A roundtrip from π-calculus to synchronous programming

Roberto Amadio

Université de Paris 7

Laboratoire Preuves, Programmes et Systèmes

Groupe de travail Concurrence: Chevaleret, Thursday 2 pm.

http://www.pps.jussieu.fr/~amadio/cc/

Introduction

> In truth, there is nothing canonical about our choice of basic combinators, 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.

> 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).

Op. cit.

Some references

- R. Milner, *Calculi for synchrony and asynchrony*, TCS, 25, 1983.
- D. Austry and G. Boudol, *Algèbre de processus et synchronisation*, TCS, 30, 1984.
- G. Berry and G. Gonthier, *The Esterel synchronous programming language*. Science of computer programming, 19, 1992.
- F. Boussinot and R. De Simone, *The SL Synchronous Language*. IEEE Trans. on Software Engineering, 22, 1996.
- R. Amadio, *The SL synchronous language, revisited*. Technical Report, Université Paris 7, Laboratoire PPS, November 2005.

NB There is a parallel thread of synchronous programming languages that is based on data flow ideas (Lustre, Signal, synchronous Kahn networks,...) that we will not be discussed here. Refer to the course on *Synchronous Programming*.

Goal of this course

- Introduce and revisit the *SL synchronous programming model* (a relaxation of the ESTEREL model).
- Apply the *mathematical framework* that has been developed for CCS/π-calculus to the SL model.
- Highlight a *research problem* arising when extending the model with data values and name mobility.

Some general ideas

Synchrony:

- Computation is regulated by a notion of *instant*.
- Threads can proceed at different speeds but periodically they go through a *global synchronisation*.

NB An instant can turn out to be considerably more complex than in Milner's description where "at every instant each agent performs a *single* action".

Reactivity:

- Each instant should *terminate* in reasonable time/space.
- If a thread loops the whole system loops!

Parallelism

- Parallelism might be useful to exploit a parallel machine.
- Parallelism is also useful to have a *modular programming style* (*e.g.* think of a graphical user interface).

Determinism

- Debugging/testing/verifying non-deterministic programs is difficult.
- Whenever possible try to work with *deterministic* programs. E.g. Kahn networks are an early example of deterministic and parallel model.

Cooperative concurrency

- In preemptive concurrency a thread can be *interrupted at any point*.
- In cooperative concurrency a thread must *explicitly return the control* to the scheduler.
- In a synchronous framework, it makes sense to adopt cooperative concurrency. This may help in making the program look more deterministic.

The SL model

- A *program* is a *multi-set of threads* interacting via *shared signals*.
- A *signal* is emitted or not, and once it is emitted it *persists* within the current instant.

Basic principle of the SL model

Reaction to the *absence* of a signal within an instant can only happen at the *next instant*.

- This is a relaxation of the ESTEREL model (Berry-Gonthier 1992) where the reaction to the absence can be *instantaneous* (SL designed to avoid ESTEREL's 'causality problem').

- For instance,

$$\nu s \text{ present } s \text{ then } 0 \text{ else } (\text{emit } s)$$

- How does one 'run' this statement?
- If terms of boolean equations, this is like

$$s = \neg s$$

  which has no solution...
- SL avoids the paradox by assuming that the branch else is only computed at the end of the instant, once the presence or absence of the signal s is settled.

Evolution

The SL model has evolved towards a *programming language* for concurrent applications:

- More *control operators*, *thread spawning*,...
- *Data types*.
- *Programming environments*: C, JAVA, SCHEME, ML,... (Boussinot et al. 95-05).
- *Multi-processing* (Boussinot 04).
- *Migration* (Boudol 04).

Philosophy

- Synchronous models like ESTEREL or LUSTRE are designed to allow a *static scheduling* and a compilation to *finite state machines*.
- The SL model:
  - more relaxed view on *efficiency*,
  - more *flexibility* in programming and compiling,
  - preserve *determinism* and *reactivity*.

Typical applications (for the basic model)

- Graphical user interfaces.
- Simulations.
- Web services.
- Network games.

A programming example (informal)

- Suppose you want to program a generic *cellular automaton*.
- Assign a *thread* to each *cell*.
- The parameters of each thread/cell are: its *current state*, its *signal*, the signals of its *neighbours*.

- Each thread performs the following actions periodically:
  1. It *emits* its current state in the signals of its neighbours.
  2. It *waits the end of the instant*.
  3. At the end of the instant it *collects* the states of its neighbours on its signal and it computes its new state.

Remarks

- Similar examples exist for the simulation of mechanical phenomena such as the n-body problem. See `http://www-sop.inria.fr/mimosa/rp` for demos.
- In these examples, signals carry *values* (*e.g.* the state of the cell).
- In the following, we will focus on a calculus where signals are *pure* (cf. CCS).
- In the end, we will consider the problems that arise when this hypothesis is dropped and we move to a kind of *synchronous* π-calculus.

The SL model (revisited)

Signals and Environments

Signal names $S = \{s, s', \ldots\}$

Interface signals

$$Int = Input \cup Output \subset S$$

assume Int finite.

Environment

$$E : S \rightharpoonup \{\mathsf{f}, \mathsf{t}\}$$

assume $dom(E)$ finite and containing Int.

Threads and Programs

$$
\begin{array}{rl}
T ::= & 0 \mid \\
& T; T \mid \\
& A(\mathbf{s}) \\
& (\nu s\ T) \mid \\
& (\mathsf{thread}\ T) \mid \\
& (\mathsf{emit}\ s) \mid \\
& (\mathsf{await}\ s) \mid \\
& (\mathsf{watch}\ s\ T)
\end{array}
$$

A *program* P is a *multi-set* of threads.

Evaluation (informal)

To execute a program during an instant, proceed as follows:

1. Schedule the execution of the threads *non-deterministically*.
2. When no thread can progress then all threads are either *terminated* or *waiting* on an await.
3. If this happens, evaluate the watching conditions in the order of their insertion.

Redexes

- Assume that sequential composition ';' associates to the right.
- A *redex* Δ is defined by the grammar:

  $$\Delta ::= 0; T \parallel (\mathsf{emit}\ s) \parallel (\nu s\ T) \parallel (\mathsf{thread}\ T) \parallel (\mathsf{await}\ s) \parallel (\mathsf{watch}\ s\ 0) \parallel A(\mathbf{s}) \ .$$

Evaluation context

An *evaluation context* C is defined by the grammar:

$$C ::= [\,] \parallel [\,]; T \parallel (\mathsf{watch}\ s\ C) \parallel (\mathsf{watch}\ s\ C); T\ .$$

Unique decomposition

A thread $T \neq 0$ admits a *unique decomposition* $T = C[\Delta]$ into an evaluation context C and a redex Δ. Moreover, if $T = 0$ then no decomposition exists.

Reduction rules

$$(0; T, E) \xrightarrow{\emptyset} (T, E)$$

$$(\mathsf{emit}\ s, E) \xrightarrow{\emptyset} (0, E[\mathsf{t}/s])$$

$$(\mathsf{watch}\ s\ 0, E) \xrightarrow{\emptyset} (0, E)$$

$$(\nu s\ T, E) \xrightarrow{\emptyset} (T, E[\mathsf{f}/s]) \quad \text{if } s \notin dom(E)$$

$$(A(\mathbf{s}), E) \quad \xrightarrow{\emptyset} ([\mathbf{s}/\mathbf{x}]T, E) \quad \text{if } A(\mathbf{x}) = T$$

$$(\mathsf{await}\ s, E) \quad \xrightarrow{\emptyset} (0, E) \quad \text{if } E(s) = \mathsf{t}$$

$$(\mathsf{thread}\ T, E) \quad \xrightarrow{\{\!|T|\!\}} (0, E)$$

$$(C[\Delta], E) \quad \xrightarrow{P} (C[T'], E') \quad \text{if } (\Delta, E) \xrightarrow{P} (T', E')$$

Program evaluation

$$(P \cup \{\!|T|\!\}, E) \to (P \cup \{\!|T'|\!\} \cup P'', E') \quad \text{if} \quad (T, E) \stackrel{P''}{\to} (T', E') \ .$$

Suspension = End of instant

- $(T, E) \downarrow$ if T cannot be reduced in the environment E according to the rules above.
- $(T, E) \downarrow$ if and only if $T = 0$ or $T = C[(\mathsf{await}\ s)]$ with $E(s) = \mathsf{f}$.
- A program is suspended if all its thread are suspended.

Computation at the end of the instant

We transform all (watch s T) instructions where the signal s is present into the terminated thread 0. Formally,

$$
\begin{aligned}
\lfloor P \rfloor_E &= \{\!| \lfloor T \rfloor_E \mid T \in P |\!\} \\
\lfloor 0 \rfloor_E &= 0 \\
\lfloor T; T' \rfloor_E &= \lfloor T \rfloor_E ; T' \\
\lfloor \mathsf{await}\ s \rfloor_E &= (\mathsf{await}\ s) \\
\lfloor \mathsf{watch}\ s\ T \rfloor_E &= \begin{cases} 0 & \text{if } E(s) = \mathsf{t} \\ (\mathsf{watch}\ s\ \lfloor T \rfloor_E) & \text{otherwise} \end{cases}
\end{aligned}
$$

Some definable instructions

$$(\mathsf{loop}\ T) \quad = A \quad \text{where: } A \ = \ T; A$$

$$(\mathsf{now}\ T) \quad = \nu s\ (\mathsf{emit}\ s); (\mathsf{watch}\ s\ T) \quad s \notin sig(T)$$

$$\mathsf{pause} \quad = \nu s\ (\mathsf{now}\ (\mathsf{await}\ s)) \quad s \notin sig(T)$$

$$
\begin{aligned}
(\text{present } s\ T_1\ T_2) \quad &= \nu s'\ (\text{thread} \\
&\qquad (\text{now } (\text{await } s); (\text{thread } T_1; (\text{emit } s'))), \\
&\qquad (\text{watch } s \text{ pause}; (\text{thread } T_2; (\text{emit } s'))) \ ); \\
&\qquad\quad (\text{await } s')
\end{aligned}
$$

$$
\begin{aligned}
(T_1 \parallel T_2) \quad &= \nu s_1, s_2, s_1', s_2' \ (\mathsf{thread} \\
&\qquad (\mathsf{watch}\ s_1'\ T_1; (\mathsf{loop}\ (\mathsf{emit}\ s_1); \mathsf{pause})), \\
&\qquad (\mathsf{watch}\ s_2'\ T_2; (\mathsf{loop}\ (\mathsf{emit}\ s_2); \mathsf{pause}))\ ); \\
&\qquad\quad (\mathsf{await}\ s_1); (\mathsf{emit}\ s_1'); (\mathsf{await}\ s_2); (\mathsf{emit}\ s_2')
\end{aligned}
$$

Remark

- The original SL model has loop, present, and $\|$ rather than recursion and thread spawning.
- This has the consequence that the number of threads and the size of the evaluation contexts is bound by a constant.
- Also the present and $\|$ constructs in SL have a different behaviour with respect to the evaluation context.

Monotonicity and determinacy

- During an instant the collection of signals emitted can only grow and all that can be tested is the presence (not the absence) of a signal.
- Technically, the reduction relation is *strongly confluent* up to some renaming.
- This implies that programs are *deterministic*, *i.e.*, for a given input they always react in the same way and recursively the property is satisfied by the continuation in the following instant (if any).

Reactivity

- It is easy to write *looping programs*.
- In the programming practice, reactivity is ensured by *instrumenting the code* with pause statements that force the computation to suspend for the current instant.
- For instance,

$$\begin{aligned} A &= (\mathsf{watch}\ s_1\ B); (\mathsf{emit}\ s_4); A \\ B &= (\mathsf{await}\ s_2); (\mathsf{emit}\ s_3); \mathsf{pause}; B \end{aligned}$$

- It is feasible to design *static analysis techniques* that guarantee reactivity.

How do we justify the choice of the synchronisation operators?

- The present operator reflects directly the design principle of the SL language.
- But why having watch and await?
- What about other synchronisation operators such as

  $$(\mathsf{when}\ s\ T)$$

  that runs T when s is present ?

A tail recursive model

$$t \quad ::= 0 \parallel A(\mathbf{s}) \parallel \mathsf{emit}\ s.t \parallel \nu s\ t \parallel \mathsf{thread}\ t.t \parallel \mathsf{present}\ s\ t\ b$$

$$b \quad ::= t \parallel \mathsf{ite}\ s\ b\ b$$

- The reduction semantics is even simpler than for the previous language (no evaluation context).
- The basic reduction rule is

$$(\mathsf{present}\ s\ t\ b, E) \xrightarrow{\emptyset} (t, E) \quad \text{if } E(s) = \mathsf{t}$$

- The basic rule at the end of the instant is

$$\langle\!|\mathsf{ite}\ s\ b_1\ b_2|\!\rangle_E = \begin{cases} \langle\!|b_1|\!\rangle_E & \text{if } E(s) = \mathsf{t} \\ \langle\!|b_2|\!\rangle_E & \text{if } E(s) = \mathsf{f} \end{cases}$$

Continuation passing style translation

- There is a *semantic preserving* CPS translation of the SL language into the tail recursive SL language.
- The translation $[\![\_]\!]$ is parameterised on a pair (t, τ):
  - t is the *default continuation*.
  - τ is a *list of pairs* $(s_1, t_1) \cdots (s_n, t_n)$ representing the signals that are under the control of a watch statement and the relative continuations.

CPS-translation

$$
\begin{aligned}
[\![0]\!](t,\tau) &= t \\
[\![T_1;T_2]\!](t,\tau) &= [\![T_1]\!]([\![T_2]\!](t,\tau),\tau) \\
[\![\mathsf{emit}\ s]\!](t,\tau) &= \mathsf{emit}\ s.t \\
[\![\nu s\ T]\!](t,\tau) &= \nu s\ [\![T]\!](t,\tau), \quad \text{where: } s \notin sig(t) \cup sig(\tau) \\
[\![\mathsf{thread}\ T]\!](t,\tau) &= \mathsf{thread}\ [\![T]\!](0,\epsilon).t
\end{aligned}
$$

$$[\![\mathsf{watch}\ s\ T]\!](t,\tau) = [\![T]\!](t,\tau\cdot(s,t))$$

$$[\![\mathsf{await}\ s]\!](t,\tau) = \mathsf{present}\ s\ t\ b,$$
where: $\tau = (s_1,t_1)\cdots(s_m,t_m), \quad b \equiv (\mathsf{ite}\ s_1\ t_1 \ldots (\mathsf{ite}\ s_m\ t_m\ A)\ldots),$
$A = \mathsf{present}\ s\ t\ b$

$$[\![A(\mathbf{s})]\!](t,\tau) = A^{(t,\tau)}(\mathbf{s},\mathbf{s}'),$$
where: $sig(t,\tau) = \{\mathbf{s}'\}, \quad A(\mathbf{x}) = T,$
$\{\mathbf{x}\}\cap\{\mathbf{s}'\} = \emptyset, \quad A^{(t,\tau)}(\mathbf{x},\mathbf{s}') = [\![T]\!](t,\tau)\ .$

Example

- Consider the program A defined by:

$$\begin{aligned} A &= (\mathsf{watch}\ s_1\ B); (\mathsf{emit}\ s_4); A \\ B &= (\mathsf{await}\ s_2); (\mathsf{emit}\ s_3); \mathsf{pause}; B \end{aligned}$$

- To keep the translation compact, assume:

$$[\![\mathsf{pause}]\!](t, (s_1, t_1) \cdots (s_n, t_n)) = \mathsf{pause.ite}\ s_1\ t_1 (\cdots (\mathsf{ite}\ s_n\ t_n\ t) \cdots)$$

- Then the program after CPS translation becomes:

$$
\begin{aligned}
A^{(0,\epsilon)} &= B^{(t_1,\tau_1)} \\
t_1 &= \mathsf{emit}\ s_4.A^{(0,\epsilon)} \\
\tau_1 &= (s_1, t_1) \\
B^{(t_1,\tau_1)} &= \mathsf{present}\ s_2\ t_2\ (\mathsf{ite}\ s_1\ t_1\ B^{(t_1,\tau_1)}) \\
t_2 &= \mathsf{emit}\ s_3.\mathsf{pause.ite}\ s_1\ t_1\ B^{(t_1,\tau_1)}\ .
\end{aligned}
$$

Towards a process algebraic treatment

Some criticism

The formalisation of the SL model we have considered so far is close to an abstract machine.

- Ad hoc α-renaming.
- A global notion of environment.
- Threads compose but do not reduce.
- Programs reduce but do not compose.

A revised syntax

We introduce a notion of program that subsumes, threads, programs, and environments.

$$
\begin{array}{lll}
P & ::= & 0 \parallel \mathsf{emit}\ s \parallel \mathsf{present}\ s\ P\ B \parallel P \mid P \parallel \nu s\ P \parallel A(\mathbf{s}) \\
B & ::= & P \parallel \mathsf{ite}\ s\ B\ B
\end{array}
$$

NB We omit ‘$\mathsf{emit}\ s.P$ and ‘$\mathsf{thread}\ P'.P$. They are syntactic sugar for $(\mathsf{emit}\ s) \mid P$ and $P' \mid P$, respectively.

A labelled transition system

Actions

$$\alpha ::= \tau \parallel s \parallel \overline{s}$$

Transitions

$$\frac{}{\mathsf{emit}\ s \xrightarrow{\overline{s}} \mathsf{emit}\ s}$$

$$\frac{}{\mathsf{present}\ s\ P\ B \xrightarrow{s} P \mid (\mathsf{emit}\ s)}$$

$$\frac{P_1 \xrightarrow{s} P_1' \quad P_2 \xrightarrow{\overline{s}} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2'}$$

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2}$$

$$\frac{P \xrightarrow{\alpha} P' \quad s \notin \alpha}{\nu s\ P \xrightarrow{\alpha} \nu s\ P'}$$

$$\frac{A(\mathbf{x}) = P}{A(\mathbf{s}) \xrightarrow{\tau} [\mathbf{s}/\mathbf{x}]P}$$

Remarks

- Emission of a signal is *persistent*.
- When a signal is present, the continuation is guaranteed to see the signal as present.
- This has the effect that:

$$(\mathsf{emit}\ s) \mid (\mathsf{present}\ s\ 0\ 0) \xrightarrow{\tau} (\mathsf{emit}\ s) \mid 0 \mid (\mathsf{emit}\ s)$$

- However this is not a problem, because emitting a signal *once* is the *same* as emitting it *twice*.

End of the instant

The idea is always the same:

- Collect all emitted signals (restricted or not).
- Reset all emissions.
- Compute the continuations B of the suspended present statements.

Towards equivalence

- Can we define *a manageable compositional equivalence* based on a *labelled bisimulation*?
- Can we *justify its definition* via a notion of *contextual bisimulation*?

Worm up: mini-'asynchronous' CCS (aCCS)

$$P ::= \overline{a} \parallel a.P \parallel (P \mid P) \parallel \nu a\ P$$

- No output prefix.
- Communication is asynchronous, *i.e.* based on *unordered and unbounded buffers* rather than on *rendez vous* (Internet vs. telephone).
- If an observer sends a message it has no way to see if it has been received. In other terms, we can observe the outputs but not the inputs.

Barbed bisimulation for aCCS

A relation R is a *barbed bisimulation* if it symmetric and whenever $P\ R\ Q$ then

1. $P \xrightarrow{\tau} P'$ then $Q \stackrel{\tau}{\Rightarrow} Q'$ and $P'\ R\ Q'$.
2. $P \xrightarrow{\overline{a}} \cdot$ then $Q \stackrel{\tau}{\Rightarrow} Q'$, $Q' \xrightarrow{\overline{a}} \cdot$, and $P\ R\ Q'$.

Denote with $\approx_B$ the largest relation that satisfies this property.

Contextual bisimulation for aCCS

A static context C is defined by

$$C ::= [\,] \parallel C \mid P \parallel \nu a\ C$$

A relation R is a *contextual bisimulation* if

1. It is a barbed bisimulation.
2. If $P\ R\ Q$ then $C[P]\ R\ C[Q]$.

Denote with $\approx_C$ the largest contextual bisimulation.

Problem

- One can prove that:

$$a.\overline{a} \approx_C 0$$

- How can we account for such equivalence in a labelled bisimulation?

Idea

If $P\ R\ Q$ and $P \xrightarrow{a} P'$ then

- Either $Q \stackrel{a}{\Rightarrow} Q'$ and $P'\ R\ Q'$ (as usual!)
- or $Q \stackrel{\tau}{\Rightarrow} Q'$ and $P'\ R\ (Q' \mid \overline{a})$ (what's that?).

Ref. A., Castellani, Sangiorgi. *On bisimulations for the asynchronous π-calculus. Theor. Comp. Sci.* 1998.

Example

$$a.\overline{a} \approx_L 0$$

- If $a.\overline{a} \xrightarrow{a} \overline{a}$,
- then $0 \stackrel{\tau}{\Rightarrow} 0$,
- and $\overline{a} \approx_L (0 \mid \overline{a})$.

Labelled bisimulation

So the full definition becomes. A relation R is a *labelled bisimulation* if it symmetric and whenever $P\ R\ Q$ then if

1. $P \xrightarrow{\tau} P'$ then $Q \stackrel{\tau}{\Rightarrow} Q'$ and $P'\ R\ Q'$.
2. $P \xrightarrow{\overline{a}} P'$ then $Q \stackrel{\overline{a}}{\Rightarrow} Q'$, and $P'\ R\ Q'$.
3. $P \xrightarrow{a} P'$ then either $Q \stackrel{a}{\Rightarrow} Q'$ and $P'\ R\ Q'$ or $Q \stackrel{\tau}{\Rightarrow} Q'$ and $P'\ R\ (Q' \mid \overline{a})$.

Congruence properties of labelled bisimulation

Proposition If $P_1 \approx_L P_2$ then $P_1 \mid Q \approx_L P_2 \mid Q$
(and $\nu a\ P_1 \approx_L \nu a\ P_2$).

Proof outline

The proof structure is *not* quite the usual one.

1. We introduce a structural equivalence $\equiv$ including equations such as $P \mid 0 \equiv P$.
2. We check that the proof technique *up to structural equivalence* is sound.
3. We show that $P_1 \approx_L P_2$ implies $(P_1 \mid \overline{a}) \approx_L (P_2 \mid \overline{a})$.

Exercise Analyse the case where $(P_1 \mid \overline{a})$ does a τ or an input move.

4. We show that $\approx_L$ is transitive.

Exercise Analyse the case where $P_1 \approx_L P_2 \approx_L P_3$ and P_1 does an input move.

5. Only then can we show that

$$\{(P_1 \mid Q, P_2 \mid Q) \mid P_1 \approx_L P_2\}$$

is a labelled bisimulation up to $\equiv$.

Exercise Analyse the case where $P_1 \mid Q$ does a τ move or an input move.

6. It follows that

$$P_1 \approx_L P_2 \text{ implies } P_1 \approx_C P_2$$

Exercise Why?

Building contexts

Proposition If $P_1 \approx_C P_2$ then $P_1 \approx_L P_2$.

- We show that

$$\{(P_1, P_2) \mid P_1 \approx_C P_2\}$$

is a labelled bisimulation. This amounts to build suitable contexts.

Exercise What happens if P_1 does an input move?

Back to SL

So why are we looking at aCCS?

- It is an interesting variation of the model to the point that *programming languages* based on the π-calculus have adopted it.
- The *reception of a signal* in SL is not directly observable just as the *reception of a message* in aCCS.
- However there are two additional complications to consider in SL:
  1. The *end of the instant* should be observable.
  2. Once a program is closed and running, an *emission* is only observable at the end of the instant.

End of the instant

- Consider the following two *suspended* programs:

  $P = \mathsf{present}\ s_1\ 0\ (\mathsf{ite}\ s_2\ (\mathsf{emit}\ s_3)\ 0) \quad$ and $\quad Q = \mathsf{present}\ s_2\ 0\ 0\ .$

- If we plug them in the context $[\ ] \mid (\mathsf{emit}\ s_2)$ they exhibit a different behaviour.

- However a definition such as:

$$\frac{P\ R\ Q \qquad P \downarrow}{Q \stackrel{\tau}{\Rightarrow} Q', \qquad Q' \downarrow, \qquad P\ R\ Q', \qquad \lfloor P \rfloor\ R\ \lfloor Q' \rfloor}$$

  is not enough to distinguish them.

- What will do is the following stronger definition:

$$\frac{P\ R\ Q \quad S = (\mathsf{emit}\ s_1) \mid \cdots \mid (\mathsf{emit}\ s_n) \quad (P \mid S)\downarrow}{\begin{array}{c}(Q \mid S) \stackrel{\tau}{\Rightarrow} (Q' \mid S), \quad (Q' \mid S)\downarrow, \\ (P \mid S)\ R\ (Q' \mid S), \quad \lfloor P \mid S \rfloor\ R\ \lfloor Q' \mid S \rfloor\end{array}}$$

Emission and end of the instant

- Suppose Ω is a looping program.
- Then in $(\mathsf{emit}\ s) \mid \Omega$, the emission on s should not be observable because the instant *never terminates* (no matter who runs in parallel).
- So we look for a predicate *Pred* such that the clause for emission is formulated as follows:

$$\frac{P\ R\ Q, \quad Pred(P), \quad P \xrightarrow{\overline{s}} P'}{Q \overset{\overline{s}}{\Rightarrow} Q' \quad \text{and } P'\ R\ Q'}$$

NB Incidentally, in SL if $P \xrightarrow{\overline{s}} P'$ then $P = P'$.

Two candidates:
Weak suspension and Labelled suspension

$P \Downarrow$ if $\exists\, P'\ P \stackrel{\tau}{\Rightarrow} P'$ and $P' \downarrow$ (weak suspension)

$P \Downarrow_L$ if $P \stackrel{\alpha_1}{\rightarrow} P_1 \cdots \stackrel{\alpha_n}{\rightarrow} P_n, \quad n \geq 0,$ and $P_n \downarrow$ (L-suspension)

Characterisation of Labelled suspension

Let P be a program and let C denote a static context made of parallel composition and restriction. The following are equivalent:

1. $P \Downarrow_L$.
2. There is a program Q such that $(P \mid Q) \Downarrow$.
3. There is a static context C such that $C[P] \Downarrow_L$.

- An important property of L-suspension is that

$$(P \mid Q) \Downarrow_L \text{ implies } P \Downarrow_L$$

- This *may fail* for weak suspension in *non-deterministic* extensions of the language.
- However, in SL the property holds for weak suspension too because thanks to *confluence* we can show:

$$P \Downarrow_L \text{ iff } P \Downarrow$$

Full definition of labelled bisimulation

If $P\ R\ Q$ then

- $P \xrightarrow{\tau} P'$ implies $Q \stackrel{\tau}{\Rightarrow} Q'$ and $P'\ R\ Q'$.
- $P \Downarrow_L$ and $P \xrightarrow{\overline{s}} P'$ implies $Q \stackrel{\overline{s}}{\Rightarrow} Q'$ and $P'\ R\ Q'$.
- $P \xrightarrow{s} P'$ implies either $Q \stackrel{s}{\Rightarrow} Q'$ and $P'\ R\ Q'$ or $Q \stackrel{\tau}{\Rightarrow} Q'$ and $P'\ R\ (Q' \mid (\mathsf{emit}\ s))$.
- $S = (\mathsf{emit}\ s_1) \mid \cdots (\mathsf{emit}\ s_n)$, $(P \mid S) \downarrow$ implies $(Q \mid S) \stackrel{\tau}{\Rightarrow} (Q' \mid S)$, $(Q' \mid S) \downarrow$, $(P \mid S)\ R\ (Q' \mid S)$, and $\lfloor P \mid S \rfloor\ R\ \lfloor Q' \mid S \rfloor$.

Denote with $\approx_L$ the largest one.

Proof of congruence

The proof outline is similar to the one for asynchronous bisimulation (but of course the details are different...):

1. We introduce a *structural equivalence*.
2. Develop *proof technique up to*.
3. Show that $P_1 \approx_L P_2$ implies $(P_1 \mid (\mathsf{emit}\ s)) \approx_L (P_2 \mid (\mathsf{emit}\ s))$.
4. Show that $\approx_L$ is *transitive*.
5. Show that

   $$\{(P_1 \mid Q, P_2 \mid Q) \mid P_1 \approx_L P_2\}$$

   is a labelled bisimulation up to $\equiv$.

Justifying labelled bisimulation

It remains to justify labelled bisimulation via a notion of contextual bisimulation.

Definition of barbed bisimulation If $P \; R \; Q$ then

- $P \xrightarrow{\tau} P'$ implies $Q \stackrel{\tau}{\Rightarrow} Q'$ and $P' \; R \; Q'$.
- $P \Downarrow_L$ and $P \xrightarrow{\overline{s}} \cdot$ implies $Q \stackrel{\tau}{\Rightarrow} Q' \xrightarrow{\overline{s}}$ and $P' \; R \; Q'$.
- $P \downarrow$ implies $Q \stackrel{\tau}{\Rightarrow} Q'$, $Q' \downarrow$, $P \; R \; Q'$, and $\lfloor P \rfloor \; R \; \lfloor Q' \rfloor$.

Definition of contextual bisimulation A barbed bisimulation such that $P\ R\ Q$ implies $C[P]\ R\ C[Q]$. Denote with $\approx_C$ the largest one.

Theorem

$$P \approx_L Q \text{ iff } P \approx_C Q.$$

NB Again, to show the direction ($\Leftarrow$), we have to build suitable contexts ...

A quick look at ongoing research

- Practical languages that have been developed on top of the model have *data types*.
- In particular, signals carry *values*. E.g. think of primitives:

$$(\mathsf{emit}\ s\ e) \quad \mathsf{present}\ s(x).P, K$$

- This introduces two sources of *non-determinism* in the language.

1. Two signals compete to be received within an instant:

$$(\mathsf{emit}\ s\ 0) \mid (\mathsf{emit}\ s\ 1) \mid \mathsf{present}\ s(x).P, K$$

   To avoid this situation one needs to make sure (static analysis) that *at most one value* is emitted on a signal at each instant.

2. Alternatively, we could wait the end of the instant to collect the *set of values* emitted within the instant (cf. Reactive ML).

   In practice, the set has to be represented by some data structure such as a list. We have to make sure that the behaviour of the continuation *does not depend* on the particular representation chosen, *i.e.*, it has to be invariant under permutations of the list.

Determinacy vs. observational equivalence

The last example shows that the notion of *determinacy* depends on the notion of *observational equivalence*. We propose the following challenge:

> Extend the theory of observational equivalence developed for SL to a kind of *synchronous* π-calculus.

- This seems interesting by itself since so far there is no analogous of Synchronous CCS for the π-calculus.
- Moreover, it may be useful to analyse the notion of determinacy.