
- Deterministic desynchronization using “endochrony”
  [Benveniste-Caillaud-LeGuernic, CONCUR 1999]
  [Potop-DeSimone-Sorel, EMSOFT 2007]

Distributing sequential code

[Caspi-Girault-Pilaud, TSE 1999]

Starting point:

- the sequential code compiled from a synchronous program
- An (abstract) architecture, made of n sites (processors), communicating pointwise through FIFOs
- An assignment of variables to sites

Result: The code for each site

Correctness: The distributed code is functionally equivalent to the centralized automaton

Distributing sequential code

[Caspi-Girault-Pilaud, TSE 1999]

Starting point:

- the sequential code compiled from a synchronous program
- An (abstract) architecture, made of *n* sites (processors), communicating pointwise through FIFOs
- An assignment of variables to sites

Result: The code for each site

Correctness: The distributed code is functionally equivalent to the centralized automaton

The distribution algorithm:

(1) code replication and pruning

(2) insertion of communications

(3) insertion of synchronizations

Distributing sequential code - Principles

Initial code, with assignment of variables to sites

(X1, Y1 computed on site 1, Y2 on site 2)

```
X1 = ...
...
X2 = ...
...
if (X1) {
    ...
    Y1 = F(X2);
}
```

Distributing sequential code - Principles

Code replication and pruning

Site 1

```
X1 = ...
...
X2 = ...
...
if (X1) {
    ...
    Y1 = F(X2);
}
```

Site 2

```
X1 = ...
...
X2 = ...
...
if (X1) {
    ...
    Y1 = F(X2);
}
```

Distributing sequential code - Principles

Code replication and pruning

Site 1

X1 = …

…

~~X2 = …~~

…

if (X1) {

    …

    Y1 = F(X2);

}

Site 2

~~X1 = …~~

…

X2 = …

…

if (X1) {

    …

    ~~Y1 = F(X2)~~;

}

Distributing sequential code - Principles

Insert communications

Site 1

```
X1 = ...
put(X1, Q12);

...
...
if (X1) {
    ...

    Y1 = F(X2);
}
```

Q12

Q21

Site 2

```
...

X2 = ...


...
get(X1, Q12);
if (X1) {
    ...
}
```

Distributing sequential code - Principles

Insert communications

Site 1

```
X1 = ...
put(X1, Q12);

...

...

if (X1) {
    ...

    get(X2, Q21);
    Y1 = F(X2);
}
else get(X2, Q21);
```

Q12

Q21

Site 2

```
...

X2 = ...

put(X2, Q21);
...

get(X1, Q12);
if (X1) {

    ...

}
```

Asynch./Synch. distributed programming

The border of synchronous world

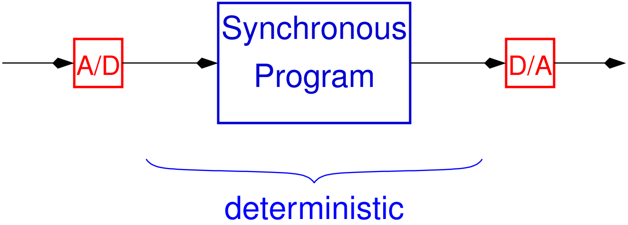
A/D

Synchronous Program

D/A

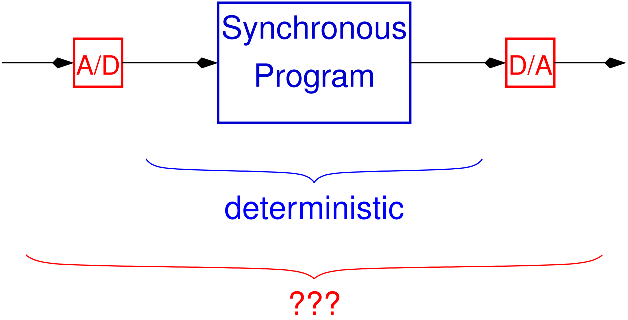
Asynch./Synch. distributed programming

The border of synchronous world



Asynch./Synch. distributed programming

The border of synchronous world



A/D

Synchronous Program

D/A

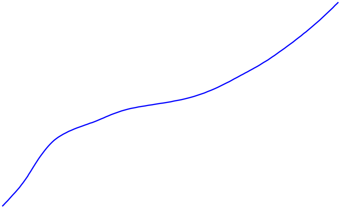
deterministic

???

The border of synchronous world

Input sampling

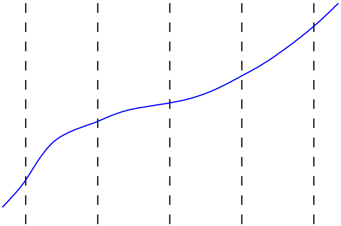
Continuous signals



The border of synchronous world

Input sampling

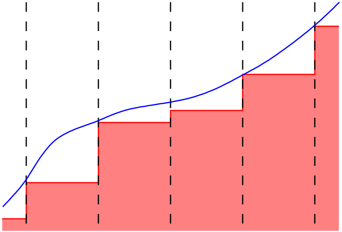
Continuous signals



The border of synchronous world

Input sampling

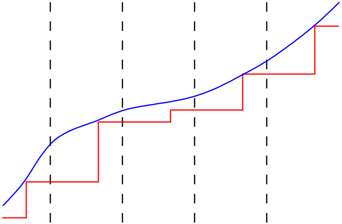
Continuous signals



The border of synchronous world

Input sampling

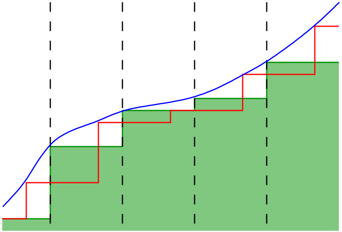
Continuous signals



The border of synchronous world

Input sampling

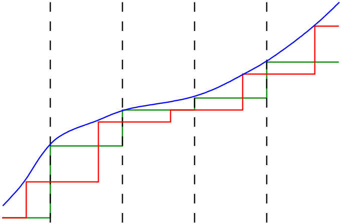
Continuous signals



The border of synchronous world

Input sampling

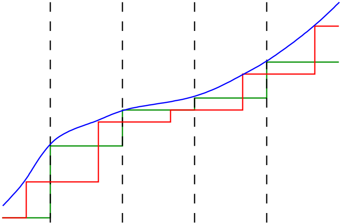
Continuous signals



The border of synchronous world

Input sampling

Continuous signals

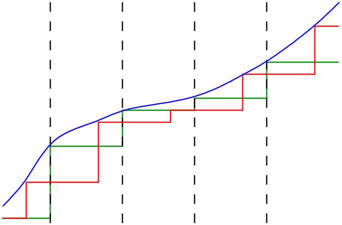


Discrete signals

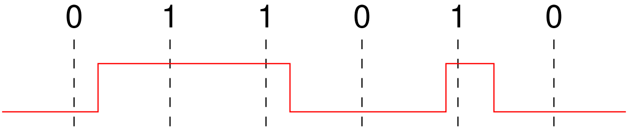
The border of synchronous world

Input sampling

Continuous signals



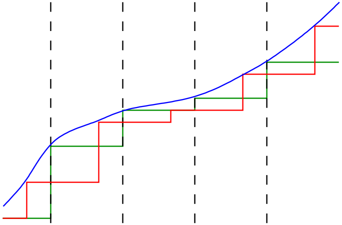
Discrete signals



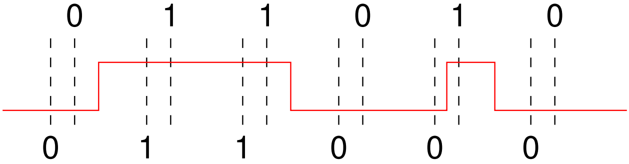
The border of synchronous world

Input sampling

Continuous signals



Discrete signals



The border of synchronous world

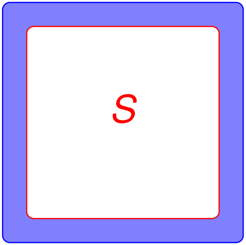
Control theory provides standard ways for keeping this non-determinism under control

- bounding the error on continuous signals thanks to uniform continuity
- using confirmation on Boolean signals (ignoring short variations)

Asynch./synch. programming

[Caspi-Salem, FTRTFT 2000]
[Caspi-Mazuet-Reynaud, SAFECOMP 2001]

Implement a synchronous program as an asynchronous composition of synchronous processes

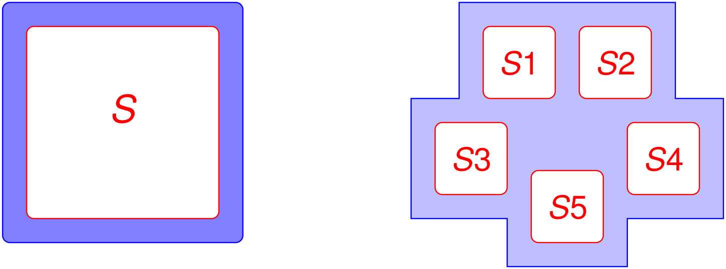


No need to preserve the synchronous (deterministic) semantics, as long as the overall (non-deterministic) semantics is preserved

Asynch./synch. programming

[Caspi-Salem, FTRTFT 2000]
[Caspi-Mazuet-Reynaud, SAFECOMP 2001]

Implement a synchronous program as an asynchronous composition of synchronous processes

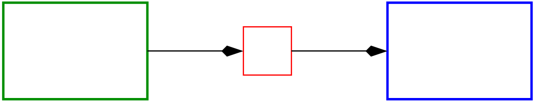


No need to preserve the synchronous (deterministic) semantics, as long as the overall (non-deterministic) semantics is preserved

Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

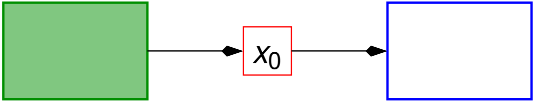
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

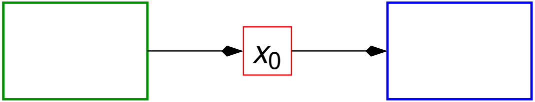
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

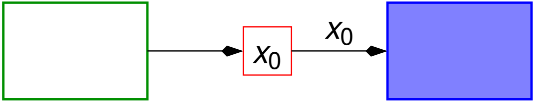
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

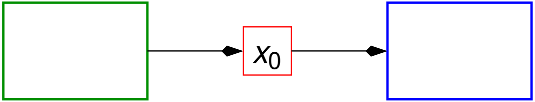
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

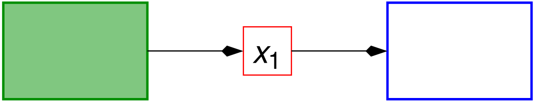
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

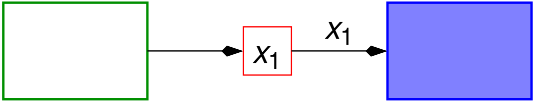
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

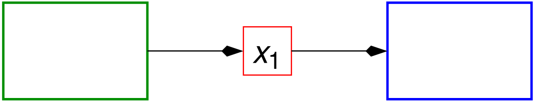
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

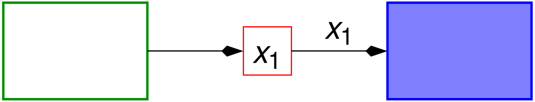
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

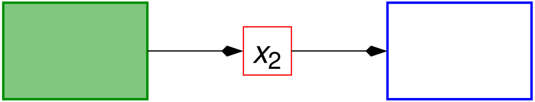
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

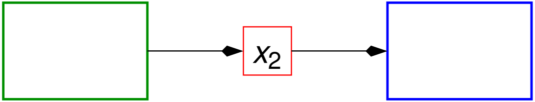
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

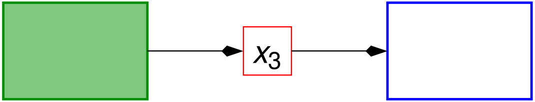
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

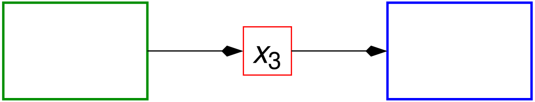
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

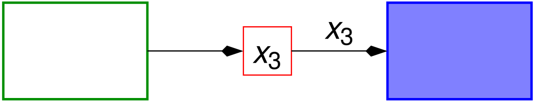
- the same output is sampled twice → duplication
- an output is not sampled → data loss



Asynch./synch. programming (1/3)

Asynchronous/synchronous distribution: Asynchronous composition of synchronous processes.
Each process has its own clock, and samples the outputs of other processes.

- the same output is sampled twice → duplication
- an output is not sampled → data loss



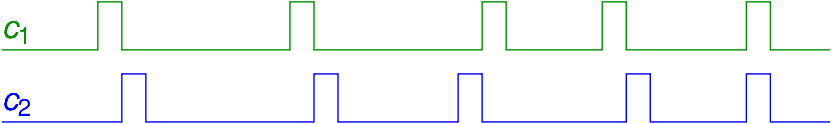
Asynch./synch. programming (2/3)

A special, very common case: Quasi-synchrony

All processes assumed to share the same clock. But, without clock synchronization, clocks may have some drift.

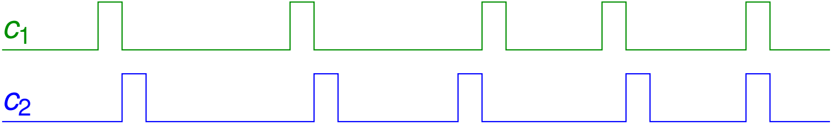
Quasi-synchrony assumption: for any pair (c_1, c_2) of clocks,

- between 2 ticks of c_1 there are at most 2 ticks of $c2$
- and conversely



Asynch./synch. programming (3/3)

Quasi-synchrony (cont)



- at most one data lost in a row
- at most one data duplicated in a row

Consequences:

- a data sent twice is surely transmitted
- be sensitive to data change rather than data read

Conclusion

Conclusion

What shall we do
now that Paul Caspi is retired ??