Determinacy in a synchronous pi-calculus

Roberto M. Amadio, Mehdi Dogguy

Determinacy in a synchronous π-calculus *

Roberto M. Amadio Mehdi Dogguy
Université Paris Diderot, PPS, UMR-7126

11th February 2008

Abstract

The $S\pi$-calculus is a *synchronous* π-calculus which is based on the SL model. The latter is a relaxation of the ESTEREL model where the reaction to the *absence* of a signal within an instant can only happen at the next instant. In the present work, we present and characterise a compositional semantics of the $S\pi$-calculus based on suitable notions of labelled transition system and bisimulation. Based on this semantic framework, we explore the notion of determinacy and the related one of (local) confluence.

1 Introduction

Let P be a program that can repeatedly interact with its environment. A *derivative* of P is a program to which P reduces after a finite number of interactions with the environment. A program *terminates* if all its internal computations terminate and it is *reactive* if all its derivatives are guaranteed to terminate. A program is *determinate* if after any finite number of interactions with the environment the resulting derivative is unique up to *semantic equivalence*.

Most conditions found in the literature that entail determinacy are rather intuitive, however the formal statement of these conditions and the proof that they indeed guarantee determinacy can be rather intricate in particular in the presence of name mobility, as available in a paradigmatic form in the π-calculus.

Our purpose here is to provide a streamlined theory of determinacy for the *synchronous* π-calculus introduced in [2]. It seems appropriate to address these issues in a volume dedicated to the memory of Gilles Kahn. First, Kahn networks [14] are a classic example of concurrent *and* deterministic systems. Second, Kahn networks have largely inspired the research on *synchronous* languages such as LUSTRE [9] and, to a lesser extent, ESTEREL [6]. An intended side-effect of this work is to illustrate how ideas introduced in concurrency theory well after Kahn networks can be exploited to enlighten the study of determinacy in concurrent systems.

Our technical approach will follow a process calculus tradition, namely:

1. We describe the interactions of a program with its environment through a *labelled transition system* to which we associate a compositional notion of *labelled bisimulation*.

2. We rely on this semantic framework, to introduce a notion of *determinacy* and a related notion of *confluence*.

*Work partially supported by ANR-06-SETI-010-02.

3. We provide *local* confluence conditions that are easier to check and that combined with *reactivity* turn out to be equivalent to determinacy.

We briefly trace the path that has lead to this approach. A systematic study of determinacy and confluence for CCS is available in [17] where, roughly, the usual theory of rewriting is generalised in two directions: first rewriting is labelled and second diagrams commute up to semantic equivalence. In this context, a suitable formulation of Newman's lemma [19], has been given in [11]. The theory has been gradually extended from CCS, to CCS with values, and finally to the π-calculus [20].

Calculi such as CCS and the π-calculus are designed to represent *asynchronous* systems. On the other hand, the $S\pi$-calculus is designed to represent *synchronous* systems. In these systems, there is a notion of *instant* (or phase, or pulse, or round) and at each instant each thread performs some actions and synchronizes with all other threads. One may say that all threads proceed at the same speed and it is in this specific sense that we will refer to *synchrony* in this work.

In order to guarantee determinacy in the context of CCS *rendez-vous* communication, it seems quite natural to restrict the calculus so that interaction is *point-to-point*, *i.e.*, it involves exactly one sender and one receiver.[1] In a synchronous framework, the introduction of *signal* based communication offers an opportunity to move from point-to-point to a more general multi-way interaction mechanism with multiple senders and/or receivers, while preserving determinacy. In particular, this is the approach taken in the ESTEREL and SL [8] models. The SL model can be regarded as a relaxation of the ESTEREL model where the reaction to the *absence* of a signal within an instant can only happen at the next instant. This design choice avoids some paradoxical situations and simplifies the implementation of the model. The SL model has gradually evolved into a general purpose programming language for concurrent applications and has been embedded in various programming environments such as C, JAVA, SCHEME, and CAML (see [7, 22, 16]). For instance, the Reactive ML language [16] includes a large fragment of the CAML language plus primitives to generate signals and synchronise on them. We should also mention that related ideas have been developed by Saraswat et al. [21] in the area of constraint programming.

The $S\pi$-calculus can be regarded as an extension of the SL model where signals can carry values. In this extended framework, it is more problematic to have both concurrency *and* determinacy. Nowadays, this question is frequently considered when designing various kind of synchronous programming languages (see, *e.g.*, [16, 10]). As we already mentioned, our purpose here is to address the question with the tool-box of process calculi following the work for CCS and the π-calculus quoted above. In this respect, it is worth stressing a few interesting variations that arise when moving from the 'asynchronous' π-calculus to the 'synchronous' $S\pi$-calculus. First, we have already pointed-out that there is an opportunity to move from a point-to-point to a multi-way interaction mechanism while preserving determinacy. Second, the notion of confluence and determinacy happen to coincide while in the asynchronous context confluence is a strengthening of determinacy which has better compositionality properties. Third, reactivity appears to be a reasonable property to require of a synchronous system, the goal being just to avoid instantaneous loops, *i.e.*, loops that take no time.[2]

[1] Incidentally, this is also the approach taken in Kahn networks but with an interaction mechanism based on unbounded, ordered buffers. It is not difficult to represent unbounded, ordered buffers in a CCS with value passing and show that, modulo this encoding, the determinacy of Kahn networks can be obtained as a corollary of the theory of confluence developed in [17].

[2] The situation is different in asynchronous systems where reactivity is a more demanding property. For

The rest of the paper is structured as follows. In section 2, we introduce the $S\pi$-calculus, in section 3, we define its semantics based on a standard notion of labelled bisimulation on a (non-standard) labelled transition system and we show that the bisimulation is preserved by static contexts, in section 4 we provide alternative characterisations of the notion of labelled bisimulation we have introduced, in section 5, we develop the concepts of determinacy and (local) confluence. Familiarity with the π-calculus [18, 23], the notions of determinacy and confluence presented in [17], and synchronous languages of the ESTEREL family [6, 8] is assumed.

2 Introduction to the $S\pi$-calculus

We introduce the syntax of the $S\pi$-calculus along with an informal comparison with the π-calculus and a programming example.

2.1 Programs

Programs $P, Q, \ldots$ in the $S\pi$-calculus are defined as follows:

$$\begin{array}{lll} P & ::= & 0 \mid A(\mathbf{e}) \mid \overline{s}e \mid s(x).P, K \mid [s_1 = s_2]P_1, P_2 \mid [u \trianglerighteq p]P_1, P_2 \mid \nu s\ P \mid P_1 \mid P_2 \\ K & ::= & A(\mathbf{r}) \end{array}$$

We use the notation $\mathbf{m}$ for a vector $m_1, \ldots, m_n$, $n \geq 0$. The informal behaviour of programs follows. 0 is the terminated thread. $A(\mathbf{e})$ is a (tail) recursive call of a thread identifier A with a vector $\mathbf{e}$ of expressions as argument; as usual the thread identifier A is defined by a unique equation $A(\mathbf{x}) = P$ such that the free variables of P occur in $\mathbf{x}$. $\overline{s}e$ evaluates the expression e and emits its value on the signal s. $s(x).P, K$ is the *present* statement which is the fundamental operator of the SL model. If the values $v_1, \ldots, v_n$ have been emitted on the signal s then $s(x).P, K$ evolves non-deterministically into $[v_i/x]P$ for some v_i ($[\_/\_]$ is our notation for substitution). On the other hand, if no value is emitted then the continuation K is evaluated at the end of the instant. $[s_1 = s_2]P_1, P_2$ is the usual matching function of the π-calculus that runs P_1 if s_1 equals s_2 and P_2, otherwise. Here both s_1 and s_2 are free. $[u \trianglerighteq p]P_1, P_2$, matches u against the pattern p. We assume u is either a variable x or a value v and p has the shape $\mathsf{c}(\mathbf{x})$, where c is a constructor and $\mathbf{x}$ is a vector of distinct variables. We also assume that if u is a variable x then x does not occur free in P_1. At run time, u is always a *value* and we run θP_1 if $\theta = match(u, p)$ is the substitution matching u against p, and P_2 if such substitution does not exist (written $match(u, p) \uparrow$). Note that as usual the variables occurring in the pattern p (including signal names) are bound in P_1. $\nu s\ P$ creates a new signal name s and runs P. $(P_1 \mid P_2)$ runs in parallel P_1 and P_2. A continuation K is simply a recursive call whose arguments are either expressions or values associated with signals at the end of the instant in a sense that we explain below. We will also write $\mathsf{pause}.K$ for $\nu s\ s(x).0, K$ with s not free in K. This is the program that waits till the end of the instant and then evaluates K.

instance, [11] notes: "*As soon as a protocol internally consists in some kind of correction mechanism (e.g., retransmission in a data link protocol) the specification of that protocol will contain a τ-loop*".

2.2 Expressions

The definition of programs relies on the following syntactic categories:

Sig	$::= s \mid t \mid \cdots$	(signal names)
Var	$::= Sig \mid x \mid y \mid z \mid \cdots$	(variables)
$Cnst$	$::= * \mid \mathsf{nil} \mid \mathsf{cons} \mid \mathsf{c} \mid \mathsf{d} \mid \cdots$	(constructors)
Val	$::= Sig \mid Cnst(Val, \ldots, Val)$	(values $v, v', \ldots$)
Pat	$::= Cnst(Var, \ldots, Var)$	(patterns $p, p', \ldots$)
Fun	$::= f \mid g \mid \cdots$	(first-order function symbols)
Exp	$::= Var \mid Cnst(Exp, \ldots, Exp) \mid Fun(Exp, \ldots, Exp)$	(expressions $e, e', \ldots$)
$Rexp$	$::= !Sig \mid Var \mid Cnst(Rexp, \ldots, Rexp) \mid$ $Fun(Rexp, \ldots, Rexp)$	(exp. with deref. $r, r', \ldots$)

As in the π-calculus, signal names stand both for signal constants as generated by the ν operator and signal variables as in the formal parameter of the present operator. Variables Var include signal names as well as variables of other types. Constructors $Cnst$ include $*$, nil, and cons. Values Val are terms built out of constructors and signal names. Patterns Pat are terms built out of constructors and variables (including signal names). If P, p are a program and a pattern then we denote with $fn(P), fn(p)$ the set of free signal names occurring in them, respectively. We also use $FV(P), FV(p)$ to denote the set of free variables (including signal names). We assume first-order function symbols $f, g, \ldots$ and an evaluation relation $\Downarrow$ such that for every function symbol f and values $v_1, \ldots, v_n$ of suitable type there is a unique value v such that $f(v_1, \ldots, v_n) \Downarrow v$ and $fn(v) \subseteq \bigcup_{i=1,\ldots,n} fn(v_i)$. Expressions Exp are terms built out of variables, constructors, and function symbols. The evaluation relation $\Downarrow$ is extended in a standard way to expressions whose only free variables are signal names. Finally, $Rexp$ are expressions that may include the value associated with a signal s at the end of the instant (which is written $!s$, following the ML notation for dereferenciation). Intuitively, this value is a *list of values* representing the *set of values* emitted on the signal during the instant.

2.3 Typing

Types include the basic type 1 inhabited by the constant $*$ and, assuming σ is a type, the type $Sig(\sigma)$ of signals carrying values of type σ, and the type $List(\sigma)$ of lists of values of type σ with constructors nil and cons. In the examples, it will be convenient to abbreviate $\mathsf{cons}(v_1, \ldots, \mathsf{cons}(v_n, \mathsf{nil}) \ldots)$ with $[v_1; \ldots; v_n]$. 1 and $List(\sigma)$ are examples of *inductive types*. More inductive types (booleans, numbers, trees,...) can be added along with more constructors. We assume that variables (including signals), constructor symbols, and thread identifiers come with their (first-order) types. For instance, a function symbols f may have a type $(\sigma_1, \sigma_2) \to \sigma$ meaning that it waits two arguments of type σ_1 and σ_2 respectively and returns a value of type σ. It is straightforward to define when a program is well-typed. We just point-out that if a signal name s has type $Sig(\sigma)$ then its dereferenced value $!s$ has type $List(\sigma)$. In the following, we will tacitly assume that we are handling well typed programs, expressions, substitutions,...

2.4 Comparison with the π-calculus

The syntax of the $S\pi$-calculus is similar to the one of the π-calculus, however there are some important *semantic* differences that we highlight in the following simple example. Assume

$v_1 \neq v_2$ are two distinct values and consider the following program in $S\pi$:

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

If we forget about the underlined parts and we regard s_1, s_2 as *channel names* then P could also be viewed as a π-calculus process. In this case, P would reduce to

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

where θ is a substitution such that $\theta(x), \theta(y) \in \{v_1, v_2\}$ and $\theta(x) \neq \theta(y)$. In $S\pi$, *signals persist within the instant* and P reduces to

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

where $\theta(x), \theta(y) \in \{v_1, v_2\}$. What happens next? In the π-calculus, P_1 is *deadlocked* and no further computation is possible. In the $S\pi$-calculus, the fact that no further computation is possible in P_2 is detected and marks the *end of the current instant*. Then an additional computation represented by the relation $\xrightarrow{N}$ moves P_2 to the following instant:

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

where $v \in \{[v_1; v_2], [v_2; v_1]\}$. Thus at the end of the instant, a dereferenced signal such as $!s_1$ becomes a list of (distinct) values emitted on s_1 during the instant and then all signals are reset.

2.5 A programming example

We introduce a programming example to illustrate the kind of synchronous programming that can be represented in the $S\pi$-calculus. We describe first a 'server' handling a list of requests emitted in the previous instant on the signal s. For each request of the shape $\mathsf{req}(s', x)$, it provides an answer which is a function of x along the signal s'.

$$\begin{array}{lcl} \mathit{Server}(s) & = & \mathtt{pause}.\mathit{Handle}(s, !s) \\ \mathit{Handle}(s, \ell) & = & [\ell \trianglerighteq \mathsf{req}(s', x) :: \ell'](\overline{s'}f(x) \mid \mathit{Handle}(s, \ell')), \mathit{Server}(s)\ . \end{array}$$

The programming of a client that issues a request x on signal s and returns the reply on signal t could be the following:

$$\mathit{Client}(x, s, t) \quad = \quad \nu s'\ (\overline{s}\mathsf{req}(s', x) \mid \mathsf{pause}.s'(x).\overline{t}x, 0)\ .$$

3 Semantics of the $S\pi$-calculus

In this section, we define the semantics of the $S\pi$-calculus by a 'standard' notion of labelled bisimulation on a 'non-standard' labelled transition system and we show that labelled bisimulation is preserved by 'static' contexts. A distinct notion of labelled bisimulation for the $S\pi$-calculus has already been studied in [2] and the following section 4 will show that the two notions are (almost) the same. A significant advantage of the presentation of labelled bisimulation we discuss here is that in the 'bisimulation game' all actions are treated in the same way. This allows allows for a considerable simplification of the diagram chasing arguments that are needed in the study of determinacy and confluence in section 5.

3.1 Actions

The actions of the forthcoming labelled transition system are classified in the following categories:

$$\begin{array}{lll} act & ::= \alpha \mid aux & \text{(actions)} \\ \alpha & ::= \tau \mid \nu \mathbf{t}\ \overline{s}v \mid sv \mid N & \text{(relevant actions)} \\ aux & ::= s?v \mid (E, V) & \text{(auxiliary actions)} \\ \mu & ::= \tau \mid \nu \mathbf{t}\ \overline{s}v \mid s?v & \text{(nested actions)} \end{array}$$

The category *act* is partitioned into relevant actions and auxiliary actions.

The *relevant actions* are those that are actually considered in the bisimulation game. They consist of: (i) an internal action τ, (ii) an emission action $\nu \mathbf{t}\ \overline{s}v$ where it is assumed that the signal names $\mathbf{t}$ are distinct, occur in v, and differ from s, (iii) an input action sv, and (iv) an action N (for *Next*) that marks the move from the current to the next instant.

The *auxiliary actions* consist of an input action $s?v$ which is coupled with an emission action in order to compute a τ action and an action (E, V) which is just needed to compute an action N. The latter is an action that can occur exactly when the program cannot perform τ actions and it amounts (i) to collect in lists the set of values emitted on every signal, (ii) to reset all signals, and (iii) to initialise the continuation K for each present statement of the shape $s(x).P, K$.

In order to formalise these three steps we need to introduce some notation. Let E vary over functions from signal names to finite sets of values. Denote with $\emptyset$ the function that associates the empty set with every signal name, with $[M/s]$ the function that associates the set M with the signal name s and the empty set with all the other signal names, and with $\cup$ the union of functions defined point-wise.

We represent a set of values as a list of the values contained in the set. More precisely, we write $v \Vdash M$ and say that v *represents* M if $M = \{v_1, \ldots, v_n\}$ and $v = [v_{\pi(1)}; \ldots; v_{\pi(n)}]$ for some permutation π over $\{1, \ldots, n\}$. Suppose V is a function from signal names to lists of values. We write $V \Vdash E$ if $V(s) \Vdash E(s)$ for every signal name s. We also write $dom(V)$ for $\{s \mid V(s) \neq []\}$. If K is a continuation, *i.e.*, a recursive call $A(\mathbf{r})$, then $V(K)$ is obtained from K by replacing each occurrence $!s$ of a dereferenced signal with the associated value $V(s)$. We denote with $V[\ell/s]$ the function that behaves as V except on s where $V[\ell/s](s) = \ell$.

With these conventions, a transition $P \xrightarrow{(E,V)} P'$ intuitively means that (1) P is suspended, (2) P emits exactly the values specified by E, and (3) the behaviour of P in the following instant is P' and depends on V. It is convenient to compute these transitions on programs where all name generations are lifted at top level. We write $P \succeq Q$ if we can obtain Q from P by repeatedly transforming, for instance, a subprogram $\nu s P' \mid P''$ into $\nu s(P' \mid P'')$ where $s \notin fn(P'')$.

Finally, the *nested actions* $\mu, \mu', \ldots$ are certain actions (either relevant or auxiliary) that can be produced by a sub-program and that we need to propagate to the top level.

3.2 Labelled transition system

The labelled transition system is defined in table 1 where rules apply to programs whose only free variables are signal names and with standard conventions on the renaming of bound names. As usual, one can rename bound variables, and the symmetric rules for (*par*) and (*synch*) are omitted. The first 12 rules from (*out*) to (ν_{ex}) are quite close to those of a polyadic π-calculus with asynchronous communication (see [12, 13, 4]) with the following exception:

$$(out)\quad \frac{e \Downarrow v}{\overline{s}e \xrightarrow{\overline{s}v} \overline{s}e} \qquad (in_{aux})\quad \frac{}{s(x).P, K \xrightarrow{s?v} [v/x]P}$$

$$(in)\quad \frac{}{P \xrightarrow{sv} (P \mid \overline{s}v)} \qquad (rec)\quad \frac{A(\mathbf{x}) = P, \quad \mathbf{e} \Downarrow \mathbf{v}}{A(\mathbf{e}) \xrightarrow{\tau} [\mathbf{v}/\mathbf{x}]P}$$

$$(=_1^{sig})\quad \frac{}{[s = s]P_1, P_2 \xrightarrow{\tau} P_1} \qquad (=_2^{sig})\quad \frac{s_1 \neq s_2}{[s_1 = s_2]P_1, P_2 \xrightarrow{\tau} P_2}$$

$$(=_1^{ind})\quad \frac{match(v, p) = \theta}{[v \trianglerighteq p]P_1, P_2 \xrightarrow{\tau} \theta P_1} \qquad (=_1^{ind})\quad \frac{match(v, p) = \uparrow}{[v \trianglerighteq p]P_1, P_2 \xrightarrow{\tau} P_2}$$

$$(comp)\quad \frac{P_1 \xrightarrow{\mu} P_1' \quad bn(\mu) \cap fn(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2} \qquad (synch)\quad \frac{P_1 \xrightarrow{\nu \mathbf{t}\ \overline{s}v} P_1' \quad P_2 \xrightarrow{s?v} P_2' \quad \{\mathbf{t}\} \cap fn(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\tau} \nu \mathbf{t}\ (P_1' \mid P_2')}$$

$$(\nu)\quad \frac{P \xrightarrow{\mu} P' \quad t \notin n(\mu)}{\nu t\ P \xrightarrow{\mu} \nu t\ P'} \qquad (\nu_{ex})\quad \frac{P \xrightarrow{\nu \mathbf{t}\ \overline{s}v} P' \quad t' \neq s \quad t' \in n(v) \backslash \{\mathbf{t}\}}{\nu t'\ P \xrightarrow{(\nu t', \mathbf{t})\overline{s}v} P'}$$

$$(0)\quad \frac{}{0 \xrightarrow{\emptyset, V} 0} \qquad (reset)\quad \frac{e \Downarrow v \quad v \text{ occurs in } V(s)}{\overline{s}e \xrightarrow{[\{v\}/s], V} 0}$$

$$(cont)\quad \frac{s \notin dom(V)}{s(x).P, K \xrightarrow{\emptyset, V} V(K)} \qquad (par)\quad \frac{P_i \xrightarrow{E_i, V} P_i' \quad i = 1, 2}{(P_1 \mid P_2) \xrightarrow{E_1 \cup E_2, V} (P_1' \mid P_2')}$$

$$(next)\quad \frac{P \succeq \nu \mathbf{s}\ P' \quad P' \xrightarrow{E, V} P'' \quad V \Vdash E}{P \xrightarrow{N} \nu \mathbf{s}\ P''}$$

Table 1: Labelled transition system

rule (*out*) models the fact that the emission of a value on a signal *persists* within the instant. The last 5 rules from (0) to (*next*) are quite specific of the $S\pi$-calculus and determine how the computation is carried on at the end of the instant (cf. discussion in 3.1).

The relevant actions different from τ, model the possible interactions of a program with its environment. Then the notion of reactivity can be formalised as follows.

Definition 1 (derivative) *A derivative of a program P is a program Q such that*

$$P \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} Q, \qquad \textit{where: } n \geq 0\ .$$

Definition 2 (reactivity) *We say that a program P is reactive, if for every derivative Q every τ-reduction sequence terminates.*

3.3 A compositional labelled bisimulation

We introduce first a rather standard notion of (weak) labelled bisimulation. We define $\stackrel{\alpha}{\Rightarrow}$ as:

$$\stackrel{\alpha}{\Rightarrow} = \begin{cases} (\stackrel{\tau}{\longrightarrow})^* & \text{if } \alpha = \tau \\ (\stackrel{\tau}{\Rightarrow}) \circ (\stackrel{N}{\longrightarrow}) & \text{if } \alpha = N \\ (\stackrel{\tau}{\Rightarrow}) \circ (\stackrel{\alpha}{\longrightarrow}) \circ (\stackrel{\tau}{\Rightarrow}) & \text{otherwise} \end{cases}$$

This is the standard definition except that we insist on *not* having internal reductions after an N action. Intuitively, we assume that an observer can control the execution of programs so as to be able to test them at the very beginning of each instant.[3] We write $P \stackrel{\alpha}{\longrightarrow} \cdot$ for $\exists\, P' \ (P \stackrel{\alpha}{\longrightarrow} P')$.

Definition 3 (labelled bisimulation) *A symmetric relation $\mathcal{R}$ on programs is a labelled bisimulation if*

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

We denote with $\approx$ the largest labelled bisimulation.

The standard variation where one considers weak reduction in the hypothesis ($P \stackrel{\alpha}{\Rightarrow} P'$ rather than $P \stackrel{\alpha}{\longrightarrow} P'$) leads to the same relation. Also, relying on this variation, one can show that the concept of bisimulation up to bisimulation makes sense, *i.e.*, a bisimulation up to bisimulation is indeed contained in the largest bisimulation. An important property of labelled bisimulation is that it is preserved by static contexts. The proof of this fact follows [2] and it is presented in appendix B.

Definition 4 *A static context C is defined as follows:*

$$C ::= [\ ] \mid C \mid P \mid \nu s\ C \tag{1}$$

Theorem 5 (compositionality of labelled bisimulation) *If $P \approx Q$ and C is a static context then $C[P] \approx C[Q]$.*

4 Characterisations of labelled bisimulation

The labelled transition system presented in table 1 embodies a number of technical choices which might not appear so natural at first sight. To justify these choices, it is therefore interesting to look for alternative characterisations of the induced bisimulation equivalence. To this end we recall the notion of *contextual* bisimulation introduced in [2].

Definition 6 *We write:*

$P \downarrow$	*if*	$\neg(\ P \stackrel{\tau}{\longrightarrow} \cdot\ )$	*(suspension)*
$P \Downarrow$	*if*	$\exists\, P' \ (\ P \stackrel{\tau}{\Rightarrow} P'$ *and* $P' \downarrow\ )$	*(weak suspension)*
$P \Downarrow_L$	*if*	$\exists\, P' \ (\ P \mid P'\ ) \Downarrow$	*(L-suspension)*

[3] This decision entails that, *e.g.*, we distinguish the programs P and Q defined as follows: $P = \mathsf{pause}.(\overline{s}_1 \oplus \overline{s}_2)$, $Q = \nu s\ (\mathsf{pause}.A(!s) \mid \overline{s}0 \mid \overline{s}1)$, where $A(x) = [x \trianglerighteq [0;1]](\overline{s}_1 \oplus \overline{s}_2), \overline{s}_1$, and $\oplus$, 0, and 1 are abbreviations for an internal choice and for two distinct constants, respectively (these concepts can be easily coded in the $S\pi$-calculus). On the other hand, P and Q would be equivalent if we defined $\stackrel{N}{\Rightarrow}$ as $\stackrel{\tau}{\Rightarrow} \circ \stackrel{N}{\longrightarrow} \circ \stackrel{\tau}{\Rightarrow}$.

Obviously, $P \downarrow$ implies $P \Downarrow$ which in turn implies $P \Downarrow_L$ and none of these implications can be reversed (see [2]). Also note that all the derivatives of a reactive program enjoy the weak suspension property.

Definition 7 (commitment) *We write $P \searrow \overline{s}$ if $P \xrightarrow{\nu \mathbf{t}\ \overline{s}v} \cdot$ and say that P commits to emit on s.*

Definition 8 (barbed bisimulation) *A symmetric relation $\mathcal{R}$ on programs is a barbed bisimulation if whenever $P\ \mathcal{R}\ Q$ the following holds:*

$(B1)$ If $P \xrightarrow{\tau} P'$ then $\exists Q'$ $(Q \stackrel{\tau}{\Rightarrow} Q'$ and $P'\ \mathcal{R}\ Q')$.

$(B2)$ If $P \searrow \overline{s}$ and $P \Downarrow_L$ then $\exists Q'$ $(Q \stackrel{\tau}{\Rightarrow} Q', Q' \searrow \overline{s},$ and $P\ \mathcal{R}\ Q')$.

$(B3)$ If $P \downarrow$ and $P \xrightarrow{N} P''$ then $\exists Q', Q''$ $(Q \stackrel{\tau}{\Rightarrow} Q', Q' \downarrow, P\ \mathcal{R}\ Q', Q' \xrightarrow{N} Q'',$ and $P''\ \mathcal{R}\ Q'')$.

We denote with $\approx_B$ the largest barbed bisimulation.

Definition 9 (contextual bisimulation) *A symmetric relation $\mathcal{R}$ on programs is a contextual bisimulation if it is a barbed bisimulation (conditions $(B1-3)$) and moreover whenever $P\ \mathcal{R}\ Q$ then*

$(C1)$ $C[P]\ \mathcal{R}\ C[Q]$, for any static context C.

We denote with $\approx_C$ the largest contextual barbed bisimulation.

We arrive at the announced characterisation of the labelled bisimulation.

Theorem 10 (characterisation of labelled bisimulation) *If P, Q are reactive programs then $P \approx Q$ if and only if $P \approx_C Q$.*

The proof of this result takes several steps summarised in Table 2 which provides 3 *equivalent* formulations of the labelled bisimulation $\approx$. In [2], the contextual bisimulation in definition 9 is characterised as a variant of the bisimulation $\approx_3$ where the condition for the output is formulated as follows:

$$\frac{P\ \mathcal{R}\ Q, \qquad P \Downarrow_L, \qquad P \xrightarrow{\nu \mathbf{t}\ \overline{s}v}_2 P', \qquad \{\mathbf{t}\} \cap fn(Q) = \emptyset}{Q \stackrel{\nu \mathbf{t}\ \overline{s}v}{\Rightarrow}_2 Q', \quad P'\ \mathcal{R}\ Q'}$$

Clearly, if P is a reactive program then $P \Downarrow_L$. Also note that the definition 2 of reactive program refers to the labelled transition system 1 for which it holds that $P \xrightarrow{sv} (P \mid \overline{s}v)$. Therefore, if P is reactive then $(P \mid \overline{s}v)$ is reactive too and if we start comparing two reactive programs then all programs that have to be considered in the bisimulation game will be reactive too. This means that on reactive programs the condition $P \Downarrow_L$ is always satisfied and therefore that the bisimulation $\approx_3$ coincides with the labelled bisimulation considered in [2].[4]

Remark 11 (on determinacy and divergence) *One may notice that the notions of labelled bisimulation and contextual bisimulation we have adopted are only* partially *sensitive to divergence. Let $\Omega = \tau.\Omega$ be a looping program. Then $\Omega \not\approx_C 0$ since 0 may suspend while Ω*

[4] On non-reactive programs, labelled bisimulation makes more distinctions than contextual bisimulation. For instance, the latter identifies all the programs that do not L-suspend.

	Labelled transition systems		Bisimulation game
$(\xrightarrow{\alpha}_1)$	Rule (in_{aux}) replaced by (in^1_{aux}) $\dfrac{}{s(x).P, K \xrightarrow{s?v} [v/x]P \mid \overline{s}v}$	$(\approx_1)$	As in definition 3
$(\xrightarrow{\alpha}_2)$	Rule (in) removed and action $s?v$ replaced by sv	$(\approx_2)$	As above if $\alpha \neq sv$. Require: (Inp) $\dfrac{P \mathcal{R} Q}{(P \mid \overline{s}v) \mathcal{R} (Q \mid \overline{s}v)}$
	As above	$(\approx_3)$	As above if $\alpha \neq sv$. Replace (Inp) with : $\dfrac{P \mathcal{R} Q, \quad P \xrightarrow{sv}_2 P'}{\exists Q' \ (\ Q \stackrel{sv}{\Rightarrow}_2 Q' \wedge P' \mathcal{R} Q') \vee (Q \stackrel{\tau}{\Rightarrow}_2 Q' \wedge P' \mathcal{R} (Q' \mid \overline{s}v)\ )}$ and for $\alpha = N$ require: $\dfrac{P \mathcal{R} Q,\ (P \mid S) \xrightarrow{N} P',\ S = \overline{s}_1 v_1 \mid \cdots \mid \overline{s}_n v_n}{\exists Q', Q'' \ (\ (Q \mid S) \stackrel{\tau}{\Rightarrow}_2 Q'', \quad (P \mid S) \mathcal{R} Q'', \ Q'' \xrightarrow{N}_2 Q', \quad P' \mathcal{R} Q'\ )}$

Table 2: Equivalent formulations of labelled bisimulation

may not. On the other hand, consider a program such as $A = \tau.A \oplus \tau.0$. *Then* $A \approx 0$ *and therefore* $A \approx_C 0$ *and we are lead to conclude that* A *is a* determinate *program. However, one may also argue that* A *is* not *determinate since it may either suspend or loop. In other words, determinacy depends on the notion of semantic equivalence we adopt. If the latter is not sensitive enough to divergence then the resulting notion of determinacy should be regarded as a* partial *property of programs,* i.e.*, it holds provided programs terminate. In practice, these distinctions do not seem very important because, as we have already argued,* reactivity *is a property one should always require of synchronous programs and once reactivity is in place the distinctions disappear.*

5 Determinacy and (local) confluence

In this section, we develop the notions of determinacy and confluence for the $S\pi$-calculus which turn out to coincide. Moreover, we note that for reactive programs a simple property of local confluence suffices to ensure determinacy.

We denote with ϵ the empty sequence and with $s = \alpha_1 \cdots \alpha_n$ a finite sequence (possibly empty) of actions different from τ. We define:

$$\stackrel{s}{\Rightarrow} = \begin{cases} \stackrel{\tau}{\Rightarrow} & \text{if } s = \epsilon \\ \stackrel{\alpha_1}{\Rightarrow} \cdots \stackrel{\alpha_n}{\Rightarrow} & \text{if } s = \alpha_1 \cdots \alpha_n \end{cases}$$

Thus s denotes a finite (possibly empty) sequence of interactions with the environment.

Following [17], a program is considered determinate if performing twice the same sequence of interactions leads to the same program up to semantic equivalence.

Definition 12 (determinacy) *We say that a program P is determinate if for every sequence s, if $P \stackrel{s}{\Rightarrow} P_i$ for $i = 1, 2$ then $P_1 \approx P_2$.*

Determinacy implies τ-inertness which is defined as follows.

Definition 13 (τ-inertness) *A program is τ-inert if for all its derivatives Q, $Q \stackrel{\tau}{\rightarrow} Q'$ implies $Q \approx Q'$.*

Next, we turn to the notion of confluence. To this end, we introduce first the notions of action compatibility and action residual.

Definition 14 (action compatibility) *The* compatibility predicate $\downarrow$ *is defined as the least reflexive and symmetric binary relation on actions such that $\alpha \downarrow \beta$ implies that either $\alpha, \beta \neq N$ or $\alpha = \beta = N$.*

In other words, the action N is only compatible with itself while any action different from N is compatible with any other action different from N.[5] Intuitively, confluence is about the possibility of commuting actions that happen in the *same instant*. To make this precise we also need to introduce a notion of action residual $\alpha\backslash\beta$ which specifies what remains of the action α once the action β is performed.

Definition 15 (action residual) *The residual operation $\alpha\backslash\beta$ on actions is only defined if $\alpha \downarrow \beta$ and in this case it satisfies:*

$$\alpha\backslash\beta = \begin{cases} \tau & \text{if } \alpha = \beta \\ \nu\mathbf{t}\backslash\mathbf{t}'\overline{s}v & \text{if } \alpha = \nu\mathbf{t}\ \overline{s}v \text{ and } \beta = \nu\mathbf{t}'\overline{s'}v' \\ \alpha & \text{otherwise} \end{cases}$$

Confluence is then about closing diagrams of compatible actions up to residuals and semantic equivalence.

Definition 16 (confluence) *We say that a program P is confluent, if for all its derivatives Q:*

$$\frac{Q \stackrel{\alpha}{\Rightarrow} Q_1, \quad Q \stackrel{\beta}{\Rightarrow} Q_2, \quad \alpha \downarrow \beta}{\exists\, Q_3, Q_4 \ (\ Q_1 \stackrel{\beta\backslash\alpha}{\Rightarrow} Q_3, \quad Q_2 \stackrel{\alpha\backslash\beta}{\Rightarrow} Q_4, \quad Q_3 \approx Q_4\ )}$$

It often turns out that the following weaker notion of *local* confluence is much easier to establish.

[5] The reader familiar with [20] will notice that, unlike in the π-calculus with *rendez-vous* communication, we do not restrict the compatibility relation on input actions. This is because of the particular form of the input action in the labelled transition system in table 1 where the input action does not actually force a program to perform an input. We expect that a similar situation would arise in the π-calculus with asynchronous communication.

Definition 17 (local confluence) *We say that a program is locally confluent, if for all its derivatives Q:*

$$\frac{Q \xrightarrow{\alpha} Q_1 \quad Q \xrightarrow{\beta} Q_2 \quad \alpha \downarrow \beta}{\exists\, Q_3, Q_4 \;\; (\; Q_1 \stackrel{\beta\backslash\alpha}{\Rightarrow} Q_3, \quad Q_2 \stackrel{\alpha\backslash\beta}{\Rightarrow} Q_4, \quad Q_3 \approx Q_4 \;)}$$

It is easy to produce programs which are locally confluent but not confluent. For instance, $A = \overline{s}_1 \oplus B$ where $B = \overline{s}_2 \oplus A$. However, one may notice that this program is *not* reactive. Indeed, for reactive programs local confluence is equivalent to confluence.

Theorem 18 (1) *A program is determinate if and only if it is confluent.*
(2) *A reactive program is determinate if and only if for all its derivatives Q:*

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

The fact that confluent programs are determinate is standard and it essentially follows from the observation that confluent programs are τ-inert. The observation that determinate programs are confluent is specific of the $S\pi$-calculus and it depends on the remark that input and output actions automatically commute with the other compatible actions.[6]

The part (2) of the theorem is proved as follows. First one notices that the stated conditions are equivalent to local confluence (again relying on the fact that commutation of input and output actions is automatic) and then following [11] one observes that local confluence plus reactivity entails confluence.

We conclude this section by noticing a strong commutation property of τ actions that suffices to entail τ-inertness and determinacy. Let $\stackrel{\alpha}{\rightsquigarrow}$ be $\xrightarrow{\alpha} \cup Id$ where Id is the identity relation.

Proposition 19 *A program is determinate if for all its derivatives Q:*

$$\frac{Q \xrightarrow{\tau} Q_1, \quad Q \xrightarrow{\tau} Q_2}{\exists\, Q' \;\; (Q_1 \stackrel{\tau}{\rightsquigarrow} Q', \quad Q_2 \stackrel{\tau}{\rightsquigarrow} Q')} \qquad \frac{Q \xrightarrow{N} Q_1, \quad Q \xrightarrow{N} Q_2}{Q_1 \approx Q_2}$$

This is proven by showing that the strong commutation of the τ-actions entails τ-inertness.

6 Conclusion

We have developed a framework to analyse the determinacy of programs in a *synchronous* π-calculus. First, we have introduced a compositional notion of labelled bisimulation. Second, we have characterised a relevant contextual bisimulation as a standard bisimulation over a modified labelled transition system. Third, we have studied the notion of confluence which turns out to be equivalent to determinacy, and we have shown that under reactivity, confluence reduces to a simple form of local confluence.

[6] We note that the commutation of the inputs arises in the π-calculus with asynchronous communication too, while the commutation of the outputs is due to the fact that messages on signals unlike messages on channels persist within an instant (for instance, in CCS, if $P = \overline{a} \mid a.\overline{b}$ then $P \xrightarrow{\overline{a}} a.\overline{b}$, $P \xrightarrow{\tau} \overline{b}$, and there is no way to close the diagram).

According to theorem 18(2), there are basically two situations that need to be analysed in order to guarantee the determinacy of (reactive) programs. (1) At least two distinct values compete to be received within an instant, for instance, consider: $\overline{s}v_1 \mid \overline{s}v_2 \mid s(x).P, K$. (2) At the end of the instant, at least two distinct values are available on a signal. For instance, consider: $\overline{s}v_1 \mid \overline{s}v_2 \mid \mathsf{pause}.A(!s)$. Based on this analysis, we are currently studying an *affine* type system in the style of [15] that avoids completely the first situation and allows the second provided the behaviour of the continuation A does not depend on the order in which the values are collected.

References

[1] R. Amadio. The SL synchronous language, revisited. *Journal of Logic and Algebraic Programming*, 70:121-150, 2007.

[2] R. Amadio. A synchronous π-calculus. *Information and Computation*, 205(9):1470–1490, 2007.

[3] R. Amadio, G. Boudol, F. Boussinot and I. Castellani. Reactive programming, revisited. In Proc. Workshop on *Algebraic Process Calculi: the first* 25 *years and beyond*, *Electronic Notes in Theoretical Computer Science*, 162:49-60, 2006.

[4] R. Amadio, I. Castellani and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In *Theoretical Computer Science*, 195:291-324, 1998.

[5] R. Amadio, F. Dabrowski. Feasible reactivity in a synchronous π-calculus. In Proc. ACM SIGPLAN Symp. on Principles and Practice of Declarative Programming, 2007.

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language. *Science of computer programming*, 19(2):87–152, 1992.

[7] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[8] F. Boussinot and R. De Simone. The SL synchronous language. *IEEE Trans. on Software Engineering*, 22(4):256–266, 1996.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In Proc. ACM-POPL, pp 178-188, 1987.

[10] S. Edwards and O. Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. IEEE Transactions on Very Large Scale Integration Systems, 14(8), 2006.

[11] J. Groote, M. Sellink. Confluence for process verification. Theor. Comput. Sci. 170(1-2):47-81, 1996.

[12] K. Honda and M. Tokoro. On asynchronous communication semantics. In *Object-based concurrent computing*, SLNCS 612, 1992.

[13] K. Honda and N. Yoshida. On reduction-based process semantics. In *Theoretical Computer Science*, 151(2):437-486, 1995.

[14] G. Kahn. The semantics of simple language for parallel programming. IFIP Congress, 1974.

[15] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the π-calculus. ACM Transactions on Programming Languages and Systems (TOPLAS), 21(5), 1999.

[16] L. Mandel and M. Pouzet. ReactiveML, a reactive extension to ML. In *Proc. ACM Principles and Practice of Declarative Programming*, pages 82–93, 2005.

[17] R. Milner. Communication and concurrency. Prentice-Hall, 1989.

[18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts 1-2. *Information and Computation*, 100(1):1–77, 1992.

[19] M. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[20] A. Philippou and D. Walker. On confluence in the π-calculus. In Proc. ICALP, pp 314-324, SLNCS 1256, 1997.

[21] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. In *Journal of Symbolic computation*, 22(5,6) 475-520, 1996.

[22] M. Serrano, F. Boussinot, and B. Serpette. Scheme fair threads. In *Proc. ACM Principles and practice of declarative programming*, pages 203-214, 2004.

[23] D. Sangiorgi and D. Walker. The π-calculus. Cambridge University Press, 2001.

A Basic properties of labelled bisimulation

We collect some basic properties of the notion of labelled bisimulation. First, we consider a standard variation of the definition 3 of bisimulation where transitions are weak on both sides of the bisimulation game.

Definition 20 (w-bisimulation) *A symmetric relation $\mathcal{R}$ on programs is a w-bisimulation if*

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

We denote with $\approx_w$ the largest w-bisimulation.

With respect to this modified definition we introduce the usual notion of bisimulation up to bisimulation.[7]

Definition 21 (w-bisimulation up to w-bisimulation) *A symmetric relation $\mathcal{R}$ on programs is a w-bisimulation up to w-bisimulation if*

$$\frac{P\ \mathcal{R}\ Q, \quad P \stackrel{\alpha}{\Rightarrow} P', \quad bn(\alpha) \cap fn(Q) = \emptyset}{\exists\, Q'\ (\ Q \stackrel{\alpha}{\Rightarrow} Q', \quad P' \approx_w \circ\, \mathcal{R} \,\circ \approx_w Q'\ )}$$

We denote with $\approx_w$ the largest w-bisimulation.

Proposition 22 (1) *The relation $\approx$ is an equivalence relation.*

(2) *The relations $\approx$ and $\approx_w$ coincide.*

(3) *If $\mathcal{R}$ is a w-bisimulation up to w-bisimulation then $\mathcal{R} \subseteq \approx_w$.*

Proof. (1) The identity relation is a labelled bisimulation and the union of symmetric relations is symmetric. To check transitivity, we prove that $\approx \circ \approx$ is a labelled bisimulation by standard diagram chasing.

(2) By definition a w-bisimulation is a labelled bisimulation, therefore $\approx_w \subseteq \approx$. To show the other inclusion, prove that $\approx$ is a w-bisimulation again by a standard diagram chasing.

(3) First note that by (1) and (2), it follows that the relation $\approx_w$ is transitive. Then one shows that if $\mathcal{R}$ is a w-bisimulation up to w-bisimulation then the relation $\approx_w \circ \mathcal{R} \circ \approx_w$ is a w-bisimulation. □

[7] We recall that it is important that this notion is defined with respect to w-bisimulation. Indeed, proposition 22(3) below fails if w-bisimulation is replaced by bisimulation.

A.1 Structural equivalence

In the diagram chasing arguments, it will be convenient to consider programs up to a notion of 'structural equivalence'. This is the least equivalence relation $\equiv$ such that (1) $\equiv$ is preserved by static contexts, (2) parallel composition is associative and commutative, (3) $\nu s\ (P \mid Q) \equiv \nu s\ P \mid Q$ if $s \notin fn(Q)$, (4) $\overline{s}v \mid \overline{s}v \equiv \overline{s}v$, and (5) $\overline{s}e \equiv \overline{s}v$ if $e \Downarrow v$. One can check for the different labelled transition systems we consider that equivalent programs generate exactly the same transitions and that the programs to which they reduce are again equivalent.

B Proof of theorem 5

The theorem follows directly from the following lemma 23(4).

Lemma 23 (1) *If $P_1 \approx P_2$ and σ is an injective renaming then $\sigma P_1 \approx \sigma P_2$.*

(2) *The relation $\approx$ is reflexive and transitive.*

(3) *If $P_1 \approx P_2$ then $(P_1 \mid \overline{s}v) \approx (P_2 \mid \overline{s}v)$.*

(4) *If $P_1 \approx P_2$ then $\nu s\ P_1 \approx \nu s\ P_2$ and $(P_1 \mid Q) \approx (P_2 \mid Q)$.*

Proof. **(1), (2)** Standard arguments.

(3) Let $\mathcal{R}' = \{((P \mid \overline{s}v), (Q \mid \overline{s}v)) \mid P \approx Q\}$ and $\mathcal{R} = \mathcal{R}' \cup \approx$. We show that $\mathcal{R}$ is a bisimulation. Suppose $(P \mid \overline{s}v) \xrightarrow{\alpha} \cdot$ and $P \approx Q$. There are two interesting cases to consider.

$(\alpha = \tau)$ Suppose $(P \mid \overline{s}v) \xrightarrow{\tau} (P' \mid \overline{s}v)$ because $P \xrightarrow{s?v} P'$. By definition of the lts, we have that $P \xrightarrow{sv} (P \mid \overline{s}v) \xrightarrow{\tau} (P' \mid \overline{s}v)$. By definition of bisimulation, $Q \overset{sv}{\Rightarrow} (Q'' \mid \overline{s}v) \overset{\tau}{\Rightarrow} (Q' \mid \overline{s}v)$ and $(P' \mid \overline{s}v) \approx (Q' \mid \overline{s}v)$. We conclude, by noticing that then $(Q \mid \overline{s}v) \overset{\tau}{\Rightarrow} (Q' \mid \overline{s}v)$.

$(\alpha = N)$ Suppose $(P \mid \overline{s}v) \xrightarrow{N} P'$. Notice that $P \xrightarrow{sv} (P \mid \overline{s}v)$. Hence:

$$Q \overset{sv}{\Rightarrow} (Q'' \mid \overline{s}v) \overset{\tau}{\Rightarrow} (Q''' \mid \overline{s}v) \xrightarrow{N} Q', \quad (P \mid \overline{s}v) \approx (Q'' \mid \overline{s}v) \approx (Q''' \mid \overline{s}v), \quad \text{and} \quad P' \approx Q' \ .$$

Then $(Q \mid \overline{s}v) \overset{N}{\Rightarrow} Q'$.

(4) We show that $\mathcal{R} = \{(\nu \mathbf{t}\ (P_1 \mid Q), \nu \mathbf{t}\ (P_2 \mid Q)) \mid P_1 \approx P_2\} \cup \approx$ is a labelled bisimulation up to the structural equivalence $\equiv$.

(τ) Suppose $\nu \mathbf{t}\ (P_1 \mid Q) \xrightarrow{\tau} \cdot$. This may happen because either P_1 or Q perform a τ action or because P_1 and Q synchronise. We analyse the various situations.

$(\tau)[1]$ Suppose $Q \xrightarrow{\tau} Q'$. Then $\nu \mathbf{t}\ (P_2 \mid Q) \xrightarrow{\tau} \nu \mathbf{t}\ (P_2 \mid Q')$ and we can conclude.

$(\tau)[2]$ Suppose $P_1 \xrightarrow{\tau} P_1'$. Then $P_2 \overset{\tau}{\Rightarrow} P_2'$ and $P_1' \approx P_2'$. So $\nu \mathbf{t}\ (P_2 \mid Q) \overset{\tau}{\Rightarrow} \nu \mathbf{t}\ (P_2' \mid Q)$ and we can conclude.

$(\tau)[3]$ Suppose $P_1 \xrightarrow{s?v} P_1'$ and $Q \xrightarrow{\nu \mathbf{t}'\ \overline{s}v} Q'$. This means $Q \equiv \nu \mathbf{t}'\ (\overline{s}v \mid Q'')$ and $Q' \equiv (\overline{s}v \mid Q'')$. By (3), $(P_1 \mid \overline{s}v) \approx (P_2 \mid \overline{s}v)$. Moreover, $(P_1 \mid \overline{s}v) \xrightarrow{\tau} (P_1' \mid \overline{s}v)$. Therefore, $(P_2 \mid \overline{s}v) \overset{\tau}{\Rightarrow} (P_2' \mid \overline{s}v)$ and $(P_1' \mid \overline{s}v) \approx (P_2' \mid \overline{s}v)$. Then we notice that the transition $\nu \mathbf{t}\ (P_1 \mid Q) \xrightarrow{\tau} \cdot \equiv \nu \mathbf{t}, \mathbf{t}'\ ((P_1' \mid \overline{s}v) \mid Q'')$ is matched by the transition $\nu \mathbf{t}\ (P_2 \mid Q) \xrightarrow{\tau} \cdot \equiv \nu \mathbf{t}, \mathbf{t}'\ ((P_2' \mid \overline{s}v) \mid Q'')$.

$(\tau)[4]$ Suppose $P_1 \xrightarrow{\nu \mathbf{t}'\ \overline{s}v} P_1'$ and $Q \xrightarrow{s?v} Q'$. Then $P_2 \overset{\nu \mathbf{t}'\ \overline{s}v}{\Rightarrow} P_2'$ and $P_1' \approx P_2'$. And we conclude noticing that $\nu \mathbf{t}\ (P_2 \mid Q) \overset{\tau}{\Rightarrow} \nu \mathbf{t}, \mathbf{t}'\ (P_2' \mid Q')$.

(*out*) Suppose $\nu\mathbf{t}\ (P_1 \mid Q) \xrightarrow{\nu\mathbf{t}'\ \overline{s}v} \cdot$. Also assume $\mathbf{t} = \mathbf{t_1}, \mathbf{t_2}$ and $\mathbf{t}' = \mathbf{t_1}, \mathbf{t_3}$ up to reordering so that the emission extrudes exactly the names $\mathbf{t}_1$ among the names in $\mathbf{t}$. We have two subcases depending which component performs the action.

(*out*)[1] Suppose $Q \xrightarrow{\nu\mathbf{t_3}\ \overline{s}v} Q'$. Then $\nu\mathbf{t}\ (P_2 \mid Q) \xrightarrow{\nu\mathbf{t}'\ \overline{s}v} \nu\mathbf{t_2}\ (P_2 \mid Q')$ and we can conclude.

(*out*)[2] Suppose $P_1 \xrightarrow{\nu\mathbf{t_3}\ \overline{s}v} P_1'$. Then $P_2 \overset{\nu\mathbf{t_3}\ \overline{s}v}{\Rightarrow} P_2'$ and $P_1' \approx P_2'$. Hence $\nu\mathbf{t}\ (P_2 \mid Q) \overset{\nu\mathbf{t}'\ \overline{s}v}{\Rightarrow} \nu\mathbf{t_2}\ (P_2' \mid Q)$ and we can conclude.

(*in*) It is enough to notice that, modulo renaming, $\nu\mathbf{t}\ (P_i \mid Q) \mid \overline{s}v \equiv \nu\mathbf{t}\ ((P_i \mid \overline{s}v) \mid Q)$ and recall that by (3), $(P_1 \mid \overline{s}v) \approx (P_2 \mid \overline{s}v)$.

(N) Suppose $\nu\mathbf{t}\ (P_1 \mid Q)\downarrow$. Up to structural equivalence, we can express Q as $\nu\mathbf{t}_Q\ (S_Q \mid I_Q)$ where S_Q is the parallel composition of emissions and I_Q is the parallel composition of receptions. Thus we have: $\nu\mathbf{t}\ (P_1 \mid Q) \equiv \nu\mathbf{t}, \mathbf{t}_Q\ (P_1 \mid S_Q \mid I_Q)$, and $\nu\mathbf{t}\ (P_2 \mid Q) \equiv \nu\mathbf{t}, \mathbf{t}_Q\ (P_2 \mid S_Q \mid I_Q)$ assuming $\{\mathbf{t}_Q\} \cap fn(P_i) = \emptyset$ for $i = 1, 2$.

If $\nu\mathbf{t}\ (P_1 \mid Q) \xrightarrow{N} P$ then $P \equiv \nu\mathbf{t}, \mathbf{t}_Q\ (P_1'' \mid Q')$ where in particular, we have that $(P_1 \mid S_Q)\downarrow$ and $(P_1 \mid S_Q) \xrightarrow{N} (P_1' \mid 0)$.

By the hypothesis $P_1 \approx P_2$, and by definition of bisimulation we derive that: (i) $(P_2 \mid S_Q) \overset{\tau}{\Rightarrow} (P_2'' \mid S_Q)$, (ii) $(P_2'' \mid S_Q)\downarrow$, (iii) $(P_2'' \mid S_Q) \xrightarrow{N} (P_2' \mid 0)$, (iv) $(P_1 \mid S_Q) \approx (P_2'' \mid S_Q)$, and (v) $(P_1' \mid 0) \approx (P_2' \mid 0)$.

Because $(P_1 \mid S_Q)$ and $(P_2'' \mid S_Q)$ are suspended and bisimilar, the two programs must commit (cf. definition 7) on the same signal names and moreover on each signal name they must emit the same set of values up to renaming of bound names. It follows that the program $\nu\mathbf{t}, \mathbf{t}_Q\ (P_2'' \mid S_Q \mid I_Q)$ is suspended. The only possibility for an internal transition is that an emission in P_2'' enables a reception in I_Q but this contradicts the hypothesis that $\nu\mathbf{t}, \mathbf{t}_Q\ (P_1 \mid S_Q \mid I_Q)$ is suspended. Moreover, $(P_2'' \mid S_Q \mid I_Q) \xrightarrow{N} (P_2' \mid 0 \mid Q')$.

Therefore, we have that

$$\nu\mathbf{t}\ (P_2 \mid Q) \equiv \nu\mathbf{t}, \mathbf{t}_Q\ (P_2 \mid S_Q \mid I_Q) \overset{\tau}{\Rightarrow} \nu\mathbf{t}, \mathbf{t}_Q\ (P_2'' \mid S_Q \mid I_Q),$$

$\nu\mathbf{t}, \mathbf{t}_Q\ (P_2'' \mid S_Q \mid I_Q)\downarrow$, and $\nu\mathbf{t}, \mathbf{t}_Q\ (P_2'' \mid S_Q \mid I_Q) \xrightarrow{N} \nu\mathbf{t}, \mathbf{t}_Q\ (P_2' \mid 0 \mid Q')$. Now $\nu\mathbf{t}, \mathbf{t}_Q\ (P_1 \mid S_Q \mid I_Q)\ \mathcal{R}\ \nu\mathbf{t}, \mathbf{t}_Q\ (P_2'' \mid S_Q \mid I_Q)$ because $(P_1 \mid S_Q) \approx (P_2'' \mid S_Q)$ and $\nu\mathbf{t}, \mathbf{t}_Q\ (P_1' \mid Q')\ \mathcal{R}\ \nu\mathbf{t}, \mathbf{t}_Q\ (P_2' \mid Q')$ because $P_1' \approx P_2'$. □

C Proof of theorem 10

We start with the labelled transition system defined in table 1 and the notion of bisimulation in definition 3. In table 2, we incrementally modify the labelled transition system and/or the conditions in the bisimulation game. This leads to three equivalent characterisations of the notion of bisimulation. We prove this fact step by step.

Lemma 24 *The bisimulation $\approx$ coincides with the bisimulation $\approx_1$.*

Proof. The only difference here is in the rule (in_{aux}), the bisimulation conditions being the same. Now this rule produces an action $s?v$ and the latter is an auxiliary action that is used to produce the relevant action τ thanks to the rule (*synch*). A simple instance of the difference follows. Suppose $P = \overline{s}e \mid s(x).Q, K$ and $e \Downarrow v$. Then:

$$P \xrightarrow{\tau} \overline{s}e \mid [v/x]Q = P' \text{ and } P \xrightarrow{\tau}_1 \overline{s}e \mid ([v/x]Q \mid \overline{s}v) = P''\ .$$

In the $S\pi$-calculus, we do not distinguish the situations where the same value is emitted once or more times within the same instant. In particular, P' and P'' are structurally equivalent (cf. section A.1). □

Next, we focus on the relationships between the labelled transitions systems $\xrightarrow{act}_1$ and $\xrightarrow{act}_2$. In $\xrightarrow{act}_2$, the rule (in) is removed and in the rule (in_{aux}), the label $s?v$ is replaced by the label sv (hence the auxiliary action $s?v$ is not used in this labelled transition system).

Lemma 25 (1) *If $P \xrightarrow{act}_1 P'$ and $act \neq sv$ then $P \xrightarrow{act'}_2 P'$ where $act' = sv$ if $act = s?v$, and $act' = act$ otherwise.*
(2) *If $P \xrightarrow{act}_2 P'$ then $P \xrightarrow{act'}_1 P'$ where $act' = s?v$ if $act = sv$, and $act' = act$ otherwise.*

We also notice that 1-bisimulation is preserved by parallel composition with an emission; the proof is similar to the one of lemma 23(3).

Lemma 26 *If $P \approx_1 Q$ then $(P \mid \overline{s}v) \approx_1 (Q \mid \overline{s}v)$.*

Lemma 27 *The bisimulation $\approx_1$ coincides with the bisimulation $\approx_2$.*

Proof. ($\approx_1 \subseteq \approx_2$) We check that $\approx_1$ is a 2-bisimulation. If $\alpha = sv$ then we apply lemma 26. Otherwise, suppose $\alpha \neq sv$, $P \approx_1 Q$, and $P \xrightarrow{\alpha}_2 P'$. By lemma 25(2), $P \xrightarrow{\alpha}_1 P'$. By definition of 1-bisimulation, $\exists\, Q'\ Q \stackrel{\alpha}{\Rightarrow}_1 Q', P' \approx_1 Q'$. By lemma 25(1), $Q \stackrel{\alpha}{\Rightarrow}_2 Q'$.

($\approx_2 \subseteq \approx_1$) We check that $\approx_2$ is a 1-bisimulation. If $\alpha = sv$ and $P \xrightarrow{sv}_1 (P \mid \overline{s}v)$ then by definition of the lts, $Q \xrightarrow{sv}_1 (Q \mid \overline{s}v)$. Moreover, by definition of 2-bisimulation, $(P \mid \overline{s}v) \approx_2 (Q \mid \overline{s}v)$. Otherwise, suppose $\alpha \neq sv$, $P \approx_2 Q$, and $P \xrightarrow{\alpha}_1 P'$. By lemma 25(1), $P \xrightarrow{\alpha}_2 P'$. By definition of 2-bisimulation, $\exists\, Q'\ Q \stackrel{\alpha}{\Rightarrow}_2 Q', P' \approx_2 Q'$. By lemma 25(2), $Q \stackrel{\alpha}{\Rightarrow}_1 Q'$. □

Next we move to a comparison of 2 and 3 bisimulations. Note that both definitions share the same lts denoted with $\xrightarrow{\alpha}_2$. First we remark the following.

Lemma 28 (1) *If $P \approx_2 Q$ and $P \xrightarrow{N} P'$ then $\exists\, Q', Q''$ ($Q \stackrel{\tau}{\Rightarrow}_2 Q'', Q'' \xrightarrow{N} Q', P \approx_2 Q'', P' \approx_2 Q'$).*
(2) *If $P \approx_3 Q$ then $(P \mid \overline{s}v) \approx_3 (Q \mid \overline{s}v)$.*

Proof. (1) If $P \xrightarrow{N} P'$ then P cannot perform τ moves. Thus if $P \approx_2 Q$ and $Q \stackrel{\tau}{\Rightarrow}_2 Q''$ then necessarily $P \approx_2 Q''$.

(2) Again we follow the proof of lemma 23(3). Let $\mathcal{R}' = \{((P \mid \overline{s}v), (Q \mid \overline{s}v)) \mid P \approx_3 Q\}$ and $\mathcal{R} = \mathcal{R}' \cup \approx_3$. We show that $\mathcal{R}$ is a 3-bisimulation. Suppose $(P \mid \overline{s}v) \xrightarrow{\alpha}_1 \cdot$ and $P \approx_3 Q$. There are two interesting cases to consider.

($\alpha = \tau$) Suppose $(P \mid \overline{s}v) \xrightarrow{\tau}_2 (P' \mid \overline{s}v)$ because $P \xrightarrow{sv}_2 P'$. By definition of 3-bisimulation, either (i) $Q \stackrel{sv}{\Rightarrow}_2 Q'$ and $P' \approx_3 Q'$ or (ii) $Q \stackrel{\tau}{\Rightarrow}_2 Q'$ and $P' \approx_3 (Q' \mid \overline{s}v)$. In case (i), $(Q \mid \overline{s}v) \stackrel{\tau}{\Rightarrow} (Q' \mid \overline{s}v)$ and we notice that $((P' \mid \overline{s}v), (Q' \mid \overline{s}v)) \in \mathcal{R}$. In case (ii), $(Q \mid \overline{s}v) \stackrel{\tau}{\Rightarrow} (Q' \mid \overline{s}v)$ and we notice that $(P' \mid \overline{s}v, (Q' \mid \overline{s}v) \mid \overline{s}v) \in \mathcal{R}$ and $(Q' \mid \overline{s}v) \mid \overline{s}v \equiv (Q' \mid \overline{s}v)$.

($\alpha = N$) Suppose $((P \mid \overline{s}v) \mid S) \xrightarrow{N} P'$. By definition of 3-bisimulation, taking $S' = (\overline{s}v \mid S)$ $(Q \mid S') \stackrel{\tau}{\Rightarrow} Q'' \xrightarrow{N} Q'$, $(P \mid S') \approx_3 Q''$, and $P' \approx_3 Q'$. □

Lemma 29 *The bisimulation $\approx_2$ coincides with the bisimulation $\approx_3$.*

PROOF. ($\approx_2 \subseteq \approx_3$) We show that $\approx_2$ is a 3-bisimulation. We look first at the condition for the input. Suppose $P \approx_2 Q$ and $P \xrightarrow{sv}_2 P'$. By definition of 2-bisimulation, $(P \mid \overline{s}v) \approx_2 (Q \mid \overline{s}v)$. Also $(P \mid \overline{s}v) \xrightarrow{\tau}_2 (P' \mid \overline{s}v) \equiv P'$. By definition of 2-bisimulation, $(Q \mid \overline{s}v) \stackrel{\tau}{\Rightarrow} (Q' \mid \overline{s}v)$ and $P' \equiv (P' \mid \overline{s}v) \approx_2 (Q' \mid \overline{s}v)$. Two cases may arise.

(1) If $Q \stackrel{sv}{\Rightarrow} Q'$ then $Q' \mid \overline{s}v \equiv Q'$ and we satisfy the first case of the input condition for 3-bisimulation.

(2) If $Q \stackrel{\tau}{\Rightarrow} Q'$ then, up to structural equivalence, we satisfy the second case of the input condition for 3-bisimulation.

Next we consider the condition for the end of the instant. Suppose $P \approx_2 Q$, $S = \overline{s_1}v_1 \mid \cdots \mid \overline{s_n}v_n$, and $(P \mid S) \xrightarrow{N}_2 P'$. By condition (*Inp*), $(P \mid S) \approx_2 (Q \mid S)$. Then, by lemma 28(1), the condition of 3-bisimulation is entailed by the corresponding condition for 2-bisimulation applied to $(P \mid S)$ and $(Q \mid S)$.

($\approx_3 \subseteq \approx_2$) We show that $\approx_3$ is a 2-bisimulation. The condition (*Inp*) holds because of lemma 28(2). The condition of 2-bisimulation for the end of the instant is a special case of the condition for 3-bisimulation where we take S empty. □

D Proof of theorem 18 and proposition 19

First, relying on proposition 22(3), one can repeat the proof in [17] that confluence implies τ-inertness and determinacy.

Proposition 30 *If a program is confluent then it is τ-inert and determinate.*

PROOF. Let $\mathcal{S} = \{(P, P') \mid P \text{ confluent and } P \stackrel{\tau}{\Rightarrow} P'\}$ and define $\mathcal{R} = \mathcal{S} \cup \mathcal{S}^{-1}$. We show that $\mathcal{R}$ is a w-bisimulation up to w-bisimulation (cf. lemma 22(3)). Clearly $\mathcal{R}$ is symmetric. Then suppose P confluent and $P \stackrel{\tau}{\Rightarrow} Q$ (the case where Q reduces to P is symmetric). If $Q \stackrel{\alpha}{\Rightarrow} Q_1$ then $P \stackrel{\alpha}{\Rightarrow} Q_1$ and $Q_1 \; \mathcal{R} \; Q_1$. On the other hand, if $P \stackrel{\alpha}{\Rightarrow} P_1$ then by confluence there are P_2, Q_1 such that $P_1 \stackrel{\tau}{\Rightarrow} P_2$, $Q \stackrel{\alpha}{\Rightarrow} Q_1$, and $P_2 \approx Q_1$. Thus $P_1 \; \mathcal{R} \circ \approx Q_1$.

Therefore if P is confluent and $P \stackrel{\tau}{\Rightarrow} P'$ then $P \approx P'$. Also recall that if Q is a derivative of P then Q is confluent. Thus we can conclude that if P is confluent then it is τ-inert.

Next, we show that:

$$\frac{P_1 \approx P_2, \quad P_1 \stackrel{\alpha}{\Rightarrow} P_3, \quad P_2 \stackrel{\alpha}{\Rightarrow} P_4}{P_3 \approx P_4} .$$

By definition of bisimulation, $\exists\, P_5$ ($P_2 \stackrel{\alpha}{\Rightarrow} P_5, P_3 \approx P_5$). By confluence, $\exists\, P_6, P_7$ ($P_5 \stackrel{\tau}{\Rightarrow} P_6, P_4 \stackrel{\tau}{\Rightarrow} P_7, P_6 \approx P_7$). By τ-inertness and transitivity, $P_3 \approx P_4$.

Finally, we can iterate this observation to conclude that if $P \stackrel{\alpha_1}{\Rightarrow} \cdots \stackrel{\alpha_n}{\Rightarrow} P_1$ and $P \stackrel{\alpha_1}{\Rightarrow} \cdots \stackrel{\alpha_n}{\Rightarrow} P_2$ then $P_1 \approx P_2$. □

We pause to point-out the particular properties of the input and output actions in the labelled transition system in table 1. It is easily verified that if $P \xrightarrow{\nu \mathbf{t} \overline{s}v} P'$ then $P \equiv \nu \mathbf{t}(\overline{s}v \mid P'')$ and $P' \equiv (\overline{s}v \mid P'')$. This entails that in the following lemma the cases that involve an output action are actually general up to structural equivalence.

Lemma 31 (input-output commutations)

$$(in-\tau)\qquad \frac{P \xrightarrow{s v} (P \mid \overline{s}v),\quad P \xrightarrow{\tau} P'}{(P \mid \overline{s}v) \xrightarrow{\tau} (P' \mid \overline{s}v),\quad P' \xrightarrow{s v} (P' \mid \overline{s}v)}$$

$$(in-in)\qquad \frac{P \xrightarrow{s v} (P \mid \overline{s}v),\quad P \xrightarrow{s' v'} (P \mid \overline{s'}v')}{\begin{array}{c}(P \mid \overline{s}v) \xrightarrow{s' v'} (P \mid \overline{s}v) \mid \overline{s'}v',\quad (P \mid \overline{s'}v') \xrightarrow{s v} (P \mid \overline{s'}v') \mid \overline{s}v,\\ (P \mid \overline{s}v) \mid \overline{s'}v' \equiv (P \mid \overline{s'}v') \mid \overline{s}v\end{array}}$$

$$(out-\tau)\qquad \frac{\nu\mathbf{t}(\overline{s}v \mid P) \xrightarrow{\nu\mathbf{t}\ \overline{s}v} (\overline{s}v \mid P),\quad \nu\mathbf{t}(\overline{s}v \mid P) \xrightarrow{\tau} \nu\mathbf{t}(\overline{s}v \mid P')}{(\overline{s}v \mid P) \xrightarrow{\tau} (\overline{s}v \mid P'),\quad \nu\mathbf{t}(\overline{s}v \mid P') \xrightarrow{\nu\mathbf{t}\ \overline{s}v} (\overline{s}v \mid P')}$$

$$(out-in)\qquad \frac{\nu\mathbf{t}(\overline{s}v \mid P) \xrightarrow{\nu\mathbf{t}\ \overline{s}v} (\overline{s}v \mid P),\quad \nu\mathbf{t}(\overline{s}v \mid P) \xrightarrow{s' v'} \nu\mathbf{t}(\overline{s}v \mid P) \mid \overline{s'}v'}{(\overline{s}v \mid P) \xrightarrow{s' v'} (\overline{s}v \mid P) \mid \overline{s'}v',\quad \nu\mathbf{t}(\overline{s}v \mid P) \mid \overline{s'}v' \xrightarrow{\nu\mathbf{t}\ \overline{s}v} (\overline{s}v \mid P) \mid \overline{s'}v'}$$

$$(out-out)\qquad \frac{\begin{array}{c}\nu\mathbf{t}(\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P) \xrightarrow{\nu\mathbf{t_1}\ \overline{s_1}v_1} \nu\mathbf{t}\backslash\mathbf{t_1}\ (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P),\\ \nu\mathbf{t}(\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P) \xrightarrow{\nu\mathbf{t_2}\ \overline{s_2}v_2} \nu\mathbf{t}\backslash\mathbf{t_2}\ (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P)\end{array}}{\begin{array}{c}\nu\mathbf{t}\backslash\mathbf{t_1}\ (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P) \xrightarrow{\nu\mathbf{t_2}\backslash\mathbf{t_1}\ \overline{s_2}v_2} (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P),\\ \nu\mathbf{t}\backslash\mathbf{t_2}\ (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P) \xrightarrow{\nu\mathbf{t_1}\backslash\mathbf{t_2}\ \overline{s_2}v_2} (\overline{s_1}v_1 \mid \overline{s_2}v_2 \mid P)\end{array}}$$

Note that, up to symmetry (and structural equivalence), the previous lemma covers *all* possible commutations of two compatible actions α, β but the 2 remaining cases where $\alpha = \beta$ and $\alpha \in \{\tau, N\}$.

Proposition 32 *If a program is deterministic then it is confluent.*

Proof. We recall that if P is deterministic then it is τ-inert. Suppose Q is a derivative of P, $\alpha \downarrow \beta$, $Q \stackrel{\alpha}{\Rightarrow} Q_1$ and $Q \stackrel{\beta}{\Rightarrow} Q_2$.

If $\alpha = \beta$ then the definition of determinacy implies that $Q_1 \approx Q_2$. Also note that $\alpha\backslash\beta = \beta\backslash\alpha = \tau$ and $Q_i \stackrel{\tau}{\Rightarrow} Q_i$ for $i = 1, 2$. So the conditions for confluence are fulfilled.

So we may assume $\alpha \neq \beta$ and, up to symmetry, we are left with 5 cases corresponding to the 5 situations considered in lemma 31.

In the 2 cases where $\beta = \tau$ we have that $Q \approx Q_2$ by τ-inertness. Thus, by bisimulation $Q_2 \stackrel{\alpha}{\Rightarrow} Q_3$ and $Q_1 \approx Q_3$. Now $\alpha\backslash\tau = \alpha$, $\tau\backslash\alpha = \tau$, and $Q_1 \stackrel{\tau}{\Rightarrow} Q_1$. Hence the conditions for confluence are fulfilled.

We are left with 3 cases where α and β are distinct input or output actions. By using τ-inertness, we can focus on the case where $Q \stackrel{\alpha}{\Rightarrow} Q_1$ and $Q \stackrel{\beta}{\rightarrow} Q_2' \stackrel{\tau}{\Rightarrow} Q_2$. Now, by iterating the lemma 31, we can prove that:

$$\frac{Q\ (\stackrel{\tau}{\rightarrow})^n\ Q_1',\quad n \geq 1,\quad Q \stackrel{\beta}{\rightarrow} Q_2'}{\exists\, Q_2''\ (\ Q_1' \stackrel{\beta}{\rightarrow} Q_2'',\quad Q_2'\ (\stackrel{\tau}{\rightarrow})^n\ Q_2''\ )}\ .$$

So we are actually reduced to consider the situation where $Q \stackrel{\alpha}{\rightarrow} Q_1' \stackrel{\tau}{\Rightarrow} Q_1$ and $Q \stackrel{\beta}{\rightarrow} Q_2' \stackrel{\tau}{\Rightarrow} Q_2$. But then by lemma 31, we have: $Q_1' \xrightarrow{\beta\backslash\alpha} Q_3$, $Q_2' \xrightarrow{\alpha\backslash\beta} Q_4$, and $Q_3 \equiv Q_4$. Then using τ-inertness and bisimulation, it is easy to close the diagram. □

This concludes the proof of the first part of the theorem (18(1)). To derive the second part, we rely on the following fact due to [11].

Fact 33 ([11]) *If a program is reactive and locally confluent then it is confluent.*

Thus to derive the second part of the theorem (18(2)) it is enough to prove.

Proposition 34 *A program is locally confluent if (and only if) for all its derivatives Q:*

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

Proof. The stated condition is a special case of local confluence thus it is a necessary condition. To show that it is sufficient to entail local confluence, it is enough to appeal again to lemma 31 (same argument given at the end of the proof of proposition 32). □

Proof of proposition 19 Say that P is *strong confluent* if it satisfies the hypotheses of proposition 19. Let $\mathcal{S} = \{(P, Q) \mid P$ strong confluent and $(P \equiv Q$ or $P \xrightarrow{\tau} Q)\}$. Let $\mathcal{R} = \mathcal{S} \cup \mathcal{S}^{-1}$. We show that $\mathcal{R}$ is a bisimulation. Hence strong confluence entails τ-inertness. Note that if $P \xrightarrow{\alpha} P_i$, for $i = 1, 2$, and α is either an input or an output action then $P_1 \equiv P_2$. By lemma 31 and diagram chasing, we show that if P is strong confluent and $P \stackrel{\alpha}{\Rightarrow} P_i$, for $i = 1, 2$, then $P_1 \approx P_2$. This suffices to show that P is determinate (and confluent). □