

Extensions of Basic Process Calculi

MPRI C-2-3- Concurrency - Lectures 5-8

2009-2010

Roberto M. Amadio

Université Paris Diderot (Paris 7)

Laboratoire Preuves, Programmes et Systèmes

Programme of these lectures

- The *first part of the course* has focused on a basic process calculus (CCS) representing at an abstract level systems of *asynchronous* processes interacting through *synchronous* channels (*rendez-vous*).
- Some things you (should) have learned:
 - The notion of *labelled transition system*.
 - How to *compare non-deterministic processes* (the trace-bisimulation spectrum).
 - How to *abstract internal behaviour* (τ -moves).
 - How processes *interact* and *compose* (CCS operators).
 - Some *algebraic laws*.
 - Equivalences vs. *modal-logic* specifications.
 - Some *proof techniques* for (bi-)simulation.

- This *second part of the course* will discuss a series of *extensions* of the basic model:
 1. The combination of *functions and processes*.
 2. An intermediate language: the *π -calculus*.
 3. The notion of *time* (or *synchrony*).
 4. The integration of *probability* and *non-determinism*.

$$\begin{array}{c}
 \lambda_{||} \quad \rightarrow \quad \pi \quad \supset \quad \text{CCS} \quad \subset \quad \text{Timed-CCS} \\
 \cap \\
 \text{Probabilistic-CCS}
 \end{array}$$

NB We keep *message-passing* as basic interaction mechanism.
 The following parts of the course will focus on *shared memory*.

Functions and Processes

Motivations and Approach

1. CCS provides a basic model of *communication* and *concurrency* while ignoring the mechanisms of *procedural* and *data abstraction* which are at the hearth of sequential programming (in this respect, CCS is close to Turing machines).
2. The (typed) λ -calculus provides a more adequate framework for sequential programming (cf. ML family). What about the *integration* of λ -calculus and CCS?
3. One standard approach, supported both by theory and by practice, is to take the λ -calculus as the *backbone* of the programming language and to add on top a few features for communication and concurrency.

A simpler framework?

- The resulting language provides a comfortable programming environment but one may question whether this is the *simplest model* one can hope for (Ockham's razor).
- It turns out that the superposition of the concepts of function and process leads to some *redundancy* and that it is possible to reduce to simpler languages.
- Specifically, we will start with a functional and concurrent language and then describe *two translations* to proper sub-languages.

Two translations (ideas)

small β -translation The usual β -reduction is decomposed into one *small β -reduction* where we only substitute a pointer to a function and by a certain number of *duplications* arising when the function is actually applied to some argument.

For instance:

$$\begin{aligned} & (\lambda x.x3)(\lambda y.y + 1) \\ & \rightarrow [\lambda y.y + 1/x](x3) \equiv (\lambda y.y + 1)3 \\ & \rightarrow [3/y](y + 1) \equiv 3 + 1 \end{aligned}$$

$$\begin{aligned} & \nu z ((\lambda x.\text{get}(x)3)z \mid z \Leftarrow \lambda y.y + 1) \\ & \rightarrow \nu z (\text{get}(z)3 \mid z \Leftarrow \lambda y.y + 1) \\ & \rightarrow \nu z ((\lambda y.y + 1)3 \mid z \Leftarrow \lambda y.y + 1) \\ & \rightarrow \nu z ((3 + 1) \mid z \Leftarrow \lambda y.y + 1) \end{aligned}$$

CPS translation A *continuation passing style* which aims at making the evaluation context *trivial* by making it an explicit parameter.

The expression:

$$(M_1 M_2) M_3$$

becomes an expression parametric in a continuation k :

$$\begin{array}{l} \nu k_1, k_2, k_{1,2}, k_3 \\ k_1(f_1). \\ k_2(f_2). \\ k_{1,2}(f_{1,2}). \\ k_3(f_3). \end{array} \quad \begin{array}{l} \overline{M_1} k_1 \mid \\ (\overline{M_2} k_2 \mid \\ \text{get}(f_1)(f_2)(k_{1,2}) \mid \\ \overline{M_3} k_3 \mid \\ \text{get}(f_{1,2})(f_3)(k))))) \end{array}$$

The result of the evaluation of the expression M is ‘sent’ on the continuation k . In the translated terms the ‘redex’ is always at top level, *i.e.* the *evaluation context is trivial*.

An intermediate language: the π -calculus

- The composition of the two (simple) translations leads to the π -calculus which is a (non-trivial) extension of CCS where interacting processes exchange channel names.
- On one hand, the π -calculus can be regarded as an *intermediate language* of the kind used in compilation; as such it has an expressive power comparable (up to some encoding!) to the one of modern programming languages.
- On the other hand, the π -calculus inherits from CCS a relatively simple and tractable theory (a labelled transition system, bisimulation proof methods, equational axiomatisations,...)



Syntax (ingredients)

- We start with a *simply typed, call-by-value λ -calculus* regarded as the core of a functional language such as ML.
- We add a mechanism to generate *fresh names* and to associate values with them thus forming “*stores*” (*e.g.* think of references in ML).
- Of course, we also need to add a mechanism to *fetch values* from the store.
- Finally, we have several terms (threads) *running in parallel* thus allowing for a *concurrent access* (potentially non-deterministic!) to the store.

Syntax (types)

- We distinguish between types of terms that may return a value and a distinct type \mathbf{B} of the terms that just act by side-effect on the store.

$$A ::= \mathbf{1} \mid (A \rightarrow \alpha) \mid Ch(A) \quad (\text{value types})$$

$$\alpha ::= A \mid \mathbf{B} \quad (\text{types})$$

- $\mathbf{1}$ is the *terminal type* inhabited by the value $*$ (the unit type in ML).
- $A \rightarrow \alpha$ is the type of functions taking a value of type A and producing a computation of type α .
- $Ch(A)$ is the type of *channels* carrying values of type A .

Syntax (terms)

Let id be the syntactic category of identifiers denoted with x, y, \dots

$$V ::= id \mid * \mid \lambda id.M \quad (\text{values})$$

$$M ::= V \mid MM \mid \nu id M \mid \text{get}(id) \mid \\ \text{set}(id, V) \mid \text{pset}(id, V) \mid (M \mid M) \quad (\text{terms})$$

$$S ::= id \leftarrow V \mid id \Leftarrow V \mid (S \mid S) \quad (\text{stores})$$

$$P ::= M \mid S \mid (P \mid P) \mid \nu id P \quad (\text{programs})$$

Example (intended reduction)

Reduction is defined on *programs*.

$$\nu f (f \Leftarrow \lambda x.x(x^*) \mid (\lambda z.\text{set}(g, z))(\underline{\text{get}(f)}(\lambda w.w)))$$

$$\rightarrow \nu f (f \Leftarrow \lambda x.x(x^*) \mid (\lambda z.\text{set}(g, z))(\underline{(\lambda x.x(x^*))(\lambda w.w)}))$$

$$\xrightarrow{*} \nu f (f \Leftarrow \lambda x.x(x^*) \mid \underline{(\lambda z.\text{set}(g, z))^*})$$

$$\xrightarrow{*} \nu f (f \Leftarrow \lambda x.x(x^*) \mid * \mid g \leftarrow *) \not\rightarrow$$

Reduction rules (ingredients)

Structural equivalence Terms which should have identical behaviour.

Evaluation contexts What can be on the stack.

Reduction rules The basic instructions of the evaluator.

NB The reduction rules just describe the *internal behaviour* of a system as opposed to labelled transitions that describe the possible *interactions* with the environment too. Sometimes it is possible to recover the labels from the reduction rules...

Example: call-by-value λ -calculus

Structural equivalence α -renaming.

Evaluation contexts $E ::= [] \mid EM \mid VE$

Decomposition A well typed closed term is either a value or has a unique decomposition as $E[\Delta]$ where Δ has the shape $(\lambda x.M)V$.

Reduction rules $E[(\lambda x.M)V] \rightarrow E[[V/x]M]$

Example: fragment of CCS

Consider the processes P specified by:

$$P ::= 0 \mid id.P \mid \overline{id}.P \mid \nu id P \mid (P \mid P)$$

Structural equivalence

1. α -renaming.
2. parallel composition is associative and commutative.
3. $\nu x P \mid Q \equiv \nu x (P \mid Q)$ if x not free in Q .

Evaluation contexts $E ::= [\]$.

Decomposition Then every term can be seen as

$$\nu x_1, \dots, x_n (\alpha_1.P_1 \mid \dots \mid \alpha_n.P_n \mid 0 \mid \dots \mid 0)$$

Reduction rule A *static context* C is specified by:

$$C ::= [] \mid (C \mid P) \mid \nu id C$$

The following reduction rule is applied *modulo structural equivalence* and in a *static context*.

$$x.P \mid \bar{x}.Q \rightarrow P \mid Q$$

Our functional-concurrent language

As in CCS, the static contexts are composed of parallel composition and name generation.

Structural equivalence We define a structural equivalence \equiv as the least equivalence relation on *programs* such that:

1. it contains α -renaming,
2. parallel composition is associative and commutative,
3. $\nu x P \mid P' \equiv \nu x P \mid P'$ if $x \notin FV(P')$,
4. $E[\nu x M] \equiv \nu x E[M]$ if x is not free in E .

The last condition is due to the fact that evaluation contexts are not trivial as in CCS.

Evaluation contexts As in call-by-value λ -calculus:

$$E ::= [] \mid EM \mid VE \mid$$

Decomposition A *well-typed program* (to be defined) whose only free variables are channel names is structurally equivalent to:

$$\nu x_1, \dots, x_n (\dots \mid V_i \mid \dots \mid S_j \mid \dots \mid E_k[\Delta_k] \mid \dots)$$

where Δ_k is a *redex* of the shape $(\lambda x.M)V$, $\text{get}(x)$, $\text{set}(x, V)$, or $\text{pset}(x, V)$.

Reduction Rules Reduction rules apply *up to structural equivalence* and in a *static context*.

$$E[(\lambda x.M)V] \rightarrow E[[V/x]M]$$

$$E[\text{set}(x, V)] \rightarrow E[*] \mid x \leftarrow V \quad E[\text{pset}(x, V)] \rightarrow E[*] \mid x \Leftarrow V$$

$$E[\text{get}(x)] \mid x \leftarrow V \rightarrow E[V] \quad E[\text{get}(x)] \mid x \Leftarrow V \rightarrow E[V] \mid x \Leftarrow V$$

Example (structural equivalence)

$$\begin{aligned} & (\nu x V)(\nu x' V') \\ & \equiv \nu x V(\nu x' V') \\ & \equiv \nu x, x' VV' \end{aligned}$$

Name generations are lifted to allow values to interact.

Typing rules (1/2)

A typing context Γ has the shape $x_1 : A_1, \dots, x_n : A_n$ where all x_i are distinct.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{}{\Gamma \vdash * : \mathbf{1}}$$

$$\frac{\Gamma, x : A \vdash M : \alpha}{\Gamma \vdash \lambda x.M : A \rightarrow \alpha}$$

$$\frac{\Gamma \vdash M : A \rightarrow \alpha \quad \Gamma \vdash N : A}{\Gamma \vdash MN : \alpha}$$

$$\frac{\Gamma \vdash x : Ch(A)}{\Gamma \vdash \text{get}(x) : A}$$

$$\frac{\Gamma, x : Ch(A) \vdash P : \alpha}{\Gamma \vdash \nu x P : \alpha}$$

Typing rules (2/2)

$$\begin{array}{c}
 \frac{\Gamma \vdash x : Ch(A) \quad \Gamma \vdash V : A}{\Gamma \vdash \text{set}(x, V) : 1} \qquad \frac{\Gamma \vdash x : Ch(A) \quad \Gamma \vdash V : A}{\Gamma \vdash \text{pset}(x, V) : 1} \\
 \\
 \frac{\Gamma \vdash x : Ch(A) \quad \Gamma \vdash V : A}{\Gamma \vdash x \leftarrow V : \mathbf{B}} \qquad \frac{\Gamma \vdash x : Ch(A) \quad \Gamma \vdash V : A}{\Gamma \vdash x \Leftarrow V : \mathbf{B}} \\
 \\
 \frac{\Gamma \vdash P : \alpha \quad \Gamma \vdash S : \mathbf{B}}{\Gamma \vdash (P \mid S) : \alpha} \qquad \frac{\Gamma \vdash P_i : \alpha_i \quad P_i \text{ not stores}}{\Gamma \vdash P_1 \mid P_2 : \mathbf{B}}
 \end{array}$$

NB As usual typing rules are applied modulo α -renaming and the symmetric rule for $S \mid P$ is omitted.

Note that a program in parallel with a store has the program's type while two programs (which are not stores) in parallel have type \mathbf{B} .

Exercise (typing and decomposition)

1. Type the following term:

$$\nu f (f \Leftarrow \lambda x.x(x*) \mid (\lambda z.\text{set}(g, z))(\text{get}(f)(\lambda w.w)))$$

2. Prove the decomposition property for well typed terms whose free variables have a channel type.

Basic properties

Weakening If $\Gamma \vdash P : \alpha$ then $\Gamma, \Gamma' \vdash P : \alpha$.

Type invariance If $\Gamma \vdash P : \alpha$ and $P \equiv P'$ then $\Gamma \vdash P' : \alpha$.

Substitution If $\Gamma, x : A \vdash P : \alpha$ and $\Gamma \vdash V : A$ then
 $\Gamma \vdash [V/x]P : \alpha$.

Subject reduction If $\Gamma \vdash P : \alpha$ and $P \rightarrow P'$ then $\Gamma \vdash P' : \alpha$.

Sequentialisation

Define

$$M; N \equiv (\lambda x.N)M \quad x \notin FV(N)$$

Derived reduction First evaluate N , if N becomes a value (terminates) then evaluate M .

Derived typing

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : \alpha}{\Gamma \vdash M; N : \alpha}$$

Input prefix

Define

$$\begin{aligned} 0 &= * \mid * \\ \bar{x}V &= \text{set}(x, V) \\ x(y).M &= (\lambda y.M)(\text{get}(x)) \end{aligned}$$

Derived reduction

$$x(y).M \mid \bar{x}V \xrightarrow{*} [V/y]M \mid * \quad (\sim [V/y]M)$$

Derived typing

$$\frac{\Gamma \vdash x : Ch(A) \quad \Gamma, y : A \vdash M : \alpha}{\Gamma \vdash x(y).M : \alpha} \qquad \frac{\Gamma \vdash x : Ch(A) \quad \Gamma \vdash V : A}{\Gamma \vdash \bar{x}V : \mathbf{1}}$$

Internal choice

Define

$$M_1 \oplus M_2 \equiv \nu x (\bar{x}(\lambda z.M_1); \bar{x}(\lambda x.M_2); x(f).x(g).f*) \quad x \notin FV(M_1 M_2)$$

Derived reduction

$$M_1 \oplus M_2 \xrightarrow{*} M_i$$

Derived typing

$$\frac{\Gamma \vdash M_i : \alpha}{\Gamma \vdash M_1 \oplus M_2 : \alpha}$$

External choice

Define

$$\begin{aligned} x_1(y_1).M_1 + x_2(y_2).M_2 &\equiv \nu \ell (\bar{\ell}\text{true} | \\ &\quad x_1(y_1).\ell(b).\text{if } b \text{ then } (M_1 | \bar{\ell}\text{false}) \\ &\quad \quad \text{else } \bar{x}_1y_1; 0 \\ &\quad x_2(y_2).\ell(b).\text{if } b \text{ then } (M_2 | \bar{\ell}\text{false}) \\ &\quad \quad \text{else } \bar{x}_2y_2; 0) \end{aligned}$$

Exercise

1. Code booleans and if-then-else in the λ -calculus.
2. Determine the derived reduction and typing rules.

Input and output prefix

Define

$$\overline{x}V.M \quad \equiv \quad (\lambda f.fV \mid M)(\text{get}(x))$$

$$x(y).M \quad \equiv \quad \text{set}(x, \lambda y.M)$$

NB Polarity inversion: rather than *transmitting the value*, one *receives its continuation*.

Exercise (1) Find the derived typing and reduction rules.

(2) Type and reduce $M \mid N$ where:

$$M \quad = \quad \overline{x_1}V. \quad \nu x_3 \overline{x_2}x_3. \quad x_3(x_4). \quad M'$$

$$N \quad = \quad x_1(y_1). \quad x_2(y_2). \quad \overline{y_2}(y_1). \quad N'$$

Divergence

Even though terms are simply typed, divergence is still possible.

Define

$$\Omega = \nu x \text{ pset}(x, \lambda y. \text{get}(x)y); \text{get}(x)*$$

Exercise Type Ω and show that there is an infinite reduction starting from Ω .

Summary functional-concurrent language

- *All values are transmissible* (including channels and functions; in these cases it is unclear what a label could be).
- We have given *reduction rules* and a *type system* which is preserved by reduction; *structural equivalence* plays an important role here (as opposed to usual sequential languages).
- A number of *control operations* have a rather direct representation in the language.

Small β -translation

Type translation (small β)

- We transform the functional types into channels types:

$$\underline{A \rightarrow \alpha} = Ch(\underline{A} \rightarrow \underline{\alpha})$$

- The rest of the translation is ‘homomorphic’:

$$\underline{\mathbf{1}} = \mathbf{1} \quad \underline{\mathbf{B}} = \mathbf{B} \quad \underline{Ch(A)} = Ch(\underline{A})$$

- The types in the codomain of the translation are generated by the following grammar:

$$\alpha ::= A \mid \mathbf{B}$$

$$A ::= \mathbf{1} \mid Ch(A) \mid Ch(A \rightarrow \alpha)$$

Some syntactic sugar

Before defining the translation on the terms, it is convenient to introduce some syntactic sugar.

$$\text{let } x = \lambda y.M \text{ in } N \quad = \nu x \text{ pset}(x, \lambda y.M); N \quad (\text{definition})$$

$$@ (M, N) \quad = ((\lambda x. \lambda y. \text{get}(x)y)M)N \quad (\text{application})$$

Term translation (small β)

The two important cases:

$$\underline{\lambda x.M} = \text{let } y = \lambda x.\underline{M} \text{ in } y$$

$$\underline{MN} = @(\underline{M}, \underline{N})$$

The rest is ‘homomorphic’:

$$\underline{x} = x, \quad \underline{*} = *, \quad \underline{\text{get}(x)} = \text{get}(x)$$

$$\underline{x \leftarrow V} = x \leftarrow V \quad \text{if } V = * \text{ or } V = x$$

$$\underline{x \leftarrow \lambda y.M} = \text{let } z = \lambda y.\underline{M} \text{ in } x \leftarrow z \quad (*)$$

$$\underline{\nu x P} = \nu x \underline{P}, \quad \underline{P \mid P'} = \underline{P} \mid \underline{P'}$$

(*) Similar, for **set**, **pset**, and persistent stores.

Type preservation (small β)

Define

$$\underline{x_1 : A_1, \dots, x_n : A_n} = x_1 : \underline{A_1}, \dots, x_n : \underline{A_n}$$

Proposition If $\Gamma \vdash M : \alpha$ then $\underline{\Gamma} \vdash \underline{M} : \underline{\alpha}$.

The target language (small- β)

It is worth spelling out the target language which is going to be the domain of the second (CPS) translation:

$$V ::= * \mid id$$

$$M ::= V \mid \text{let } id = \lambda id.M \text{ in } M \mid @(M, M) \mid \\ \nu id M \mid \text{get}(id) \mid \text{set}(id, V) \mid \text{pset}(id, V) \mid (M \mid M)$$

$$S ::= id \leftarrow V \mid id \Leftarrow V \mid (S \mid S)$$

$$P ::= M \mid S \mid (P \mid P) \mid \nu id P$$

A difficulty in the simulation proof

- We would like to show (at least):

$$\frac{P \rightarrow P'}{\underline{P} \xrightarrow{*} Q \quad Q \text{ 'equivalent' to } \underline{P}'}$$

- Consider $P = (V_1 V_2)$, where:

$$V_1 = \lambda x_1. x_1(x_1^*) \quad V_2 = \lambda x_2. x_2$$

- A little computation shows that:

$$\underline{P} \xrightarrow{*} Q \equiv \nu y_1, y_2 (y_1 \Leftarrow \lambda x_1. \underline{x_1(x_1^*)} \mid y_2 \Leftarrow \lambda x_2. x_2 \mid \underline{y_2(y_2^*)})$$

- On the other hand,

$$P \rightarrow [V_2/x_1](x_1(x_1*))$$

and

$$\underline{P'} = \underline{[V_2/x_1](x_1(x_1*))} = \underline{[V_2/x_1]x_1(x_1*)}$$

is not quite the same as Q .

- For instance, the translation of V_2 is duplicated in $\underline{P'}$.

Read-back translation (small- β)

- In order to relate the reductions of a term and its translations it is useful to introduce a *read-back translation*:

$$\begin{aligned} (@(M, N))^o &= M^o N^o \\ (\text{let } x = \lambda y.M \text{ in } N)^o &= [\lambda y.M^o / x]N^o \end{aligned}$$

The translation acts as an homomorphism elsewhere.

- One can check that $\underline{M}^o = M$.

- The idea is to define a binary relation \mathcal{R} between terms in the source and target language as follows:

$$M\mathcal{R}N \quad \text{if } M \equiv N^o$$

- Then one shows that

$$\frac{M \rightarrow M' \quad M\mathcal{R}N}{N \xrightarrow{*} N' \quad M'\mathcal{R}N'}$$

Exercise

We focus on the purely functional fragment of the *source language*:

$$M ::= id \mid \lambda id.M \mid MM$$

Then the related *target language* of the small- β translation is:

$$M ::= id \mid \text{let } id = \lambda id.M \text{ in } M \mid @(M, M)$$

We can make explicit the computation rules of the target language.

- The *evaluation contexts* of the target language are:

$$E ::= [] \mid \text{let } x = \lambda y.M \text{ in } E \mid @(E, M) \mid @(id, E)$$

- The *structural equivalences* are:

$$E[@(\text{let } x = \lambda y.M_1 \text{ in } M_2, N)] \equiv E[\text{let } x = \lambda y.M_1 \text{ in } @(M_2, N)]$$

provided $x \notin FV(N)$, and

$$E[@(x, \text{let } x' = \lambda y.N_1 \text{ in } N_2)] \equiv E[\text{let } x' = \lambda y.N_1 \text{ in } @(x, N_2)]$$

provided $x \neq x'$.

- The *reduction rule* is:

$$E[\text{let } x = \lambda y.M \text{ in } E' [@(x, z)]] \rightarrow E[\text{let } x = \lambda y.M \text{ in } E' [[z/y]M]]$$

if x is not bound in E' .

Using this explicit presentation of the target language, prove in detail the *simulation result*:

$$\frac{M \rightarrow M' \quad M = N^o}{N \xrightarrow{*} N' \quad M' = N'^o}$$

A good starting point is to characterise the situations where $M = N^o$.

- If $M \equiv x$ then

$$N \equiv \begin{array}{l} \text{let } x_1 = \lambda y_1.N_1 \text{ in } \dots \\ \quad \text{let } x_n = \lambda y_n.N_n \text{ in } \dots \\ \quad x \end{array}$$

where $x \neq x_i$ for $i = 1, \dots, n$.

- If $M \equiv \lambda x.M'$ then

$$N \equiv \text{let } x_1 = \lambda y_1.N_1 \text{ in } \dots \\ \text{let } x_n = \lambda y_n.N_n \text{ in } \dots \\ x_i$$

where $1 \leq i \leq n$ and M' can be obtained as a suitable readback of $N_i \dots$

- If $M \equiv M'M''$ then

$$N \equiv \text{let } x_1 = \lambda y_1.N_1 \text{ in } \dots \\ \text{let } x_n = \lambda y_n.N_n \text{ in } \dots \\ @(N, N')$$

and M' and M'' can be obtained as suitable readbacks of N' and $N'' \dots$

CPS translation

Type translation (CPS)

- We recall the shape of the types in the target language of the first translation:

$$\alpha ::= A \mid \mathbf{B}$$

$$A ::= \mathbf{1} \mid Ch(A) \mid Ch(A \rightarrow \alpha)$$

- The idea is to transform an expression of type A into a function that takes a ‘channel’ k carrying values of type A as argument and producing a behaviour where the value of the expression is stored in k .

$$\overline{A} = Ch(A) \rightarrow \mathbf{B}$$

Term translation (CPS)

The main cases of the term translations are as follows:

$$\overline{\text{let } x = \lambda y.M \text{ in } N} = \lambda k.\text{let } x = \lambda y.\lambda k'.\overline{M}k' \text{ in } \overline{N}k$$

$$\begin{aligned} \overline{@(M, N)} &= \lambda k.\nu k_1, k_2 (\overline{M}k_1 \mid \\ &\quad k_1(f).\overline{N}k_2 \mid \\ &\quad k_2(x).((\text{get}(f)x)k)) \end{aligned}$$

where $k(x).M$ stands for $(\lambda x.M)\text{get}(k)$.

NB When we fetch a function from the store we always apply it to an argument *and* to a continuation.

It is convenient to have a specialised notation *à la CCS* that allows to get rid of the *administrative* reduction steps.

The target language (CPS)

$$V ::= id \mid *$$

$$M ::= \overline{id}(V, \dots, V) \mid id(id, \dots, id).M \mid !id(id, \dots, id).M \mid \nu id M \mid (M \mid M)$$

These terms can be regarded as abbreviations:

$$x(y_1, \dots, y_n).M = \text{set}(x, \lambda y_1, \dots, y_n.M); 0$$

$$!x(y_1, \dots, y_n).M = \text{pset}(x, \lambda y_1, \dots, y_n.M); 0$$

$$\overline{x}(V_1, \dots, V_n) = \text{get}(x)V_1 \dots V_n$$

This is a kind of restricted CCS (no output prefix) where channel names (and ground values) can be transmitted along channels.

Typing in the target language (CPS)

- All well-typed terms M have the *behaviour* type \mathbf{B} .
- We can introduce a special syntax to type the variables:

$$A ::= \mathbf{1} \mid Ch(A, \dots, A)$$

So we have either the terminal type or the type of a channel carrying n values.

- Then a typing judgment for a term M has the shape:

$$x_1 : A_1, \dots, x_n : A_n \vdash M$$

Exercise Write down the typing rules.

Reduction in the target language (CPS)

- *Static contexts* and *structural equivalence* are as in CCS.
- There are two *reductions rules*:

$$\bar{x}(V_1, \dots, V_n) \mid x(y_1, \dots, y_n).M \rightarrow [V_1/y_1, \dots, V_n/y_n]M$$

$$\bar{x}(V_1, \dots, V_n) \mid !x(y_1, \dots, y_n).M \rightarrow [V_1/y_1, \dots, V_n/y_n]M \mid !x(y_1, \dots, y_n).M$$

Exercise (programming in π)

The obtained language is both a restriction of CCS (no output prefix, no sum) and an extension (channel names can be transmitted along channel names).

Reprogramme in π :

- Internal choice.
- Booleans and If-then-else.
- External choice.
- Output prefix.

More programming tricks

- The *replication* $!(x(\mathbf{y}).M)$ can be regarded as a restricted form of *parametric recursive definition*:

$$A(x, \dots) = x(\mathbf{y}).(M \mid A(x, \dots))$$

- Conversely, a *recursive definition*

$$\text{letrec } A(\mathbf{x}) = M[A(\mathbf{y})] \text{ in } N[A(\mathbf{z})]$$

can be simulated by *replication*:

$$\nu A (!A(\mathbf{x}).M[\bar{A}(\mathbf{y})] \mid N[\bar{A}(\mathbf{z})])$$

- Finally, putting typing on the side, it is possible to simulate a *polyadic* synchronisation with a series of *monadic* ones.

$$\frac{\bar{x}(y_1, y_2) \quad \mid \quad x(z_1, z_2).M}{x(k).\bar{k}y_1 \mid x(k').\bar{k}'y_2 \quad \mid \quad \nu k \bar{x}k \mid k(z_1).\nu k' \bar{x}k' \mid k'(z_2).M}$$

Translation revisited (CPS)

We rewrite the CPS translation for the functional kernel of the let-language by making it parametric in a (fresh) channel k . So we write $\overline{M}k = \dots$ rather than $\overline{M} = \lambda k. \dots$

$$\overline{\text{let } x = \lambda y. M \text{ in } N}k = \nu x (!(x(y, k'). \overline{M}k') \mid \overline{N}k)$$

$$\begin{aligned} \overline{@(M, N)}k &= \nu k_1, k_2 (\overline{M}k_1 \mid \\ &\quad k_1(f).(\overline{N}k_2 \mid \\ &\quad k_2(x). \overline{f}(x, k)) \end{aligned}$$

$$\overline{x}k = \overline{k}x$$

Simulation (example)

- One can show:

$$\frac{M \rightarrow M'}{\overline{M}k \xrightarrow{*} N \quad N \text{ 'is equivalent to' } \overline{M'}k}$$

- For instance, suppose:

$$M = \text{let } y_1 = \lambda x_1.x_1 \text{ in let } y_2 = \lambda x_2.x_2 \text{ in } @(@(y_1, y_2), x_3)$$

- Then

$$M \rightarrow M' = \text{let } y_1 = \lambda x_1.x_1 \text{ in let } y_2 = \lambda x_2.x_2 \text{ in } @(y_2, x_3)$$

- A little computation shows that:

$$\overline{M}k \xrightarrow{*} \overline{M'}k$$

Composing the two translations (small- β +CPS)

If we define

$$\llbracket M \rrbracket = \overline{\llbracket M \rrbracket}$$

we get (for the purely functional fragment):

$$\begin{aligned}\llbracket x \rrbracket k &= \bar{k}x \\ \llbracket \lambda x.M \rrbracket k &= \nu f (!f(x, k'). \llbracket M \rrbracket k' \mid \bar{k}f) \\ \llbracket MN \rrbracket k &= \nu k_1, k_2 (\llbracket M \rrbracket k_1 \mid \\ &\quad k_1(f). (\llbracket N \rrbracket k_2 \mid \\ &\quad k_2(x). \bar{f}(x, k)))\end{aligned}$$

A functional expression becomes a process sending (a pointer to) its value on a channel k .

Exercise

1. Determine the corresponding type translation.
2. State and prove directly the fact that the translation preserves the typing.

The situation for call-by-name

Suppose we look at a *call-by-name* λ -calculus.

Evaluation context

$$E ::= [] \mid EM$$

Reduction rule

$$(\lambda x.M)N \rightarrow [N/x]M$$

We can define a similar translation where $\llbracket \lambda x.M \rrbracket k$ is a process waiting on the channel k (a pointer to) its next argument:

$$\llbracket \lambda x.M \rrbracket k = k(x, k'). \llbracket M \rrbracket k'$$

$$\llbracket MN \rrbracket k = \nu k', x (!x(k''). \llbracket N \rrbracket k'' \mid \llbracket M \rrbracket k' \mid \bar{k}'(x, k))$$

$$\llbracket x \rrbracket k = \bar{x}k$$

Exercise

1. Find a type translation.
2. Verify that the translation is type preserving.
3. Compute $\llbracket (\lambda f.f)(\lambda y.y) \rrbracket k$.

Summary

- We have described an integration of *synchronisation primitives à la CCS* in a *programming language à la ML*.
- The resulting language can be compiled to an intermediate language called *π -calculus* that looks like CCS with name passing.
- The compilation can be decomposed in *two main steps*:
 1. Store functions and duplicate them only when they are actually applied to an argument (a kind of *call-by-need*).
 2. Use a *continuation passing translation* to make the evaluation contexts trivial.

References

- The **functional-concurrent language** described in this lecture is inspired by early versions of concurrent ML built on top of SML:
 - Facile: A symmetric integration of concurrent and functional programming, A. Giacalone, P. Mishra, S. Prasad, 1989.
 - CML: A higher concurrent language, J. Reppy, PhD 1992.
- The origin of the **small β translation** can be traced back to the problem of giving a formal representation of *lazy evaluation* where one has to account for *shared resources*. See, e.g.,
 - A natural semantics for lazy evaluation, J. Launchbury, 1993.

- Early studies on the **CPS translation** include:
 - Definitional interpreters for higher-order programming, J. Reynolds, 1972.
 - Call-by-name, call-by-value and the lambda-calculus, G. Plotkin, 1975.

It was later realised that in their *typed version* CPS translations are actually instances of *double negation translations* from classical to intuitionistic logic. Nowadays CPS translations are a standard tool in compilation.

- The π -calculus was introduced in:

A calculus of mobile processes, Parts I and II. R. Milner, J. Parrow, D. Walker, 1989.

following earlier work by Engberg and Nielsen. The expressive power of the π -calculus was soon realised and led to a number of encodings, notably of the *call-by-name* and *call-by-value* λ -calculus. See, *e.g.*,

Functions as processes, R. Milner, 1992.

- The **two steps translation** discussed in this lecture is inspired by

The π -calculus in direct style, G. Boudol, 1998

which focuses on call-by-name rather than on call-by-value.

π -calculus

Motivation

- In the previous lecture, we have isolated an extension of CCS, known as *π -calculus*, where processes exchange names when synchronising.
- The π -calculus can be seen as an *intermediate language* for a rather general functional-concurrent language of the ML family.

$$\lambda_{||} \quad \rightarrow \quad \pi$$

- Our purpose in this lecture is to show that the *bisimulation proof techniques* developed for CCS can actually be extended to the π -calculus.

$$\pi \quad \supset \quad \text{CCS}$$

- More generally, while there seems to be no simple translation from π to CCS, many results developed for CCS can be *lifted* to π .

What is the labelled transition system for the π -calculus?

In order to answer this question, we will proceed as follows:

1. We introduce a notion of *contextual bisimulation* whose definition does *not* depend on labels.
2. We introduce a notion of *labelled transition system* and a related notion of *labelled bisimulation*.
3. We show that the notions of contextual and labelled bisimulation *coincide* thus justifying *a posteriori* the definition of label.

The simple case first!

- The construction is *sensitive* to the choice of the initial language (the contexts of contextual bisimulation depend on the language...).
- To reduce the technical details we pick a simple version of the π -calculus which is *monadic*, *untyped*, and with *rendez-vous* communication.

$$id ::= x \mid y \mid \dots \quad (\text{names})$$

$$P ::= 0 \mid \overline{id}id.P \mid id(id).P \mid \\ (P \mid P) \mid \nu id P \mid [id = id]P \quad (\text{processes})$$

$$C ::= [] \mid (C \mid P) \mid \nu id P \quad (\text{static contexts})$$

- Note that:
 1. channel names have *no type*,
 2. in all communications exactly *one name* is exchanged,
 3. both the input and output communication actions have a *continuation*,
 4. names can be compared for *equality*.
- The reduction rules are up to the usual structural equivalence and in a static context:

$$x(y).P \mid \bar{x}z.P' \rightarrow [z/y]P \mid P' \quad [x = x]P \rightarrow P$$

If we forget about the names exchanged and the test for equality we are back to a fragment of CCS.

Commitments

We introduce some candidates for the notion of *basic observable*:

$P \downarrow x$ if the process P is ready to perform a visible communication action on channel x (and $P \Downarrow x$ if $P \Rightarrow Q$ and $Q \downarrow x$).

We may also distinguish the *polarity* of the communication (input or output).

$P \downarrow_{\exists}$ if $\exists x P \downarrow_x$ (and $P \Downarrow_{\exists}$ if $\exists x P \Downarrow_x$).

$P \downarrow$ if $P \not\rightarrow$ (and $P \Downarrow$ if $\exists P' P \Rightarrow P'$ and $P' \downarrow$).

Contextual bisimulation

A binary symmetric relation \mathcal{R} on processes is a *strong* contextual bisimulation if whenever $P \mathcal{R} Q$ the following conditions hold:

(cxt) For all static contexts C , $C[P] \mathcal{R} C[Q]$.

(red) If $P \rightarrow P'$ then for some Q' , $Q \rightarrow Q'$ and $P' \mathcal{R} Q'$.

(cmt) If $P \downarrow x$ then $Q \downarrow x$.

For the *weak* version replace \rightarrow by \Rightarrow and \downarrow by \Downarrow .

Denote with \sim_C (\approx_C) the largest contextual (weak) bisimulation.

Exercise (barbed bisimulation)

Consider the variant of contextual equivalence where we do *not* require that the relation is closed under static contexts.

Call *barbed bisimulation* the largest such equivalence and denote it with \sim_{BB} .

In the framework of CCS, show that barbed bisimulation is *not* preserved by parallel composition.

Exercise (barbed equivalence)

Say that two processes are *barbed equivalent* if put in any static context they are barbed bisimilar. Denote such equivalence with \sim_{BE} .

Show that if two processes are contextually bisimilar then they are barbed equivalent.

NB The converse can be quite tricky to prove. See, *e.g.*,

Fournet-Gonthier, A hierarchy of equivalences for asynchronous calculi, 1998.

The characterisation of labelled bisimulation we are aiming at is more direct/natural when working with *contextual bisimulation* than with *barbed equivalence*.

Exercise (variations on commitment)

1. In the framework of CCS, show that we get an equivalent notion of contextual bisimulation if the condition [cmt] is replaced by

$$P \Downarrow_{\exists} \text{ implies } Q \Downarrow_{\exists}$$

2. On the other hand, show that we get an incomparable notion of contextual bisimulation if the condition [cmt] is replaced by

$$P \Downarrow \text{ implies } Q \Downarrow$$

Exercise (labelled bisimulation is a contextual bisimulation)

Consider the notion of (strong or weak) labelled bisimulation for CCS.

Prove that the largest labelled bisimulation is a contextual bisimulation (both in the strong and weak case).

Contextual and labelled bisimulation coincide (for CCS)

- The previous exercise shows that labelled bisimulation is contained in contextual bisimulation.
- We show the converse by proving that the largest contextual bisimulation \sim_C (or \approx_C in the weak case) is a labelled (weak) bisimulation.
- We consider directly the *weak case*.
- Recall that *internal choice* can be defined as follows:

$$P \oplus Q = \nu x (x.P \mid x.Q \mid \bar{x}) = \tau.P + \tau.Q$$

- If $P \approx_C Q$ and $P \xrightarrow{\tau} P'$ then $Q \xrightarrow{\tau} Q'$ and $P' \approx_C Q'$, by condition [red] of contextual bisimulation.

- So suppose $P \xrightarrow{x} P'$.
- Take o_1, o_2 two distinct fresh names (not in P and Q) and define the static context

$$C = [] \mid \bar{x}.(o_1 \oplus (o_2 \oplus 0))$$

- By hypothesis, $C[P] \approx_C C[Q]$.
- Clearly, $P \xrightarrow{\tau} P' \mid (o_2 \oplus 0)$ and again by hypothesis (condition [red]) $C[Q] \xrightarrow{\tau} Q''$ and $P' \mid (o_2 \oplus 0) \approx_C Q''$.
- Now we argue that Q'' must be of the shape $Q' \mid (o_2 \oplus 0)$ where $Q \xrightarrow{x} Q'$.

- The case $Q'' = C[Q']$ and $Q \xrightarrow{\tau} Q'$ is impossible because $P' \Downarrow o_2$ entails $Q' \Downarrow o_2$, and the latter entails $Q' \Downarrow o_1$ which cannot be matched by P' .
- The cases $Q \xrightarrow{x} Q'$ and $Q'' = Q' \mid R'$ where $R' \in \{o_1 \oplus (o_2 \oplus 0), o_1, o_2, 0\}$ are also impossible for similar reasons.
- Thus we must have $Q'' = Q' \mid (o_2 \oplus 0)$ and $P' \mid (o_2 \oplus 0) \approx_C Q''$.
- It is easy to argue that since $P' \mid (o_2 \oplus 0) \xrightarrow{\tau} P' \mid 0$ we must have $Q' \mid (o_2 \oplus 0) \xrightarrow{\tau} Q_1 \mid 0$ and $Q_1 \mid 0 \approx_C P' \mid 0$.
- Thus $Q \xrightarrow{x} Q_1$ and $P' \equiv (P' \mid 0) \approx_C (Q_1 \mid 0) \equiv Q_1$. Strictly speaking, we use an *up to technique*.

Exercises (easy)

1. Do the previous proof in the strong case. Can you simplify the context C in this case?
2. Show that (weak) labelled bisimulation is a (weak) barbed bisimulation.
3. Show that (weak) labelled bisimulation is a (weak) barbed equivalence.

Summary

- Contextual bisimulation requires: reduction, static contexts, and commitments.
- Labelled bisimulation requires: labels, labelled transitions, labelled bisimulation.
- For CCS, choosing as commitment \Downarrow_x or \Downarrow_{\exists} , the two notions coincide.

What about the π -calculus?

Labelled bisimulation for π

Actions

We have *four types* of actions:

Action α	Example
τ	$\bar{x}y.P \mid x(z).Q \xrightarrow{\tau} P \mid [y/z]Q$
input xz	$x(y).P \xrightarrow{xz} [z/y]P$
output $\bar{x}y$	$\bar{x}y.P \xrightarrow{\bar{x}y} P$
bound output $\bar{x}(y)$	$\nu y \bar{x}y.P \xrightarrow{\bar{x}(y)} P$

NB The first three cases would also arise in an extension of CCS with ground values. The real novelty is the *bound output case*. Note that an effect of the action is to *free* the restricted name y .

Some tricky examples

The bound output action carries a bound name and one has to be careful to *avoid conflicts*.

- $\nu y \ x(y).P.$
- $\nu y \ \bar{x}y.P.$
- $\nu y \ \bar{x}y.P \mid x(z).Q.$
- $\nu y \ \bar{x}y.P \mid x(z).(y(w).Q).$

Free and bound names

- All occurrences of a name in an action are free except y in a bound output action $\bar{x}(y)$.
- Define $fn(\alpha)$ ($bn(\alpha)$) as the set of names occurring free (bound) in α .
- Let $n(\alpha) = fn(\alpha) \cup bn(\alpha)$.
- In processes, the formal parameter of an input and the ν bind names.
- Define $fn(P)$ as the name occurring free in a process P .

Labelled transition system

$$\begin{array}{c}
 \frac{}{x(y).P \xrightarrow{xz} [z/y]P} \qquad \frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P} \\
 \\
 \frac{P \xrightarrow{\alpha} P' \quad x \notin n(\alpha)}{\nu x P \xrightarrow{\alpha} \nu x P'} \qquad \frac{P \xrightarrow{\bar{y}x} P' \quad x \neq y}{\nu x P \xrightarrow{\bar{y}(x)} P'} \\
 \\
 \frac{P \xrightarrow{xy} P' \quad Q \xrightarrow{\bar{x}y} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \frac{P \xrightarrow{xy} P' \quad Q \xrightarrow{\bar{x}(y)} Q' \quad y \notin fn(P)}{P \mid Q \xrightarrow{\tau} \nu y (P' \mid Q')} \\
 \\
 \frac{P \xrightarrow{\alpha} P' \quad bn(\alpha) \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{}{[x = x]P \xrightarrow{\tau} P}
 \end{array}$$

NB Rules apply up to α -renaming and symmetric rules are omitted.

Exercise (on the lts for π)

Apply the definition to compute the labelled transitions of the following processes:

- $\nu y \ x(y).P.$
- $\nu y \ \bar{x}y.P.$
- $\nu y \ \bar{x}y.P \mid x(z).Q.$
- $\nu y \ \bar{x}y.P \mid x(z).(y(w).Q).$

Exercise (τ -transitions vs. reduction)

In the previous lecture we have defined a reduction relation \rightarrow on the π -calculus. The basic rules are:

$$\bar{x}y.P \mid x(z).Q \rightarrow (P \mid [y/z]Q) \quad [x = x]P \rightarrow P$$

and they apply up to structural equivalence and in a static context.

Show that reduction and τ -transitions are the same up to structural equivalence. Namely:

1. If $P \rightarrow Q$ then $P \xrightarrow{\tau} Q'$ and $Q \equiv Q'$.
2. If $P \xrightarrow{\tau} Q$ then $P \rightarrow Q'$ and $Q \equiv Q'$.

For instance, consider:

$$(\nu y \bar{x}y.P_1 \mid P_2) \mid x(z).P_3$$

Towards labelled bisimulation

- Suppose we want to show that P and Q are ‘labelled’ bisimilar.
- Further suppose $P \xrightarrow{\bar{x}(y)} P'$ makes a bound output.
- What is the condition on Q ?
- Note that bisimilar processes may have different sets of free names.
- For instance, suppose $P = \nu y \bar{x}y.P'$ and $Q = \nu y \bar{x}y.Q' \mid R(y)$.
- The transition $P \xrightarrow{\bar{x}(y)}$ cannot be matched (literally) by Q because y is free in Q .

Labelled bisimulation

- A symmetric binary relation \mathcal{R} on processes is a strong labelled bisimulation if

$$\frac{P \mathcal{R} Q, \quad P \xrightarrow{\alpha} Q, \quad bn(\alpha) \cap fn(Q) = \emptyset}{Q \xrightarrow{\alpha} Q' \quad P' \mathcal{R} Q'}$$

- For the weak case replace $\xrightarrow{\alpha}$ by $\xRightarrow{\alpha}$.
- Denote with \sim_L (\approx_L) the largest strong (weak) labelled bisimulation.

Labelled bisimulation is preserved by static contexts

- We want to show that strong (weak) labelled bisimulation is preserved by static contexts.
- We define a binary relation

$$\mathcal{R} = \{(\nu y P \mid R, \nu y Q \mid R \mid P \sim_L Q)\}$$

- We show that \mathcal{R} is a labelled bisimulation.
- First let us see what goes wrong with the relation one would define for CCS:

$$\mathcal{R}' = \{(P \mid R, Q \mid R) \mid P \sim_L Q\}$$

- We can have $P \mid R \xrightarrow{\tau} \nu y P' \mid R'$ because $P \xrightarrow{\bar{x}(y)} P'$ and $R \xrightarrow{xy} R'$.
- Then we just have $Q \mid R \xrightarrow{\tau} \nu y Q' \mid R'$ and P' is bisimilar to $Q' \dots$

- Let us look again at this case when working with the larger relation \mathcal{R} .
- Suppose $\nu \mathbf{y} P \mid R \xrightarrow{\tau} \nu y \nu \mathbf{y} P' \mid R'$ because $P \xrightarrow{\bar{x}(y)} P'$ and $R \xrightarrow{xy} R'$.
- Then $Q \xrightarrow{\bar{x}(y)} Q'$ and $P' \sim_L Q'$.
- Therefore $\nu \mathbf{y} Q \mid R \xrightarrow{\tau} \nu y \nu \mathbf{y} Q' \mid R'$ and now

$$\nu y \nu \mathbf{y} P' \mid R' \mathcal{R} \nu y \nu \mathbf{y} Q' \mid R'$$

NB It is possible to develop a little bit of ‘bisimulation-up-to-context’ techniques to get rid once and for all of these technicalities.

Exercise

- Complete the proof that labelled bisimulation is preserved by static contexts.
- Generalise the proof to the weak case.

Contextual bisimulation is a labelled bisimulation (π case)

- The notion of contextual bisimulation is immediately lifted to the π -calculus.
- Since labelled bisimulation is preserved by static contexts, we can easily conclude that the largest labelled bisimulation is a contextual bisimulation.
- We are left to show that the largest contextual bisimulation is a labelled bisimulation.

- The static contexts we have to build are now slightly more elaborate.

Action	Static Context
xy	$[] \mid \bar{x}y.o_1 \oplus (o_2 \oplus 0)$
$\bar{x}y$	$[] \mid x(z).[z = y](o_1 \oplus (o_2 \oplus 0))$
$\bar{x}(y)$	$[] \mid x(z).([z = y_1]w_1 \mid \cdots \mid [z = y_k]w_k \mid (o_1 \oplus (o_2 \oplus 0)))$

where y_1, \dots, y_k are the free names of the pair of processes under consideration and $w_1, \dots, w_k, o_1, o_2$ are fresh names.

Exercise

Complete the proof that the largest contextual bisimulation is a labelled bisimulation.

Variations on π

A number of variations of the π -calculus have been considered in the literature. We mention a few and discuss the impact on the presented construction (*i.e.*, contextual vs. labelled bisimulation).

Polyadic channels

- When allowing polyadic channels, one needs to make sure that in a synchronisation there are as many actual as formal parameters (some channel typing).
- In the formalisation of the labelled transition system, the formalisation of the actions is a bit more complicated as several names can be extruded as the result of a communication.
- The actions are:

$$\alpha ::= \tau \mid x(y_1, \dots, y_n) \mid \nu z_1, \dots, z_m \bar{x}(y_1, \dots, y_n)$$

with the requirement for the output that the z_i are distinct and belong to $\{y_1, \dots, y_n\}$.

Asynchronous communication

- In this language, an output action cannot prefix another action (the intermediate language we obtained in the first lecture has this property).
- In this case, a context cannot observe directly an input action. For instance:

$$x.\bar{x} \approx 0$$

- In the labelled case, the bisimulation game must be adapted:

$$\frac{P \approx Q \quad P \xrightarrow{x} P'}{(Q \xrightarrow{x} Q' \text{ and } P' \approx Q') \text{ or } (Q \xrightarrow{\tau} Q' \text{ and } P' \approx (Q' \mid \bar{x}))}$$

No name comparison

- In the asynchronous case, assume one is not allowed to compare names (again this was the case in the intermediate language).
- Further impose that on a received name one can only send messages.
- Then distinct names may have the same ‘observable behaviour’.
For instance:

$$\nu x (\bar{y}x \mid \bar{y}x \mid!(x.z)) \approx \nu x, x' (\bar{y}x \mid \bar{y}x' \mid!(x.z) \mid!(x'.z))$$

- Again the labelled transition system we have considered discriminates too much. What matters is not the name but the ‘service’ one can trigger with it.

Late lts

- It is tempting to define an input action as

$$x(y).P \xrightarrow{x} \lambda y.P$$

so that a process rewrites to a function from names to processes...

- ...and then go on to define a *bisimulation on functions* and use it to handle the input case:

$$\frac{P \sim_L Q \quad P \xrightarrow{x} F}{Q \xrightarrow{x} G \quad \forall b \ F(b) \sim_L G(b)}$$

- This gives a reasonable equivalence which is a bit *more discriminating* than the contextual one.

- Suppose $P \sim Q$.
- In the *late bisimulation* one requires:

$$\forall F \exists G \forall b \quad P \xrightarrow{x} F \supset \\ (Q \xrightarrow{x} G \wedge F(b) \sim G(b))$$

- While in the bisimulation we have studied (also called *early*) one requires:

$$\forall F \forall b \exists G \quad P \xrightarrow{x} F \supset \\ (Q \xrightarrow{x} G \wedge F(b) \sim G(b))$$

- For instance consider:

$$P = x(y).\text{if } y = 0 \text{ then } o_1 \text{ else } o_2$$

$$Q = x(y).\text{if } y = 0 \text{ then } o_1 + \\ x(y).\text{if } y \neq 0 \text{ then } o_2$$

- Then P and Q are early bisimilar but not late bisimilar.
- Contextual bisimulation (= early bisimulation) cannot test the function in isolation. It can only test the function applied to an argument.

Exercise Can you find a similar example in the syntax of the π -calculus?

Summary

- The notion of **contextual bisimulation** is built on the notions of *reduction*, *commitment* (basic observable), and *static context*.
- For **CCS**, the notion of *contextual bisimulation* coincides with the notion of *labelled bisimulation* studied in the first part of the course (modulo a suitable choice of the basic observable).
- We have seen that for (a variety of) the **π -calculus** it is possible to define a labelled transition system and a related notion of labelled bisimulation such that the latter coincides with contextual bisimulation.
- The definition of the labelled transition system and the labelled bisimulation for the π -calculus (and related calculi) is not trivial. Then the corresponding notion of contextual bisimulation acts as a **guiding principle**.

References

- As already mentioned the π -**calculus** was introduced in:

A calculus of mobile processes, Parts I and II. R. Milner, J. Parrow, D. Walker, 1989.

following earlier work by Engberg and Nielsen. This paper actually introduces the *late* labelled transition system and bisimulation.

- A few **books** have been written on the subject.

An introduction:

Communicating and Mobile Systems: the π -calculus. R. Milner. 1999.

A rather systematic treatment:

D. Sangiorgi, D. Walker, The π -calculus: a theory of mobile processes, 2001.

A revised treatment oriented towards the ‘asynchronous’ π -calculus and its ‘distributed’ versions:

M. Hennessy, The distributed π -calculus, 2007.

- The **asynchronous version** of the π -calculus was put forward independently in
 - K. Honda, M. Tokoro, A object calculus for asynchronous communication, 1991
 - G. Boudol, Asynchrony and the π -calculus, 1992.
- The notion of **contextual bisimulation** was studied in
 - K. Honda, N. Yoshida, On reduction-based process semantics, 1993.
- The notion of **asynchronous bisimulation** sketched here is introduced in
 - R.A, I. Castellani, D. Sangiorgi, On bisimulations for the asynchronous π -calculus, 1994.
- The earlier definition of **barbed equivalence** can be found in
 - R. Milner, D.Sangiorgi, Barbed bisimulation, 1992.

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

Consider the following fragment of SCCS:

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

Say that P is *fireable* if $P \xrightarrow{1} P$. Show that 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):

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

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 by 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. 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.

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 $P \xrightarrow{\tau} \cdot$ and $P \Downarrow$ if $P \xrightarrow{\tau} Q$ 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 extensions 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 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.

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

Probability and Non-determinism

Motivations

Some standard topics in the area of (deterministic) probabilistic systems.

- Analysis of probabilistic programs/protocols, *i.e.*, programs/protocols that toss coins at some point in the computation.
- Reliability evaluation: estimate probability of failure of certain components and determine overall system reliability.
- Performance evaluation: determine average waiting time.

In *concurrent systems*, *non-determinism* arises to account for *race conditions* and as a *specification device*.

It is then natural to *lift* methods for (deterministic) probabilistic systems to non-deterministic ones.

A short story: from the origins to our problem

- Let S be a *countable* set of *states*.
- A (*discrete*) *distribution* on the states S is a function $\Delta : S \rightarrow [0, 1]$ such that $\sum_{s \in S} \Delta(s) = 1$.
- We denote with $Dist(S)$ the collection of distributions on S .
- If $S' \subseteq S$ we denote with $\Delta[S']$ the sum $\sum_{s' \in S'} \Delta(s')$.
NB The value of the sums *do not* depend on the enumeration of the states.

- We may represent a distribution as a *formal sum*

$$\sum_{i \in I} [p_i] s_i$$

where $\sum_{i \in I} p_i = 1$

- The *binary* version is also written as

$$s_1 +_p s_2$$

which stands for $[p]s_1 + [1 - p]s_2$.

- The *unary* version is shortened to s which stands for $[1]s$.

Discrete Time Markov Chains (Markov 1906)

- A discrete notion of time $t = 0, 1, 2, \dots$
- A transition ‘matrix’

$$P : S \rightarrow \text{Dist}(S)$$

where $p(i)(j)$ is the probability that the system being at state i at time t moves into state j at time $t + 1$ (for any t).

A beautiful theory! Linear algebra for probability theorists...

Markov Decision Processes Elaboration on Markov chains produced in the 50's (Bellman 1957,...)

- Add a countable set of actions Act .
- Redefine the transition matrix as

$$P : (S \times Act) \rightarrow Dist(S)$$

$P(s, a)(s')$ is the *probability* of moving from s to s' given that the decision to perform action a has been taken.

- Possibly add a *reward function*

$$R : (S \times Act) \rightarrow [S \rightarrow \mathbf{R}]$$

$R(s, a)(s')$ is the *reward* the decision maker gets if being in s and taking the decision a the system moves into s' .

Typical problem what is the *policy* for the decision maker that maximizes some cumulative function of the rewards?

Probabilistic Labelled Transitions Systems Defined in the 80's (Vardi 1985,...)

- Key point: being in state s and taking the decision a the system may end up in different distributions of states.
- We now have a transition *relation*:

$$\rightarrow \subseteq S \times Act \times Dist(S)$$

- If all the distributions are trivial (Dirac) then we are back to *Labelled Transition Systems*.
- If for a given state and action there is only one distribution then we are back to *Markov Decision Processes*.

The focus is on probabilistic lts

- Our *modest goal* is to analyse how the process calculus framework adapts to the move from *lts* to *probabilistic lts*.
- In particular we look at:
 1. The definition of *strong bisimulation* on probabilistic lts.
 2. How to associate a probabilistic lts with a *Probabilistic CCS*, namely a CCS enriched with a probabilistic choice operator.
 3. How to compose transitions (notions of *weak transition* and *weak bisimulation* can be derived)

Warning

- Probabilistic lts are *widely accepted* as a way to combine probabilities and non-determinism.
- However, looking at the literature you may also find more *exotic definitions*.
- For instance, in so called *generative models* the probability distribution also includes a distribution over the possible actions:

$$\rightarrow \subseteq S \times \text{Dist}(\text{Act} \times S)$$

- It is *not* so clear what the computational meaning of the process $[0.5]a + [0.5]b$ is and how it composes in parallel with, say, $[0.3]\bar{a} + [0.7]\bar{b}$.

Lifting a relation on states to a relation on distributions

- Let $\mathcal{R} \subseteq S \times S$ be an equivalence relation on states.
- Let $[s]_{\mathcal{R}}$ denote the equivalence class of s .
- We lift \mathcal{R} to an equivalence relation $\mathcal{D}(\mathcal{R})$ on $Dist(S)$ as follows:

$$\Delta \mathcal{D}(\mathcal{R}) \Delta' \quad \text{if} \quad \forall s \in S \quad \Delta[[s]_{\mathcal{R}}] = \Delta'[[s]_{\mathcal{R}}]$$

Thus two distributions are *equivalent* with respect to \mathcal{R} if they are the same modulo the equivalence induced by \mathcal{R} .

This is also called *lumping equivalence* in Markov processes literature (lumping=aggregate).

Bisimulation

- Let (S, Act, \rightarrow) be a *probabilistic* lts.
- An equivalence relation \mathcal{R} over S is a *bisimulation* if:

$$\frac{\forall s, s', \alpha, \Delta \quad s \mathcal{R} s' \text{ and } s \xrightarrow{\alpha} \Delta}{\exists \Delta' \quad s' \xrightarrow{\alpha} \Delta' \text{ and } \Delta \mathcal{D}(R) \Delta'}$$

- We denote with \sim_P the largest bisimulation.

Example (bisimilar states)

The following states A and A' are bisimilar:

$$A = in.([0.8]B + [0.2]C) \quad A' = in.([0.8]B + [0.1]D + [0.1]E)$$

$$B = out.[1]A \quad D = err.[1]A$$

$$C = err.[1]A \quad E = err.[1]A$$

We may represent this system as a *bipartite graph* where

$$Nodes = States \cup Distributions$$

The edges from *States* to *Distributions* are labelled with (CCS) *actions* and the labels in the other direction with *probabilities*.

Example (non-bisimilar states)

The following states A and A' are *not* bisimilar.

$$\begin{array}{l|l} A = in.\Delta_1 + in.\Delta_2 & \Delta_1 = (B +_{0.9} C) \\ B = out.[1]A & \Delta_2 = (B +_{0.5} C) \\ C = err.[1]A & \\ A' = in.\Delta_1 + in.\Delta_2 + in.\Delta_3 & \Delta_3 = (B +_{0.7} C) \end{array}$$

One may argue that Δ_3 is a *convex combination* of Δ_1 and Δ_2 .
Indeed taking $\lambda = 0.5$:

$$\Delta_3 = \lambda \cdot \Delta_1 + (1 - \lambda) \cdot \Delta_2$$

There exists a more *relaxed* definition of bisimulation that takes this into account.

Example (vending machine)

Here is an *unreliable vending machine* with *slow* and *fast* users where we write P for $[1]P$.

$$VM = \text{coin}.(VM +_{0.1} VM')$$

$$VM' = ((\overline{\text{tea}}.VM + \overline{\text{coffee}}.VM) \triangleright VM)$$

$$SlowU = \overline{\text{coin}}.\text{tick}.\text{tea}.0$$

$$FastU = \overline{\text{coin}}.\text{tea}.0$$

Consider $(VM \mid SlowU)$ and $(VM \mid FastU)$.

Who may get the tea?

Example (communication protocol)

We introduce some TCCS notation:

$$a.P \triangleright_n Q = a.P \triangleright (a.P \triangleright \cdots (a.P \triangleright Q) \cdots)$$

We model a communication medium that may lose messages (but acknowledgments are never lost):

$$\begin{aligned} S &= \text{send}.S' && \text{(sender)} \\ S' &= \overline{\text{in}}.(ack.S \triangleright_2 S') \\ M &= \text{in}.(M +_{0.1} \text{tick} .(\overline{\text{out}}M)) && \text{(medium)} \\ R &= \text{out}.\overline{\text{rec}}.\overline{\text{ack}}.R && \text{(receiver)} \end{aligned}$$

The medium transmits at most 1 message/instant and whenever the message is lost 2 instants pass without any message being transmitted.

A formal language: PCCS

We distinguish *processes* (states) and (formal) *distributions*:

$$P ::= 0 \mid \sum_{i \in I} \alpha_i . \Delta_i \mid (P \mid P) \mid \nu a P \mid A(\mathbf{a}) \quad (\text{processes})$$

$$\Delta ::= \sum_{i \in I} [p_i] P_i \quad (\text{distributions})$$

Some notation for distributions

It is convenient to use the following abbreviations for distributions:

$$\nu a \Sigma_{i \in I} [p_i] P_i \quad \equiv \quad \Sigma_{i \in I} [p_i] \nu a P_i$$

$$\Sigma_{i \in I} [p_i] P_i \mid \Sigma_{j \in J} [q_j] Q_j \quad \equiv \quad \Sigma_{(i,j) \in I \times J} [p_i \cdot q_j] (P_i \mid Q_j)$$

Associated probabilistic lts

Then it is not difficult to write down the rules of the probabilistic lts.

$$\frac{}{\sum_{i \in I} \alpha_i \Delta_i \xrightarrow{\alpha_i} \Delta_i} \quad \frac{P \xrightarrow{a} \Delta \quad P' \xrightarrow{\bar{a}} \Delta'}{(P \mid P') \xrightarrow{\tau} (\Delta \mid \Delta')}$$

$$\frac{P \xrightarrow{\alpha} \Delta}{(P \mid P') \xrightarrow{\alpha} \Delta \mid [1]P'} \quad \frac{P \xrightarrow{\alpha} \Delta \quad a, \bar{a} \neq \alpha}{\nu a P \xrightarrow{\alpha} \nu a \Delta}$$

$$\frac{[\mathbf{b}/\mathbf{a}]P \xrightarrow{\alpha} \Delta}{A(\mathbf{b}) \xrightarrow{\alpha} \Delta} \quad A(\mathbf{a}) = P$$

How do we compose transitions?

- To define weak transitions and bisimulation we need to *compose* transitions.
- Suppose $\rightarrow \subseteq S \times Dist(S)$.
- Then $\mapsto \subseteq Dist(S) \times Dist(S)$ is defined as follows:

$$\begin{aligned} \Delta \mapsto \Delta' \quad \text{if} \quad & \Delta = \sum_{i \in I} [p_i] s_i \quad I \text{ finite} \\ & \forall i \in I \exists \Delta_i \quad s_i \rightarrow \Delta_i \\ & \Delta' = \sum_{i \in I} [p_i] \Delta_i \end{aligned}$$

where

$$\sum_{i \in I} [p_i] (\sum_{j \in J_i} [q_{i,j}] s_{i,j}) \equiv \sum_{i \in I, j \in J_i} [p_i \cdot q_{i,j}] s_{i,j}$$

- For instance, suppose:

$$P_1 \rightarrow P_{11} +_{p_1} P_{12} \quad P_2 \rightarrow P_{21} +_{p_2} P_{22}$$

Then

$$P_1 +_p P_2 \mapsto [pp_1]P_{11} + [p(1 - p_1)]P_{12} + \\ [(1 - p)p_2]P_{21} + [(1 - p)(1 - p_2)]P_{22}$$

Summary

- Probabilistic lts are from processes to distributions on processes.
- Do *not* confuse non-deterministic and probabilistic choice.
- Construction to *lift* a relation on processes to a relation on distributions (notion of strong bisimulation is derived).
- Construction to *compose* transitions (notions of weak transition and weak equivalence can be derived).

References

A reasonable **introduction** to the subject is:

B. Jonnson, K. Larsen W. Yi. Probabilistic extensions of process algebras. 2001.

The notion of **probabilistic lts** has been put forward in

M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. 1985.

under the name of *concurrent Markov chains* in the framework of model-checking and it has been later revisited by

Hansson and Jonnson. A calculus of communicating systems with time and probabilities. 1990.

in the framework of CCS (actually of TCCS).

The notion of **bisimulation** is introduced for Markov Decision Processes in:

K. Larsen and A. Skou. Bisimulation through probabilistic testing. 1989.

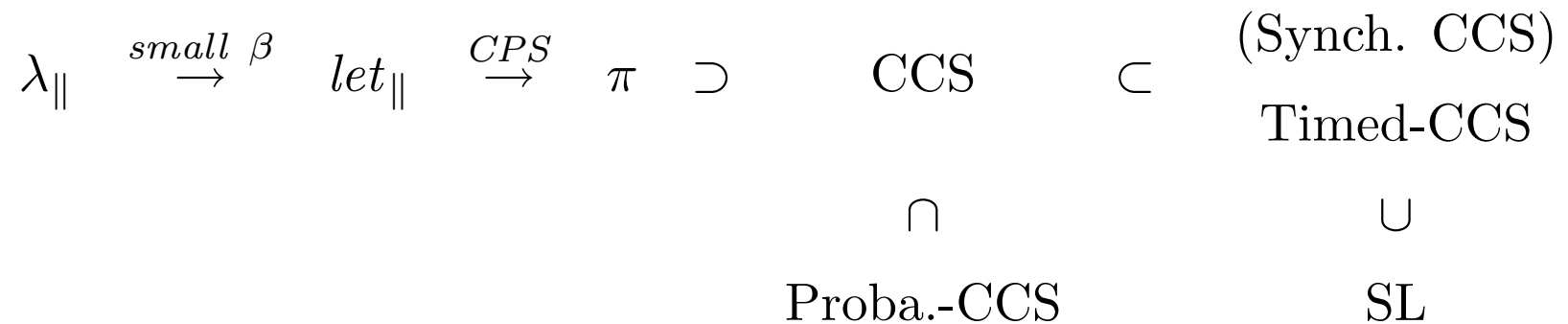
while the weaker notion of ‘convex’ bisimulation for probabilistic lts is introduced in:

R. Segala and N. Lynch. Probabilistic simulation for probabilistic processes. 1995.

The following book covers the situation where distributions are on **continuous state spaces** and requires a non-trivial amount of measure theory:

P. Panangaden. Labelled Markov Chains, 2009.

Summary of lectures 5-8: Extensions of CCS



These extensions have been considered in *isolation* for the sake of *simplicity*.

However they are rather *robust* and can be *combined*.

For instance, one could define a *probabilistic* and *timed π -calculus*.