Determinacy in a synchronous pi-calculus

Roberto Amadio, Mehdi Dogguy

Determinacy in a synchronous π-calculus *

Roberto M. Amadio Mehdi Dogguy
Université Paris 7, PPS, UMR-7126

4th July 2007

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 study the notions of determinacy and (local) confluence for the $S\pi$-calculus and we introduce a typing system that guarantees determinacy.

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 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 can associate a compositional notion of bisimulation equivalence.

2. We provide local confluence conditions that combined with reactivity turn out to be equivalent to determinacy.

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

3. We introduce a typing system which is preserved by labelled transitions and such that typable programs enjoy a strong confluence property.

We briefly trace the path that has lead to this approach. A systematic study of determinacy and confluence for CCS is available in [18] 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 [20], has been given in [11]. The theory has been gradually extended from CCS, to CCS with values, and finally to the π-calculus [22]. Typing systems appear to be a natural way to formalise decidable conditions that preserve confluence. The sorting system in [18] is an early example of this point of view, and more recent proposals for the π-calculus include [15, 21].

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, 23, 26, 17]). For instance, the Reactive ML language [17] 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. [24] 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.*, [17, 10, 25]). 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.

[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 [18].

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] Fourth, one needs to design a type system that accounts for 'resource usage' in the specific framework of a synchronous model with signal-based communication.

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, in section 4, we develop the concepts of determinacy and (local) confluence, and in section 5, we present a typing system that guarantees determinacy. Proofs are available in the appendix along with a simple *size-change* criterion that ensures reactivity. Familiarity with the π-calculus [19, 27], the notions of determinacy and confluence presented in [18], 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 some programming examples and an informal comparison with the π-calculus.

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 the 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.

[2] The situation is different in asynchronous systems where reactivity is a more demanding property. For 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,... A more refined type system including information on signal usage will be presented in section 5.

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 Programming examples

We introduce a few programming examples to illustrate the kind of synchronous programming that can be represented in the $S\pi$-calculus. We will come back to these examples in section 5 to discuss the scope of the typing system.

Example 1 (cell) *We describe the behaviour of a generic cell that might be used in the simulation of a dynamic system. Each cell relies on three parameters: its state q, its own activation signal s, and the list ℓ of activation signals of its neighbours. The cell performs the following operations in a cyclic fashion: (i) it emits its current state and a fresh identifier along the activation signals of its neighbours, (ii) it waits till the end of the current instant, and (iii) it collects the values emitted by its neighbours and computes its new state.*

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

where next is a function that computes the following state of the cell according to its current state and the state of its neighbours. Assuming that the function next is invariant under permutations of the list of states, we would like to show that the evolution of the simulation is deterministic.

Example 2 (synchronous data flow) *We provide an example of synchronous data-flow computation. The network is described by the program*

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

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

Assuming that at each instant a value is emitted on the input signal s_1, we would like to show that at each instant at most one value will be emitted on every other signal.

Example 3 (client-server) *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'.*

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

Assuming that parallel composition is an associative and commutative operation with respect to semantic equivalence, we would like to show that the behaviour of the server is deterministic. 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 means of a labelled transition system and a notion of bisimulation. This topic has already been studied in [2]. The main contribution here is the introduction of a *modified* labelled transition system that coupled with a *standard* notion of bisimulation allows to characterise *on reactive program* the *contextual bisimulation* introduced in [2]. The fact that the notion of bisimulation introduced here is *standard* (as opposed to [2]) will allow for a considerable simplification of the diagram chasing arguments that are needed in the study of determinacy and confluence in section 4.

3.1 Actions

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

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

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) and 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 only occur when the program cannot perform τ actions and it amounts to (i) 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 only 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: 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 4 (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 5 (reactivity) *We say that a program P is reactive, if for every derivative Q every τ-reduction sequence terminates.*

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

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

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

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

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

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

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

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

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

Table 1: Labelled transition system

3.3 Labelled bisimulation and its characterisation

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 6 (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.

Next, we recall the notion of contextual bisimulation introduced in [2].

Definition 7 *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)*

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

Definition 9 (barbed bisimulation) *A symmetric relation $\mathcal{R}$ on programs is a barbed bisimulation if whenever $P\ \mathcal{R}\ Q$ the following holds:*

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

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

[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 $\stackrel{\tau}{\Rightarrow} \circ \stackrel{N}{\longrightarrow} \circ \stackrel{\tau}{\Rightarrow}$.

	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 6
$(\xrightarrow{\alpha}_2)$	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: $\dfrac{P \mathcal{R} Q, S = \overline{s}_1 v_1 \mid \cdots \mid \overline{s}_n v_n, \quad (P \mid S) \xrightarrow{N} P'}{\exists Q', Q'' \ (\ (Q \mid S) \stackrel{\tau}{\Rightarrow}_2 Q'', \quad (P \mid S) \mathcal{R} Q'', \ Q'' \xrightarrow{N}_2 Q', \quad P' \mathcal{R} Q'\ )}$

Table 2: Equivalent formulations of labelled bisimulation

Definition 10 *A static context C is defined as follows:*

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

Definition 11 (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 12 (characterisation of labelled bisimulation) *If P, Q are reactive programs then $P \approx Q$ iff $P \approx_C Q$.*

This result provides two arguments in favour of labelled bisimulation. First it is preserved by static contexts and second it can be characterised by just appealing to internal transitions. 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 11 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 5 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]

4 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 [18], a program is considered determinate if performing twice the same sequence of interactions leads to the same program up to semantic equivalence.

Definition 13 (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 14 (τ-inertness) *A program is τ-inert if for all its derivatives Q, $Q \xrightarrow{\tau} Q'$ 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 15 (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.

[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.

[5] The reader familiar with [22] 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 16 (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 17 (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.

Definition 18 (local confluence) *We say that a program is locally 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 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 19 (1) *A program is determinate iff it is confluent.*
(2) *A reactive program is determinate iff for all its derivatives Q:*

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

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. The typable programs introduced in section 5 will enjoy this kind of property. Let $\stackrel{\alpha}{\rightsquigarrow}$ be $\stackrel{\alpha}{\rightarrow} \cup Id$ where Id is the identity relation.

[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 $P \stackrel{\overline{a}}{\rightarrow} a.\overline{b}$, $P \stackrel{\tau}{\rightarrow} \overline{b}$, and there is no way to close the diagram.

Proposition 20 *A program is determinate if for all its derivatives Q:*

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

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

5 A typing system for determinacy

We refer to [27, 14] for general introductions to type systems for the π-calculus. Our contribution here is to adapt some of the ideas on *usages* to the specific framework of the $S\pi$-calculus. Here are some issues that do *not* arise in the π-calculus: (i) the notion of usage may depend on the instant, (ii) we have to deal with inductive types and set types (a kind of quotient type), and (iii) in general the emission of a 'linear' resource on a signal destroys the linearity since the emitted value can be received an arbitrary number of times.

Our analysis builds on theorem 19(2) according to which 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: $\overline{s}v_1 \mid \overline{s}v_2 \mid \mathsf{pause}.A(!s)$.

In this section, we introduce a type system 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.

5.1 Usages, signals, and set types

In first approximation, we may regard a *usage* as an element of the set $L = \{0, 1, \infty\}$ with the intuition that 0 corresponds to no usage at all, 1 to at most one usage, and ∞ to any usage. We *order* usages as follows $0 < 1 < \infty$, going from the most restrictive to the most liberal one. We also *add* usages with a symmetric operation $\oplus$ which respects the order and such that $0 \oplus x = x$, $1 \oplus 1 = \infty$, and $\infty \oplus x = \infty$.

A signal usage can be refined to include information about whether a signal is used (i) to emit, (ii) to receive during the instant, or (iii) to receive at the end of the instant. Then a usage becomes an element of L^3. Further, a usage may depend on time, *i.e.*, it can be regarded as an infinite word over L^3, namely an element of $(L^3)^\omega$. We say that a usage is *uniform* if it is invariant under time, *i.e.*, it is represented by a word $x^\omega = x \cdot x \cdots x \cdots$ with $x \in L^3$.

We will focus on certain uniform usages, however, to reason compositionally about them, we are lead to introduce certain usages which are *not* uniform. For instance, a proof that a given program uses a given signal at most once each instant, is naturally decomposed into a proof that the signal is used at most once in the current instant and at most once in all the following instants.

Within an instant, we consider the usages $e = (\infty, 0, \infty)$, $o_1 = (1, \infty, \infty)$ and $o_0 = (0, \infty, \infty)$. Over all instants, we introduce two *main uniform usages*: (i) o_1^ω is a usage where at most one value is emitted at each instant, and (ii) e^ω is a usage where no signal can be read before the end of the instant. To reason about the first usage, we also introduce the 3

usages o_0^ω, $o_1 \cdot o_0^\omega$, and $o_0 \cdot o_1^\omega$. Then the set U of usages is

$$U = \{e^\omega, o_1^\omega, o_0^\omega, o_1 \cdot o_0^\omega, o_0 \cdot o_1^\omega\}$$

with generic elements $u, u', \ldots$ We consider that the addition operation $\oplus$ is defined only if the result is in the set U, *e.g.*, $o_1^\omega \oplus o_1^\omega$ is undefined. If $u \in U$ then $\uparrow u$, the *shift* of u, is the infinite word in U obtained from u by removing the first character (the shift is always defined).

We assume that a signal type constructor carries the information u about signal usage and write $Sig_u(\sigma)$. An important question, is whether a signal with usage u may carry a value containing a signal with another usage u'. Previous work on linear typing suggests that *not* all combinations are legal. For instance, one should not put on top of a type constructor with 'linear information' a 'non-linear' type constructor because then the linearity information is lost. In our case, the usage $(1, \infty, \infty)^\omega$ carries some linear information and this information is not preserved if, for instance, we emit this signal along another signal with the same usage because then the signal can be received (and used) an arbitrary number of times.[7]

Another issue to be considered is the type to be assigned to a dereferenced signal. If the usage of the signal is o_1^ω then the list of values at the end of the instant contains at most one element and we can just assign a *list type* to the dereferenced signal. On the other hand, if the usage is e^ω then the list may contain many elements and we need to make sure that the processing of the list does not depend on the order of the elements of the list. For this reason, we assign to the dereferenced signal a *set type*.

5.2 Types and type contexts

We summarise these considerations in a type system which enjoys two basic properties: (i) the typing is preserved by labelled transitions as long as they are compatible with the typing and (ii) a typable program P behaves 'deterministically' with respect to the actions τ and N. The combination of these two properties and proposition 20 then entails that typable programs are determinate. Moreover, the typing rules define a discipline to compose programs while preserving determinacy. The language of types is defined as follows:

$$\begin{array}{lll} \sigma & ::= 1 \mid List(\sigma) \mid Set(\sigma) \mid Sig_u(\sigma) & \text{where } u \in \{e^\omega, o_0^\omega\} \\ \rho & ::= \sigma \mid Sig_u(\sigma) & \text{where } u \in U \end{array}$$

We denote with $\sigma, \sigma', \ldots$ types that can be arbitrarily nested. In particular, $Set(\sigma)$ is the type of sets of values of type σ. We also denote with $\rho, \rho', \ldots$ more general types that allow certain special usages to appear at top level. The partial operation of addition $\oplus$ is extended to types so that $\sigma \oplus \sigma = \sigma$, $Sig_u(\sigma) \oplus Sig_{u'}(\sigma) = Sig_{u \oplus u'}(\sigma)$ if $u \oplus u'$ is defined, and $\rho \oplus \rho'$ is undefined otherwise. We call the σ types *neutral* since if $\rho \oplus \sigma$ is defined then it equals ρ. We also say that a type ρ is uniform if it only contains uniform usages.

A type context (or simply a context) Γ is a partial function with finite domain $dom(\Gamma)$ from variables to types. An addition operation $\Gamma_1 \oplus \Gamma_2$ on contexts is only defined if for all x such that $\Gamma_1(x) = \rho_1$ and $\Gamma_2(x) = \rho_2$, the type $\rho_1 \oplus \rho_2$ is defined. If this is the case then:

[7] An interesting question that we do not pursue here is what kind of usages *do preserve linear information*. For instance, one possibility is to consider a usage such as $(1, 1, 0)^\omega$ which corresponds to a signal where at most one emission and one reception is performed at each instant.

$$(\Gamma_1 \oplus \Gamma_2)(x) = \begin{cases} \Gamma_1(x) \oplus \Gamma_2(x) & \text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_1(x) & \text{otherwise, if } x \in dom(\Gamma_1) \\ \Gamma_2(x) & \text{otherwise, if } x \in dom(\Gamma_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The shift operation is extended to contexts so that $(\uparrow \Gamma)(x) = Sig_{(\uparrow u)}(\sigma)$ if $\Gamma(x) = Sig_u(\sigma)$ and $(\uparrow \Gamma)(x) = \Gamma(x)$ otherwise. We also denote with $\Gamma, x : \sigma$ the context Γ *extended* with the pair $x : \sigma$ (so $x \notin dom(\Gamma)$). We say that a context is *neutral* (*uniform*) if it assigns to variables neutral (uniform) types.

5.3 Typing expressions and value equivalence

The rules for typing expressions with dereferenciation are as follows. We denote with k either a constructor or a function symbol and we assume that its type is explicitly given. We may label the constructor nil both with the types $List(\sigma)$ and $Set(\sigma)$ and the constructor cons both with the types $(\sigma, List(\sigma)) \to List(\sigma)$ and $(\sigma, Set(\sigma)) \to Set(\sigma)$.

$$(var) \quad \frac{\Gamma \text{ neutral}}{\Gamma, x : \rho \vdash x : \rho} \qquad (k) \quad \frac{\Gamma \vdash r_i : \sigma_i, \quad i = 1, \ldots, n, \quad \Gamma' \text{ neutral} \quad k : (\sigma_1, \ldots, \sigma_n) \to \sigma, \quad k = f \text{ or } k = \mathsf{c}}{\Gamma \oplus \Gamma' \vdash k(r_1, \ldots, r_n) : \sigma}$$

$$(!_{Set}) \quad \frac{u = e^\omega, \quad \Gamma \text{ neutral}}{\Gamma, s : Sig_u(\sigma) \vdash !s : Set(\sigma)} \qquad (!_{List}) \quad \frac{u = o_0^\omega, \quad \Gamma \text{ neutral}}{\Gamma, s : Sig_u(\sigma) \vdash !s : List(\sigma)}$$

We define an equivalence relation $\sim_\rho$ on values as the least equivalence relation such that $s \sim_{Sig_u(\sigma)} s$, $* \sim_1 *$, $\mathsf{nil} \sim_{List(\sigma)} \mathsf{nil}$, and

$$\begin{array}{ll} \mathsf{cons}(v_1, v_2) \sim_{List(\sigma)} \mathsf{cons}(u_1, u_2) & \text{if } v_1 \sim_\sigma u_1 \text{ and } v_2 \sim_{List(\sigma)} u_2 \ . \\ [v_1; \ldots; v_n] \sim_{Set(\sigma)} [u_1; \ldots; u_m] & \text{if } \{v_1, \ldots, v_n\} \sim_{Set(\sigma)} \{u_1, \ldots, u_m\}, \\ \text{where: } \{v_1, \ldots, v_n\} \sim_{Set(\sigma)} \{u_1, \ldots, u_m\} & \text{if for a permutation } \pi, v_i \sim_\sigma u_{\pi(i)} \ . \end{array}$$

We assume that each function symbol f, coming with a type $(\sigma_1, \ldots, \sigma_n) \to \sigma$, *respects* the typing in the following sense: (1) if $v_i \sim_{\sigma_i} u_i$, $i = 1, \ldots, n$, $f(v_1, \ldots, v_n) \Downarrow v$ and $f(u_1, \ldots, u_n) \Downarrow u$ then $v \sim_\sigma u$. (2) If $\Gamma \vdash v_i : \sigma_i$, for $i = 1, \ldots, n$, and $f(v_1, \ldots, v_n) \Downarrow v$ then $\Gamma \vdash v : \sigma$.

5.4 Typing programs

We assume that each signal name generation comes with its type whose usage can be either e^ω or o_1^ω. We also assume that each thread identifier A, defined by an equation $A(x_1, \ldots, x_n) = P$, comes with a type $(\rho_1, \ldots, \rho_n)$ where the types ρ_i are uniform for $i = 1, \ldots, n$. We require that A has the property that: (i) if $v_i \sim_{\rho_i} u_i$ for $i = 1, \ldots, n$ then $A(v_1, \ldots, v_n) \approx A(u_1, \ldots, u_n)$

and (ii) $x_1 : \rho_1, \ldots, x_n : \rho_n \vdash P$ is derivable in the following typing system.

$$(0)\quad \frac{\Gamma \text{ neutral}}{\Gamma \vdash 0} \qquad (out)\quad \frac{u \in \{e^\omega, o_1 o_0^\omega\}, \quad \Gamma \vdash e : \sigma}{\Gamma, s : Sig_u(\sigma) \vdash \overline{s}e}$$

$$(\nu)\quad \frac{\begin{array}{c} u \in \{e^\omega, o_1^\omega\}, \\ \Gamma, s : Sig_u(\sigma) \vdash P \end{array}}{\Gamma \vdash \nu s : Sig_u(\sigma)\ P} \qquad (in)\quad \frac{\begin{array}{c} u \in \{o_0^\omega, o_0 o_1^\omega\}, \quad \Gamma, s : Sig_u(\sigma), x : \sigma \vdash P, \\ \uparrow (\Gamma, s : Sig_u(\sigma)) \vdash K \end{array}}{\Gamma, s : Sig_u(\sigma) \vdash s(x).P, K}$$

$$(par)\quad \frac{\Gamma_i \vdash P_i, \quad i = 1, 2}{\Gamma_1 \oplus \Gamma_2 \vdash P_1 \mid P_2} \qquad (rec)\quad \frac{\begin{array}{c} A : (\rho_1, \ldots, \rho_n), \\ \Gamma_i \vdash r_i : \rho_i, \quad i = 1, \ldots, n \end{array}}{\Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash A(r_1, \ldots, r_n)}$$

$$(m_s)\quad \frac{\Gamma \vdash P_i, \quad i = 1, 2}{\Gamma \vdash [s_1 = s_2]P_1, P_2} \qquad (m_{\mathsf{c}})\quad \frac{\begin{array}{c} \mathsf{c} : (\sigma_1, \ldots, \sigma_n) \to \sigma, \quad \Gamma' \vdash u : \sigma, \quad \Gamma \oplus \Gamma' = \Gamma \\ \Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash P_1, \quad \Gamma \vdash P_2 \end{array}}{\Gamma \vdash [u \trianglerighteq \mathsf{c}(x_1, \ldots, x_n)]P_1, P_2}$$

With reference to example 1, 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 we can require: $Cell : (State, S_1, L_1)$ and $Send : (State, S_1, L_1, L_1)$. Because, the usage of the signals under consideration is e^ω, the type of their dereferenciation is $Set(State)$ and therefore we must require $next : (State, Set(State)) \to State$, which means that the result of the function *next* must be invariant under permutations of the list of (distinct) states.

With reference to example 2, we assume an inductive type D of *data* and let $I = Sig_{o_0^\omega}(D)$ and $O = Sig_{o_1^\omega}(D)$. Then we can require: $A : (I, O, I, O)$, $C : (I, O, I, O)$, and $C : (I, O)$. The restricted signals $s_2, \ldots, s_5$ take the type O and the overall system is well-typed with respect to the context $s_1 : I, s_6 : O$. Note that the typing system guarantees determinacy by making sure that at every instant *at most one* value is emitted on every signal. One could consider a more refined type system that guarantees that *exactly one* value is emitted on a signal at every instant.

With reference to example 3, assume an inductive type D of data and let $S_1 = Sig_u(D)$, $\mathsf{req} : (Sig_u(D), D) \to Req$, and $S_2 = Sig_{e^\omega}(Req)$. Note that in our type system we are forced to take $u = e^\omega$, otherwise the types are not well-formed. Then we could type the server assuming: $Server : S_2$ and $Handle : (S_2, Set(Req))$. However, the usage $u = e^\omega$ is incompatible with the definition of the client, as it can receive the result during an instant. It appears that in order to type the client we need to introduce several new 'linear' types. First, the signal on which the client waits for a reply should allow at most one emission per instant and second each request should be received and handled at most once. The development of such system appears to be feasible but beyond the scope of the present paper.

5.5 Formal properties of the typing system

We omit some preliminary results on weakening and substitution properties. The following proposition states how the typing is preserved by labelled transitions (we omit the cases for the auxiliary actions).

Proposition 21 (subject reduction) *Suppose* $\Gamma \vdash P$. *Then:*

(1) *If* $P \xrightarrow{sv} P'$, $\Gamma' \vdash \overline{s}v$, *and* $\Gamma \oplus \Gamma'$ *is defined then* $\Gamma \oplus \Gamma' \vdash P'$.

(2) *If* $P \xrightarrow{\nu \mathbf{t}:\rho \overline{s}v} P'$ *then* $\Gamma, \mathbf{t} : \rho \vdash P'$.

(3) *If* $P \xrightarrow{\tau} P'$ *then* $\Gamma \vdash P'$.

(4) *If* $P \xrightarrow{N} P'$ *then* $\uparrow(\Gamma) \vdash P'$.

One can summarize this property by saying that labelled transitions preserve the typing provided that the input transitions are compatible with the hypotheses in the context. In particular, an input transition on the signal s is disallowed whenever the usage of s in Γ is either o_1^ω or $o_1 o_0^\omega$. Next we observe that typable programs have strong confluence properties with respect to τ and N transitions.

Proposition 22 *Suppose* $\Gamma \vdash P$ *and* $P \xrightarrow{\alpha} P_i$, *for* $i = 1, 2$.

(1) *If* $\alpha = \tau$ *then* $\exists\, Q$ $(P_i \overset{\tau}{\rightsquigarrow} Q)$.

(2) *If* $\alpha = N$ *then* $P_1 \approx P_2$.

In the proof of (1), the only interesting case arises when the two τ reductions are generated by a synchronisation. However, the typing forbids situations such as: $\overline{s}e \mid \overline{s}e' \mid s(x).P, K$. The only possible super-position is of the form: $\overline{s}e \mid s(x).P, K \mid s(x).P', K'$. But this is innocuous because the emitted signal persists.

The proof of (2), relies on the observation that the only essential difference between P_1 and P_2 is in the order in which values emitted on a signal, say s, are collected at the end of the instant. If s has a linear usage, then at most one value can be emitted on it and therefore there is only one possible ordering. On the other hand, if s has a usage e^ω then $!s$ has a set-type which forces a processing which is order-independent.

Now we would like to conclude that a typable program is determinate, but we have to give a careful interpretation to this statement. A typable program is a program P typable in a context Γ and then the labelled transitions that we consider in the definitions of determinacy and (local) confluence are only those compatible with the context Γ. One can check that proposition 20 still holds in this restricted framework. Then this result coupled with the proposition 22 entails the following result.

Theorem 23 (typability implies determinacy) *If the program* P *is typable in the context* Γ *then* $\Gamma \vdash P$ *is determinate.*

6 Conclusion

We have developed a framework to analyse the determinacy of programs in a *synchronous* π-calculus. First, we have characterised a relevant contextual bisimulation as a standard bisimulation over a modified labelled transition system. Second, 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. Third, we have engineered a typing system that guarantees a form of strong confluence and is preserved by the labelled transition system. To achieve this goal we have adapted to our framework ideas on resource usage, linearity, and quotient types. The resulting type system provides a static and compositional analysis to check the determinacy of programs.

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*, in press. Also available as Technical report, Université Paris 7, June 2006.

[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] N. Kobayashi. Type systems for concurrent programs. In Proc. *10th Anniversary Colloquium of UNU/IIST*, Springer LNCS 2757, 2003.

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

[16] C. Lee, N. Jones, A. Ben-Amram. The size-change principle for program termination. In Proc. ACM-POPL, 2004.

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

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

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

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

[21] U. Nestmann. On determinacy and nondeterminacy in concurrent programming. PhD thesis, Universität Erlangen, 1996.

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

[23] Reactive programming, INRIA, Mimosa Project. `http://www-sop.inria.fr/mimosa/rp`.

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

[25] V. Saraswat, R. Jagadeesan, A. Solar-Lezama, and C. von Praun. Determinate imperative programming: a clocked interpretation of imperative syntax. Draft, 2006.

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

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

[28] R. Thiemann, J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. In Applicable Alg. in Eng., Com., and Comp., in press.

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 6 of bisimulation where transitions are weak on both sides of the bisimulation game.

Definition 24 (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.[8]

Definition 25 (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 26 (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. □

[8] We recall that it is important that this notion is defined with respect to w-bisimulation. Indeed, proposition 26(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 12

We start with the labelled transition system defined in table 1 and the notion of bisimulation in definition 6. 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 27 *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 28 (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.

Lemma 29 *If $P \approx_1 Q$ then $(P \mid \overline{s}v) \approx_1 (Q \mid \overline{s}v)$.*

Proof. Let $\mathcal{R}' = \{((P \mid \overline{s}v), (Q \mid \overline{s}v)) \mid P \approx_1 Q\}$ and $\mathcal{R} = \mathcal{R}' \cup \approx_1$. We show that $\mathcal{R}$ is a 1-bisimulation. Suppose $(P \mid \overline{s}v) \xrightarrow{\alpha}_1 \cdot$ and $P \approx_1 Q$. There are two interesting cases to consider.

$(\alpha = \tau)$ Suppose $(P \mid \overline{s}v) \xrightarrow{\tau}_1 (P' \mid \overline{s}v)$ because $P \xrightarrow{s?v}_1 P'$. By definition of the lts, we have that $P \xrightarrow{sv}_1 (P \mid \overline{s}v) \xrightarrow{\tau}_1 (P' \mid \overline{s}v)$. By definition of 1-bisimulation, $Q \overset{sv}{\Rightarrow}_1 (Q'' \mid \overline{s}v) \overset{\tau}{\Rightarrow}_1 (Q' \mid \overline{s}v)$ and $(P' \mid \overline{s}v) \approx_1 (Q' \mid \overline{s}v)$. We conclude, by noticing that then $(Q \mid \overline{s}v) \overset{\tau}{\Rightarrow}_1 (Q' \mid \overline{s}v)$.

($\alpha = N$) Suppose $(P \mid \overline{s}v) \xrightarrow{N}_1 P'$. Then also $P \xrightarrow{sv}_1 (P \mid \overline{s}v) \xrightarrow{N}_1 P'$. By definition of 1-bisimulation, $Q \stackrel{sv}{\Rightarrow}_1 (Q'' \mid \overline{s}v) \stackrel{N}{\Rightarrow}_1 Q'$ and $P' \approx_1 Q'$. We conclude by noticing that then $(Q \mid \overline{s}v) \stackrel{N}{\Rightarrow}_1 Q'$. □

Lemma 30 *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 29. Otherwise, suppose $\alpha \neq sv$, $P \approx_1 Q$, and $P \xrightarrow{\alpha}_2 P'$. By lemma 28(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 28(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 28(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 28(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 31 (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) We follow the pattern of lemma 29. Let $\mathcal{R}' = \{((P \mid \overline{s}v), (Q \mid \overline{s}v)) \mid P \approx_1 Q\}$ and $\mathcal{R} = \mathcal{R}' \cup \approx_1$. 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 32 *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 31(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 31(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. □

In [2], the contextual bisimulation in definition 11 is characterised as a variant of the bisimulation $\approx_3$ where the condition for the output is formulated as follows:

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

Clearly, if P is a reactive program then $P \Downarrow_L$. Also note that the definition 5 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].

Remark 33 (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 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.*

C Proof of theorem 19 and proposition 20

First, relying on proposition 26(3), one can repeat the proof in [18] that confluence implies τ-inertness and determinacy.

Proposition 34 *If a program is confluent then it is τ-inert and determinate.*

Proof. Let $\mathcal{S} = \{(P, P') \mid P \text{ confluent and } P \overset{\tau}{\Rightarrow} P'\}$ 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 26(3)). Clearly $\mathcal{R}$ is symmetric. Then suppose P confluent and $P \overset{\tau}{\Rightarrow} Q$ (the case where Q reduces to P is symmetric). If $Q \overset{\alpha}{\Rightarrow} Q_1$ then $P \overset{\alpha}{\Rightarrow} Q_1$ and $Q_1 \mathcal{R} Q_1$. On the other hand, if $P \overset{\alpha}{\Rightarrow} P_1$ then by confluence there are P_2, Q_1 such that $P_1 \overset{\tau}{\Rightarrow} P_2$, $Q \overset{\alpha}{\Rightarrow} Q_1$, 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 35 (input-output commutations)

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

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

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

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

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

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 36 *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 35.

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 35, 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 35, we have: $Q'_1 \stackrel{\beta\backslash\alpha}{\longrightarrow} Q_3$, $Q'_2 \stackrel{\alpha\backslash\beta}{\longrightarrow} Q_4$, 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 (19(1)). To derive the second part, we rely on the following result due to [11].

Theorem 37 ([11]) *If a program is reactive and locally confluent then it is confluent.*

Thus to derive the second part of the theorem (19(2)) it is enough to prove.

Proposition 38 *A program is locally confluent if (and only if) for all its derivatives Q:*

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

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 35 (same argument given at the end of the proof of proposition 36). □

Proof of proposition 20 Say that P is *strong confluent* if it satisfies the hypotheses of proposition 20. Let $\mathcal{S} = \{(P, Q) \mid P$ strong confluent and $(P \equiv Q$ or $P \stackrel{\tau}{\rightarrow} Q)\}$. 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 $P \stackrel{\alpha}{\rightarrow} P_i$, for $i = 1, 2$, and α is either an input or an output action then $P_1 \equiv P_2$. By lemma 35 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). □

D Proof of theorem 23

D.1 Preliminaries

We note that if $\Gamma \oplus \Gamma'$ is defined and Γ' is neutral then $\Gamma \oplus \Gamma'$ is just an *extension* of Γ with neutral assignments. In other terms, we can decompose Γ' in Γ'_1, Γ'_2 and write $\Gamma \oplus \Gamma'$ as Γ, Γ'_2.

The typing rules are formulated so that the linear hypotheses, *i.e.*, those of the shape $s : \rho$ where ρ is *not* neutral, *must* be used in the typing (note that this does *not* imply that these

resources will actually be used in the computation). Therefore, if judgments of the shape $\Gamma \vdash r : \sigma$ or $\Gamma, s : \rho \vdash s : \rho$ are derivable then Γ is a neutral context. In particular, expressions of neutral type can only be typed with a neutral context.

We remark that the typing judgments can always be weakened by extending them with a neutral context.

Lemma 39 (weakening) *Suppose Γ, Γ' are contexts such that $dom(\Gamma) \cap dom(\Gamma') = \emptyset$ and Γ' is a neutral context. Then:*

(1) *If $\Gamma \vdash r : \rho$ then $\Gamma, \Gamma' \vdash r : \rho$.*

(2) *If $\Gamma \vdash P$ then $\Gamma, \Gamma' \vdash P$.*

Proof. In both cases we proceed by induction on the proof of the typing judgment.

(1) (var) From $\Gamma_1, x : \rho \vdash x : \rho$ with Γ_1 neutral, we derive $\Gamma_1, x : \rho, \Gamma' \vdash x : \rho$ since Γ' is neutral too.

(k) Suppose $\Gamma \oplus \Gamma'' \vdash k(\mathbf{r}) : \sigma$ with Γ'' neutral. Then Γ'' can be decomposed in Γ''_1, Γ''_2 so that $\Gamma \oplus \Gamma'' = \Gamma, \Gamma''_1$. But then $(\Gamma \oplus \Gamma''), \Gamma' \vdash k(\mathbf{r}) : \sigma$.

$(!_{Set})$ If $\Gamma'', s : Sig_u(\sigma) \vdash !s : List(\sigma)$ with Γ'' neutral then $\Gamma'', \Gamma', s : Sig_u(\sigma) \vdash !s : List(\sigma)$.

$(!_{List})$ Same argument as in the previous case.

(2) (0) If $\Gamma \vdash 0$ then Γ is neutral and since Γ' is neutral too then $\Gamma, \Gamma' \vdash 0$.

(out) Suppose $\Gamma, s : Sig_u(\sigma) \vdash \overline{s}e$. Then $\Gamma \vdash e : \sigma$. By (1), $\Gamma, \Gamma' \vdash e : \sigma$. Hence $\Gamma, \Gamma', s : Sig_u(\sigma) \vdash \overline{s}e$.

(ν) Suppose $\Gamma \vdash \nu s : Sig_u(\sigma)\ P$. Then $\Gamma, s : Sig_u(\sigma) \vdash P$ and by inductive hypothesis, $\Gamma, s : Sig_u(\sigma), \Gamma' \vdash P$. Hence $\Gamma, \Gamma' \vdash \nu s : Sig_u(\sigma)\ P$.

(in) Suppose $\Gamma_1, s : Sig_u(\sigma) \vdash s(x).P, K$. Then $\Gamma_1, s : Sig_u(\sigma), x : \sigma \vdash P$ and $\uparrow (\Gamma_1, s : Sig_u(\sigma)) \vdash K$. By inductive hypothesis, $\Gamma_1, s : Sig_u(\sigma), x : \sigma, \Gamma' \vdash P$ and $\uparrow (\Gamma_1, s : Sig_u(\sigma)), \Gamma' \vdash K$. Since Γ' is neutral, $\uparrow \Gamma' = \Gamma'$. Therefore $\uparrow (\Gamma_1, s : Sig_u(\sigma)), \Gamma' = \uparrow (\Gamma_1, s : Sig_u(\sigma), \Gamma')$, and by applying the typing rule (in) we conclude that $\Gamma_1, s : Sig_u(\sigma), \Gamma' \vdash s(x).P, K$.

(par) Suppose $\Gamma_1 \oplus \Gamma_2 \vdash P_1 \mid P_2$ with $\Gamma_i \vdash P_i$, for $i = 1, 2$. By inductive hypothesis, $\Gamma_i, \Gamma' \vdash P_i$, for $i = 1, 2$. Then $(\Gamma_1 \oplus \Gamma_2), \Gamma' \vdash P_1 \mid P_2$, noticing that $(\Gamma_1, \Gamma') \oplus (\Gamma_2, \Gamma') = (\Gamma_1 \oplus \Gamma_2), \Gamma'$ since Γ' is neutral.

(rec) Suppose $\Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash A(r_1, \ldots, r_n)$, with $A : (\rho_1, \ldots, \rho_n)$, $\Gamma_i \vdash r_i : \rho_i$, for $i = 1, \ldots, n$. By inductive hypothesis, $\Gamma_i, \Gamma' \vdash r_i : \rho_i$. Again, noticing that Γ' is neutral we can conclude that $(\Gamma_1 \oplus \cdots \oplus \Gamma_n), \Gamma' \vdash A(r_1, \ldots, r_n)$.

(m_s) Suppose $\Gamma \vdash [s_1 = s_2]P_1, P_2$, with $\Gamma \vdash P_i$, for $i = 1, 2$. By inductive hypothesis, $\Gamma, \Gamma' \vdash P_i$, for $i = 1, 2$. Therefore, $\Gamma, \Gamma' \vdash [s_1 = s_2]P_1, P_2$

(m_{c}) Suppose $\Gamma \vdash [u \trianglerighteq \mathsf{c}(\mathbf{x})]P_1, P_2$, with $\mathsf{c} : (\sigma_1, \ldots, \sigma_n) \to \sigma$, $\Gamma'' \oplus \Gamma = \Gamma$, $\Gamma'' \vdash u : \sigma$, $\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash P_1$, and $\Gamma \vdash P_2$. By inductive hypothesis, $\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n, \Gamma' \vdash P_1$, and $\Gamma, \Gamma' \vdash P_2$. Moreover, $\Gamma'' \oplus (\Gamma, \Gamma') = \Gamma, \Gamma'$. Therefore, $\Gamma, \Gamma' \vdash [u \trianglerighteq \mathsf{c}(\mathbf{x})]P_1, P_2$. □

Next we study how typing is preserved by the substitution of a value for a variable.

Lemma 40 (substitution) (1) *If* $\Gamma, s : \rho \vdash s : \rho$ *and* $s' \notin dom(\Gamma)$ *then* $\Gamma, s' : \rho \vdash s' : \rho$.

(2) *If* $\Gamma, x : \sigma \vdash r : \rho$, $\Gamma' \vdash v : \sigma$ *and* $\Gamma \oplus \Gamma'$ *is defined then* $\Gamma \oplus \Gamma' \vdash [v/x]r : \rho$.

(3) *If* $\Gamma, s : \rho \vdash P$ *and* $s' \notin dom(\Gamma)$ *then* $\Gamma, s' : \rho \vdash [s'/s]P$.

(4) *If* $\Gamma, x : \sigma \vdash P$, $\Gamma' \vdash v : \sigma$ *and* $\Gamma \oplus \Gamma'$ *is defined then* $\Gamma \oplus \Gamma' \vdash [v/x]P$.

Proof. (1) By the definition of the (var) typing rule.

(2) By induction on the typing of r.

(var) Suppose $\Gamma, x : \sigma \vdash y : \rho$. We distinguish two cases.

$(x = y)$ Then the judgment is $\Gamma, x : \sigma \vdash x : \sigma$ with Γ neutral. Then $[v/x]x = v$ and $\Gamma' \vdash v : \sigma$ imply $\Gamma \oplus \Gamma' \vdash v : \sigma$ by the weakening lemma 39(1).

$(x \neq y)$ The judgment is $\Gamma, y : \rho, x : \sigma \vdash y : \rho$ with Γ neutral. Then $(\Gamma, y : \rho) \oplus \Gamma' \vdash y : \rho$.

(k) Suppose $\Gamma, x : \sigma, \Gamma'' \vdash k(r_1, \ldots, r_n) : \sigma$. Then, by the typing rule (k) and weakening we can suppose $\Gamma, x : \sigma, \Gamma'' \vdash r_i : \sigma_i$, for $i = 1, \ldots, n$. By inductive hypothesis, $(\Gamma, \Gamma'') \oplus \Gamma' \vdash [v/x]r_i : \sigma_i$. Then $(\Gamma, \Gamma'') \oplus \Gamma' \vdash [v/x]k(r_1, \ldots, r_n) : \sigma$.

$(!_{Set})$ Suppose $\Gamma, x : \sigma' \vdash !s : Set(\sigma)$. We distinguish two cases.

$(x = s)$ Then the judgments have the form $\Gamma, s : Sig_u(\sigma) \vdash !s : Set(\sigma)$ and $\Gamma', s' : Sig_u(\sigma) \vdash s' : Sig_u(\sigma)$ with Γ and Γ' neutral. Then $(\Gamma \oplus \Gamma'), s' : Sig_u(\sigma) \vdash !s' : Set(\sigma)$.

$(!_{List})$ Same argument as in the previous case applies.

(3) By induction on the typing of P.

(0) If $\Gamma, s : \rho \vdash 0$ and $s' \notin dom(\Gamma)$ then $\Gamma, s' : \rho \vdash 0$.

(out) Suppose $\Gamma, s : \rho \vdash \overline{s''}e$. We distinguish two cases.

$(s = s'')$ Then $[s'/s](\overline{s''}e) = \overline{s'}e$ and $\Gamma, s' : \rho \vdash \overline{s'}e$.

$(s \neq s'')$ Then $[s'/s](\overline{s''}e) = \overline{s''}[s'/x]e$ and $\Gamma = \Gamma', s'' : \rho''$. Since $\Gamma', s : \rho \vdash e : \sigma$, by (1), $\Gamma', s' : \rho \vdash [s'/s]e : \sigma$. Therefore $\Gamma', s' : \rho, s'' : \rho'' \vdash \overline{s''}[s'/x]e$.

(ν) Suppose $\Gamma, s : \rho \vdash \nu t : \rho'\ P$ and $s' \notin dom(\Gamma)$. Then $\Gamma, s : \rho, t : \rho' \vdash P$. By inductive hypothesis, $\Gamma, s' : \rho, t : \rho' \vdash [s'/s]P$. Hence $\Gamma, s' : \rho \vdash \nu t : \rho'[s'/s]P$.

(in) Suppose $\Gamma, s : \rho \vdash s''(x).P, K$ and $s' \notin dom(\Gamma)$. We distinguish two cases.

$(s = s'')$ Then $\rho = Sig_u(\sigma)$, $\Gamma, s : \rho, x : \sigma \vdash P$, and $\uparrow (\Gamma, s : \rho) \vdash K$. By inductive hypothesis, $\Gamma, s' : \rho, x : \sigma \vdash [s'/s]P$ and $\uparrow (\Gamma, s' : \rho) \vdash [s'/s]K$. Then $\Gamma, s' : \rho \vdash [s'/s](s(x).P, K)$

$(s \neq s'')$ Then $\Gamma = \Gamma', s'' : Sig_u(\sigma)$, $\Gamma, s : \rho, x : \sigma \vdash P$, $\uparrow (\Gamma, s : \rho) \vdash K$. By inductive hypothesis, $\Gamma, s' : \rho, x : \sigma \vdash [s'/s]P$ and $\uparrow (\Gamma, s' : \rho) \vdash [s'/s]K$. Then $\Gamma, s' : \rho \vdash [s'/s](s''(x).P, K)$.

(par) Suppose $\Gamma, s : \rho \vdash P_1 \mid P_2$ and $s' \notin dom(\Gamma)$, with $(\Gamma_1, s : \rho_1) \oplus (\Gamma_2, s : \rho_2) = \Gamma, s : \rho$. By inductive hypothesis, $\Gamma_i, s' : \rho_i \vdash [s'/s]P_i$, for $i = 1, 2$. Hence, $\Gamma, s' : \rho \vdash [s'/s](P_1 \mid P_2)$.

(rec) Suppose $\Gamma, s : \rho \vdash A(r_1, \ldots, r_n)$, $s' \notin dom(\Gamma)$, and $A : (\rho'_1, \ldots, \rho'_n)$. Then $\Gamma_i, s : \rho_i \vdash r_i : \rho'_i$, for $i = 1, \ldots, n$, and $\Gamma, s : \rho = (\Gamma_1, s : \rho_1) \oplus \cdots \oplus (\Gamma_n, s : \rho_n)$. By (2), $\Gamma_i, s' : \rho_i \vdash [s'/s]r_i : \rho'_i$, for $i = 1, \ldots, n$. Therefore, $\Gamma, s' : \rho \vdash [s'/s](A(r_1, \ldots, r_n))$.

(m_s) Suppose $\Gamma, s : \rho \vdash [s_1 = s_2]P_1, P_2$ and $s' \notin dom(\Gamma)$. Then $\Gamma, s : \rho \vdash P_i$, for $i = 1, 2$. By inductive hypothesis, $\Gamma, s' : \rho \vdash [s'/s]P_i$, for $i = 1, 2$. Therefore $\Gamma, s' : \rho \vdash [s'/s]([s_1 = s_2]P_1, P_2$.

(m_{c}) Suppose $\Gamma, s : \rho \vdash [u \trianglerighteq \mathsf{c}(\mathbf{x})]P_1, P_2$, $\mathsf{c} : (\sigma_1, \ldots, \sigma_n) \to \sigma$, and $s' \notin dom(\Gamma)$. Then there is some $\Gamma', s : \rho'$ such that $\Gamma', s : \rho' \oplus (\Gamma, s : \rho) = \Gamma, s : \rho$, $\Gamma', s : \rho' \vdash u : \sigma$. Also, $\Gamma, s : \rho, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash P_1$ and $\Gamma, s : \rho \vdash P_2$. By inductive hypothesis, $\Gamma', s' : \rho' \vdash [s'/s]u : \sigma$, $\Gamma, s' : \rho, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash [s'/s]P_1$, and $\Gamma, s' : \rho \vdash [s'/s]P_2$. Then then $\Gamma, s' : \rho \vdash [s'/s]([u \trianglerighteq \mathsf{c}(\mathbf{x})]P_1, P_2)$.

(4) By induction on the typing of P.

(0) If $\Gamma, x : \sigma \vdash 0$, $\Gamma' \vdash v : \sigma$, and $\Gamma \oplus \Gamma'$ is defined then $\Gamma \oplus \Gamma' \vdash 0$.

(out) Suppose $\Gamma, x : \sigma \vdash \overline{s}e$, $\Gamma' \vdash v : \sigma$ and $\Gamma \oplus \Gamma'$ is defined. We distinguish two cases.

($x = s$) Suppose $\Gamma, s : \sigma \vdash \overline{s}e$, $\Gamma \vdash e : \sigma'$, and $\Gamma' \vdash s' : \sigma$, where $\sigma = Sig_u(\sigma')$. Note that $[s'/x](\overline{s}e) = \overline{s'}e$. By weakening, $\Gamma \oplus \Gamma' \vdash e : \sigma'$. Therefore $\Gamma \oplus \Gamma' \vdash \overline{s'}e$.

($x \neq s$) Suppose $\Gamma, s : Sig_u(\sigma'), x : \sigma \vdash \overline{s}e$ and $\Gamma, x : \sigma \vdash e : \sigma'$. By (2), $\Gamma \oplus \Gamma' \vdash [v/x]e : \sigma'$. Therefore, $(\Gamma, s : Sig_u(\sigma')) \oplus \Gamma' \vdash \overline{s}[v/x]e$ and we note that $[v/x]\overline{s}e = \overline{s}[v/x]e$.

(ν) Suppose $\Gamma, x : \sigma \vdash \nu t : \rho\ P$, $\Gamma' \vdash v : \sigma$ and $\Gamma \oplus \Gamma'$ is defined. Then $\Gamma, x : \sigma, t : \rho \vdash P$ and by inductive hypothesis, $\Gamma \oplus \Gamma', t : \rho \vdash [v/x]P$. Thus $\Gamma \oplus \Gamma' \vdash [v/x](\nu t : \rho\ P)$.

(in) Suppose $\Gamma, x : \sigma \vdash s(y).P, K$, $\Gamma' \vdash v : \sigma$ and $\Gamma \oplus \Gamma'$ is defined. We distinguish two cases.

($x = s$) Suppose $\Gamma, s : \sigma \vdash s(y).P, K$ and $\Gamma' \vdash s : \sigma$ where $\sigma = Sig_u(\sigma')$. Then $\Gamma, s : \sigma, y : \sigma' \vdash P$ and $\uparrow (\Gamma, s : \sigma) \vdash K$. By inductive hypothesis, $(\Gamma \oplus \Gamma'), y : \sigma \vdash [s'/s]P$ and $\uparrow (\Gamma \oplus \Gamma') \vdash [s'/x]K$. Therefore $\Gamma \oplus \Gamma' \vdash [s'/s](s(y).P, K)$.

($x \neq s$) Suppose $\Gamma, s : Sig_u(\sigma'), x : \sigma \vdash s(y).P, K$. Then $\Gamma, s : Sig_u(\sigma'), x : \sigma, y : \sigma' \vdash P$ and $\uparrow (\Gamma, s : Sig_u(\sigma'), x : \sigma) \vdash K$. By inductive hypothesis, $(\Gamma, s : Sig_u(\sigma')) \oplus \Gamma', y : \sigma' \vdash P$ and $\uparrow ((\Gamma, s : Sig_u(\sigma')) \oplus \Gamma') \vdash K$. Therefore, $(\Gamma, s : Sig_u(\sigma')) \oplus \Gamma' \vdash s(y).P, K$.

(par) Suppose $\Gamma, x : \sigma \vdash P_1 \mid P_2$, $\Gamma' \vdash v : \sigma$, $\Gamma = \Gamma_1 \oplus \Gamma_2$, $\Gamma_i, x : \sigma \vdash P_i$, for $i = 1, 2$. By inductive hypothesis, $\Gamma_i \oplus \Gamma' \vdash [v/x]P_i$, for $i = 1, 2$. Thus $\Gamma \oplus \Gamma' \vdash [v/x](P_1 \mid P_2)$.

(rec) Suppose $\Gamma, x : \sigma \vdash A(r_1, \ldots, r_n)$, $A : (\rho_1, \ldots, \rho_n)$, $\Gamma' \vdash v : \sigma$, and $\Gamma \oplus \Gamma'$ is defined. Then $\Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_n$, $\Gamma_i, x : \sigma \vdash r_i : \rho_i$, for $i = 1, \ldots, n$. By inductive hypothesis, $\Gamma_i \oplus \Gamma' \vdash [v/x]r_i : \rho_i$. Thus $\Gamma \oplus \Gamma' \vdash [v/x]A(r_1 \ldots, r_n)$.

(m_s) Suppose $\Gamma, x : \sigma \vdash [s_1 = s_2]P_1, P_2$, $\Gamma' \vdash v : \sigma$ and $\Gamma \oplus \Gamma'$ is defined. Then $\Gamma, x : \sigma \vdash P_i$, for $i = 1, 2$, and by inductive hypothesis $\Gamma \oplus \Gamma' \vdash [v/x]P_i$. Thus $\Gamma \oplus \Gamma' \vdash [v/x]([s_1 = s_2]P_1, P_2)$.

(m_{c}) Suppose $\Gamma, x : \sigma \vdash [u \trianglerighteq \mathsf{c}(\mathbf{x})]P_1, P_2$, $\mathsf{c} : (\sigma_1, \ldots, \sigma_n) \to \sigma'$, $\Gamma' \vdash v : \sigma$, $\Gamma \oplus \Gamma'$ is defined, $\Gamma, x : \sigma \vdash u : \sigma'$, $\Gamma, x : \sigma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash P_1$, and $\Gamma, x : \sigma \vdash P_2$. By inductive hypothesis and (1), $\Gamma \oplus \Gamma' \vdash [v/x]u : \sigma'$, $(\Gamma \oplus \Gamma'), x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash [v/x]P_1$, $\Gamma \oplus \Gamma' \vdash [v/x]P_2$. Thus $(\Gamma \oplus \Gamma') \vdash [v/x]([u \trianglerighteq \mathsf{c}(\mathbf{x})]P_1, P_2)$. □

A form of substitution also arises at the end of the instant when we compute $V(K)$. First, given a pair (E, V) we define $P_{V \setminus E}$ as the program composed of the parallel composition of the emissions of the values in V but not in E. For instance, suppose $v_1 \neq v_2$, $V(s_1) = [v_2; v_1]$, $E(s_1) = \{v_1\}$, $V(s_2) = [v_2]$, $E(s_2) = \emptyset$. Then $P_{V \setminus E}$ is defined as the program $(\overline{s_1}v_2 \mid \overline{s_2}v_2)$.

Lemma 41 (1) *If* $\uparrow (\Gamma) \vdash A(\mathbf{r})$, $\Gamma' \vdash P_V$ *and* $\Gamma \oplus \Gamma'$ *is defined then* $\uparrow (\Gamma \oplus \Gamma') \vdash V(A(\mathbf{r}))$.
(2) *Moreover, suppose there are* V', E *such that* $V, V' \Vdash E$. *Then* $V(A(\mathbf{r})) \approx V(A(\mathbf{r}'))$.

PROOF. (1) Note that $\uparrow(\Gamma')$ is neutral and that $\uparrow(\Gamma \oplus \Gamma') = \uparrow(\Gamma) \oplus \uparrow(\Gamma')$. Suppose $A : (\rho_1, \ldots, \rho_n)$, $\uparrow(\Gamma) = \Gamma_1 \oplus \cdots \oplus \Gamma_n$, and $\Gamma_i \vdash r_i : \rho_i$, for $i = 1, \ldots, n$. If ρ_i is not neutral then $V(r_i) = r_i$ and $\Gamma_i \oplus \uparrow(\Gamma') \vdash r_i : \rho_i$. On the other hand, if ρ_i is not neutral then $V(r_i)$ is obtained from r_i by replacing each occurence of a dereferenced signal $!s$ with $V(s)$. We know that if $\Gamma_i \vdash !s : \sigma$ then $\Gamma' \vdash V(s) : \sigma$ where σ has the shape $Set(\sigma')$ or $List(\sigma')$ depending on the usage associated with the signal s. In particular, if the usage is not e^ω then the list $V(s)$ contains at most one element, for if it contained two or more we could not have $\Gamma' \vdash P_V$. By induction on r_i, we show that $\Gamma_i \oplus \Gamma' \vdash V(r_i) : \rho_i$. Then we conclude that $\uparrow(\Gamma \oplus \Gamma') \vdash V(A(\mathbf{r}))$.

(2) We show by induction on the typing of $\Gamma_i \vdash r_i : \rho_i$ that $V(r_i) \sim_{\rho_i} V'(r_i)$, for $i = 1, \ldots, n$. Then we use the hypothesis that $v_i \sim_{\rho_i} v'_i$ implies that $A(v_1, \ldots, v_n) \approx A(v'_1, \ldots, v'_n)$. □

Finally, we have to check that typing is preserved by expression evaluation.

Lemma 42 *If $\Gamma \vdash e : \rho$ and $e \Downarrow v$ then $\Gamma \vdash v : \rho$.*

PROOF. We proceed by induction on the typing of e.

(*var*) Then the typing judgment must have the form $\Gamma, s : \rho \vdash s : \rho$ and we know that $s \Downarrow s$.

(k) Suppose $e = k(e_1, \ldots, e_n)$. Then $\Gamma = \Gamma' \oplus \Gamma''$, Γ'' is neutral and $\Gamma' \vdash e_i : \sigma_i$ for $i = 1, \ldots, n$. By inductive hypothesis, $e_i \Downarrow v_i$ and $\Gamma' \vdash v_i : \sigma_i$ for $i = 1, \ldots, n$. We distinguish two cases.

($k = \mathsf{c}$) Then $v = \mathsf{c}(v_1, \ldots, v_n)$ evaluates to itself and by the rule (k), $\Gamma \vdash v : \sigma$.

($k = f$) Notice that by weakening $\Gamma \vdash v_i : \sigma_i$. By hypothesis on f, we know that $f(v_1, \ldots, v_n) \Downarrow v$ and $\Gamma \vdash v : \sigma$. □

D.2 Subject reduction

We can now prove proposition 21 which describes how labelled transitions preserve the typing. We restate the proposition including some additional information and the cases that concern the auxiliary transitions.

Proposition 43 (subject reduction) *Suppose $\Gamma \vdash P$. Then:*

(1) *If $P \xrightarrow{s v} P'$, $\Gamma' \vdash \overline{s} v$, and $\Gamma \oplus \Gamma'$ is defined then $\Gamma \oplus \Gamma' \vdash P'$.*

(2) *If $P \xrightarrow{s ? v} P'$ then $\Gamma(s) = Sig_u(\sigma)$ and $u \neq e^\omega$. Moreover if $\Gamma' \vdash v : \sigma$, and $\Gamma \oplus \Gamma'$ is defined then $\Gamma \oplus \Gamma' \vdash P'$.*

(3) *If $P \xrightarrow{\nu \mathbf{t} : \rho \overline{s} v} P'$ then $\Gamma(s) = Sig_u(\sigma)$, $u \in \{e^\omega, o_1^\omega, o_1 \cdot o_0^\omega\}$, and $\Gamma, \mathbf{t} : \rho \vdash P'$. Moreover, there is a Γ' such that $\Gamma' \vdash v : \sigma$ and $(\Gamma, \mathbf{t} : \rho) \oplus \Gamma' = (\Gamma, \mathbf{t} : \rho)$.*

(4) *If $P \xrightarrow{\tau} P'$ then $\Gamma \vdash P'$.*

(5) *If $P \xrightarrow{E, V} P'$, $\Gamma' \vdash P_{V \setminus E}$, and $\Gamma \oplus \Gamma'$ is defined then $\uparrow(\Gamma \oplus \Gamma') \vdash P'$.*

(6) *If $P \xrightarrow{N} P'$ then $\uparrow(\Gamma) \vdash P'$.*

PROOF. In each case, we proceed by induction on the proof of the labelled transition.

(1) The only one way to derive an observable input action is with the rule (*in*). Then P' is $(P \mid \overline{s} v)$ which is typable applying the hypotheses and the typing rule (*par*).

(2) (in_{aux}) If $\Gamma \vdash s(x).P, K$ then by the typing rule (in), $\Gamma(s) = Sig_u(\sigma)$ and $u \neq e^\omega$. Moreover if $\Gamma' \vdash v : \sigma$, and $\Gamma \oplus \Gamma'$ is defined then $\Gamma \oplus \Gamma' \vdash [v/x]P'$ by the substitution lemma 40(4).

($comp$) Suppose, for instance, $\Gamma_1 \oplus \Gamma_2 \vdash (P_1 \mid P_2)$, $\Gamma_i \vdash P_i$, for $i = 1, 2$, and $P_1 \xrightarrow{s?v} P_1'$. By inductive hypothesis, $\Gamma_1(s) = Sig_{u'}(\sigma)$ and $u' \neq e^\omega$. Thus $(\Gamma_1 \oplus \Gamma_2)(s) = Sig_u(\sigma)$ and $u \neq e^\omega$. Moreover, if $\Gamma' \vdash v : \sigma$ and $\Gamma_1 \oplus \Gamma_2 \oplus \Gamma'$ is defined then $\Gamma_1 \oplus \Gamma' \vdash P_1'$, by inductive hypothesis, and therefore $\Gamma_1 \oplus \Gamma_2 \oplus \Gamma' \vdash (P_1' \mid P_2)$.

(ν) Suppose, $\Gamma \vdash \nu t : \rho\ P$. Then $\Gamma, t : \rho \vdash P$ with $t \notin fn(s?v)$. By inductive hypothesis, $(\Gamma, t : \rho)(s) = Sig_u(\sigma)$ and $u \neq e^\omega$. Since $t \neq s$, we also have $\Gamma(s) = Sig_u(\sigma)$. Suppose $\Gamma' \vdash v : \sigma$ and $\Gamma \oplus \Gamma'$ is defined. We can assume $t \notin dom(\Gamma')$ so $(\Gamma, t : \rho) \oplus \Gamma' = (\Gamma \oplus \Gamma'), t : \rho$ is defined too. If $P \xrightarrow{s?v} P'$ then, by inductive hypothesis, $(\Gamma \oplus \Gamma'), t : \rho \vdash P'$. Then we can conclude $\Gamma \oplus \Gamma' \vdash \nu t : \rho\ P'$.

(3) Let $U' = \{e^\omega, o_1^\omega, o_1 \cdot o_0^\omega\}$ be the set of expected usages for the signal on which the output occurs.

(out) Suppose $\overline{s}e \xrightarrow{\overline{s}v} \overline{s}e$ with $e \Downarrow v$. By the typing rule (out), we must have $\Gamma, s : Sig_u(\sigma) \vdash \overline{s}e$ with $u \in U'$, Γ neutral, and $\Gamma \vdash e : \sigma$.

($comp$) Suppose $\Gamma_1 \oplus \Gamma_2 \vdash P_1 \mid P_2$ with $\Gamma_i \vdash P_i$, for $i = 1, 2$, and $P_1 \xrightarrow{\nu \mathbf{t} : \rho \overline{s} v} P_1'$. By inductive hypothesis, $\Gamma_1(s) = Sig_{u'}(\sigma)$, $u \in U'$, $\Gamma_1, \mathbf{t} : \rho \vdash P_1'$, and there is a Γ' such that $\Gamma' \oplus (\Gamma_1, \mathbf{t} : \rho) = \Gamma_1, \mathbf{t} : \rho$ and $\Gamma' \vdash v : \sigma$. Now it must be that $(\Gamma_1 \oplus \Gamma_2)(s) = Sig_u(\sigma)$ with $u \in U'$. Since $(\Gamma_1, \mathbf{t} : \rho) \oplus \Gamma_2 = (\Gamma_1 \oplus \Gamma_2), \mathbf{t} : \rho$, we can conclude that $(\Gamma_1 \oplus \Gamma_2), \mathbf{t} : \rho \vdash (P_1' \mid P_2)$.

(ν) Suppose $\nu t' : \rho'\ P \xrightarrow{\nu \mathbf{t} : \rho \overline{s} v} \nu t' : \rho'\ P'$ because $P \xrightarrow{\nu \mathbf{t} : \rho \overline{s} v} P'$ and t' does not interfere with the action. If $\Gamma \vdash \nu t' : \rho'\ P$ then $\Gamma, t' : \rho' \vdash P$. By inductive hypothesis, $(\Gamma, t' : \rho')(s) = Sig_u(\sigma)$, $u \in U'$, and there is a Γ' such that t does not occur in Γ', $\Gamma' \oplus (\Gamma, t' : \rho', \mathbf{t} : \rho) = (\Gamma, t' : \rho', \mathbf{t} : \rho)$, and $\Gamma' \vdash v : \sigma$. Since $s \neq t'$, we have $\Gamma(s) = Sig_u(\sigma)$. Also by inductive hypothesis, $\Gamma, t' : \rho', \mathbf{t} : \rho \vdash P'$, and therefore $\Gamma, \mathbf{t} : \rho \vdash \nu t' : \rho'\ P'$.

(ν_{ex}) Suppose $\nu t' : \rho'\ P \xrightarrow{\nu t' : \rho', \mathbf{t} : \rho \overline{s} v} P'$ because $P \xrightarrow{\nu \mathbf{t} : \rho \overline{s} v} P'$ and t' occurs free in v. If $\Gamma \vdash \nu t' : \rho'\ P$ then $\Gamma, t' : \rho' \vdash P$. By inductive hypothesis, $(\Gamma, t' : \rho')(s) = Sig_u(\sigma)$, $u \in U'$, and there is a Γ' such that $\Gamma' \oplus (\Gamma, t' : \rho', \mathbf{t} : \rho) = (\Gamma, t' : \rho', \mathbf{t} : \rho)$, and $\Gamma' \vdash v : \sigma$. Also by inductive hypothesis, $\Gamma, t' : \rho', \mathbf{t} : \rho \vdash P'$.

(4) ($synch$) Suppose $\Gamma_1 \oplus \Gamma_2 \vdash P_1 \mid P_2$, where $\Gamma_i \vdash P_i$, for $i = 1, 2$, and $P_1 \xrightarrow{\nu \mathbf{t} : \rho \overline{s} v} P_1'$ $P_2 \xrightarrow{s?v} P_2'$, with the usual side conditions on $\mathbf{t}$. By inductive hypothesis, $\Gamma_1(s) = Sig_u(\sigma)$, $\Gamma_1, \mathbf{t} : \rho \vdash P_1'$, and there is Γ' such that $\Gamma' \oplus (\Gamma_1, \mathbf{t} : \rho) = (\Gamma_1, \mathbf{t} : \rho)$ and $\Gamma' \vdash v : \sigma$. Also by inductive hypothesis, $\Gamma_2(s) = Sig_{u'}(\sigma')$, $u \neq e^\omega$. Since $\Gamma_1 \oplus \Gamma_2$ is defined, we know that $\sigma = \sigma'$, $u \oplus u'$, and $\Gamma_2 \oplus \Gamma'$ are defined. Thus $\Gamma_2 \oplus \Gamma' \vdash P_2'$. Since $(\Gamma_1, \mathbf{t} : \rho) \oplus (\Gamma_2 \oplus \Gamma') = (\Gamma_1, \mathbf{t} : \rho) \oplus \Gamma_2$, we conclude that $(\Gamma_1, \mathbf{t} : \rho) \oplus \Gamma_2 \vdash (P_1' \mid P_2')$ and therefore $\Gamma_1 \oplus \Gamma_2 \vdash \nu \mathbf{t} : \rho\ (P_1' \mid P_2')$.

(rec) Suppose $A(x_1, \ldots, x_n) = P$, $A : (\rho_1, \ldots, \rho_n)$, $x_1 : \rho_1, \ldots, x_n : \rho_n \vdash P$, $\Gamma_i \vdash e_i : \rho_i$, $e_i \Downarrow v_i$, for $i = 1, \ldots, n$, and $\Gamma_1 \oplus \cdots \oplus \Gamma_n$ is defined. Without loss of generality, suppose the first m types ρ_i are *not* neutral and the remaining ones are. Remember that the types ρ_i must be uniform. Then we must have $e_i = s_i$, $\Gamma_i = \Gamma_i', s_i : \rho_i$, Γ_i' neutral, for $i = 1, \ldots, m$ and the names s_i are all distinct, for otherwise, the sum of the contexts is not defined. Iterating the substitution lemma 40(3), for $i = 1, \ldots, m$, we obtain that $s_1 : \rho_1, \ldots, s_m :$

$\rho_m, x_{m+1} : \rho_{m+1}, \ldots, x_n : \rho_n \vdash [s_1/x_1, \ldots, s_m/x_m]P$. We can further weaken, and obtain $(\Gamma_1 \oplus \cdots \oplus \Gamma_m), x_{m+1} : \rho_{m+1}, \ldots, x_n : \rho_n \vdash [s_1/x_1, \ldots, s_m/x_m]P$. Concerning, the values v_i, for $i = m+1, \ldots, n$, we know that $\Gamma_i \vdash v_i : \rho_i$. Then iterating the substitution lemma 40(4), for $i = m+1, \ldots, n$, we obtain that $\Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash [s_1/x_1, \ldots, s_m/x_m, v_{m+1}/x_{m+1}, \ldots, v_n/x_n]P$ as required.

(*matching*) We just spell out the case for $(=_1^{ind})$, the other 3 cases being simpler. Suppose $P = [\mathsf{c}(\mathbf{v}) \trianglerighteq \mathsf{c}(\mathbf{x})]P_1, P_2$ and $P \xrightarrow{\tau} [\mathbf{v}/\mathbf{x}]P_1$. If $\Gamma \vdash P$ then $\mathsf{c} : (\sigma_1, \ldots, \sigma_n) \to \sigma$, and $\Gamma' \vdash \mathsf{c}(\mathbf{v}) : \sigma$, with $\Gamma' \oplus \Gamma = \Gamma$. Moreover, $\Gamma' \vdash v_i : \sigma_i$, for $i = 1, \ldots, n$ and $\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash P_1$. By the substitution lemma 40(4), $\Gamma \vdash [v_1/x_1, \ldots, v_n/x_n]P_1$.

(*comp*) Suppose $\Gamma_1 \oplus \Gamma_2 \vdash P_1 \mid P_2$, where $\Gamma_i \vdash P_i$, for $i = 1, 2$, and $P_1 \xrightarrow{\tau} P_1'$. By inductive hypothesis, $\Gamma_1 \vdash P_1'$ and therefore $\Gamma_1 \oplus \Gamma_1 \vdash P_1' \mid P_2$.

(ν) Suppose $\Gamma \vdash \nu t : \rho\ P$ and $P \xrightarrow{\tau} P'$. Then $\Gamma, t : \rho \vdash P$ and by inductive hypothesis, $\Gamma, t : \rho \vdash P'$. Therefore, $\Gamma \vdash \nu t : \rho P'$.

(5) (0) Suppose $\Gamma \vdash 0$, $0 \xrightarrow{\emptyset, V} 0$, $\Gamma' \vdash P_V$, and $\Gamma \oplus \Gamma'$ is defined. Then $\uparrow (\Gamma \oplus \Gamma')$ is neutral and therefore $\uparrow (\Gamma \oplus \Gamma') \vdash 0$.

(*reset*) Suppose $e \Downarrow v$, $\overline{s}e \xrightarrow{E,V} 0$, $\Gamma \vdash \overline{s}e$, $\Gamma' \vdash P_{V \setminus E}$, and $\Gamma \oplus \Gamma'$ is defined. Then $\uparrow (\Gamma \oplus \Gamma')$ is neutral and therefore $\uparrow (\Gamma \oplus \Gamma') \vdash 0$.

(*cont*) Suppose $s \notin dom(V)$, $s(x).P, K \xrightarrow{\emptyset, V} V(K)$, $\Gamma \vdash s(x).P, K$, $\Gamma' \vdash P_{V \setminus E}$, and $\Gamma \oplus \Gamma'$ is defined. Then by lemma 41(1), $\uparrow (\Gamma \oplus \Gamma') \vdash V(K)$.

(*par*) Suppose $\Gamma_i \vdash P_i$, $P_i \xrightarrow{E_i, V} P_i'$, for $i = 1, 2$, $\Gamma' \vdash V \setminus (E_1 \cup E_2)$, and $\Gamma_1 \oplus \Gamma_2 \oplus \Gamma'$ is defined. Up to commutation and associativity of parallel composition, we can consider that P_1 has the shape $P_{1\cap 2} \mid P_{1\setminus 2} \mid P_{11}$ and P_2 has the shape $P_{1\cap 2} \mid P_{2\setminus 1} \mid P_{22}$, where $P_{1\cap 2}$, $P_{1\setminus 2}$, and $P_{2\setminus 1}$ are the emissions that are in $E_1 \cap E_2$, $E_1 \setminus E_2$, and $E_2 \setminus E_1$, respectively. Denote with $\Gamma_{1\cap 2}$,$\Gamma_{1\setminus 2}$, $\Gamma_{2\setminus 1}$, Γ_{11}, Γ_{22} the contexts that type $P_{1\cap 2}$, $P_{1\setminus 2}$, $P_{2\setminus 1}$, P_{11}, P_{22}, respectively. Note that $\Gamma_{1\cap 2}$, $\uparrow \Gamma_{1\setminus 2}$, and $\uparrow \Gamma_{2\setminus 1}$ are neutral. Then apply the inductive hypothesis to P_i for $i = 1, 2$ and add the contexts to get the assertion.

(6) First, we notice that if $\Gamma \vdash P$ and $P \succeq P'$ then $\Gamma \vdash P'$. Suppose $\Gamma \vdash \nu \mathbf{t} : \rho\ P$ and $P \xrightarrow{E,V} P'$ with $V \Vdash\!\!- E$. Then $P_{V \setminus E} = 0$ and we have $\emptyset \vdash 0$. Hence, by (5), $\uparrow (\Gamma, \mathbf{t} : \rho) \vdash P'$. But the types ρ are uniform and therefore $\uparrow (\Gamma, \mathbf{t} : \rho) = \uparrow \Gamma, \mathbf{t} : \rho$ and we can conclude $\uparrow \Gamma \vdash \nu \mathbf{t} : \rho\ P'$. □

D.3 Commutations

Next we restate and prove proposition 22 which shows that typable programs have strong confluence properties with respect to τ and N transitions.

Proposition 44 *Suppose* $\Gamma \vdash P$ *and* $P \xrightarrow{\alpha} P_i$, *for* $i = 1, 2$.

(1) *If* $\alpha = \tau$ *then* $\exists Q\ (P_i \overset{\tau}{\rightsquigarrow} Q)$, *for* $i = 1, 2$.

(2) *If* $\alpha = N$ *then* $P_1 \approx P_2$.

Proof. (1) Except in the case of a synchronisation, a program can perform a τ transition if it has the shape $C[\Delta]$ where C is a static context and Δ can be reduced according to 5

rules, namely the rule (*rec*) and the 4 rules for matching. In the case of a synchronisation, the program has the shape $C[\overline{s}e][s(x).P, K]$ where C is a two holes (static) context. It is clear that a τ transition according to the first 5 rules cannot interfere with another τ transition. The only possibility that remains to be considered is the one where the program can perform two distinct synchronisations. To have an interference, the synchronisation must be on the same signal name s and according to the typing rules the usage of this signal cannot be e^ω. But then again by the typing rules, we must have only one emission, so that, up to structural equivalence, and assuming $e \Downarrow v$, we are in the following situation:

$$C[\overline{s}e \mid [v/x]P_1 \mid s(y).P_2, K_2], C[\overline{s}e \mid s(x).P_1, K_1 \mid [v/y]P_2] \xrightarrow{\tau} C[\overline{s}e \mid [v/x]P_1 \mid [v/y]P_2] \ .$$

Because an emitted value persists within the instant, one can close the diagram in one step.

(2) We rely on lemma 41(2) and the fact that bisimulation is preserved by static contexts. □

E A size-change termination principle for reactivity

In this appendix, we discuss a variant of the size-change termination (SCT) principle for first-order functional programs discussed in [16] whose goal is to check reactivity of programs in the $S\pi$-calculus. We assume the reader is familiar with [16] (see also [28] for a discussion of the SCT principle from a term-rewriting perspective).

We provide a short introduction to the SCT principle and we fix the notation. Consider a first-order functional program with call-by-value evaluation specified by rules of the shape:

$$f(p_1, \ldots, p_n) \to e$$

where f is a function symbol, p_i are patterns and e an expression. Whenever $e \equiv C[g(e_1, \ldots, e_m)]$ for some context C, we have a potential call from f to g and it is assumed that we have a *symbolic criteria* to compare the arguments $p_1, \ldots, p_n$ of f with the arguments $e_1, \ldots, e_m$ of g.

Next consider a program in the $S\pi$-calculus specified by a system of recursive equations $A(\mathbf{x}) = P$. If P contains a thread identifier B which is not in the 'else' branch of a present statement then we may say that that we have a potential call from A to B within the same instant. Before a call to A produces a call to B, the program may perform a certain number of inputs and branching operations which we need to over-approximate. The general idea is that, modulo this over-approximation, we can regard the system of recursive equations as a system of (pseudo-)term rewriting rules of the shape $A(\mathbf{p}) \to B(\mathbf{e})$.

E.1 Generation of the rewriting rules

We formalise the generation of the (pseudo-)term rewriting rules following ideas already presented in [5].[9] Consider a system of recursive equations $A(\mathbf{x}) = P$ in the $S\pi$-calculus. Without loss of generality, we may assume that the shape of the program at the beginning of each instant is:

$$\nu\mathbf{s}(A_1(\mathbf{e}_1) \mid \cdots \mid A_n(\mathbf{e}_n)) \tag{2}$$

The only information we will keep of a signal name is its type $Sig(\sigma)$. Thus we know the type of the values it may carry. Formally, we select a distinct canonical *constant*, say s, for every

[9] [5] provides static conditions that guarantee *feasible* reactivity, *i.e.*, reactivity in polynomial time.

type $Sig(\sigma)$ and replace in the system every occurrence of a signal name of this type with s. Following this operation, we remove all name generation instructions νs.

As for the operation $[s_1 = s_2]P_1, P_2$ that compares signal names, we will simply disregard it and systematically explore the situations where one of the programs P_1 or P_2 is executed. This is like replacing a conditional $[s_1 = s_2]P_1, P_2$ with an internal choice $P_1 \oplus P_2$.

Now, consider the equation:

$$A(x) = [x \trianglerighteq \mathsf{cons}(x_1, x_2)]A(x_2), B(x) \tag{3}$$

If a call to $A(x)$ leads to a call to $A(x_2)$ then it must be the case that $x = \mathsf{cons}(x_1, x_2)$ for some x_1. We can abstract this situation with the term rewriting rules:

$$A(\mathsf{cons}(x_1, x_2)) \to A(x_2) \qquad A(x) \to B(x) \ .$$

Next, consider the equation:

$$A(x) = s(y).[y \trianglerighteq \mathsf{cons}(y_1, y_2)]B(x, y_1), (C(x) \mid D(x)), E(x) \ . \tag{4}$$

Here a call to $A(x)$ may lead to a call of either $B(x, y_1)$ or $C(x) \mid D(x)$ within the same instant. We may describe this situation with the rules:

$$A(x) \to B(x, y_1), \qquad A(x) \to C(x), \qquad A(x) \to D(x) \ .$$

In this case, the first rule is *not* a term rewriting rule because the variables on the right-hand side are *not* contained in the variables in the left hand side.

Following this discussion, we define a method to associate a set of rules with a system of recursive equations in $S\pi$. For each equation $A(\mathbf{x}) = P$, we compute the function $\mathcal{R}(P, A(\mathbf{x}))$ which is defined on the structure of P as follows (we recall that in $[x \trianglerighteq p]P_1, P_2$ the variable x cannot occur free in P_1):

$\mathcal{R}(P, A(\mathbf{p}))$	$=$ case P of
0	$: \emptyset$
$[x \trianglerighteq p]P_1, P_2$	$: \mathcal{R}(P_1, A([p/x]\mathbf{p})) \cup \mathcal{R}(P_2, A(\mathbf{p}))$
$[s_1 = s_2]P_1, P_2$	$: \mathcal{R}(P_1, A(\mathbf{p})) \cup \mathcal{R}(P_2, A(\mathbf{p}))$
$(P_1 \mid P_2)$	$: \mathcal{R}(P_1, A(\mathbf{p})) \cup \mathcal{R}(P_2, A(\mathbf{p}))$
$\nu s \ P'$	$: \mathcal{R}(P', A(\mathbf{p}))$
$\overline{s}e$	$: \emptyset$
$B(\mathbf{e})$	$: \{A(\mathbf{p}) \to B(\mathbf{e})\}$
$s(y).P', B(\mathbf{r})$	$: \mathcal{R}(P', A(\mathbf{p}))$

Example 45 *For example 1, we derive:*

$$Cell(q, s, \ell) \to Send(q, s, \ell, \ell), \qquad Send(q, s, \ell, \mathsf{cons}(s', \ell'')) \to Send(q, s, \ell, \ell'')) \ .$$

For example 2, no rewriting rule is derived. For example 3, we derive:

$$Handle(s, \mathsf{cons}(\mathsf{req}(s', x), \ell')) \to Handle(s, \ell')), \qquad Handle(s, \ell) \to Server(s) \ .$$

E.2 Size-change termination principle

The generated rules have the shape $A(\mathbf{p}) \to B(\mathbf{e})$. As already mentioned, we assume we have an effective way to 'compare' the arguments $\mathbf{p}$ with the arguments $\mathbf{e}$. Specifically, we assume we have a well-founded order $>_v$ on values and we denote with $\theta, \theta', \ldots$ ground substitutions, *i.e.*, substitutions mapping variables to values. We define a *flow graph* whose nodes are thread identifiers and such that there is an edge from A to B if a call to A may lead to a call of B. Suppose A has n parameters and B has m parameters. Each edge form A to B in the flow graph is labelled with a *size change graph* which is a function

$$G : \{1, \ldots, n\} \times \{1, \ldots, m\} \to \{?, \geq, >\} \ .$$

Intuitively, the abstract value $G(i, j)$ expresses the relationship between the i^{th} parameter of A and j^{th} parameter of B, with $>$ meaning 'strictly greater', $\geq$ meaning 'greater or equal', and ? meaning that the relationship is "unknown'. Formally, we require that: (1) $G(i, j) = >$ implies that $\forall\, \theta \ \ (\theta e_j \Downarrow u \ \supset \theta p_i >_v u)$, and (2) $G(i, j) = \geq$ implies that $\forall\, \theta \ \ (\theta e_j \Downarrow u \ \supset \theta p_i \geq_v u)$. A simple method to compare arguments is to consider the *homeomorphic embedding*. Another possibility is to introduce *quasi-interpretations* as in [5].

The possible computations of the program are now *abstracted* into a *finite* labelled graph. We note that between a node with n arguments and another node with m arguments there are at most $3^{n \cdot m}$ size-change graphs. There is a natural way to compose the abstract values $?, >, \geq$ which is specified as follows:

$\cdot$	$?$	$\geq$	$>$
$?$	$?$	$?$	$?$
$\geq$	$?$	$\geq$	$>$
$>$	$?$	$>$	$>$

This induces a partial operation of composition on size-change graphs:

$$(G_1; G_2)(i, k) = \begin{cases} > & \exists\, j \ \ (G_1(i, j) \cdot G_2(j, k) = >) \\ \geq & \text{o.w. and } \exists\, j \ \ (G_1(i, j) \cdot G_2(j, k) = \geq) \\ ? & \text{o.w.} \end{cases}$$

Note that this operation is *associative*. Given a flow graph F, let $\mathcal{G}(F)$ be the least set containing the size change graphs appearing as labels in F and closed under composition. Note that this is a *finite* set.

A *multi-graph* is a sequence (possibly infinite) of composable size-change graphs. A *thread* in a multi-graph is a directed path in the multi-graph without '?' labels. An *infinitely descending thread* in a multi-graph is a thread that crosses edges of the strict type $>$ infinitely often.

Definition 46 ([16]) *A program satisfies the* size-change termination principle *(SCT principle) if every* infinite *multi-graph associated with the flow graph has an* infinitely descending *thread.*

The following characterisation of the flow graphs that satisfy the SCT principle relies on Ramsey theorem and it is also due to [16].

Proposition 47 ([16]) *A flow graph F satisfies the SCT principle if and only if for any size change graph $G \in \mathcal{G}(F)$ such that $G; G = G$ we have an index l where $G(l, l) = >$.*

Example 48 *We continue the example 45 assuming that we compare the arguments via the homeomorphic embedding,*

In the example 1, we have two nodes $\{Cell, Send\}$ *and edges* $\{(Cell, Send), (Send, Send)\}$. *The only way to have an infinite path in this graph is to loop on the edge* $(Send, Send)$. *The size-change graph* G *that labels this edge is such that* $G(4,4) =>$. *Then, following the characterisation in proposition 47, it is clear that the flow graph satisfies the SCT principle.*

In the example 3, we have two nodes $\{Server, Handle\}$ *and edges* $\{(Handle, Handle), (Handle, Server)\}$. *The only way to have an infinite path in this graph is to loop on the edge* $(Handle, Handle)$. *The size-change graph* G *that labels this edge is such that* $G(2,2) =>$. *Again, it is easily checked that the flow graph satisfies the SCT principle.*

E.3 From the SCT principle to reactivity

We show that the validity of the SCT principle entails reactivity.

Proposition 49 *If the flow graph associated with a system of equations satisfies the SCT principle then every program built over the system is reactive.*

Proof. The construction of the flow graph abstracts the possible computations of a program and in particular it assures that whenever $A(\theta p_1, \ldots, \theta p_n)$ calls $B(\theta e_1, \ldots, \theta e_m)$ there is an edge from A to B in the flow graph with a label G such that if $\theta e_j \Downarrow u$ and $G(i,j) =>$ $(G(i,j) =\geq)$ then $\theta p_i >_v u$ $(\theta p_i \geq_v u)$. At this abstract level, the configuration of a program can be regarded as finite multi-set of the shape:

$$\{\!|A_1(\mathbf{v}_1), \ldots, A_n(\mathbf{v}_n)|\!\}$$

and each abstract computation step amounts to replace an element $A_i(\mathbf{v}_i)$ with a finite multi-set $\{\!|B_{i,1}(\mathbf{u}_1), \ldots, B_{i,n}(\mathbf{u}_n)|\!\}$. By König lemma, an infinite computation must correspond to an infinite sequence of calls:

$$A_1(\mathbf{v}_1) \rightarrow A_2(\mathbf{v}_2) \rightarrow A_3(\mathbf{v}_3) \rightarrow \cdots$$

and there is at least an infinite path in the flow graph and an associated infinite multi-graph $\mathcal{M} = G_1; G_2; G_3; \ldots$ that corresponds to this sequence of calls. By the SCT principle, we should have an infinitely descending thread in the multi-graph. By the properties of size-change graphs, this implies that there is a strictly descending sequence of values. But this contradicts the hypothesis that the order $>_v$ is well-founded. □