

Lectures on Determinacy and Synchrony

2008-2009

MPRI C-2-3- Concurrency - Lectures 9-12

Roberto AMADIO

Université Paris Diderot (Paris 7)

Laboratoire Preuves, Programmes et Systèmes

Programme of these lectures

We will cover the notions of:

- Determinacy, Confluence, and Linearity.
- Synchrony and Time.

In the framework of *process calculi* (specifically, CCS, π -calculus, and variations thereof).

Determinacy

What is a deterministic system?

In automata theory, one can consider various definitions. For instance, look at *finite automata*:

Def 1 There is no *word* w that admits two computation paths in the graph such that one leads to an accepting state and the other to a non-accepting state.

Def 2 Each *reachable configuration* admits at most one successor.

Def 3 For each *state*:

- either there is exactly one outgoing transition labelled with ϵ ,
- or all outgoing transitions are labelled with distinct symbols of the input alphabet.

Thus one can go from '*extensional*' conditions (intuitive but hard to verify) to '*syntactic*' conditions (verifiable but not as general).

Why did we allow non-determinism?

Race conditions Two clients request the same service.

$$\nu a (\bar{a}.P_1 \mid \bar{a}.P_2 \mid a)$$

General specification and portability We do not want to commit on a particular behaviour. For instance, consider

$$\nu a, b (\tau.\bar{a}.\bar{b}.\bar{c} \mid a.\bar{b}.\bar{d} \mid b)$$

Depending on the compilation, the design of the virtual machine, the processors timing, . . . we might always run \bar{d} rather than \bar{c} (or the other way around).

Why is determinism desirable?

- Easier to *test and debug*.
- Easier to *prove correct*.

NB Often the implementation seems ‘deterministic’ because:

- either the program is inherently deterministic,
- or the scheduler determinizes the program’s behaviour.

However this kind of determinism is *not portable*.

Towards a definition of determinacy

- If P and P' are '*equivalent*' then one is determinate if and only if the other is.
- If we run an '*experiment*' twice we always get the same 'result'.
- If P is determinate and we run an experiment then *the residual of P* after the experiment should still be determinate.

- For the time being, we will place ourselves in the context of a simple model such as *CCS*.
- We take *equivalent* to mean *weak bisimilar*.
- We take *experiment* to be a finite sequence of *labelled transitions*.

A formal definition of determinacy

- Denote with \mathcal{L} the set of *visible actions and co-actions* with generic elements ℓ, ℓ', \dots
- Denote with $Act = \mathcal{L} \cup \{\tau\}$ the set of *actions*, with generic elements α, β, \dots
- Let $s \in \mathcal{L}^*$ denote a finite word over \mathcal{L} . Then:

$$\begin{aligned} P \xRightarrow{\epsilon} P' & \quad \text{if } P \xRightarrow{\tau} P' \\ P \xRightarrow{\ell_1 \dots \ell_n} P', \ n \geq 1 & \quad \text{if } P \xRightarrow{\ell_1} \dots \xRightarrow{\ell_n} P' \end{aligned}$$

- If $P \xRightarrow{s} Q$ we say that Q is a *derivative* of P .

Definition A process P is *determinate* if for any $s \in \mathcal{L}^*$,

$$\frac{P \xrightarrow{s} P' \quad P \xrightarrow{s} P''}{P' \approx P''}$$

NB This definition relies on the notion of *labelled transition system*.

In $P \xrightarrow{\ell} P'$, ℓ represents a *minimal* and *deterministic* interaction with the environment and P' is the *residual* after the interaction.

Exercise

Are the following CCS processes determinate?

1. $a.(b + c)$.

2. $a.b + ac$.

3. $a + a.\tau$.

4. $a + \tau.a$.

5. $a + \tau$.

Proposition

1. If P is determinate and $P \xrightarrow{\alpha} P'$ then P' is determinate.
2. If P is determinate and $P \approx P'$ then P' is determinate.

Proof idea

1. Suppose $P \xrightarrow{\alpha} P'$ and $P' \xrightarrow{s} P_i$ for $i = 1, 2$.
 - If $\alpha = \tau$ then $P \xrightarrow{s} P_i$ for $i = 1, 2$. Hence $P_1 \approx P_2$.
 - If $\alpha = \ell$ then $P \xrightarrow{\ell \cdot s} P_i$ for $i = 1, 2$. Hence $P_1 \approx P_2$.

2. Suppose $P \approx P'$ and $P' \xrightarrow{s} P'_i$ for $i = 1, 2$.

- By definition of *weak bisimulation*:

$$P \xrightarrow{s} P_i \text{ and } P_i \approx P'_i$$

for $i = 1, 2$.

- Since P is *determinate*, we have $P_1 \approx P_2$.
- Therefore, we conclude by *transitivity* of \approx :

$$P'_1 \approx P_1 \approx P_2 \approx P'_2$$

NB Most proofs in this lecture will be by *diagram chasing*.

τ -inertness and determinacy

Definition We say that a process P is τ -*inert* if for all its derivatives Q , if $Q \xrightarrow{\tau} Q'$ then $Q \approx Q'$.

Proposition If P is determinate then it is τ -inert.

Proof idea

- Suppose $P \xRightarrow{s} Q$ and $Q \xRightarrow{\tau} Q'$.
- Then $P \xRightarrow{s} Q$ and $P \xRightarrow{s} Q'$.
- Thus by determinacy, $Q \approx Q'$.

Trace equivalence

We define the *traces* of a process P as

$$tr(P) = \{s \in \mathcal{L}^* \mid P \xrightarrow{s} \cdot\}$$

and say that two processes P, Q are *trace equivalent* if $tr(P) = tr(Q)$.

NB The traces of a process form a *non-empty, prefix-closed* set of *finite words* over \mathcal{L} .

Exercise

Are the following equations valid for *trace equivalence* and/or *weak bisimulation*?

1. $a + \tau = a.$

2. $\alpha.(P + Q) = \alpha.P + \alpha.Q.$

3. $(P + Q) \mid R = P \mid R + Q \mid R.$

4. $P = \tau.P.$

Proposition

1. If $P \approx Q$ then $\text{tr}(P) = \text{tr}(Q)$.
2. Moreover, if P, Q are *determinate* then $\text{tr}(P) = \text{tr}(Q)$ implies $P \approx Q$.

Proof idea

1. Suppose $P \approx Q$ and $P \xRightarrow{s} \cdot$. Then $Q \xRightarrow{s} \cdot$ by induction on $|s|$ using the properties of weak bisimulation.
2. Suppose P, Q determinate and $tr(P) = tr(Q)$.
 - We show that

$$\{(P, Q) \mid tr(P) = tr(Q)\}$$

is a bisimulation.

- If $P \xrightarrow{\tau} P'$ then $P \approx P'$ by *determinacy*.
- Thus taking $Q \xrightarrow{\tau} Q$ we have:

$$P' \approx P \quad \text{tr}(P) = \text{tr}(Q) .$$

- By (1), we conclude:

$$\text{tr}(P') = \text{tr}(P) = \text{tr}(Q) .$$

- If $P \xrightarrow{\ell} P'$ then we note that:

$$tr(P) = \{\epsilon\} \cup \{\ell\} \cdot tr(P') \cup \bigcup_{\ell \neq \ell', P \xRightarrow{\ell'} P''} \{\ell'\} \cdot tr(P'')$$

- This is because all the processes P' such that $P \xRightarrow{\ell} P'$ are bisimilar, hence trace equivalent.
- A similar reasoning applies to $tr(Q)$.
- Thus there must be a Q' such that $Q \xRightarrow{\ell} Q'$ and $tr(P') = tr(Q')$.

How do we build deterministic systems?

- Start with *deterministic components*.
- Look for *methods to combine* them that *preserve determinacy*.

Exercise

Consider the process $P \mid Q$ where P, Q are as follows.

1. $P = a.b, Q = a.$

2. $P = a, Q = \bar{a}.$

3. $P = a + b, Q = \bar{a}.$

Are $P, Q,$ and $(P \mid Q)$ determinate?

Sorting

Sorting information is useful when trying to combine processes so as to preserve some property such as determinacy.

Let \mathcal{L} be the set of visible actions and L, L', \dots range over $2^{\mathcal{L}}$.

Definition We say that a process P has sort L if all the actions performed by P and its derivatives lie in $L \cup \{\tau\}$.

Remarks on sorting

- In CCS, it is easy to provide an *upper bound for sorting* since:

$$P : fn(P) \cup \overline{fn(P)}$$

where $fn(P)$ are the *free names* in P .

- Sorting is *closed under intersection*: if $P : L_i$ for $i = 1, 2$ then $P : L_1 \cap L_2$.
- Thus each process has a *minimum sort*.
- In general, the minimum sort *cannot be computed* because CCS can simulate Turing machines (TM) and the firing of a transition may correspond to the TM reaching the halting state...
- We discuss a method to compute an *over-approximation* of the minimum sort that we denote with $\mathcal{L}(P)$.

Computing the over-approximation

- Non-trivial programs in CCS are given via a *system of recursive equations*:

$$A(a_1, \dots, a_n) = P$$

where the names a_1, \dots, a_n are all distinct and $fn(P) \subseteq \{a_1, \dots, a_n\}$.

- An *assignment* ρ is a function that associates with every thread identifier A of arity n a function $\rho(A)$ that takes a vector of n names (b_1, \dots, b_n) and produces a subset $\rho(A)(b_1, \dots, b_n)$ of

$$\{b_1, \dots, b_n, \bar{b}_1, \dots, \bar{b}_n\}$$

- The *least assignment* ρ_\emptyset is the function where the ‘subset’ produced is always the empty set: $\rho_\emptyset(A)(b_1, \dots, b_n) = \emptyset$.

- We define the sort $\llbracket P \rrbracket \rho$ of a process P *relatively* to an assignment ρ :

$$\llbracket 0 \rrbracket \rho = \emptyset$$

$$\llbracket \alpha.P \rrbracket \rho = \begin{cases} \llbracket P \rrbracket \rho & \text{if } \alpha = \tau \\ \{\alpha\} \cup \llbracket P \rrbracket \rho & \text{otherwise} \end{cases}$$

$$\llbracket P_1 + P_2 \rrbracket \rho = \llbracket P_1 \rrbracket \rho \cup \llbracket P_2 \rrbracket \rho$$

$$\llbracket P_1 \mid P_2 \rrbracket \rho = \llbracket P_1 \rrbracket \rho \cup \llbracket P_2 \rrbracket \rho$$

$$\llbracket \nu a P \rrbracket \rho = \llbracket P \rrbracket \rho \setminus \{a, \bar{a}\}$$

$$\llbracket A(\mathbf{b}) \rrbracket \rho = \rho(A)(\mathbf{b})$$

- Now we compute iteratively $\rho_0 = \rho_\emptyset$ and ρ_{n+1} so that:

$$\rho_{n+1}(A)(\mathbf{a}) = \llbracket P \rrbracket \rho_n$$

for all identifiers A defined by an equation $A(\mathbf{a}) = P$.

- This defines a *growing sequence* (check this!) that is guaranteed *to converge* after finitely many steps to a *least fixed point* ρ since $\rho_n(A)(\mathbf{a}) \subseteq \{\mathbf{a}\} \cup \overline{\{\mathbf{a}\}}$ which is a *finite set*.

Example

- We consider the system composed of one equation:

$$A(a, b) = a.\nu c (A(a, c) \mid \bar{b}.A(c, b))$$

- Then

$$\begin{aligned} & \rho_1(A)(a, b) \\ &= \llbracket a.\nu c (A(a, c) \mid \bar{b}.A(c, b)) \rrbracket \rho_\emptyset \\ &= \{a\} \cup (\rho_\emptyset(A)(a, c) \cup \{\bar{b}\} \cup \rho_\emptyset(A)(c, b)) \setminus \{c, \bar{c}\} \\ &= \{a, \bar{b}\} \end{aligned}$$

- The following iteration reaches the *fixed point*:

$$\begin{aligned}
& \rho_2(A)(a, b) \\
&= \llbracket a.\nu c (A(a, c) \mid \bar{b}.A(c, b)) \rrbracket \rho_1 \\
&= \{a\} \cup (\rho_1(A)(a, c) \cup \{\bar{b}\} \cup \rho_1(A)(c, b)) \setminus \{c, \bar{c}\} \\
&= \{a\} \cup (\{a, \bar{c}\} \cup \{\bar{b}\} \cup \{c, \bar{b}\}) \setminus \{c, \bar{c}\} \\
&= \{a, \bar{b}\}
\end{aligned}$$

Thus $\mathcal{L}(P) = \{a, \bar{b}\}$.

Some sufficient conditions for building determinate processes

Proposition Suppose P, Q, P_i are determinate processes for $i \in I$. Then:

1. $0, \alpha.P, \nu a P$ are determinate.
2. $\Sigma_{i \in I} \ell_i.P_i$ is determinate if the ℓ_i are *all distinct*.
3. $P \mid Q$ is determinate if P, Q *do not communicate* and *do not share actions* (that is $\mathcal{L}(P) \cap \mathcal{L}(Q) = \emptyset$ and $\mathcal{L}(P) \cap \overline{\mathcal{L}(Q)} = \emptyset$).
4. σP is determinate if σ is an *injective substitution* on the free names in P .

Proof idea

1. For instance, for $\nu a P$ one checks that if $\nu a P \xRightarrow{s} Q$ then $P \xRightarrow{s} P'$ and $Q = \nu a P'$.
2. Routine. Note that it is essential that all the actions are distinct and visible.
3. Because of the hypothesis on the sorting, an action of $(P_1 \mid P_2)$ can be attributed *uniquely* to either P_1 or P_2 . Then we can rely on the determinacy of P_1 and P_2 .
4. The transitions of P and σP are in perfect correspondence as long as σ is injective. Note that if σ is not injective then σP could perform some additional synchronisations.

Summary on determinacy

1. Deterministic processes are τ -inert

$$P \xrightarrow{s} P' \xrightarrow{\tau} P'' \Rightarrow P' \approx P''$$

2. For *deterministic* processes,

$$\text{bisimulation} = \text{trace equivalence.}$$

3. A simple iterative method to extract from a process P an *approximated sorting information*

$$\mathcal{L}(P) \subseteq fn(P) \cup \overline{fn(P)} .$$

4. We rely on the approximated sorting information to *build deterministic processes*.
5. Unfortunately, rules for *parallel composition* are *too restrictive*: no synchronisation.

Confluence

Refining the conditions

We want to allow some form of *communication*, but...

- We have to *avoid race conditions*: two processes compete on the same resource.
- We also have to *avoid that an action preempts* other actions.
- We introduce a notion of *confluence* that strengthens determinacy and is preserved by some form of communication (parallel composition + restriction).
- For instance,

$$\nu a ((a + b) \mid \bar{a})$$

will be *rejected* because $a + b$ is *not* confluent (while being *deterministic*).

Confluence: rewriting vs. concurrency

- Notion reminiscent of *confluence* in term rewriting systems and λ -calculus (Church-Rosser theorem)

$$\frac{t \xrightarrow{*} t_1, \quad t \xrightarrow{*} t_2}{\exists s \ (t_1 \xrightarrow{*} s, \quad t_2 \xrightarrow{*} s)}$$

- By analogy one calls confluence the related theory in process calculi but bear in mind that:
 1. Confluence is relative to a *labelled transition system*.
 2. We close diagrams *up to equivalence*.

Definition of confluence

We define a notion of *action difference*:

$$\alpha \setminus \beta = \begin{cases} \alpha & \text{if } \alpha \neq \beta \\ \tau & \text{otherwise} \end{cases}$$

Definition (Conf 0) A process P is *confluent* if *for every derivative* Q of P we have:

$$\frac{Q \xRightarrow{\alpha} Q_1 \quad Q \xRightarrow{\beta} Q_2}{\exists Q'_1, Q'_2 (Q_1 \xRightarrow{\beta \setminus \alpha} Q'_1 \quad Q_2 \xRightarrow{\alpha \setminus \beta} Q'_2 \quad Q'_1 \approx Q'_2)}$$

NB If $\alpha = \beta$ then we close the diagram with τ actions only.

Some properties

A first *sanity check* is to verify that the definition is invariant under *transitions* and *equivalence*.

Proposition

1. If P is confluent and $P \xrightarrow{\alpha} P'$ then P' is confluent.
2. If P is confluent and $P \approx P'$ then P' is confluent.

Proof idea (cf. similar proof for determinacy)

1. If Q is a derivative of P' then it is also a derivative of P .
2. It is enough to apply the fact that:

$$(P \approx P' \text{ and } P \xrightarrow{\alpha} P_1) \text{ implies } (P' \xrightarrow{\alpha} P'_1 \text{ and } P_1 \approx P'_1)$$

and the transitivity of \approx .

Determinacy vs. Confluence

Confluence implies τ -inertness, and from this we can show that it implies determinacy too.

Proposition Suppose P is *confluent*. Then P is:

1. τ -*inert*, and
2. *determinate*.

Reminder

A relation R is a *weak bisimulation up to* \approx if

$$\frac{P R Q \quad P \xrightarrow{\alpha} P'}{Q \xrightarrow{\alpha} Q' \quad P'(\approx \circ R \circ \approx)Q'}$$

(and symmetrically for Q).

NB It is important that we work with the *weak moves* on both sides, otherwise the relation R is *not* guaranteed to be contained in \approx . E.g. consider

$$R = \{(\tau.a, 0)\}$$

Proof idea

1. We want to show that $P \xrightarrow{\tau} Q$ implies $P \approx Q$.

- We show that

$$R = \{(P, Q) \mid P \xrightarrow{\tau} Q\}$$

is a weak bisimulation up to \approx .

- It is clear that whatever Q does, P can do too with some extra moves.
- Suppose, for instance, $P \xrightarrow{\alpha} P_1$ with $\alpha \neq \tau$ (case $\alpha = \tau$ left as exercise).
- By (Conf 0),

$$Q \xrightarrow{\alpha} Q_1 \quad P_1 \xrightarrow{\tau} P_2 \quad Q_1 \approx P_2$$

- That is

$$P_1(R \circ \approx)Q_1$$

2. We want to show that if P is confluent then it is determinate.

- Suppose $P \xRightarrow{s} P_i$ for $i = 1, 2$ and $s \in \mathcal{L}^*$.
- We proceed by *induction* on the length $|s|$ of s .
- If $|s| = 0$ and $P \xRightarrow{\tau} P_i$ for $i = 1, 2$ then *by τ -inertness*

$$P_1 \approx P \approx P_2 .$$

- For the inductive case, suppose $P \xRightarrow{\ell} P'_i \xRightarrow{r} P_i$ for $i = 1, 2$.
- By confluence and τ -inertness, we derive that $P'_1 \approx P'_2$.
- By *weak bisimulation*, $P'_2 \xRightarrow{r} P''_2$ and $P''_2 \approx P_1$.
- By *inductive hypothesis*, $P_2 \approx P''_2$.
- Thus $P_2 \approx P''_2 \approx P_1$ as required.

Exercise

We have seen that confluence implies determinacy which implies τ -inertness. Give examples that show that these implications cannot be reversed.

Characterisations of Confluence

A first characterisation

We consider a first ‘asymmetric’ characterisation where the move from Q to Q_1 just concerns a *single action*.

Proposition (Conf 1) A process P is confluent iff for every derivative Q of P , we have:

$$\frac{Q \xrightarrow{\alpha} Q_1 \quad Q \xrightarrow{\beta} Q_2}{\exists Q'_1, Q'_2 \left(Q_1 \xrightarrow{\beta \setminus \alpha} Q'_1 \quad Q_2 \xrightarrow{\alpha \setminus \beta} Q'_2 \quad Q'_1 \approx Q'_2 \right)}$$

Proof idea

- The diagrams of (Conf 1) are a particular case of (Conf 0).
- Thus we just have to show that the diagrams of (Conf 1) suffice to complete the diagrams of (Conf 0).
- We may proceed by induction on the length of the transition $Q \xRightarrow{\alpha} Q_1$. For instance suppose $\alpha \neq \beta$, $\beta \neq \tau$, and

$$Q \xrightarrow{\tau} Q_1 \xRightarrow{\alpha} Q_2 \quad Q \xRightarrow{\beta} Q_3$$

- By (Conf 1),

$$Q_1 \xRightarrow{\beta} Q_4 \quad Q_3 \xRightarrow{\tau} Q_5 \quad Q_4 \approx Q_5$$

- By inductive hypothesis

$$Q_2 \xrightarrow{\beta} Q_6 \quad Q_4 \xrightarrow{\alpha} Q_7 \quad Q_4 \approx Q_7$$

- From $Q_4 \approx Q_5$ and $Q_4 \xrightarrow{\alpha} Q_7$ we derive

$$Q_5 \xrightarrow{\alpha} Q_8 \quad Q_7 \approx Q_8$$

- Therefore

$$Q_2 \xrightarrow{\beta} Q_6 \quad Q_3 \xrightarrow{\alpha} Q_8 \quad Q_6 \approx Q_8$$

as required.

Exercise

Consider another case of the proof. For instance, when
 $Q \xrightarrow{\alpha} Q_1 \xrightarrow{\tau} Q_2$.

Difference of sequences

In another direction we seek a more general definition of confluence where one commutes *sequences of actions*.

- Let $r, s \in \mathcal{L}^*$. To compute the *difference* $r \setminus s$ of r by s we scan r from left to right deleting each label which occurs in s taking into account the multiplicities (cf. difference of *multi-sets*).

$$\begin{aligned}(\epsilon \setminus s) &= \epsilon \\ (\ell r \setminus s) &= \begin{cases} \ell \cdot (r \setminus s) & \text{if } \ell \notin s \\ r \setminus (s_1 \cdot s_2) & \text{if } s = s_1 \ell s_2, \ell \notin s_1 \end{cases}\end{aligned}$$

- For instance

$$aba \setminus ca = ba \quad ca \setminus aba = c$$

Exercise

Let $r, s, t \in \mathcal{L}^*$. Show that:

1. $(rs) \setminus (rt) = s \setminus t$.
2. $r \setminus (st) = (r \setminus s) \setminus t$.
3. $(rs) \setminus t = (r \setminus t)(s \setminus (t \setminus r))$.

A final characterisation of confluence

Proposition (Conf 2) A process P is *confluent* iff for all $r, s \in \mathcal{L}^*$ we have:

$$\frac{P \xRightarrow{r} P_1 \quad P \xRightarrow{s} P_2}{\exists P'_1, P'_2 \quad P_1 \xRightarrow{s \setminus r} P'_1 \quad P_2 \xRightarrow{r \setminus s} P'_2 \quad P'_1 \approx P'_2}$$

Proof idea

(\Leftarrow) It suffices to check that if P has property (Conf 2) then its derivatives have it too.

- Suppose $P \xRightarrow{t} Q$ for $t \in \mathcal{L}^*$.
- Suppose further $Q \xRightarrow{r} Q_1$ and $Q \xRightarrow{s} Q_2$.
- By composing diagrams and applying (Conf 2) we get:

$$Q_1 \xRightarrow{(ts \setminus tr)} Q'_1 \quad Q_2 \xRightarrow{(tr \setminus ts)} Q'_2 \quad Q'_1 \approx Q'_2$$

- Applying the previous exercise we derive, *e.g.*:

$$ts \setminus tr = s \setminus r$$

(\Rightarrow) We proceed in three steps.

1. By induction on $|s|$ we show that:

$$\frac{P \xrightarrow{\tau} P_1 \quad P \xrightarrow{s} P_2}{\exists P'_1, P'_2 \quad P_1 \xrightarrow{s} P'_1 \quad P_2 \xrightarrow{\tau} P'_2 \quad P'_1 \approx P'_2}$$

2. Then, again by induction on $|s|$, we show that:

$$\frac{P \xrightarrow{\ell} P_1 \quad P \xrightarrow{s} P_2}{\exists P'_1, P'_2 \quad P_1 \xrightarrow{s \setminus \ell} P'_1 \quad P_2 \xrightarrow{\ell \setminus s} P'_2 \quad P'_1 \approx P'_2}$$

3. Finally we prove the commutation of diagram (Conf 2) by induction on $|r|$ when $P \xrightarrow{r} P_1$

Exercise

Complete the proof.

**Summary: equivalent definitions of
confluence for a process P**

Conf 0

For all derivatives Q :

$$\frac{Q \xrightarrow{\alpha} Q_1 \quad Q \xrightarrow{\beta} Q_2}{\exists Q'_1, Q'_2 (Q_1 \xrightarrow{\beta \setminus \alpha} Q'_1 \quad Q_2 \xrightarrow{\alpha \setminus \beta} Q'_2 \quad Q'_1 \approx Q'_2)}$$

Conf 1

For all derivatives Q

$$\frac{Q \xrightarrow{\alpha} Q_1 \quad Q \xrightarrow{\beta} Q_2}{\exists Q'_1, Q'_2 (Q_1 \xrightarrow{\beta \setminus \alpha} Q'_1 \quad Q_2 \xrightarrow{\alpha \setminus \beta} Q'_2 \quad Q'_1 \approx Q'_2)}$$

Conf 2

$$\frac{P \xrightarrow{r} P_1 \quad P \xrightarrow{s} P_2}{\exists P'_1, P'_2 (P_1 \xrightarrow{s \setminus r} P'_1 \quad P_2 \xrightarrow{r \setminus s} P'_2 \quad P'_1 \approx P'_2)}$$

Building confluent processes

Building confluent processes

Next, we return to the issue of building confluent (and therefore determinate) processes.

Proposition If P, Q are confluent processes then so are:

1. $0, \alpha.P$.
2. $\nu a P$.
3. σP where σ is an injective substitution on the free names of P .

Proof Routine analysis of transitions (cf. similar statement for determinacy).

Remark on sum

- In general, $a + b$ is *determinate* but it is *not confluent* for $a \neq b$
- To have confluence, one may consider a special kind of ‘commuting sum’

$$(a \mid b).P =_{def} a.b.P + b.a.P$$

Restricted composition

We allow a parallel composition with restriction

$$\nu a_1, \dots, a_n (P \mid Q)$$

when:

1. P and Q do not share visible actions:

$$\mathcal{L}(P) \cap \mathcal{L}(Q) = \emptyset$$

2. P and Q may interact only on the names in $\{\mathbf{a}\}$:

$$\mathcal{L}(P) \cap \overline{\mathcal{L}(Q)} \subseteq \{a_1, \dots, a_n\}$$

Proposition Confluence is preserved by restricted composition.

Proof idea

- First we observe that any derivative of $\nu \mathbf{a} (P \mid Q)$ will have the shape $\nu \mathbf{a} (P' \mid Q')$ where P' is a derivative of P and Q' is a derivative of Q .
- Since sorting is preserved by transitions, the two conditions on sorting will be satisfied.
- Therefore, it is enough to show that the diagrams in (Conf 1) commute for processes of the shape $R = \nu \mathbf{a} (P \mid Q)$ under the given hypotheses.

- We consider one case. Suppose:

$$R \xrightarrow{\ell} \nu a (P_1 \mid Q), \quad \text{because } P \xrightarrow{\ell} P_1$$

- Also assume:

$$R \xRightarrow{\ell} \nu a (P_2 \mid Q_2)$$

because $P \xRightarrow{slr} P_2$ and $Q \xRightarrow{\bar{s}\cdot\bar{r}} Q_2$ with $s \cdot r \in \{\mathbf{a}, \bar{\mathbf{a}}\}^*$ and $\ell \notin \{\mathbf{a}, \bar{\mathbf{a}}\}$.

- Since P is confluent we have:

$$\frac{P \xrightarrow{\ell} P_1 \quad P \xRightarrow{slr} P_2}{P_1 \xRightarrow{sr} P'_1 \quad P_2 \xRightarrow{\tau} P'_2 \quad P'_1 \approx P'_2}$$

- Then we have that:

$$\nu\mathbf{a} (P_1 \mid Q) \stackrel{\tau}{\Rightarrow} \nu\mathbf{a} (P'_1 \mid Q_2) \approx \nu\mathbf{a} (P'_2 \mid Q_2)$$

thus closing the diagram (note that we use the congruence properties of \approx).

Exercise

Consider another case of the proof, for instance:

$$\begin{aligned} \nu a (P \mid Q) &\xrightarrow{\tau} \nu a (P \mid Q) && \text{as } P \xrightarrow{a} P_1, && Q \xrightarrow{\bar{a}} Q_1 \\ \nu a (P \mid Q) &\xRightarrow{\tau} \nu a (P_2 \mid Q_2) && \text{as } P \xrightarrow{s} P_2, && Q \xRightarrow{\bar{s}} Q_2 \end{aligned}$$

A case study: Kahn networks

Interaction model

Point-to-point communication for every channel there is at most one sender and one receiver.

Ordered buffers of unbounded capacity send is non blocking and the order of emission is preserved at the reception.

Each thread may:

1. perform arbitrary *sequential deterministic computation*,
2. *insert a message in a buffer*,
3. *receive a message from a buffer*. If the buffer is empty then the thread *must* suspend,

A thread *cannot* try to receive a message from several channels at once.

Semantics (informal)

- We regard the unbounded buffers as finite or infinite words over some data domain.
- The nodes of the networks are functions over words.
- Kahn observes that the associated system of equations has a least fixed point.

- Kahn networks is an important (practical) case where *concurrency* and *determinism* coexist. For instance, they are frequently used in the *signal processing* community.
- We refer to the course on Synchronous Systems for more information on Kahn networks and related applications.
- Our modest goal is to:
 1. Formalise Kahn networks as a fragment of CCS.
 2. Apply the developed theory to show that the fragment is confluent and therefore deterministic.

CCS formalisation of Kahn networks

- We will work with a ‘data domain’ that contains just one element.
- The generalisation to arbitrary data domains is not difficult, but we would need to formalise determinacy and confluence in the framework of *CCS with values* (a word on this later...).
- First problem: how do we model *unbounded buffers* in CCS?

Representing an unbounded buffer in CCS

A unbounded buffer taking inputs on a and producing outputs on b can be written as (yes, you have already seen this!):

$$Buf(a, b) = a.\nu c (Buf(a, c) \mid \bar{b}.Buf(c, b))$$

- We will write more suggestively $a \mapsto b$ for $Buf(a, b)$, assuming $a \neq b$.
- We have already analysed the sorting of this system:

$$\mathcal{L}(a \mapsto b) = \{a, \bar{b}\}$$

- Moreover, this system falls within the class of *confluent processes* we have considered as it relies on *restricted composition*:

$$\begin{aligned} \mathcal{L}(a \mapsto c) \cap \mathcal{L}(\bar{b}.c \mapsto b) &= \emptyset \\ \mathcal{L}(a \mapsto c) \cap \overline{\mathcal{L}(\bar{b}.c \mapsto b)} &\subseteq \{c, \bar{c}\} \end{aligned}$$

- We would like to show that $a \mapsto b$ works indeed as an unbounded buffer.
- Let $\bar{c}^n = \bar{c} \dots \bar{c}$, n times, $n \geq 0$.
- We should have:

$$P(n) = \nu a (\bar{a}^n \mid a \mapsto b) \approx \bar{b}^n$$

- This is an interesting exercise because:
 - The process $P(n)$ has a non trivial *dynamics*.
 - We can prove the statement just by considering *finite traces*.

Computing the trace of $P(n)$

- Obviously:

$$tr(\bar{b}^n) = \{\epsilon, \bar{b}, \bar{b}\bar{b}, \dots, \bar{b}^n\}$$

- We have $\mathcal{L}(P(n)) = \{\bar{b}\}$, thus $tr(P(n))$ is a non-empty prefix closed set of finite words over \bar{b} .
- For $n = 0$, $P(n)$ can do no transition.
- For $n > 0$ we need to generalise a bit the form of the process $P(n)$. Let $Q(n, m)$ be a process of the form:

$$Q(n, m) = \nu a, c_1, \dots, c_m (\bar{a}^n \mid a \mapsto c_1 \mid \dots \mid c_m \mapsto b)$$

for $m \geq 0$. Note that $P(n) = Q(n, 0)$ and $Q(0, k) \approx 0$ for any k .

- Moreover

$$Q(n, m) \xrightarrow{\bar{b}} Q(n-1, 2m+1)$$

- Thus

$$P(n) \xrightarrow{\bar{b}} \cdots \xrightarrow{\bar{b}} Q(0, 2^n - 1) \approx 0$$

- Because $P(n)$ is confluent we can conclude that:

$$\text{tr}(P(n)) = \text{tr}(\bar{b}^n)$$

CCS processes representing Kahn networks

We define a class of CCS processes sufficient to represent Kahn networks.

- Let KP be the least set of processes such that $0 \in KP$ and if $P, Q \in KP$ and α is an action then
 1. $\alpha.P \in KP$,
 2. $\nu \mathbf{a} (P \mid Q) \in KP$ provided $\mathcal{L}(P) \cap \mathcal{L}(Q) = \emptyset$ and $\mathcal{L}(P) \cap \overline{\mathcal{L}(Q)} \subseteq \{\mathbf{a}, \bar{\mathbf{a}}\}$,
 3. $B(\mathbf{b}) \in KP$ if the names \mathbf{b} are all distinct.
- We admit a recursive equation $A(\mathbf{a}) = P$ only if $P \in KP$.
- We admit processes that are in KP and that depend on recursive equations of the shape above.
- It is easily checked that $a \mapsto b$ is admissible and that Kahn processes are confluent.

From a Kahn network to a CCS process

Suppose we have a Kahn network with three nodes, and the following ports and behaviours where we use ! for output and ? for input.

Node	Ports	Behaviours
1	?a, ?b, ?c, !d, !e, !f	$A_1 = ?a.!d.!e.?b.?c.!f.A_1$
2	!b, ?d	$A_2 = ?d.!b.A_2$
3	!c, ?e	$A_3 = ?e.!c.A_3$

The corresponding CCS system relies on the equations for *Buf* plus:

$$\begin{aligned}A_1(a, b, c, d, e, f) &= a.\bar{d}.\bar{e}.b.c.\bar{f}.A_1(a, b, c, d, e, f) \\A_2(b, d) &= d.\bar{b}.A_2(b, d) \\A_3(c, e) &= e.\bar{c}.A_3(c, e)\end{aligned}$$

The sorting is easily derived:

$$\begin{aligned}\mathcal{L}(A_1(a, b, c, d, e, f)) &= \{a, b, c, \bar{d}, \bar{e}, \bar{f}\} \\ \mathcal{L}(A_2(b, d)) &= \{\bar{b}, d\} \\ \mathcal{L}(A_3(c, e)) &= \{\bar{c}, e\}\end{aligned}$$

To build the system, we have to introduce a buffer before every input channel. Thus the initial configuration is:

$$\begin{aligned} & \nu a', b, b', c, c', d, d', e, e' \\ & (a \mapsto a' \mid b \mapsto b' \mid c \mapsto c' \mid d \mapsto d' \mid e \mapsto e' \mid \\ & A_1(a', b', c', d, e, f) \mid A_2(b, d') \mid A_3(c, e')) \end{aligned}$$

It is easily checked that the resulting processes belong to the class *KP*.

NB Via recursion, we can represent Kahn networks with a dynamically changing number of nodes (*e.g.*, the buffer).

Summary on building confluent processes

To build confluent processes we can use:

- nil and input prefix,
- restricted composition,
- injective recursive calls,
- recursive equations $A(\mathbf{a}) = P$, where P is built according to the rules above.

This class of processes is enough to represent Kahn networks.

Reactivity and Local Confluence

Termination and Local confluence

- In rewriting theory, local confluence is the following condition:

$$\frac{t \rightarrow t_1, \quad t \rightarrow t_2}{\exists s \ (t_1 \xrightarrow{*} s, \quad t_2 \xrightarrow{*} s)}$$

Thus with local confluence we just have to close *one step* diagrams.

- Newman's lemma states that *local confluence* plus *termination* entails *confluence*.
- We present a suitable generalisation of this result to our framework.
- We begin by recalling the classic proof of Newman's lemma.

Well-founded sets and induction principle

Well-founded set A *well-founded* set $(W, >)$ is a set W equipped with a transitive relation $>$ which does not contain infinite descending sequences:

$$x_0 > x_1 > x_2 > \cdots$$

Hence $>$ must be *strict*. If $x \in W$ let $\downarrow(x) = \{y \in W \mid x > y\}$.

Induction principle Let $(W, >)$ be a well-founded set and let $A \subseteq W$.

$$\frac{\forall x (\downarrow(x) \subseteq A \supset x \in A)}{A = W}$$

Exercise (general culture, optional)

Let $(X, >)$ be a set X with a transitive relation $>$. Show that $(X, >)$ is well-founded if and only if the induction principle holds on $(X, >)$.

Local confluence and Newman lemma

Let (X, \rightarrow) be a rewriting system, *i.e.*, a set X with a binary relation \rightarrow . (X, \rightarrow) is:

confluent if for all $x \in X$

$$\frac{x \xrightarrow{*} x_1, \quad x \xrightarrow{*} x_2}{\exists y \ (x_1 \xrightarrow{*} y, \quad x_2 \xrightarrow{*} y)}$$

locally confluent if for all $x \in X$

$$\frac{x \rightarrow x_1, \quad x \rightarrow x_2}{\exists y \ (x_1 \xrightarrow{*} y, \quad x_2 \xrightarrow{*} y)}$$

terminating if all reduction sequences $x_0 \rightarrow x_1 \rightarrow \dots$ are finite.

Newman's lemma

If a rewriting system is *locally confluent* and *terminates* then it is *confluent*.

Proof

- Consider a rewriting system (X, \rightarrow) .
- If \rightarrow^+ denotes the transitive closure of \rightarrow then (X, \rightarrow) terminates iff (X, \rightarrow^+) is well-founded.
- Let $A \subseteq X$ be the set of *elements* for which the confluence condition holds.

- We know that $(X, \overset{+}{\rightarrow})$ is well-founded. We show that $A = X$ by applying the induction principle.

$$\frac{\forall x (\downarrow (x) \subseteq A \supset x \in A)}{A = X}$$

- If x is a normal form then $x \in A$.

- Otherwise, suppose

$$x \rightarrow x_1 \xrightarrow{*} x_2, \quad x \rightarrow x_3 \xrightarrow{*} y_4$$

- By local confluence,

$$\exists x_5 \ (x_1 \xrightarrow{*} x_5, \quad x_3 \xrightarrow{*} x_5)$$

- By inductive hypothesis on x_1 ,

$$\exists x_6 \ (x_2 \xrightarrow{*} x_6, \quad x_5 \xrightarrow{*} x_6)$$

- Again by inductive hypothesis on x_3 ,

$$\exists x_7 \ (x_6 \xrightarrow{*} x_7, \quad x_4 \xrightarrow{*} x_7)$$

Back to processes...

- A process P is *terminating* (or strongly normalising) if there is no infinite sequence

$$P \xrightarrow{\tau} P_1 \xrightarrow{\tau} \dots$$

- A process P is *reactive* (or fully terminating) if all its derivatives are terminating.
- A process P is *locally confluent* if for all its derivatives Q :

$$\frac{Q \xrightarrow{\alpha} Q_1 \quad Q \xrightarrow{\beta} Q_2}{\exists Q'_1, Q'_2 \quad (Q_1 \xRightarrow{\beta \setminus \alpha} Q'_1 \quad Q_2 \xRightarrow{\alpha \setminus \beta} Q'_2 \quad Q'_1 \approx Q'_2)}$$

Exercise

Consider again the process

$$A(a, b) = a.\nu c (A(a, c) \mid \bar{b}.A(c, b))$$

Is the process $A(a, b)$ reactive? Consider the cases $a \neq b$ and $a = b$.

Exercise

Consider the process:

$$A = a.b + \tau.(a.c + \tau.A)$$

Check whether A is:

1. τ -inert,
2. locally confluent,
3. terminating.
4. reactive.
5. determinate.
6. confluent.

A generalisation of Newman's lemma

- Suppose P is a *reactive* process and let W be the set of its derivatives.
- For $Q, Q' \in W$ write $Q > Q'$ if Q rewrites to Q' by a positive number of τ -actions.
- Then $(W, >)$ is a well founded order.

Proposition If a process is *reactive* and *locally confluent* then it is *confluent*.

Proof outline

Let B be the relation $\xrightarrow{\tau} \cup (\xrightarrow{\tau})^{-1} \cup \approx$ (restricted to W) and B^* its reflexive and transitive closure. Note B^* is symmetric too.

1. For every derivative Q of P it holds:

$$\frac{Q \xrightarrow{\tau} Q_1, \quad Q \xrightarrow{\alpha} Q_2}{\exists Q_3 \ (Q_1 \xrightarrow{\alpha} Q_3, \quad Q_2 B^* Q_3)}$$

2. The relation B^* is a weak-bisimulation.
3. The process P is τ -inert.
4. The process P is confluent.

NB B^* is a binary relation on W (the derivatives of P).

Step 1

For every derivative Q of P it holds:

$$\frac{Q \xrightarrow{\tau} Q_1, \quad Q \xrightarrow{\alpha} Q_2}{\exists Q_3 \ (Q_1 \xrightarrow{\alpha} Q_3, \quad Q_2 B^* Q_3)}$$

Proof By induction on the well founded order $(W, >)$.

- If $Q = Q_1$ then the statement holds trivially.
 - So assume $Q \xrightarrow{\tau} Q_3 \xrightarrow{\tau} Q_1$ and consider 2 cases.
 1. If $Q \xrightarrow{\tau} Q_4 \xrightarrow{\alpha} Q_2$.
 - By local confluence, $Q_3 \xrightarrow{\tau} Q_5$, $Q_4 \xrightarrow{\tau} Q_6$, and $Q_5 \approx Q_6$.
 - By ind. hyp., $Q_6 \xrightarrow{\alpha} Q_7$ and $Q_2 B^* Q_7$.
 - By def. of bis. $Q_5 \xrightarrow{\alpha} Q_8$ and $Q_7 \approx Q_8$.
 - By ind. hyp., $Q_1 \xrightarrow{\alpha} Q_9$ and $Q_8 B^* Q_9$.
- So $Q_2 B^* Q_7 \approx Q_8 B^* Q_9$ and by def. of B , $Q_2 B^* Q_9$.

2. If $Q \xrightarrow{\alpha} Q_4 \xrightarrow{\tau} Q_2$ with $\alpha \neq \tau$.

– By local confluence, $Q_3 \xrightarrow{\alpha} Q_5$, $Q_4 \xrightarrow{\tau} Q_6$, $Q_5 \approx Q_6$.

– By ind. hyp., $Q_1 \xrightarrow{\alpha} Q_7$ and $Q_5 B^* Q_7$.

So $Q_2 \xleftarrow{\tau} Q_4 \xrightarrow{\tau} Q_6 \approx Q_5 B^* Q_7$. Hence $Q_2 B^* Q_7$.

Step 2

The relation B^* is a weak-bisimulation.

Proof Suppose $Q_0 B Q_1 \cdots B Q_n B Q_{n+1}$ and $Q_0 \xrightarrow{\alpha} Q'_0$. Proceed by ind. on n and case analysis on $Q_n B Q_{n+1}$. By ind. hyp. we know that $Q_n \xrightarrow{\alpha} Q'_n$ and $Q'_0 B^* Q'_n$.

1. If $Q_n \approx Q_{n+1}$ then $Q_{n+1} \xrightarrow{\alpha} Q'_{n+1}$ and $Q'_n \approx Q'_{n+1}$. So $Q'_0 B^* Q'_n \approx Q'_{n+1}$ and we use $B^* \circ \approx \subseteq B^*$.
2. If $Q_n \xleftarrow{\tau} Q_{n+1}$ then $Q_{n+1} \xrightarrow{\alpha} Q'_n$.
3. If $Q_n \xrightarrow{\tau} Q_{n+1}$ then by Step (1), $Q_{n+1} \xrightarrow{\alpha} Q'_{n+1}$ and $Q'_n B^* Q'_{n+1}$.
So $Q'_0 B^* Q'_n B^* Q'_{n+1}$ and we use $B^* \circ B^* \subseteq B^*$.

Step 3

The process P is τ -inert.

Proof By def., $\xrightarrow{\tau} \subseteq B^*$ and by Step (2), $B^* \subseteq \approx$.

Step 4

The process P is confluent.

Proof By ind. on the well-founded order.

1. Suppose $Q \xrightarrow{\alpha} Q_3 \xRightarrow{\tau} Q_1$ and $Q \xrightarrow{\beta} Q_4 \xRightarrow{\tau} Q_2$, with $\alpha, \beta \neq \tau$.
 - By local confluence, $Q_3 \xRightarrow{\beta \setminus \alpha} Q_5$, $Q_4 \xRightarrow{\alpha \setminus \beta} Q_6$, and $Q_5 \approx Q_6$.
 - By Step (3), $Q_4 \approx Q_2$, and by weak bis., $Q_2 \xRightarrow{\alpha \setminus \beta} Q_8$,
 $Q_6 \approx Q_8$.
 - By Step (3), $Q_3 \approx Q_1$, and by weak bis., $Q_1 \xRightarrow{\beta \setminus \alpha} Q_7$,
 $Q_5 \approx Q_7$.

So we have $Q_8 \approx Q_6 \approx Q_5 \approx Q_7$ as required.

2. Suppose $Q \xrightarrow{\tau} Q_3 \xrightarrow{\alpha} Q_1$ and $Q \xrightarrow{\beta} Q_2$.

- By Step (3), $Q \approx Q_3$, and by weak bis., $Q_3 \xrightarrow{\beta} Q_5$, $Q_2 \approx Q_5$.
- By ind. hyp., $Q_1 \xrightarrow{\beta \setminus \alpha} Q_6$, $Q_5 \xrightarrow{\alpha \setminus \beta} Q_7$, and $Q_6 \approx Q_7$.
- By weak bis., $Q_2 \xrightarrow{\alpha \setminus \beta} Q_4$ and $Q_4 \approx Q_7$.

So $Q_4 \approx Q_7 \approx Q_6$ as required.

Summary on confluence

1. We have 3 alternative characterisations of confluence.
2. A *confluent* process is always τ -*inert* and *determinate*.
3. *Restricted parallel composition* preserves confluence.
4. A *reactive* and *locally confluent* process is *confluent*.

A typing approach to determinacy:
the case of the π -calculus

Confluence in CCS with value passing

Consider the process P

$$P = a(b).\bar{a}b$$

- It seems reasonable to regard P as *determinate*.
- However, according to a straightforward extension of the concept of confluence to CCS with values, P is *not confluent*.
- Possible relaxation: do not require confluence for distinct input actions with the same subject.

Confluence in the π -calculus

- Consider

$$P = \nu a (\bar{b}a \mid \bar{c}a)$$

- Again, a straightforward definition of confluence would lead us to conclude that P is *not* confluent.
- One has to take into account the fact that an output may free names bound in another output action.

This is a bit *ad hoc* and does not scale up very well.

Two approaches to determinacy

As a property of the lts

- Define *determinacy/confluence* at the level of the lts.
- Show that certain constructions preserve confluence of the lts.

Typing processes

- Define *well-typed processes*. Note that the typing rules provide a way to build processes.
- Restrict the attention to interactions with the environment that *respect the typing constraints*.
- Require a form of *typed equivalence* (as opposed to a type-free one).

Contrasting the two approaches (semi-formally)

Determinacy with respect to the lts A process P is *determinate* if:

$$\frac{P \xrightarrow{s} P' \quad P \xrightarrow{s} P''}{P' \approx P''}$$

Determinacy with respect to a type system A typable process P is *determinate* if

$$\frac{C \text{ static context} \quad C[P] \text{ typable} \quad C[P] \xrightarrow{\tau} P' \quad C[P] \xrightarrow{\tau} P''}{P' \approx_{\text{typed}} P''}$$

Example

- Suppose the typing system guarantees *point-to-point* interaction and consider:

$$P = a(x).a(y).\bar{b}x \mid \bar{a}3$$

- With respect to the first definition, P fails to be deterministic as:

$$P \xrightarrow{a2} \bar{b}2 \quad P \xrightarrow{a2} \bar{b}3$$

- However, if the environment plays by the typing rules it should *not* send a message on a .
- In other words $C[P]$, where $C = [] \mid \bar{a}2$ fails to be typable.

Exercise

In the context of CCS, suppose we decide that a ‘typable static context’ C is defined as follows:

$$C ::= [] \mid \ell.C$$

Say that a process P is *context-deterministic* (c-deterministic for short) if it satisfies:

$$\frac{C[P] \xrightarrow{\tau} P_1 \quad C[P] \xrightarrow{\tau} P_2}{P_1 \approx P_2}$$

1. Show that if P is c-deterministic and $P \xrightarrow{\tau} P'$ then $P \approx P'$.
2. Show that if P is c-deterministic and $P \xrightarrow{\alpha} P'$ or $P \approx P'$ then P' is c-deterministic.
3. Show that if P is c-deterministic then it is deterministic.
4. What’s wrong with the definition of c-determinacy?

From sorting to affine resource usage

Initial goal Generalise the *sorting* system so that we keep the invariant that on every channel, at any time, at most one process can send and at most one process can receive.

Refining the goal: an example of process we want to type

$$P = \nu b \bar{a}b.Prod(in, b)$$

$$Prod(in, b) = in(c).\bar{b}c.Prod(in, b)$$

$$Q = a(b).Cons(b, out)$$

$$Cons(b, out) = b(c).\overline{out}c.Cons(b, out)$$

Process P exports towards process Q a channel ‘ b ’ that will be used in output by $Prod$ and in input by $Cons$.

Affine channel usage

- Let $L = \{0, 1\}$ with a partial addition operation such that

$$x \oplus 0 = 0 \oplus x = x$$

and $1 \oplus 1$ is *undefined*. Notation: in the following, $(X \oplus Y) \downarrow$ means that the sum is defined.

- A channel usage u is an element of L^2 . At any time, a channel with usage (i, j) can be used by at most i processes to send and at most j processes to receive.
- The addition operation \oplus is extended to L^2 componentwise.
- Subtraction and inequality are derived:

$$x \ominus y = z \quad \text{if } x = y \oplus z .$$

$$x \geq y \quad \text{if } \exists z (x = y \oplus z) .$$

- We annotate every channel type constructor Ch with a usage. Thus the (monadic, simple) types are:

$$\sigma ::= o \mid Ch_u(\sigma)$$

- The addition operation \oplus is extended to types so that:

$$o \oplus o = o, \quad Ch_{u_1}(\sigma) \oplus Ch_{u_2}(\sigma) = Ch_{u_1 \oplus u_2}(\sigma)$$

(the sum being undefined otherwise; note that we add usages at top level only).

- Similarly, for subtraction and inequality.

Type judgements

- As usual, a *typing context* has the shape:

$$\Gamma = a_1 : \sigma_1, \dots, a : \sigma_n$$

- The sum of contexts $\Gamma_1 \oplus \Gamma_2$ is defined if

$$a : \sigma_1 \in \Gamma_1, a : \sigma_2 \in \Gamma_2 \quad \Rightarrow \quad (\sigma_1 \oplus \sigma_2) \downarrow$$

- Type judgement for channel names:

$$\frac{\sigma \geq \sigma'}{\Gamma, a : \sigma \vdash a : \sigma'}$$

Type judgements (continued)

- Consider the following processes:

$$P ::= 0 \mid a(b).P \mid \bar{a}b.P \mid \nu a : \sigma P \mid (P \mid P) \mid A(\mathbf{a})$$

We assume that generated names and process identifiers carry a type annotation ($\nu a : \sigma$ and $A : (\sigma_1, \dots, \sigma_n)$, respectively).

- The type judgement for processes is:

$$\Gamma \vdash P$$

- The typing rules are:

$$\frac{}{\Gamma \vdash 0}$$

$$\frac{\Gamma_i \vdash P_i \quad i = 1, 2}{(\Gamma_1 \oplus \Gamma_2) \vdash (P_1 \mid P_2)}$$

$$\frac{\Gamma, a : \sigma \vdash P}{\Gamma \vdash \nu a : \sigma P}$$

$$\frac{A : (\sigma_1, \dots, \sigma_n) \quad \Gamma_i \vdash b_i : \sigma_i, i = 1, \dots, n}{(\Gamma_1 \oplus \dots \oplus \Gamma_n) \vdash A(b_1, \dots, b_n)}$$

$$\frac{\Gamma \vdash a : Ch_u(\sigma) \quad \pi_2(u) = 1 \quad \Gamma, b : \sigma \vdash P}{\Gamma \vdash a(b).P}$$

$$\frac{\Gamma_1 \vdash a : Ch_u(\sigma) \quad \pi_1(u) = 1 \quad \Gamma_2 \vdash b : \sigma \quad \Gamma_1 \vdash P}{(\Gamma_1 \oplus \Gamma_2) \vdash \bar{a}b.P}$$

Typing the motivating example

$$P = \nu b: Ch_{(1,1)}(o) \bar{a}b.Prod(in, b)$$

$$Prod(in, b) = in(c).\bar{b}c.Prod(in, b)$$

$$Q = a(b).Cons(b, out)$$

$$Cons(b, out) = b(c).\overline{out}c.Cons(b, out)$$

$$\Gamma_P = a : Ch_{(1,0)}(Ch_{(0,1)}(o)), in : Ch_{(0,1)}(o)$$

$$\Gamma_Q = a : Ch_{(0,1)}(Ch_{(0,1)}(o)), out : Ch_{(1,0)}(o)$$

$$Prod : (Ch_{(0,1)}(o), Ch_{(1,0)}(o))$$

$$Cons : (Ch_{(0,1)}(o), Ch_{(1,0)}(o))$$

Remarks

- We can split the usages of a channel (input/output).
- When we send, we consume the usage of the name we send.
- When we receive, we assume the usage of the name we receive.
- It is crucial that a name which is sent is received at most once, otherwise the affinity information is lost.

Exercise

Suppose we have a situation where we import a channel on which *Prod* and *Cons* will start interacting:

$$\nu b : Ch_{(1,1)}(o) \bar{a}b.0 \mid a(b).(Prod(in, b) \mid Cons(b, out))$$

Is there Γ such that $\Gamma \vdash P$?

Exercise

Suppose we emit *twice* a fresh name b as in the following process:

$$P = \nu b : Ch_{(1,1)}(o) \bar{a}b.0 \mid \bar{a}'b.0 .$$

Is there a Γ such that $\Gamma \vdash P$?

Exercise

Suppose $\Gamma \vdash P$ and $P \xrightarrow{\tau} P_i$ for $i = 1, 2$ and suppose that the two τ transitions are generated by two distinct synchronisations.

- Explain why the transitions do not ‘superpose’ and conclude that either $P_1 = P_2$ or $\exists Q (P_1 \xrightarrow{\tau} Q, P_2 \xrightarrow{\tau} Q)$.
- Is this enough to conclude that typable processes are confluent with respect to τ transitions?

Formal properties (outline)

A list of definitions and properties to show that *typable processes* enjoy a *strong confluence* property with respect to *typable transitions*.

1. Weakening.
2. Substitution.
3. Actions compatible with a given context.
4. Typed transitions.
5. Subject reduction.
6. Typed equivalence and strong confluence of typed transitions.

Weakening

$$\frac{\Gamma \vdash P \quad (\Gamma \oplus \Gamma') \downarrow}{(\Gamma \oplus \Gamma') \vdash P}$$

Exercise Prove this by induction on $\Gamma \vdash P$. For instance, consider the case where $(\Gamma_1 \oplus \Gamma_2) \vdash \bar{a}b.P$.

Substitution

$$\frac{\Gamma, a : \sigma \vdash P \quad \Gamma' \vdash b : \sigma \quad (\Gamma \oplus \Gamma') \downarrow}{(\Gamma \oplus \Gamma') \vdash [b/a]P}$$

Exercise Prove this by induction on $\Gamma, a : \sigma \vdash P$. For instance, consider the case where $\Gamma, a : Ch_u(\sigma) \vdash a(c).P$.

Actions compatible with a given context

We define P_α as a ‘minimal’ environment that allows the action α to happen:

$$P_\alpha = \begin{cases} 0 & \text{if } \alpha = \tau \\ a(b).0 & \text{if } \alpha = \bar{a}b \text{ or } \alpha = \nu b\bar{a}b \\ \bar{a}b.0 & \text{if } \alpha = ab \end{cases}$$

Then we use this to define when an action is compatible with a given typing context:

$$(\Gamma, \alpha) \downarrow \quad \text{if } \exists \Gamma' \ (\Gamma' \vdash P_\alpha \text{ and } (\Gamma \oplus \Gamma') \downarrow)$$

Exercise

Suppose $\Gamma = a : Ch_{(1,0)}(Ch_{(0,1)}(o)), b : Ch_{(0,1)}(o)$. Prove or disprove the following:

(1) $(\Gamma, \bar{a}b) \downarrow$

(2) $(\Gamma, ac) \downarrow$.

Suppose now $\Gamma = a : Ch_{(0,1)}(Ch_{(0,1)}(o)), b : Ch_{(0,1)}(o)$. Prove or disprove the following:

(3) $(\Gamma, \bar{a}b) \downarrow$.

Typed transitions

We introduce a notion of ‘typed’ transition.

$$P \xrightarrow{\Gamma, \alpha} P' \text{ if } P \xrightarrow{\alpha} P', \quad \Gamma \vdash P, \text{ and } (\Gamma, \alpha) \downarrow$$

Subject reduction

Suppose $(\Gamma, \alpha) \downarrow$. We define the residual $\Gamma(\alpha)$ of Γ after the action α :

$$\Gamma(\alpha) = \begin{cases} \Gamma & \text{if } \alpha = \tau \\ \Gamma \oplus (b : \sigma) & \text{if } \alpha = ab, \Gamma \vdash a : Ch_u(\sigma) \\ \Gamma \ominus (b : \sigma) & \text{if } \alpha = \bar{a}b, \Gamma \vdash a : Ch_u(\sigma) \\ (\Gamma, b : \sigma') \ominus (b : \sigma) & \text{if } \alpha = \nu b : \sigma' \bar{a}b, \Gamma \vdash a : Ch_u(\sigma) \end{cases}$$

Subject reduction (continued)

Typing is preserved by typed transitions.

$$\frac{P \xrightarrow{\Gamma, \alpha} P'}{\Gamma(\alpha) \vdash P'}$$

Exercise Prove this by induction on $P \xrightarrow{\alpha} P'$. For instance, consider the case where $P \xrightarrow{\Gamma, \tau} P'$ by a synchronisation action.

Typed bisimulation

- We can define a notion of *typed equivalence* as a *family* of symmetric relations $\{S_\Gamma \mid \Gamma \text{ context}\}$ such that:
 1. $(P, Q) \in S_\Gamma$ implies that P and Q are typable in the context Γ , and
 2. a *typed* transition of P with respect to the context Γ is matched by a typed transition of Q in the usual way.
- With respect to this notion of typed equivalence we would like to show that $P \xrightarrow{\Gamma, \tau} Q$ implies $P \approx_\Gamma Q$.

NB We omit the formal development of this part because it is a bit technical and conceptually quite close to the one for the ‘type-free’ case.

Example

In general, one expects (type-free) bisimilarity to entail typed bisimilarity while the converse fails. For instance,

- Take $\Gamma = a : Ch_{(1,1)}(o)$ and $P = a \mid \bar{a}$.
- Then $\Gamma \vdash P$.
- Also $P \xrightarrow{\Gamma, \tau} 0$ and $\Gamma \vdash 0$.
- Moreover, $P \approx_{\Gamma} 0$ while $P \not\approx 0$.

Key commutation property

Typed transitions enjoy a *strong* confluence property.

$$\frac{P \xrightarrow{\Gamma, \tau} Q \quad P \xrightarrow{\Gamma, \alpha} P'}{\exists P' \left(P \xrightarrow{\Gamma(\alpha), \tau} P' \quad Q \xrightarrow{\Gamma, \alpha} P' \right)}$$

NB Strong confluence of τ transitions is a special case.

Summary: typing approach to confluence

1. Define a *type system* such that typed processes do not have ‘race conditions’.
2. Show that *typing is preserved by reduction* (needs to look at actions too):

$$\frac{P \xrightarrow{\Gamma, \alpha} P'}{\Gamma(\alpha) \vdash P'}$$

3. Define *typed equivalence* \approx_{Γ} .
4. Show that typed equivalence is *invariant under internal reduction* (needs to look at commutation of τ actions with the other actions):

$$\frac{P \xrightarrow{\Gamma, \tau} P'}{P \approx_{\Gamma} P'}$$

References and historical remarks

- The notions of determinacy and confluence presented are based on chapter 11 of:
Robin Milner. *Communication and Concurrency*,
Prentice-Hall, 1989.

- Amazingly, this book does not refer to Kahn networks which were introduced in:

Gilles Kahn. The semantics of a simple language for parallel programming, IFIP Conf. on Information Processing 74, North-Holland, 1974.

Incidentally, synchronous data flow languages such as LUSTRE can be regarded as a refinement of this model.

- A rather complete study of the notion of confluence in the more general framework of the π -calculus is in:

Anna Philippou, David Walker. On confluence in the pi-Calculus. ICALP 1997: 314-324. (See also Anna Philippou PhD thesis, University of Warwick 1996).
- This builds on the PhD thesis of Sanderson and Tofts (in Edinburgh in the early 90's) where notions of confluence for CCS with value passing were proposed.
- The presented generalisation of Newman's lemma is due to J. Groote, M. Sellink. Confluence for process verification. Theor. Comput. Sci. 170(1-2):47-81, 1996.

- The classical reference for Linear Logic is:

J.-Y. Girard. Linear Logic. Theoretical Computer Science, 50(1), 1987.

Several up-to-date tutorials on this topic are available (Curien, Danos-Di Cosmo,...)

- This work has influenced a number of works on the static analysis of programs which exploit the notion of linearity. An early example in the framework of the π -calculus is:

Naoki Kobayashi, Benjamin C. Pierce, David N. Turner. Linearity and the pi-calculus. ACM Transactions on Programming Languages and Systems (TOPLAS), 21(5), 1999.

- The type system discussed here is a bit different in that:
 - the usage of resources in the typing is affine (at most once) rather than linear (exactly once).
 - we maintain the invariant:

At any time at most one process can send (receive) on a channel.

rather than:

a channel is used at most once to send (receive).

(which is more in line with the sorting system we have previously considered).

- Beware that going from logic to programming some properties are lost. For instance, intuitionistic logic inspires ML type systems. However in ML, all types are inhabited and programs do not always normalise!

Exercise (revision)

Recall that in the context of CCS, the *traces* of a process P are defined as

$$tr(P) = \{s \in \mathcal{L}^* \mid P \xRightarrow{s} \cdot\}$$

We write $P =_{tr} Q$ if $tr(P) = tr(Q)$. Show that if P, Q, R are CCS processes and $P =_{tr} Q$ then $(P \mid R) =_{tr} (Q \mid R)$.

Exercise (revision)

Suppose P is a CCS process that is reactive and such that for every derivative Q of P we have:

$$\frac{Q \xrightarrow{\tau} Q_1 \quad Q \xrightarrow{\tau} Q_2}{Q_1 \approx Q_2}$$

Show that this implies that for every derivative Q of P we have:

$$\frac{Q \xRightarrow{\tau} Q_1 \quad \text{and} \quad Q \xRightarrow{\tau} Q_2}{\exists Q'_1, Q'_2 \ (Q_1 \xRightarrow{\tau} Q'_1, \ Q_2 \xRightarrow{\tau} Q'_2, \ \text{and} \ Q'_1 \approx Q'_2)}$$

Exercise (revision)

We consider a *linear* variant of the typing system.

A *type* is *neutral* if it is either o or $Ch_{(0,0)}(\sigma)$. Thus if σ is neutral and $\sigma \oplus \sigma'$ is defined then $\sigma \oplus \sigma' = \sigma'$.

A *context* is *neutral* if it contains only neutral types.

The rule for typing names is:

$$\frac{\Gamma \text{ neutral}}{\Gamma, a : \sigma \vdash a : \sigma}$$

The rules for (some of) the processes are:

$$\begin{array}{c}
 \frac{\Gamma \text{ neutral}}{\Gamma \vdash 0} \\
 \\
 \frac{\Gamma_1 \vdash a : Ch_{(0,1)}(\sigma) \quad \Gamma_2, b : \sigma \vdash P}{\Gamma_1 \oplus \Gamma_2 \vdash a(b).P} \quad \frac{\Gamma_i \vdash P_i \quad i = 1, 2}{(\Gamma_1 \oplus \Gamma_2) \vdash (P_1 \mid P_2)} \\
 \\
 \frac{\Gamma_1 \vdash a : Ch_{(1,0)}(\sigma) \quad \Gamma_2 \vdash b : \sigma \quad \Gamma_3 \vdash P}{(\Gamma_1 \oplus \Gamma_2 \oplus \Gamma_3) \vdash \bar{a}b.P}
 \end{array}$$

For this system, prove the following versions of weakening and substitution:

$$\frac{\Gamma \vdash P \quad \Gamma' \text{ neutral} \quad (\Gamma \oplus \Gamma') \downarrow}{(\Gamma \oplus \Gamma') \vdash P} \quad \frac{\Gamma, a : \sigma \vdash P \quad \Gamma' \vdash b : \sigma \quad (\Gamma \oplus \Gamma') \downarrow}{(\Gamma \oplus \Gamma') \vdash [b/a]P}$$

What's wrong with the weakening rule for the affine system?

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 π -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 $out : Q \rightarrow M^n$.
 - A *next state* function $next : Q \times M^m \rightarrow Q$.
- 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, \dots, x_m in the incoming edges and places itself in the state $next(q, x_1, \dots, x_m)$
- 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 $\mathbf{Z}_n = \{0, 1, \dots, n - 1\}$ are arranged in a *ring*. Process i receives from process $(i - 1) \bmod n$ and sends to process $(i + 1) \bmod n$.

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 $(i_{max} + r) \bmod n$ 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 $\alpha : \mathcal{A} \rightarrow \mathbf{Z}$ 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 α 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 α 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 α 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:

$P \times 0$ is equivalent to 0

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

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

Example: product of actions

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

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

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

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

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

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*

$1 : a : 0$ 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 .

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

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.

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

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):

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

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 δ , and asynchroniser Δ .

Meije operators asynchronous composition \parallel , 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 $S(a, b) = \bar{a} : S(a, b) + \bar{b} : S(a, b)$.

- We have actually built a λ -term $F : Pr \rightarrow Pr \rightarrow Pr$:

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

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 λ -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 δ 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 δP is strongly bisimilar to $A(\mathbf{a})$.

Representing SCCS in Meije

- Let $T_\alpha = \alpha : T_\alpha$ for α action.
- We want to define $+$, \times , δ , Δ from \parallel , $(a \Rightarrow -)$, and $(a * -)$.
- Any guesses?

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

$$P \times Q = \nu a((a * P) \parallel (a * Q) \parallel T_{\bar{a}a})$$

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

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

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

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

Exercise

Check that strong bisimulation holds.

Representing Meije in SCCS

- We want to define \parallel , $(a \Rightarrow -)$, and $(a * -)$ from $+$, \times , δ , and Δ .
- 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: $S(a, b) = \bar{a} : S(a, b) + \bar{b} : S(a, b) + \overline{ab} : S(a, b)$.

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. Austrey 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.

A second generation model

The notion of instant reconsidered

- In SCCS, at each instant, each thread performs exactly 1 action.
- In the semi-formal model for synchronous algorithms, at each instant, each thread writes and reads exactly once in the point-to-point channels.

A more liberal viewpoint At each instant, each thread performs an arbitrary (but hopefully finite) number of actions. The instant ends when each thread has either terminated its task for the current instant or it is suspended waiting for events that cannot arise.

Timed CCS

A formalisation of this viewpoint in the framework of CCS.

$$\begin{aligned}\alpha & ::= \tau \mid \ell && \text{(usual actions)} \\ \mu & ::= \alpha \mid \text{tick} && \text{(extended actions)} \\ P & ::= \dots \mid (P \triangleright Q) && \text{(extended processes)}\end{aligned}$$

`tick` represents the move to the following instant.

$(P \triangleright Q)$ *else_next* operator: if cannot run P now, run Q at the following instant.

Labelled transition system

Usual rules for the α actions plus:

$$\frac{P \xrightarrow{\alpha} P'}{(P \triangleright Q) \xrightarrow{\alpha} P'}$$

Plus special rules for the tick action that say that a process can tick if and only if it cannot perform τ actions.

$$\frac{P \not\rightarrow \cdot}{(P \triangleright Q) \xrightarrow{\text{tick}} Q} \quad \frac{}{0 \xrightarrow{\text{tick}} 0}$$

$$\frac{}{\ell.P \xrightarrow{\text{tick}} \ell.P} \quad \frac{P_i \xrightarrow{\text{tick}} P'_i \quad i = 1, 2 \quad (P_1 \mid P_2) \not\rightarrow \cdot}{(P_1 \mid P_2) \xrightarrow{\text{tick}} (P'_1 \mid P'_2)}$$

$$\frac{P_i \xrightarrow{\text{tick}} P'_i \quad i = 1, 2}{(P_1 + P_2) \xrightarrow{\text{tick}} (P'_1 + P'_2)} \quad \frac{P \xrightarrow{\text{tick}} P'}{\nu a P \xrightarrow{\text{tick}} \nu a P'}$$

Exercise (on formalising tick actions)

1. Check that $P \xrightarrow{\text{tick}} \cdot$ if and only if $P \not\rightarrow \cdot$. (Indeed, this is in perfect agreement with the usual feeling that you do not see time passing when you have something to do)
2. The previous lts uses the negative condition $P \not\rightarrow \cdot$. Show that this condition can be formalised in a positive way by defining a formal system to derive judgements of the shape $P \downarrow L$ where L is a set of observable actions and $P \downarrow L$ if and only if $P \not\rightarrow \cdot$ and $L = \{\ell \mid P \xrightarrow{\ell} \cdot\}$.

Exercise (continuations of tick action)

We say that P is a ‘CCS process’ if it does not contain the *else_next* operator. Show that:

1. If $P \xrightarrow{\text{tick}} Q_1$ and $P \xrightarrow{\text{tick}} Q_2$ then $Q_1 = Q_2$.
2. If P is a CCS process and $P \xrightarrow{\text{tick}} Q$ then $P = Q$.

Exercise (programming a switch)

Let $\text{tick} .P = (0 \triangleright P)$ and $\text{tick}^n .P = \text{tick} \cdots \text{tick} .P$, n times.

1. Program a *light switch*

Switch(*press*, *off*, *on*, *brighter*)

that behaves as follows:

- Initially the switch is off.
- If the switch is off and it is pressed then the light turns on.
- If the switch is pressed again in the following 2 instants then the light becomes brighter while if it is pressed at a later instant it turns off again.
- If the light is brighter and the switch is pressed then it becomes off.

2. Program a *fast user* $Fast(\textit{press})$ that presses the switch every 2 instants and a *slow user* $Slow(\textit{press})$ that presses the switch every 4 instants.

3. Consider the systems:

$$\nu\textit{press} (\textit{Switch}(\textit{press}, \textit{off}, \textit{on}, \textit{brighter}) \mid \textit{Fast}(\textit{press}))$$
$$\nu\textit{press} (\textit{Switch}(\textit{press}, \textit{off}, \textit{on}, \textit{brighter}) \mid \textit{Slow}(\textit{press}))$$

and determine when the light is going to be off, on, and bright.

Exercise (bisimulation for TCCS)

Let \approx_{tick} be the notion of weak bisimulation where as expected

$$\xRightarrow{\text{tick}} = \xRightarrow{\tau} \circ \xrightarrow{\text{tick}} \circ \xRightarrow{\tau} .$$

1. Show that \approx_{tick} is preserved by parallel composition.
2. Show that $((P_1 \triangleright P_2) \triangleright P_3) \approx_{\text{tick}} (P_1 \triangleright P_3)$.

Exercise (more on congruence of \approx_{tick})

Suppose $P_i \approx_{\text{tick}} Q_i$ for $i = 1, 2$. Can we conclude that:

1. $P_1 + Q_1 \approx_{\text{tick}} P_2 + Q_2$?
2. $((\ell.P_1) \triangleright Q_1) \approx_{\text{tick}} ((\ell.P_2) \triangleright Q_2)$?
3. $(P_1 \triangleright Q_1) \approx_{\text{tick}} (P_2 \triangleright Q_2)$?

Exercise (CCS vs. TCSS)

Recall that a process is *reactive* if every derivative strongly normalizes with respect to τ reduction. Let Ω be the always diverging process $\tau.\tau.\tau \cdots$.

1. Is $0 \approx_{\text{tick}} \Omega$?
2. Suppose P, Q are CCS processes. Does $P \approx_{\text{tick}} Q$ imply $P \approx Q$?
3. Suppose further that P, Q are reactive. Does $P \approx Q$ imply $P \approx_{\text{tick}} Q$?

Exercise

We write $P \downarrow$ if $P \not\stackrel{\tau}{\rightarrow} \cdot$ and $P \Downarrow$ if $P \stackrel{\tau}{\Rightarrow} Q$ and $Q \downarrow$.

Show that on CCS processes the bisimulation \approx_{tick} can be characterised as the largest relation \mathcal{R} which is a weak labelled bisimulation (in the usual CCS sense) and such that if $P \mathcal{R} Q$ and $P \downarrow$ then $Q \Downarrow$.

References

- A notion of ‘timed’ CCS is first introduced in
Yi Wang. A calculus of real time systems. PhD thesis.
1991.

This calculus has a tick (x) operator that describes the passage of x time units where x is a non-negative real.

- A kind of *else_next* operator is proposed in
Nicolin, Sifakis. The algebra of timed processes.
Information and Computation, 1994.
- A *testing* semantics of a process calculus very close to the one presented here is given in
Hennessy, Reagan. A process algebra of timed systems.
Information and Computation, 1995.

However, it seems fair to say that all these works generalise to CCS ideas that were presented in

Berry, Cosserat. The Esterel synchronous programming language and its mathematical semantics. INRIA report, 1988.

Two basic differences in the Esterel approach are that:

1. Threads interact through *signals*.
2. The resulting calculus is *deterministic*.

Next, we will take sometime to discuss this approach.

Another popular formalism for describing timed systems is

Alur, Dill. A theory of timed automata. Theoretical Computer Science, 1994.

- This is an enrichment of finite state automata with timing constraints which still enjoys decidable model-checking properties.
- It is more a *specification language* for finite control systems than a *programming language*.
- The *implementability* of certain specifications is actually *problematic*.
- For instance, threads can perform atomic read-and-reset operations on clocks and this leads to unnatural race conditions.

Signal based interaction and determinacy

SL model

- Threads interact through *signals* (rather than channels).
- A signal is either emitted or not. Once it is emitted it *persists* during the instant and it is *reset* at the end of it.
- Thus the collection of emitted signals grows *monotonically* during each instant.

A simple calculus for the SL model

We present the calculus as a fragment of timed CCS. Write s, s', \dots for *signal* names.

Processes $P ::= 0 \mid s.P, P \mid (\text{emit } s) \mid (P \mid P) \mid \nu s P \mid A(\mathbf{s})$

where:

$$s.P, Q = (s.P \triangleright Q)$$

$$(\text{emit } s) = (\bar{s}.Emit(s) \triangleright 0)$$

$$\text{where: } Emit(s) = (\bar{s}.Emit(s) \triangleright 0)$$

Remarks on the calculus

- There is no sum and no prefix for emission (cf. asynchronous π -calculus).
- The input is a specialised form of the input prefix and the $(_ \triangleright _)$ operator.
- The derived synchronisation rule is:

$$(\text{emit } s) \mid s.P, Q \xrightarrow{\tau} \xrightarrow{\tau} (\text{emit } s) \mid P$$

(the second τ is just recursion unfolding and we will ignore it in the following).

- Note that:

$$(\text{emit } s) \mid s.P_1, Q_1 \mid s.P_2, Q_2 \xrightarrow{\tau} (\text{emit } s) \mid P_1 \mid P_2$$

Coding tick and await

- The tick action can be expressed as

$$\text{tick } .P = \nu s \ s.0, P \quad s \notin \text{fn}(P)$$

- A persistent input (as in TCCS) is expressed as:

$$\text{await } s.P = A(\mathbf{s}), \quad \text{where } A(\mathbf{s}) = s.P, A(\mathbf{s})$$

where $\text{fn}(P) \cup \{s\} = \{\mathbf{s}\}$.

Coding an *if_then_else* on signals

- For every signal s , we can program a process $Echo(s, s_-, s_+)$ as follows:

$$Echo(s, s_-, s_+) = s.\text{tick} .Echo_+(s, s_-, s_+), Echo_-(s, s_-, s_+)$$

$$Echo_+(s, s_-, s_+) = (\text{emit } s_+) \mid Echo(s, s_-, s_+)$$

$$Echo_-(s, s_-, s_+) = (\text{emit } s_-) \mid Echo(s, s_-, s_+)$$

that tells us in the following instant whether the signal s was emitted or not.

- Then we can define an *if_then_else* as follows:

$$(\text{ite } s \ P \ Q) = s_+.P, 0 \mid s_-.Q, 0$$

Example: programming a NOR gate in SL

- Input signals: s_0, s_1 .
- Output signal: s .
- At instant $i + 1$, emit s iff neither s_0 nor s_1 were emitted at instant i .

$$N = N_0 \mid \text{Echo}(s_0) \mid \text{Echo}(s_1)$$

$$N_0 = \text{tick} .(\text{ite } s_0 \ N_0 \ (\text{ite } s_1 \ N_0 \ N_1))$$

$$N_1 = (\text{emit } s) \mid N_0$$

Remarks

- We can *program* the boolean function *NOR* rather than writing down its *truth table*.
- Several threads can *share the same signal*:

$$(\text{emit } s) \mid s.P_1, Q_1 \mid s.P_2, Q_2$$

- We can react to the *absence of a signal* at the end of the instant and therefore we can regard a signal as a *binary information*.

Exercise (programming the light switch in SL)

- Reprogram the light switch in SL.
- Compare with the solution based on TCCS.

(Very) Strong confluence

A basic property is:

$$\frac{P \xrightarrow{\tau} P_1 \quad P \xrightarrow{\tau} P_2}{P_1 = P_2 \text{ or } \exists Q (P_1 \xrightarrow{\tau} Q, P_2 \xrightarrow{\tau} Q)}$$

NB We close the diagram in *at most one step* and *up to α -renaming*.

Proof idea

- Internal reductions are due either to *unfolding* or to *synchronisation*.
- The only possibility for a *superposition* of the redexes is:

$$(\text{emit } s) \mid s.P_1, Q_1 \mid s.P_2, Q_2$$

- And we exploit the fact that emission is *persistent*.

Remarks on a compositional semantics for the SL model

- We have defined a bisimulation semantics \approx_{tick} for TCCS. This semantics can be applied to SL too.
- In SL one can expect additional equations to hold. For instance,

$s.(\text{emit } s), 0$ should be ‘equivalent’ to 0

(cf. asynchronous communication).

Exercise Check that the equation does not hold in the TCCS embedding.

- More generally, because SL is deterministic one can expect a collapse of the *bisimulation* semantics with a *trace* semantics.

SL with data types

- The language with *pure* signals is *deterministic*.
- There are reasonable extension to *(infinite) data domains*. In general, the resulting language becomes *non-deterministic*.
- Efficient *implementation model*.
- Embedded in many *programming environments*: C, C++, Scheme, ML.
- Significant *applications*: event-driven control, data flow, GUI, simulations, web services, multiplayer games.

Two references

- Boussinot. Reactive C: an extension of C to program reactive systems. *Soft. Practice and Experience*, 1991.
- Mandel-Pouzet. Reactive ML, a reactive extension to ML. In *Proc. ACM PPDP*, 2005.

Some historical remarks

- In the ESTEREL model it is actually possible to *react immediately* (rather than at the end of the instant) to the absence of a signal.
- This requires some semantic care, to avoid writing paradoxical programs such as:

$s.0, (\text{emit } s)$

which are supposed to emit s when s is not there (cf. stabilization problems in synchronous circuits).

- It also requires some clever *compilation techniques* to determine whether a signal is not emitted. Infact these techniques (so far!) are specific to *finite state models*.

- SL is a *relaxation* of the ESTEREL model where the *absence of a signal* can only be detected at the end of the instant.
- If we forget about *name generation*, then the SL model essentially defines a kind of *monotonic Mealy machine*. Monotonic in the sense that output signals can only depend *positively* on input signals (within the same instant).
- The monotonicity restriction allows to *avoid the paradoxical programs* (monotonic boolean equations do not have a least fixed point!).
- The SL model has a natural and *efficient implementation model* that works well for *general programs* (not just finite state machines).

- The ESTEREL/SL models were conceived in Sophia-Antipolis shortly after the *SCCS/Meije models* and in the *same research team*.
- In spite of this, there is *no* strong formal result on the possibility/impossibility of embedding one model into the other up to some reasonable equivalence (*e.g.*, SCCS vs. TCCS).

Some references

- G. Berry and G. Gonthier, *The Esterel synchronous programming language*. Science of computer programming, 1992.

It introduces an imperative language to program reactive systems. The language can be compiled to finite automata. The semantics allows to react immediately to the absence of a signal. Static analysis is required to avoid ‘causality problems’.
- F. Boussinot and R. De Simone, *The SL Synchronous Language*. IEEE Trans. on Software Engineering, 1996.

Relaxation of the Esterel model. It allows reaction to the absence of a signal only at the end of the instant.
- A., *The SL synchronous language, revisited*. *Journal of Logic and Algebraic Programming*, 2007.

A process calculus description of the SL model with pure signals.
- A., *A synchronous π -calculus*. *Information and Computation*, 2007.

The generalisation of the previous work to signals carrying data values.

Exercise (revision)

In the context of SCCS/Meije, we specify a ternary operator rr (round robin) by the rule:

$$\frac{P_0 \xrightarrow{\alpha} P'_0}{rr(P_0, P_1, P_2) \xrightarrow{\alpha} rr(P_1, P_2, P'_0)}$$

Show that the operator rr is definable as a SCCS/Meije process. This amounts to define a SCCS/Meije process $RR(P_0, P_1, P_2)$, parametric in P_0, P_1, P_2 , which is strongly bisimilar to $rr(P_0, P_1, P_2)$.

Exercise (revision)

We can embed SL terms in TCCS and then compare them using the weak bisimulation for TCCS. Prove or disprove:

1. $s.(s.P, Q), Q \approx_{\text{tick}} s.P, Q$.
2. $(\text{emit } s) \mid s.P, Q \approx_{\text{tick}} (\text{emit } s) \mid P$.

Reactivity

Motivations

- Reactivity is *necessary* in synchronous/timed systems (otherwise time cannot progress!).
- It is also *desirable* in asynchronous systems.
- As we have seen, it helps in proving *confluence* (hence *determinacy*).

- We seek simple *syntactic conditions* that guarantee reactivity.
- In practice, these syntactic conditions are more effective in synchronous/timed systems than in asynchronous ones:
 - In synchronous/timed programs each thread is supposed to accomplish some task at each instant. The programs are naturally *annotated* to exhibit termination of the instant.
 - In asynchronous/untimed programs proving reactivity may require sophisticated ‘global’ arguments.
E.g., measure on the states and the messages in transit in the network.

Multi-set order (reminder)

- Denote with $\mathcal{M}(A)$ the finite multi-sets on a set A .
- If $(A, >)$ is a strict order then $\mathcal{M}(A)$ inherits a strict order $>_{\mathcal{M}(A)}$ defined by:

$$\frac{X \subseteq M, \quad X \neq \emptyset, N = (M \setminus X) \cup Y, \quad \forall y \in Y \exists x \in X \ x > y}{M >_{\mathcal{M}(A)} N}$$

- E.g. $\{5, 3, 1, 1\} >_{\mathcal{M}(\mathbf{N})} \{4, 3, 3, 1\}$,
with $X = \{5, 1\}$ and $Y = \{4, 3\}$,
or, alternatively, $X = \{5, 3, 3, 1\}$ and $Y = \{4, 3, 3, 1\}$.
- **Fact:** The order $(A, >)$ is well-founded iff the order $(\mathcal{M}(A), >_{\mathcal{M}(A)})$ is well-founded.

Reactivity in (T)CCS: a simple static analysis

- We assume the instruction tick is *explicitly* used in the program.
- We compute an *over-approximation* of the *control flow* of the system of equations.
- We check that within an instant it is *not possible to loop* through a thread identifier.

Call graph

If P is a process then $Call(P)$ is an *over-approximation* of the set of process identifiers that P may possibly call within the current instant:

$$\begin{aligned} Call(P) &= \text{case } P \\ 0 &: \emptyset \\ \text{tick } .P &: \emptyset \\ B(\mathbf{a}) &: \{B\} \\ \ell.P &: Call(P) \\ (P \triangleright Q) &: Call(P) \\ \nu s P &: Call(P) \\ P_1 + P_2 &: Call(P_1) \cup Call(P_2) \\ P_1 \mid P_2 &: Call(P_1) \cup Call(P_2) \end{aligned}$$

Given a system of equations:

$$A_1(\mathbf{a}_1) = P_1$$

...

$$A_n(\mathbf{a}_1) = P_n$$

build a (directed) *call graph* with *nodes* $\{A_1, \dots, A_n\}$ and such that

$$(A_i, A_j) \text{ is an edge iff } A_j \in \text{Call}(P_i)$$

Proposition If the call graph has no loops then any process relying on the related system of equations is reactive.

Proof idea

- If the graph has no loops then we can define a *well-founded order* $>$ on thread identifiers such that $A > B$ whenever there is an edge from A to B in the call graph.
- A process is essentially a *multi-set* of threads:

$$\{P_1, \dots, P_n\}$$

- Whenever we perform an internal reduction either we *reduce the size* of the threads or we unfold a recursive equation $A_i(\mathbf{a}) = P_i$ and then we have:

$$Call(A_i) = \{A_i\} >_{mset} Call(P_i)$$

Exercise

Define a well-founded measure on processes that shows that all internal reductions terminate.

Refining the conditions: directions

There are two main classes of techniques for proving termination/reactivity.

Combinatorial For instance, polynomial interpretation, simplification ordering (Kruskal theorem).

Logical Reducibility candidates/Logical relations techniques, imported from proof theory.

NB Porting these techniques to process calculi is still work in progress.

References on Reactivity

- On the *combinatorial* approach:
 - Deng, Sangiorgi. Ensuring termination by typability.
- On the *logical* approach:
 - Sangiorgi. Termination of processes.
 - Yoshida *et al.* Strong normalisation in the π -calculus.
 - Boudol. Typing termination in a higher-order concurrent imperative language,

Exercise (revision)

Suppose (after some abstraction) we study recursive equations of the shape:

$$A(\mathbf{a}) = b_1 \dots b_h . (\overline{c_1} \mid \dots \mid \overline{c_k} \mid A_1(\mathbf{d}_1) \mid \dots \mid A_n(\mathbf{d}_j))$$

where $h, k, j \geq 0$. A *level* is a natural number denoted with n, m, \dots

Suppose we can statically associate with every channel and every thread identifier a level. In particular we write $a : n$ if the channel name a has level n and $A : n[n_1, \dots, n_k]$ if A is a thread identifier of level n which takes as parameters k channel names of level n_1, \dots, n_k , respectively.

Propose syntactic criteria that:

1. guarantee that systems of the shape $A_1(\mathbf{d}_1) \mid \cdots \mid A_n(\mathbf{d}_j)$ are reactive.
2. allow (under suitable conditions) *recursive* calls such as:

$$A(\dots) = \dots A(\dots)$$

or

$$A(\dots) = \dots B(\dots) \quad B(\dots) = \dots A(\dots)$$

Conclusion

- We have addressed two basic questions in process calculi:
 - What is *determinacy*?
 - What is *time*?
- To answer these questions, we have found useful to explore some related concepts such as *confluence*, *linear/affine typing*, and *reactivity*.
- We have strived for *simplicity* and tried to give a *working knowledge* of the concepts.
- Other properties of interest include, *e.g.*, *absence of deadlock*, and analysis of the *information flow*. Linear typing and reactivity play a key role in these properties too!