1

Determinacy in a synchronous π-calculus

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

Mehdi Dogguy
Université Paris Diderot, PPS, UMR-7126

Abstract

The $S\pi$-calculus is a *synchronous* π-calculus which is based on the SL model. The latter is a relaxation of the ESTEREL model where the reaction to the *absence* of a signal within an instant can only happen at the next instant. In the present work, we present and characterize a compositional semantics of the $S\pi$-calculus based on suitable notions of labelled transition system and bisimulation. Based on this semantic framework, we explore the notion of determinacy and the related one of (local) confluence.[1]

1.1 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

[1] Work partially supported by ANR-06-SETI-010-02.

From Semantics to Computer Science Essays in Honour of Gilles Kahn, eds Yves Bertot, Gérard Huet, Jean-Jacques Lévy and Gordon Plotkin. Published by Cambridge University Press.

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

Our technical approach will follow a process calculus tradition, as listed here.

(i) We describe the interactions of a program with its environment through a *labelled transition system* to which we associate a compositional notion of *labelled bisimulation*.
(ii) We rely on this semantic framework, to introduce a notion of *determinacy* and a related notion of *confluence*.
(iii) We provide *local* confluence conditions that are easier to check and that combined with *reactivity* turn out to be equivalent to determinacy.

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

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

In order to guarantee determinacy in the context of CCS *rendez-vous* communication, it seems quite natural to restrict the calculus so that interaction is *point-to-point*, *i.e.*, it involves exactly one sender and one receiver.[2] In a synchronous framework, the introduction of *signal*-based

[2] 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

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

The $S\pi$-calculus can be regarded as an extension of the SL model where signals can carry values. In this extended framework, it is more problematic to have both concurrency *and* determinacy. Nowadays, this question is frequently considered when designing various kind of synchronous programming languages (see, e.g. [16, 10]). As we have already mentioned, our purpose here is to address the question with the tool-box of process calculi following the work for CCS and the π-calculus quoted above. In this respect, it is worth stressing a few interesting variations that arise when moving from the 'asynchronous' π-calculus to the 'synchronous' $S\pi$-calculus. First, we have already pointed-out that there is an opportunity to move from a point-to-point to a multi-way interaction mechanism while preserving determinacy. Second, the notion of confluence and determinacy happen to coincide while in the asynchronous context confluence is a strengthening of determinacy which has better compositionality properties. Third, reactivity appears to be a reasonable property to require of a synchronous system, the goal being just to avoid instantaneous loops, i.e. loops that take no time.[3]

The rest of the paper is structured as follows. In Section 1.2, we introduce the $S\pi$-calculus, in Section 1.3, we define its semantics based on

unbounded, ordered buffers in a CCS with value passing and show that, modulo this encoding, the determinacy of Kahn networks can be obtained as a corollary of the theory of confluence developed in [17].

[3] 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*".

a standard notion of labelled bisimulation on a (non-standard) labelled transition system and we show that the bisimulation is preserved by static contexts, in Section 1.4 we provide alternative characterisations of the notion of labelled bisimulation we have introduced, in Section 1.5, we develop the concepts of determinacy and (local) confluence. Familiarity with the π-calculus [18, 23], the notions of determinacy and confluence presented in [17], and synchronous languages of the ESTEREL family [6, 8] is assumed.

1.2 Introduction to the $S\pi$-calculus

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

1.2.1 Programs

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

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

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

usual the variables occurring in the pattern p (including signal names) are bound in P_1. $\nu s\ P$ creates a new signal name s and runs P. $(P_1 \mid P_2)$ runs in parallel P_1 and P_2. A continuation K is simply a recursive call whose arguments are either expressions or values associated with signals at the end of the instant in a sense that we explain below. We will also write $\mathsf{pause}.K$ for $\nu s\ s(x).0, K$ with s not free in K. This is the program that waits till the end of the instant and then evaluates K.

1.2.2 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.

1.2.3 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, ...

1.2.4 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 $S\pi$:

$$P = \nu\ s_1, s_2 \left(\begin{array}{ccccc} \overline{s_1}v_1 & | & \overline{s_1}v_2 & | & \\ s_1(x).\ (s_1(y).\ (s_2(z).\ A(x,y)\ \underline{,B(!s_1))}\ \underline{,0})\ \underline{,0} & & & & \end{array} \right)$$

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.

1.2.5 A programming example

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

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

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

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

1.3 Semantics of the $S\pi$-calculus

In this section, we define the semantics of the $S\pi$-calculus by a 'standard' notion of labelled bisimulation on a 'non-standard' labelled transition system and we show that labelled bisimulation is preserved by 'static' contexts. A distinct notion of labelled bisimulation for the $S\pi$-calculus has already been studied in [2] and the following Section 1.4 will show that the two notions are (almost) the same. A significant advantage of the presentation of labelled bisimulation we discuss here is that in the 'bisimulation game' all actions are treated in the same way. This allows

for a considerable simplification of the diagram chasing arguments that are needed in the study of determinacy and confluence in Section 1.5.

1.3.1 Actions

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

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

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

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

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

In order to formalize 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.

1.3.2 Labelled transition system

The labelled transition system is defined in Table 1.1 where rules apply to programs whose only free variables are signal names and with standard conventions on the renaming of bound names. As usual, one can rename bound variables, and the symmetric rules for (*par*) and (*synch*) are omitted. The first 12 rules from (*out*) to (ν_{ex}) are quite close to those of a polyadic π-calculus with asynchronous communication (see [4, 12, 13]) with the following exception: rule (*out*) models the fact that the emission of a value on a signal *persists* within the instant. The last five 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 Section 1.3.1).

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

Definition 1.1 (derivative) *A derivative of a program P is a program Q such that*

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

Definition 1.2 (reactivity) *We say that a program P is reactive, if for every derivative Q every τ-reduction sequence terminates.*

Table 1.1. *Labelled transition system.*

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

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

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

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

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

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

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

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

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

1.3.3 A compositional labelled bisimulation

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

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

This is the standard definition except that we insist on *not* having internal reductions after an N action. Intuitively, we assume that an