# An introduction to synchrony

## What is a synchronous model?

Concurrent/distributed systems are classified according to *two main parameters*.

1. The *relative speed* of the processes (or threads, or components, or…):
   - asynchronous,
   - synchronous,
   - partially synchronous,
   - …

2. The way the processes *interact*:
    - shared memory,
    - message based: rendez-vous (also known as synchronous) or bounded/unbounded, ordered/unordered buffers,
    - signals,
    - ...

    See, *e.g.*

    L. Lamport, N. Lynch. Distributed computing: models and methods. Handbook of Theoretical Computer Science.

- So far we have considered models (CCS, the $\pi$-calculus) where:
  - processes are *asynchronous*, *i.e.*, proceed at independent speeds,
  - interaction is either *rendez-vous/synchronous* or *asynchronous* message passing

  **NB** In particular, processes can only *synchronise* through communication.

- In the following we are going to discuss models where:
  - processes are *synchronous*,
  - interaction is either *rendez-vous/synchronous* or *signal based*.

- In first approximation, in a synchronous concurrent/distributed system all processes *proceed in lockstep* (at the same speed).
- In other words, the computation is regulated by a notion of *instant* (or *round*, or *phase*, or *pulse*,. . .). As we will see, what constitutes an instant can vary considerably from one model to another.
- Though synchronous circuits are typical examples of synchronous systems, one should *not* conclude that synchronous systems are *hardware*.

- Notions of synchrony are quite useful in the design of *software systems* too.
- The programming of many problems in distributed/parallel computation can be 'simplified' or even 'made possible' by a synchronous assumption. E.g.
  - Leader election.
  - Minimum spanning tree.
  - Consensus in the presence of failures.

  In general, the notion of synchrony is a useful *logical concept* that can make *programming easier*.

## An example of synchronous model in distributed algorithms

*Synchronous network model* described, *e.g.*, in:

*N. Lynch, Distributed algorithms. Morgan Kaufmann*, and

*G. Tel, Introduction to distributed algorithms. Cambridge University Press.*

- Each node/process is a Moore automaton modulo the fact that the sets of *states*, *inputs*, and *outputs* can be infinite.
- Each node/process has a set of *states* $Q$ and an *initial state* $q_o \in Q$.
- Each node/process has $m$ *incoming edges* (inputs) and $n$ *outgoing edges* (outputs).
- There is a set $M$ of *messages*.

- Each node/process has:
  - An *output* function $out : Q \to M^n$.
  - A *next state* function $next : Q \times M^m \to Q$.
- At each instant, each node/process being in state $q$:
  - Computes $out(q)$ and writes the 'outputs' in the $n$ outgoing edges.
  - Reads the inputs $x_1, \ldots, x_m$ in the incoming edges and places itself in the state $next(q, x_1, \ldots, x_m)$
- The execution model guarantees that when a node tries to read the inputs, all the other nodes have already written their outputs.
- The execution of a network of synchronous processes is (strongly) *deterministic*. There is essentially only one execution path.

## Remark

Models used to describe algorithms are usually over-simplified. For instance, in this case:

- Fixed number of participants and fixed communication topology.
- No explanation on how the algorithm interacts with the external world (usually hard-coded in the initial and final state).

## A synchronous algorithm in the synchronous network model

We describe a *leader election algorithm* (due to Le Lann *et al.*)

**Ring topology** Processes $\mathbf{Z}_n = \{0, 1, \ldots, n-1\}$ are arranged in a *ring*. Process $i$ receives from process $(i-1)$ *mod* $n$ and sends to process $(i+1)$ *mod* $n$.

**UID** Processes do not know necessarily their position or the size of the network but they do know a *unique process identifier* (UID). UID's can be compared.

**Goal** Run a protocol that will elect as leader the process with the highest UID (and no other).

## Informal description

1. Initially, all processes send their UID to the next process.
2. To compute the next state, each process $i$ reads the UID $u$ of the previous process and compares it to its own UID $u_{my}$:

| | |
|---|---|
| $u = u_{my}$ : | $i$ becomes leader |
| $u < u_{my}$ : | repeat step 2, sending nothing to the next process |
| $u > u_{my}$ : | repeat step 2, sending $u$ to the next process |

## Execution: an example

Processes $0, 1, 2, 3$ with UID $5, 7, 4, 3$:

| Round | 0(5) | 1(7) | 2(4) | 3(3) |
|---|---|---|---|---|
| 0 | $(?, 5)$ | $(?, 7)$ | $(?, 4)$ | $(?, 3)$ |
| 1 | $(?, \_)$ | $(?, \_)$ | $(?, 7)$ | $(?, 4)$ |
| 2 | $(?, \_)$ | $(?, \_)$ | $(?, \_)$ | $(?, 7)$ |
| 3 | $(?, 7)$ | $(?, \_)$ | $(?, \_)$ | $(?, \_)$ |
| 4 | $(?, \_)$ | $(L, \_)$ | $(?, \_)$ | $(?, \_)$ |

**NB** Here termination for non-leader processes is *implicit*. To get *explicit* termination, let the leader announce the result.

## Exercise

Formalise the algorithm in the synchronous network model.

## Analysis (informal)

Let $i_{max}$ be the process with the largest UID $u_{max}$ and $n$ the size of the ring. Show that:

1. After $r$ rounds, $0 \leq r \leq (n-1)$, the process $(i_{max} + r)\ mod\ n$ sends $u_{max}$. Thus at round $n$, $i_{max}$ becomes leader.
2. $i_{max}$ never forwards a UID. So no other process different from $i_{max}$ ever becomes leader.

## The same algorithm in an asynchronous framework

- The same ‘algorithm’ works in an asynchronous network, provided each channel is a FIFO queue holding up to $n$ messages (to avoid problems, one could use a Kahn network here).

- It is an instructive exercise to prove the correctness in this framework and compare the proof with the one in the synchronous case.

- The analysis is at the level of the single events rather than at the level of the rounds.

- For instance, the invariant needs to keep track of what is in the queues and termination cannot rely on the number of rounds.

## Synchrony in process calculi: SCCS

## Goals

- Review possible formalisations of the concept of synchrony from a *process calculus perspective* ...
- ... with an eye towards *synchronous programming languages*.

We will follow the historical development up to some recent contributions.

## SCCS: synchronous CCS (Milner 1983)

> We now wish to discuss a calculus [...] It arose from the author's attempt to relate *asynchrony* to *synchrony*. The contrast between these terms may be understood in more than one way. Here, we mean the contrast between the assumption which we have hitherto made that concurrent agents proceed at indeterminate relative speeds (asynchrony), and the alternative assumption that they proceed in lockstep - i.e. that at every instant each agent performs *a single action* (synchrony).

R. Milner, Communication and Concurrency, Prentice-Hall, 1989.

## SCCS: actions

- We start with a set of *particulate actions* $\mathcal{A}$.
- An *action* is a function $\alpha : \mathcal{A} \to \mathbf{Z}$ which is equal to 0 almost everywhere.
- Actions constitute an abelian group:

$$(\alpha \cdot \beta)(a) = \alpha(a) + \beta(a)$$

- This is the *free abelian group* generated by $\mathcal{A}$.
- For instance, we can write $\alpha = a\bar{b}a$ for an action $\alpha$ such that

$$\alpha(c) = \begin{cases} 2 & \text{if } c = a \\ -1 & \text{if } c = b \\ 0 & \text{otherwise} \end{cases}$$

- Thus

$$(a\bar{b}a) \cdot (\bar{a}bb) = ab = ba$$

- By convention we write 1 for the identity, *i.e.*, for the action $\alpha$ which is 0 everywhere.

- The CCS ‘special’ case is when $\alpha(a) \in \{0, 1, -1\}$. But then we do not have a group structure.
- A generalisation is to assume that on a subset $\mathcal{A}$, $\alpha(a)$ is non-negative which means that some actions have no inverse. Then we have a commutative monoid structure with an abelian subgroup.

## SCCS: synchronous product

- Write $\alpha : P$ for the process that must do $\alpha$ in the first instant and run $P$ in the following. Thus

$$\frac{}{\alpha : P \xrightarrow{\alpha} P}$$

- We also have the possibility of choosing among several actions

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

- At each instant, each parallel component *must do an action*:

$$\frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\beta} Q'}{P \times Q \xrightarrow{\alpha \cdot \beta} P' \times Q'}$$

- If one process cannot perform an action the whole system is stuck. Thus, if 0 is the usual process that does no action then:

$$P \times 0 \text{ is equivalent to } 0$$

- The neutral process is the one that runs the identity action at each instant $\mathbf{1} = 1 : 1 : 1 : \cdots$ and that can be defined recursively:

$$P \times \mathbf{1} \text{ is equivalent to } P$$

## Example: product of actions

- Let $\alpha$ abbreviate $\alpha : 0$.
- Consider:

$$P = (a+c) \times (b+c) \times (a+d) \times (\overline{aa} + \overline{a} + 1 + \overline{b}) \times (\overline{cc} + \overline{c} + 1 + \overline{d})$$

- Do we have $P \xrightarrow{1} P'$, for some $P'$?

- Yes, if for instance we fire the subprocesses in red:

$$P = (a + c) \times (b + c) \times (a + d) \times (\overline{aa} + \overline{a} + 1 + \overline{b}) \times (\overline{cc} + \overline{c} + 1 + \overline{d})$$

## Bisimulation for SCCS

- The theory of *bisimulation* developed in the asynchronous case applies equally well in the synchronous case.
- Notice that in a synchronous calculus an observer has a way to *measure time*

$$1 : a : 0 \text{ is observably different from } a : 0$$

- Thus we cannot abstract away the actions 1. In other terms, we rely on the *strong labelled transition system*.
- We regard two processes $P, Q$ 'equivalent' if they are *strongly bisimilar* and write $P \sim Q$.

## Exercise

1. Consider the following fragment of SCCS:

   $$P ::= 0 \mid \alpha : P \mid P \times P$$

   Can we regard these processes as *deterministic*?

2. Next, consider the following larger fragment of SCCS:

   $$P ::= 0 \mid \alpha : P \mid P + P \mid P \times P$$

   Can we regard these processes as *deterministic*?

3. Say that $P$ is *fireable* if $P \xrightarrow{1} P$. Show that for the larger fragment deciding whether $P \xrightarrow{1} P'$ is an NP-complete problem (by reduction of 3-SAT).

## Programming a NOR gate in SCCS

| $a$ | $b$ | $c = NOR(a,b)$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |
| 1 | 0 | 0 |

The process $A_i$ for $i = 0, 1$ has 4 inputs $a_0, a_1, b_0, b_1$ and 2 outputs $c_0, c_1$.

$$A_i = \overline{c}_i : \mathbf{1} \quad \times \quad \Sigma_{i,j \in \{0,1\}}\, a_i \cdot b_j :\ A_{NOR(i,j)} \qquad i = 0, 1$$

The result is emitted in the following instant.

## Programming in SCCS is awkward

- We need two channels for every signal. This is because the data representation is *unary*.
- For instance, to represent a 16 bits integer we need $2^{16}$ channels...
- Worse, it is not possible to *program* the NOR gate. We have to represent its truth table.
- Again, imagine what happens when the input is a 16 bits integer...
- And what about infinite data domain?

## Desynchronisation, or how to wait indefinitely?

- It may be hard to predict the exact computation time of each thread.
- It may even be impossible if the event is generated from an ‘asynchronous’ component.
- We need to express the possibility to wait for an event an arbitrary number of instants

## Desynchronisation operators

**Delay (délai)** The first action can be delayed arbitrarily many instants.

$$\frac{}{\delta P \xrightarrow{1} \delta P} \qquad \frac{P \xrightarrow{\alpha} P'}{\delta P \xrightarrow{\alpha} P'}$$

**Asynchroniser (désynchronisation)** After the first action, all following actions can be delayed arbitrarily many instants.

$$\frac{P \xrightarrow{\alpha} P'}{\Delta P \xrightarrow{\alpha} \delta \Delta P'}$$

## Exercise

Prove or give a counter-example to the following equalities when the equality is interpreted as strong bisimulation.

1. $\delta\delta P = \delta P$
2. $\delta\Delta P = \Delta P$
3. $\Delta\Delta P = \Delta P$
4. $\Delta\delta P = \Delta P$
5. $\delta(P + Q) = \delta P + \delta Q$
6. $\Delta(P + Q) = \Delta P + \Delta Q$
7. $\delta(P \times Q) = \delta P \times \delta Q$
8. $\Delta(P \times Q) = \Delta P \times \Delta Q$

## Embedding CCS in SCCS

- Intuitively, the CCS process

$$a.b.0$$

corresponds to the SCCS process

$$\delta(a:\ \delta(b:\ (\delta 0)))$$

- Following this intuition, it is possible to encode CCS in SCCS.

**NB** Again, the delay/desynchronisation operators are *not* a practical programming notation.

## Meije, a complementary view of SCCS

## Meije (Austry-Boudol 1984)

We keep the same *action structure*. However:

- In SCCS, we start with a *synchronous product* and then we introduce some *desynchronisation operators*.
- In Meije, we start with an *asynchronous product* and then we introduce some *synchronisation operators*.

## Meije operators: Asynchronous composition, Trigger, and Driver

**Asynchronous composition** Components proceed at independent speeds (but multi-way synchronisations are possible):

$$\frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\beta} Q'}{P \parallel Q \xrightarrow{\alpha \cdot \beta} P' \parallel Q'}$$

**Trigger (Déclencheur)** The first action is triggered by a particulate action:

$$\frac{P \xrightarrow{\alpha} P'}{(a \Rightarrow P) \xrightarrow{a\alpha} P'}$$

**Driver (Pilote)** All actions are driven from a particulate action:

$$\frac{P \xrightarrow{\alpha} P'}{(a * P) \xrightarrow{a\alpha} (a * P')}$$

## Summary SCCS/Meije

**Common part** nil $0$, prefix $\alpha : P$, restriction $\nu a\ P$, and recursive definitions $A(\mathbf{a}) = P$.

**SCCS operators** sum $+$, synchronous composition $\times$, delay $\delta$, and asynchroniser $\Delta$.

**Meije operators** asynchronous composition $\|$, trigger $(a \Rightarrow P)$, and driver $(a * P)$.

## Definability

- We show that SCCS and Meije operators are *inter-definable*.
- But what does *definability* mean exactly ?

## Example

- Suppose we want to define an operator $(P \; I \; Q)$ that *interleaves* the actions of $P$ and $Q$.
- We can describe $I$ with the rules:

$$\frac{P \xrightarrow{\alpha} P'}{(P \; I \; Q) \xrightarrow{\alpha} (P' \; I \; Q)} \qquad \frac{Q \xrightarrow{\alpha} Q'}{(P \; I \; Q) \xrightarrow{\alpha} (P \; I \; Q')}$$

- Now we can actually *define* the interleaving operator as follows:

$$P \; I \; Q = \nu a, b \; ((a * P) \parallel (b * Q) \parallel S(a, b))$$

where $S(a, b) = \overline{a} : S(a, b) + \overline{b} : S(a, b)$.

- We have actually built a $\lambda$-term $F : Pr \to Pr \to Pr$:

$$F = \lambda x.\lambda y.\nu a, b\ ((a * x) \parallel (b * y) \parallel S(a, b))$$

  that for any pair of processes $P, Q$ builds a process $F(P, Q)$ that *behaves* as $P\ I\ Q$.

- More precisely, for any $P, Q$ the transition diagram associated with $F(P, Q)$ is *strongly bisimilar* to the one associated with $P\ I\ Q$.

- Note that the $\lambda$-term $F$ does *not* depend on $P, Q$.
- A *weaker definition of definability* could require that for all $P, Q$ there is a process $F_{P,Q}$ that behaves as $P \; I \; Q$.
- For instance, in this weaker sense the operator $\delta$ could be defined as follows.
  - Given $P$ we introduce a fresh identifier $A$ with parameters $\{\mathbf{a}\} = fn(P)$ and a new equation:

$$A(\mathbf{a}) = 1 : A(\mathbf{a}) + P$$

  - Then $\delta P$ is strongly bisimilar to $A(\mathbf{a})$.

## Representing SCCS in Meije

- Let $T_\alpha = \alpha : T_\alpha$ for $\alpha$ action.
- We want to define $+, \times, \delta, \Delta$ from $\|$, $(a \Rightarrow \_)$, and $(a * \_)$.
- Any guesses?

$$P + Q = \nu a((a \Rightarrow P) \parallel (a \Rightarrow Q) \mid \overline{a} : 0)$$

$$P \times Q = \nu a \; ((a * P) \parallel (a * Q) \parallel T_{\overline{aa}})$$

$$\delta P = \nu a((a \Rightarrow P) \parallel D(a))$$

where: $D(a) = (1 : D(a) + \overline{a} : 0)$.

$$\Delta P = \nu a \ ((a * P) \parallel \overline{a} : S(a))$$

where: $S(a) = 1 : S(a) + \overline{a} : S(a)$.

## Exercise

Check that strong bisimulation holds.

## Representing Meije in SCCS

- We want to define $\|$, $(a \Rightarrow \_)$, and $(a * \_)$ from $+$,$\times$,$\delta$, and $\Delta$.
- Any guesses?

$$(a \Rightarrow P) = a : T_1 \times P$$

$$(a * P) = T_a \times P$$

Also let $\nabla P = \delta \Delta P$.

$$P \parallel Q = \nu a, b\ (\ \nabla(a * P) \times \nabla(b * Q) \times S(a,b)\ )$$

where: $S(a,b) = \overline{a} : S(a,b) + \overline{b} : S(a,b) + \overline{ab} : S(a,b)$.

## Exercise

Again check that strong bisimulation holds.

## A methodological remark

> In truth, there is *nothing canonical about* our choice of basic combinators [in CCS], even though they were chosen with great attention to economy. What characterises our calculus is not the exact choice of combinators, but rather the choice of interpretation and of *mathematical framework*.

R. Milner, Communication and Concurrency, Prentice-Hall, 1989, page 195.

So CCS should not be regarded as a *canonical calculus* but rather as an *inspiring example*.

## Some limitations of the SCCS/Meije model

1. Not a programming notation.
2. Implementation model?
3. Generalisation to infinite data domain?

May be we are too short-sighted, but the fact is that more than 20 years later, there is no synchronous programming language that builds directly on the SCCS/Meije model.

## Main references

- R. Milner, *Calculi for synchrony and asynchrony*, TCS, 25, 1983.

  Introduces SCCS and shows how CCS can be embedded into it.

- D. Austry and G. Boudol, *Algèbre de processus et synchronisation*, TCS, 30, 1984.

  Introduces Meije and shows that it is equivalent to SCCS.

- R. De Simone, *Higher-level synchronising devices in Meije-SCCS*, TCS, 37, 1985.

  Shows that a whole class of *finitely presented* operators can be realised in SCCS/Meije.