Determinacy in a synchronous π-calculus

Roberto Amadio

Université de Paris 7

Laboratoire Preuves, Programmes et Systèmes

Joint work with Mehdi Dogguy

Plan

- What is determinacy in interactive systems?
- The *synchronous* π-calculus.
- Results.

Towards a definition of determinacy

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

- We place this preliminary discussion in the context of a simple model such as *CCS*.
- Take *equivalent* to mean *weak bisimilar*.
- Take *experiment* to be a finite sequence of observable actions.

Ref Milner 89, Groote-Sellink 96, Philippou-Walker 97

A formal definition of determinacy

- Let $s = \ell_1 \cdots \ell_n$ be a finite word of *observable* actions.
- Define

$$
\begin{array}{ll}
P \overset{\epsilon}{\Rightarrow} P' & \text{if } P \overset{\tau}{\Rightarrow} P' \\
P \overset{\ell_1 \ldots \ell_n}{\Rightarrow} P',\ n \geq 1 & \text{if } P \overset{\ell_1}{\Rightarrow} \cdots \overset{\ell_n}{\Rightarrow} P'
\end{array}
$$

- A process P is *determinate* if for any s,

$$
\frac{P \overset{s}{\Rightarrow} P' \quad P \overset{s}{\Rightarrow} P''}{P' \approx P''}
$$

NB This definition entails invariance under internal reductions.

Wish list

We want more:

1. Manageable method to prove determinacy. For instance, *confluence* and even better *local confluence*.

2. Compositional and effective method to build deterministic systems. For instance, a *typing system*.

Confluence and Local Confluence

- A process P is *confluent* if for every *derivative* Q of P we have:

$$\frac{Q \stackrel{\alpha}{\Rightarrow} Q_1 \quad Q \stackrel{\beta}{\Rightarrow} Q_2 \quad \alpha \downarrow \beta}{\exists\, Q_1', Q_2' \ (\ Q_1 \stackrel{\beta\backslash\alpha}{\Rightarrow} Q_1' \quad Q_2 \stackrel{\alpha\backslash\beta}{\Rightarrow} Q_2' \quad Q_1' \approx Q_2'\ )}$$

- A process P is *locally confluent* if for every *derivative* Q of P we have:

$$\frac{Q \stackrel{\alpha}{\rightarrow} Q_1 \quad Q \stackrel{\beta}{\rightarrow} Q_2 \quad \alpha \downarrow \beta}{\exists\, Q_1', Q_2' \ (\ Q_1 \stackrel{\beta\backslash\alpha}{\Rightarrow} Q_1' \quad Q_2 \stackrel{\alpha\backslash\beta}{\Rightarrow} Q_2' \quad Q_1' \approx Q_2'\ )}$$

NB $\alpha \downarrow \beta$ and $\alpha\backslash\beta$ stand for *action compatibility* and *action residual*, respectively.

Facts in CCS

Call a process *reactive* if the τ reductions of every derivative always terminate.

- A confluent process is deterministic (converse fails).
- A reactive and locally confluent process is confluent (a kind of Newman lemma).

Rudimentary typing (sample)

- Let Γ be a set of observable actions.

- We write $\Gamma \vdash P$ if all the observable actions a derivative of P may perform belong to Γ.

- A typing rule for parallel composition:

$$\frac{\Gamma_1 \vdash P_1, \quad \Gamma_2 \vdash P_2, \quad \Gamma_1 \cap \Gamma_2 = \emptyset, \quad \Gamma_1 \cap \overline{\Gamma_2} \subseteq \{a_1, \ldots, a_n\}}{(\Gamma_1 \cup \Gamma_2) \backslash \{a_1, \ldots, a_n\} \vdash \nu a_1, \ldots, a_n \ (P \mid Q)}$$

Fact in CCS A typable program is confluent.

The $S\pi$-calculus: a *synchronous* π-calculus

Assume $v_1 \neq v_2$ are two distinct values and

$$
\begin{aligned}
P = \nu\ s_1, s_2\ (&\overline{s_1}v_1 \mid \overline{s_1}v_2 \mid \\
&\quad s_1(x).\ (s_1(y).\ (s_2(z).\ A(x,y)\ , B(!s_1)) \\
&\qquad\qquad , 0) \\
&\quad , 0)
\end{aligned}
$$

P is a π-calculus process if we forget about the else branches of the read instructions.

Ref Boussinot-De Simone 96, A. 05, A. 06

Spot the differences...

$P = \nu\ s_1, s_2\ (\overline{s_1}v_1 \mid \overline{s_1}v_2 \mid s_1(x).\ (s_1(y).\ (s_2(z).\ A(x,y)\ , B(!s_1)), 0), 0)$

- In π, P reduces to

$$P_1 = \nu s_1, s_2\ s_2(z).A(\sigma(x), \sigma(y))$$

where $\sigma(x), \sigma(y) \in \{v_1, v_2\}$ and $\sigma(x) \neq \sigma(y)$.

- In $S\pi$, *signals persist within the instant* and P reduces to

$$P_2 = \nu s_1, s_2\ (\overline{s_1}v_1 \mid \overline{s_1}v_2 \mid (s_2(z).A(\sigma(x), \sigma(y)), B(!s_1)))$$

where $\sigma(x), \sigma(y) \in \{v_1, v_2\}$.

- In π, P_1 is now *deadlocked*.
- In $S\pi$, the *current instant ends* and we move to the following one

$$P_2 \xrightarrow{N} P_2' = \nu s_1, s_2\ B(\ell)$$

where $\ell \in \{[v_1; v_2], [v_2; v_1]\}$ and N is the *next action*.

- Thus at the end of the instant, $!s_1$ becomes *a list of (distinct) values* emitted on s_1 during the instant.
- For this reason, $S\pi$ includes *lists has a primitive data structure*.

Deterministic programs: a cellular automaton

$$
\begin{array}{lcl}
Cell(q, s, \ell) & = & Send(q, s, \ell, \ell) \\
Send(q, s, \ell, \ell') & = & [\ell' \geq \mathsf{cons}(s', \ell'')] \quad (\overline{s'}q \mid Send(q, s, \ell, \ell'')), \\
 & & \qquad\qquad\qquad\qquad \mathsf{pause}.Cell(next(q, !s), s, \ell)
\end{array}
$$

Deterministic, assuming *next* is invariant under permutations of the list of states.

Deterministic programs: synchronous data flow

$$\xrightarrow{s_1} A \xrightarrow{s_2} C \xrightarrow{s_3} A \xrightarrow{s_4} B \xrightarrow{s_5} C \xrightarrow{s_6}$$

$$\nu s_2, s_3, s_4, s_5(\ A(s_1, s_2, s_3, s_4) \mid B(s_4, s_5) \mid C(s_2, s_3, s_5, s_6)\ )$$

$$\begin{aligned}
A(s_1, s_2, s_3, s_4) \quad &= s_1(x).(\overline{s_2} f(x) \mid s_3(y).(\overline{s_4} g(y) \mid \mathsf{pause}.A(s_1, s_2, s_3, s_4)), 0), 0 \\
B(s_4, s_5) \quad &= s_4(x).(\overline{s_5} h(x) \mid \mathsf{pause}.B(s_4, s_5)), 0 \\
C(s_2, s_3, s_5, s_6) \quad &= s_2(x).(\overline{s_3} i(x) \mid s_5(y).(\overline{s_6} l(y)) \mid \mathsf{pause}.C(s_2, s_3, s_5, s_6)), 0), 0
\end{aligned}$$

Deterministic, assuming at every instant at most one value is emitted on signal s_1.

Deterministic programs: client server

$$
\begin{array}{lcl}
Server(s) & = & \mathsf{pause}.Handle(s, !s) \\
Handle(s, \ell) & = & [\ell \geq \mathsf{cons}(\mathsf{req}(s', x), \ell')](\overline{s'} f(x) \mid Handle(s, \ell')), Server(s) \\
Client(x, s, t) & = & \nu s' \ (\bar{s}\mathsf{req}(s', x) \mid \mathsf{pause}.s'(x).\bar{t}x, 0)
\end{array}
$$

Deterministic, assuming ??

Ref Mandel-Pouzet 05, Saraswat et al. 06, Edwards-Tardieu 07.

Results (informal)

We manage to follow the 'CCS approach' above. Some highlights:

- We find a *modified* labelled transition system that allows for a *standard* definition of bisimulation.
- In $S\pi$, *determinacy=confluence* and we have simple local confluence conditions that coupled with reactivity imply confluence.
- We design a typing system for analysing signal usage.

Modified lts and standard bisimulation

For diagram chasing, it is nice to have a *standard bisimulation*.

$$\frac{P \mathrel{\mathcal{R}} Q, \quad P \xrightarrow{\alpha} P', \quad bn(\alpha) \cap fn(Q) = \emptyset}{\exists Q' \; (\; Q \stackrel{\alpha}{\Rightarrow} Q', \qquad P' \mathrel{\mathcal{R}} Q' \;)}$$

This is possible with a *modified lts*. The input rule is replaced by two:

$$\frac{}{s(x).P, K \xrightarrow{s?v} [v/x]P} \qquad \frac{}{P \xrightarrow{sv} (P \mid \overline{s}v)}$$

- The action $s?v$ is *not* observable. It is an *auxiliary* action needed to compute the internal synchronisation.
- The observable action is sv. Note that this action is always *enabled*.

Ref Honda-Yoshida 95, A.-Castellani-Sangiorgi 98

A simple condition for determinacy

- In the modified lts, (observable) *inputs* commute because they are always *enabled* and outputs commute because they are *persistent*.
- Then one just needs to check that τ-actions commute and N-actions commute.
- For instance, under reactivity, the following suffices to guarantee confluence: for all derivatives Q,

$$\frac{Q \xrightarrow{\alpha} Q_1, \quad Q \xrightarrow{\alpha} Q_2, \quad \alpha \in \{\tau, N\}}{\exists\, Q_3, Q_4 \ \ (Q_1 \stackrel{\tau}{\Rightarrow} Q_3, \quad Q_2 \stackrel{\tau}{\Rightarrow} Q_4, \quad Q_3 \approx Q_4)}$$

Signal usage

A signal type, $Sig_u(\sigma)$, carries an information u on the signal *usage*.

- Start with $L = \{0, 1, \infty\}$ where $0 < 1 < \infty$.
- Refine into $x \in L^3$ for output, input, and input at the end of the instant.
- Further refine, into $u \in (L^3)^\omega$ for usage at instant $0, 1, 2, \ldots$.

Ref Kobayashi-Pierce-Turner 99, Kobayashi 02,...

Two main usages

For the time being we have focused on just *two main usages*.

$e^\omega, e = (\infty, 0, \infty)$ We can receive only at the end of the instant. Moreover, the processing of the list recovered at the end of the instant must be order independent. We use *set types* to enforce this.

$o_1^\omega, o_1 = (1, \infty, \infty)$ At every instant, at most one emission is performed on the signal. To reason on this, we also rely on $o_1 o_0^\omega$, $o_0 o_1^\omega$, and o_0^ω, where $o_0 = (0, \infty, \infty)$.

Typing for the cellular automaton

$$\begin{array}{lcll} Cell(q,s,\ell) & = & Send(q,s,\ell,\ell) & \\ Send(q,s,\ell,\ell') & = & [\ell' \geq \mathsf{cons}(s',\ell'')] & (\overline{s'}q \mid Send(q,s,\ell,\ell'')), \\ & & & \mathsf{pause}.Cell(next(q,!s),s,\ell) \end{array}$$

Assume an inductive type $State$ to represent the state of a cell and let $S_1 = Sig_{e^\omega}(State)$ and $L_1 = List(S_1)$. Then the program is typable, assuming:

$$Cell : (State, S_1, L_1), \qquad Send : (State, S_1, L_1, L_1),$$
$$next : (State, Set(State)) \to State$$

Typing for the data flow

$$\nu s_2, s_3, s_4, s_5(\ A(s_1, s_2, s_3, s_4) \mid B(s_4, s_5) \mid C(s_2, s_3, s_5, s_6)\ )$$

$$\begin{aligned}
A(s_1, s_2, s_3, s_4) \quad &= s_1(x).(\overline{s_2} f(x) \mid s_3(y).(\overline{s_4} g(y) \mid \mathsf{pause}.A(s_1, s_2, s_3, s_4)), 0), 0 \\
B(s_4, s_5) \quad &= s_4(x).(\overline{s_5} h(x) \mid \mathsf{pause}.B(s_4, s_5)), 0 \\
C(s_2, s_3, s_5, s_6) \quad &= s_2(x).(\overline{s_3} i(x) \mid s_5(y).(\overline{s_6} l(y)) \mid \mathsf{pause}.C(s_2, s_3, s_5, s_6)), 0), 0
\end{aligned}$$

Assume an inductive type D of *data* and let $I = Sig_{o_0^\omega}(D)$ and $O = Sig_{o_1^\omega}(D)$. Then the program is typable assuming:

$$A : (I, O, I, O), \quad B : (I, O), \quad C : (I, O, I, O)\ .$$

(Problems with) Typing the client-server

$$
\begin{array}{lcl}
Server(s) & = & \mathsf{pause}.Handle(s, !s) \\
Handle(s, \ell) & = & [l \geq \mathsf{cons}(\mathsf{req}(s', x), \ell')](\overline{s'} f(x) \mid Handle(s, \ell')), Server(s) \\
Client(x, s, t) & = & \nu s' \ (\overline{s}\mathsf{req}(s', x) \mid \mathsf{pause}.s'(x).\overline{t}x, 0)
\end{array}
$$

- Assume an inductive type D of data and let

$$S_1 = Sig_u(D), \quad \mathsf{req} : (Sig_u(D), D) \to Req, \quad S_2 = Sig_{e^\omega}(Req)$$

- Since there are many clients, we are forced to take $u = e^\omega$. Then the server is typable as follows:

$$Server : (S_2), \quad Handle : (S_2, Set(Req))$$

- However the usage $u = e^{\omega}$ is incompatible with the programming of the client, as it can receive *during* the instant.
- It seems one needs more *usages* to type this.
- The client will get at most one reply on the return signal assuming that its request is *received at most once* and it is handled in a *linear way*.
- Thus, we need *many-to-one* usage and *linear inductive types*.