The reflexive CHAM and the join-calculus

Cédric Fournet
Cedric.Fournet@inria.fr

Georges Gonthier
Georges.Gonthier@inria.fr

INRIA Rocquencourt [*]
78153 Le Chesnay Cedex
FRANCE

October 1995 [†]

Abstract

By adding reflexion to the chemical machine of Berry and Boudol, we obtain a formal model of concurrency that is consistent with mobility and distribution. Our model provides the foundations of a programming language with functional and object-oriented features. It can also be seen as a process calculus, the join-calculus, which we prove equivalent to the π-calculus of Milner, Parrow and Walker.

1 Introduction

There is a mismatch between calculi for concurrent processes and languages for programming distributed and mobile systems. Calculi such as CCS or the π-calculus [16, 19] introduce a small number of constructs, and have a thoroughly studied metatheory. However, they are mostly based on atomic non-local interaction (typically *rendez-vous*), which is difficult to implement fully in a distributed setting. Programming languages such as Actors [1] or Obliq [8] have separate primitives for transmission and synchronization, for instance remote procedure call and semaphores. However, they also have a much larger set of constructs, usually including imperative primitives, and this hinders their formal investigation.

To bridge this gap, we introduce a new elementary model of concurrency, the reflexive chemical abstract machine. We both use this model as the basis for a practical programming language design, and study this model formally using a process calculus, the join-calculus.

The reflexive CHAM model is obtained from the generic CHAM [6] by imposing locality and adding reflexion. Locality is achieved by barring non-linear reaction patterns; this implies that each reaction rule or molecule can be associated with

[†]To appear in POPL'96. Also available electronically from INRIA

[*]This work is partly supported by the ESPRIT Basic Research Action 6454 - CONFER.

a single reaction site. Reflexion is added by letting reactions extend a machine with new kinds of molecules along with their reaction rules; this lets our model be computationally complete. Our model is more effective than the generic CHAM: molecules travel to their reaction site, instead of having to mix and match. It also turns out that the sequential deterministic subset of our model is basically the continuation-passing style λ-calculus; hence we can embed the λ-calculus using any CPS transform.

Our language design extends a higher-order sequential language with parallelism in expressions (with fork calls) and in function patterns (with join patterns). Jointly defined functions provide the same synchronization capabilities as synchronous channels or concurrent objects. Moreover, join patterns are more consistent with lexical scope: they statically bind (joint) function calls to a body of code, whereas the binding of messages to receptors is dynamic.

The join-calculus is simply the syntactic description of the reflexive CHAM molecules. It is quite similar to the π-calculus, except that it combines restriction, reception, and replication in a single (joint) receptor definition. Our main theorem states that the π-calculus and the join-calculus have the same expressive power, up to weak barbed congruence; it is obtained by exhibiting fully abstract encodings in each direction. As a result we can expect most of the π-calculus metatheory to carry over directly to the join-calculus.

2 Overview

Most process calculi are based on synchronous channels. A channel is an abstraction of the communication media on which data is exchanged; send and receive operations on channels provide a concise denotation for the transmission, routing, and synchronization that actually occur in a concurrent system. The π-calculus [19, 17] has demonstrated that, in combination with an elegant scope management technique, channel operations are computationally complete. The PICT experiment [21, 24, 23] has further shown that the π-calculus, more specifically its asynchronous fragment, can be used as the basis of a useful higher-order concurrent language, in a non-distributed setting.

In a distributed setting, however, channels introduce atomic interaction between distant emitters and receivers (communication *in the ether* [16]). This can be difficult to implement, even more so if recovery from local failures is also supported: unless the channel implementation includes a sophisticated fault-tolerant consensus protocol, some of the implementation details will be revealed by failures. This problem occurs even in the asynchronous setting, as there is interaction between distant receivers, through contention.

On the other hand, channels are not absolutely required for high-level distributed programming. For instance, they are not primitive in object-based languages [1, 8]; unfortunately, these languages lack an abstract foundation as simple and precise as the π-calculus. It is such a model that we purport to develop in this paper. Our starting point will be the chemical abstract machine, which can be regarded as the computational model of the π-calculus.

Litterally, CHAM interaction is local, since *molecules* simply move around in a *solution*, until they meet in matching pairs and react; but the random motion in this description is not very effective. Assuming that the chemical rules have disjoint domains, the CHAM also has a more operational interpretation: all molecules travel to a *reaction site* associated to their rule, where they are sorted, matched, and made to react (figuratively, reactions are "catalyzed" at the sites).

Under this interpretation, however, the CHAM is not very concurrent: communication is centralized in a fixed set of sites (catalyzers are bottlenecks), and the management of each site is complex, as the number of different expected molecule shapes can grow arbitraly (catalyzers clog up). It would be much better to have a larger number of sites with simpler matching instead, and this is exactly what the reflexive CHAM modifications bring in, by allowing dynamic creation of sites and restricting reaction patterns.

We now sketch the basic mechanisms of the reflexive CHAM; the formal definition is exposed in section 3. Our model operates on higher-order solutions $\mathcal{R} \vdash \mathcal{M}$ comprising two multisets. The molecules $\mathcal{M}$ represent the processes running in parallel; the reactions $\mathcal{R}$ define the current reduction rules.

Names are the only values in our model, as in the π-calculus. They have a twofold usage: port names, and transmitted values. We write $x\langle y\rangle$ to mean that the name y is sent on the name x.

An atom is a pending message $x\langle y\rangle$. A compound molecule consists of several sub-molecules, glued by the join operator "|". Molecules can be heated into smaller ones, in a reversible way. As a first example, we consider a print spooler with two ports: available printers like laser send their name on the port named ready, while users send the filenames 1, 2 to be printed on the port named job. There are three atoms in solution on the first line, versus one atom and one compound molecule on the second line, where the molecule joins the laser-printer and the file 1. The structural equivalence $\rightleftharpoons$ relates these two solutions, without reactions yet.

$$\begin{array}{rcl} & \emptyset & \vdash \quad \mathsf{ready}\langle\mathsf{laser}\rangle, \quad \mathsf{job}\langle 1\rangle, \quad \mathsf{job}\langle 2\rangle \\ \rightleftharpoons & \emptyset & \vdash \quad \mathsf{ready}\langle\mathsf{laser}\rangle \mid \mathsf{job}\langle 1\rangle, \quad \mathsf{job}\langle 2\rangle \end{array}$$

Denoted D or $J \triangleright P$, a reaction consumes compound molecules that have a specific join pattern J, and produces new molecules in the solution that are copies of P where the formal parameters of J have been instantiated to the transmitted values. This corresponds to reduction steps on the whole solution $(\mathcal{R} \vdash \mathcal{M}) \longrightarrow (\mathcal{R} \vdash \mathcal{M}')$. Continuing our example, we add a reaction that matches printers and jobs, then sends the filename to the printer:

$$D \quad = \quad \mathsf{ready}\langle\mathsf{printer}\rangle \mid \mathsf{job}\langle\mathsf{file}\rangle \triangleright \mathsf{printer}\langle\mathsf{file}\rangle$$

We now add this chemical reaction in our solution, and we use it to reduce our previous molecule and generate a new atom. Notice that non-determinism comes from $\rightleftharpoons$, and is just committed by the reaction.

$$\begin{array}{rcl} & D & \vdash \quad \mathsf{ready}\langle\mathsf{laser}\rangle \mid \mathsf{job}\langle 1\rangle, \quad \mathsf{job}\langle 2\rangle \\ \longrightarrow & D & \vdash \quad \mathsf{laser}\langle 1\rangle, \quad \mathsf{job}\langle 2\rangle \end{array}$$

Our model is reflexive, meaning that reactions can be dynamically created. This is done by our last kind of molecule. The defining molecule $\texttt{def}\, D\ \texttt{in}\ P$ can be heated in two parts, a new reaction D and a molecule P. In this case, the newly defined ports can be used in both Q and P. The solution we just considered could have come from a single molecule, with the structural rules:

$$\begin{array}{rcl} \emptyset & \vdash & \texttt{def}\, D\ \texttt{in}\ \mathsf{ready}\langle\mathsf{laser}\rangle \mid \mathsf{job}\langle 1\rangle \mid \mathsf{job}\langle 2\rangle \\ \rightleftharpoons\ D & \vdash & \mathsf{ready}\langle\mathsf{laser}\rangle \mid \mathsf{job}\langle 1\rangle \mid \mathsf{job}\langle 2\rangle \\ \rightleftharpoons\ D & \vdash & \mathsf{ready}\langle\mathsf{laser}\rangle \mid \mathsf{job}\langle 1\rangle, \quad \mathsf{job}\langle 2\rangle \end{array}$$

A more realistic spooler would send the name job to its users, and the name ready to its printer drivers. This corresponds to the well-known scope-extrusion of the π-calculus. However, our definitons have a strict lexical discipline: the behaviour of ready and job may not be deterministic, but it is statically defined. Other processes that receive these names may send messages, but they cannot add new reactions for them. This essential restriction to reflexion lets us extend the language safely. For instance, special names used only in δ-rules may be added to the machine without special care, while in the π-calculus any process may mistakenly alter their behaviour.

In section 4, we expand the model into a simple programming language with mobility, and we illustrate some of its features. From the programmer's point of view, it is a high-level concurrent language with lexical scope and asynchronous messages. We identify function calls as a special case of message passing with CPS: we analyze two reduction strategies for the λ-calculus, then we define some convenient syntactic sugar for sequential control.

The language also has object-oriented features. Elementary objects are defined by new names and new reaction rules: methods are the names that are returned, behaviours are declared in the rules, states are held in messages on internal names. Elaborate synchronization schemes can be expressed among these concurrent objects by pattern-matching on their rules. Our firm commitment to lexical scoping makes our objects very static, meaning that more imperative features such as cloning must be explicitly encoded.

In sections 5 and 6 we explore the properties of the join-calculus and its relation to the π-calculus. The join-calculusis the process calculus induced by the reflexive CHAM. We first define the observational equivalence, then we use it as a basis to compare the relative expressive powers of different calculi. Our translations between calculus are proved fully abstract with regards to weak barbed congruence; in that sense, our technical results are precise up-to substitution of encodings in any context of the host calculus. In section 5, we strip the join-calculus of convenient but unnecessary features: recursion, join patterns including more than two messages, polyadic messages, definitions with several clauses, and we obtain a *core join-calculus* that retain the expressive power of our model. In section 6, we compare this core join-calculus and the asynchronous π-calculus [11]. In spite of significant differences, both calculi provide exactly the same expressive power. However, their scoping conventions makes the accurate encodings surprisingly complex. We present both simple and accurate encodings, and we discuss their characteristics, which illuminate what separates the two calculi.

We conclude the paper with a few words on future work. An implementation is under way, to evaluate our language in practice, and we mention the extensions to support types, explicit distribution, failure-detection and migration.

In the annexes A and B, we sketch the proofs of full abstraction for the two encodings between the π-calculus and the join-calculus that are described in section 6. These results are obtained using auxiliary encodings and bisimulation-based techniques, in particular weak bisimulation up-to expansion as proposed in [25].

2.1 Related work

To our knowledge, Banâtre [5] was the first to suggest "multi-functions" as primitives for synchronization. They correspond to a first-order version of our join definitions, in a procedural and synchronous language. Our work is more directly related to the recent "asynchronous" trend of the π-calculus [11, 10, 7], and from its first applications [23, 21].

Our calculus focuses on mobility in a minimal setting. This contrast with extensions for concurrency from an object-oriented or a functional kernel [1, 9, 8]. Likewise, distributed systems built on the actor paradigm [1, 2] proposed a two-layered architecture with a functional kernel wrapped in an imperative extension for communication.

Other calculi introduce concurrency and/or distribution using different primitives. Instead of directed communication with a functional flavour, they rely for instance on unification and broadcast. This is the case for Oz [26], and for linear objects [4].

3 The reflexive chemistry

We first give the syntax of processes, and the scope for their names. Then we present the reflexive chemical machine, and we illustrate it on a few simple examples.

3.1 Names, Processes, Definitions

Values in the reflexive CHAM are only names, as this is the case in the π-calculus. Let $\mathcal{N}$ be an infinite set of names; we use name variables in lowercase letters $x \in \mathcal{N}$ to denote its elements. In the following, $\widetilde{x}$ is a notation for a tuple of name variables $x_1, x_2, \cdots x_n$.

The following grammar defines processes, join-patterns and definitions. A process P is an emission of an asynchronous polyadic message $x\langle\widetilde{v}\rangle$, a definition of new names, or a parallel composition of processes. A definition D consists of one or several elementary definitions $J \triangleright P$ that match patterns J joining messages to guarded processes P, connected by the $\wedge$ operator. It entirely describe the behaviour of its defined names.

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
P & \stackrel{\text{def}}{=} & x\langle\widetilde{v}\rangle & J & \stackrel{\text{def}}{=} & x\langle\widetilde{v}\rangle & D & \stackrel{\text{def}}{=} & J \triangleright P \\
 & & \texttt{def}\, D \;\texttt{in}\; P & & & J|J & & & D \wedge D \\
 & & P|P & & & & & &
\end{array}$$

Names that appear in a process P may be captured by an enclosing definition. The only binder is the join pattern J, but the scope of its names depends of their position in messages. The formal parameters that are received are bound in the corresponding guarded process. The defined port names are bound in the whole defining process, that is, the main process and recursively all the guarded processes. Received variables $rv(J)$, defined variables $dv(J)$ and $dv(D)$, and free variables $fv(D)$ and $fv(P)$ are specified by structural induction. Notice the syntactic restriction for processes: No received variable may appear twice in the same pattern J. This rules out any comparison on names, and guarantees that join patterns remain linear.

$$
\begin{array}{lcl}
rv(x\langle\tilde{v}\rangle) & \stackrel{\text{def}}{=} & \{u \in \tilde{v}\} \\
rv(J|J') & \stackrel{\text{def}}{=} & rv(J) \uplus rv(J') \\
\\
dv(x\langle\tilde{v}\rangle) & \stackrel{\text{def}}{=} & \{x\} \\
dv(J|J') & \stackrel{\text{def}}{=} & dv(J) \cup dv(J') \\
\\
dv(J \triangleright P) & \stackrel{\text{def}}{=} & dv(J) \\
dv(D \wedge D') & \stackrel{\text{def}}{=} & dv(D) \cup dv(D') \\
\\
fv(J \triangleright P) & \stackrel{\text{def}}{=} & dv(J) \cup (fv(P) - rv(J)) \\
fv(D \wedge D') & \stackrel{\text{def}}{=} & fv(D) \cup fv(D') \\
\\
fv(x\langle\tilde{v}\rangle) & \stackrel{\text{def}}{=} & \{x\} \cup \{u \in \tilde{v}\} \\
fv(\texttt{def}\, D \;\texttt{in}\; P) & \stackrel{\text{def}}{=} & (fv(P) \cup fv(D)) - dv(D) \\
fv(P|P') & \stackrel{\text{def}}{=} & fv(P) \cup fv(P')
\end{array}
$$

A name is fresh with regards to a process or a solution when it is not free in them. In the following, we use substitutions σ and $\{^x/_y\}$, with possibly implicit α-renaming on non-free variables to avoid name clashes.

While this is not needed in the join-calculus, we will assume that for any given name variable the number of arguments is the same in every message and in every join-pattern. Formally, this amounts to use a recursive sort discipline and to consider only well-sorted processes, as for the π-calculus[18, 22].

3.2 Operational semantics

We extend the chemical approach of Berry and Boudol [6] with reflexion. We first give some heating/cooling reversible rules $\rightleftharpoons$. This corresponds to the underlying structural equivalence on processes, and includes reflexion. Once molecules have been suitably dissolved, the single reduction rule $\longrightarrow$ expresses the mechanism of communication in a much simpler way than for the π-calculus.

Rules operate on higher-order solutions $\mathcal{R} \;\vdash \mathcal{M}$. On the right-hand-side, active processes are "molecules" in the multiset $\mathcal{M}$, on the left-hand-side, active definitions are "reactions" in the multiset $\mathcal{R}$. For the sake of simplicity, we only mention the elements of both multisets that participate in the rule, separated by commas.

$$\begin{array}{llclcll}
\text{(str-join)} & & \vdash & P|Q & \rightleftharpoons & & \vdash & P, Q \\
\text{(str-and)} & D \wedge E & \vdash & & \rightleftharpoons & D, E & \vdash & \\
\text{(str-def)} & & \vdash & \texttt{def}\, D \;\texttt{in}\; P & \rightleftharpoons & D\sigma_{\text{dv}} & \vdash & P\sigma_{\text{dv}} \\
\\
\text{(red)} & J \triangleright P & \vdash & J\sigma_{\text{rv}} & \longrightarrow & J \triangleright P & \vdash & P\sigma_{\text{rv}}
\end{array}$$

Side-conditions for substitutions:
(str–def) σ_{dv} instantiates the port variables $dv(D)$ to distinct, fresh names: $Dom(\sigma_{\text{dv}}) \cap (fv(\mathcal{R} \vdash \mathcal{M})) = \emptyset$
(red) σ_{rv} substitutes the transmitted names for the distinct received variables $rv(J)$.

The first two structural rules express that "|" and "∧" are commutative and associative. (str-def) describes the heating of a molecule that defines new names. The restriction on σ_{dv} is reminiscent of the restriction prefix ν in the π-calculus, with regards to scope extrusion, and at the same time enforces a strict static scope for the definitions. (red) is a meta reduction rule that associates the actual reduction rule to each reaction in $\mathcal{R}$. In one computation step, such reductions consume any molecule with a given port pattern, make a fresh copy of their guarded process, substitute its received parameters for the sent names, and release the process as a new floating molecule.

3.3 Some examples

We now give examples of processes and definitions, along with an intuitive description of their meaning. The formal treatment of observations is deferred until section 5.

$$\texttt{def}\, x\langle u\rangle \triangleright y\langle u\rangle \;\texttt{in}\; P \tag{1}$$

$$\texttt{def}\, y\langle u\rangle \triangleright x\langle u\rangle \;\texttt{in}\; \texttt{def}\, x\langle u\rangle \triangleright y\langle u\rangle \;\texttt{in}\; P \tag{2}$$

$$\texttt{def}\, x_1\langle u\rangle | x_2\langle v\rangle \triangleright x\langle u, v\rangle \;\texttt{in}\; P \tag{3}$$

$$\texttt{def}\, x\langle v\rangle | y\langle \kappa\rangle \triangleright \kappa\langle v\rangle \;\texttt{in}\; P \tag{4}$$

$$\texttt{def}\, s\langle\rangle \triangleright P \wedge s\langle\rangle \triangleright Q \;\texttt{in}\; s\langle\rangle \tag{5}$$

$$\texttt{def}\, once\langle\rangle | y\langle v\rangle \triangleright x\langle v\rangle \;\texttt{in}\; y\langle 1\rangle | y\langle 2\rangle | y\langle 3\rangle | once\langle\rangle \tag{6}$$

$$\texttt{def}\, loop\langle\rangle \triangleright P | loop\langle\rangle \;\texttt{in}\; loop\langle\rangle | Q \tag{7}$$

The simpler definitions perform some wiring between names: in (1) messages on the local name x in P are forwarded to the outside as messages on y; in (2) the leftmost x is a free name, while the rightmost one is locally bound in P, and will require renaming; however, messages on the local x are still forwarded in two steps on the external one; (3) performs multiplexing of messages on x whose parts are supplied on x_1 and x_2; (4) was introduced as a print spooler in the overview, but it more generally models π-calculus-like channels, as values are sent on x and requests for values are sent on y, to me matched in the definition; (5) and (6) both express *internal non-determinism* $P + Q$ using a compound definition, and $x\langle 1\rangle + x\langle 2\rangle + x\langle 3\rangle$ using the message on *once* as a lock; (7) replicates the process P, starting a new copy each time the definition is used.

We finish our series with a longer example that illustrates both higher-order and the use of internal messages to store some local state. A reference cell abstraction is defined as:

$$\texttt{def}\; mkcell\langle v_0, \kappa_0\rangle \;\triangleright\; \left(\begin{array}{rl} \texttt{def} & get\langle\kappa\rangle | s\langle v\rangle \triangleright \kappa\langle v\rangle | s\langle v\rangle \\ \wedge & set\langle u, \kappa\rangle | s\langle v\rangle \triangleright \kappa\langle\rangle | s\langle u\rangle \\ \texttt{in} & s\langle v_0\rangle \mid \kappa_0\langle get, set\rangle \end{array}\right)$$

Each *mkcell* message triggers the external definition, which in turn defines three fresh names get, set, s. The first two are sent back on κ_0 for later access or update to the new cell. Thanks to lexical scoping, the last name s remains local, and the initial message $s\langle v_0\rangle$ together with the two internal rules guarantee the invariant of the cell: there is exactly one message on s, which contains the current value.

4 Programming in the join-calculus

We now use the reflexive CHAM as the foundation of a concurrent programming language. While our model already provides enough expressive power, its features are too low-level for actual programming. For instance, there is no convenient way to express sequential control in a process, which strongly suggest the use of some syntactic sugar. We first study the embedding of higher-order functional programming. using continuation-passing styles, we encode two reduction strategies for the λ-calculus in clear-cut subsets of the join-calculus. Then we describe a toy concurrent language based on these ideas, and we give some programming examples. We finally discuss object-oriented features. programming.

4.1 Two encodings of the λ-calculus

Definitions of the form "$\texttt{def}\, f\langle x\rangle \triangleright P \;\texttt{in}\; Q$" seems to be very similar to the "$\texttt{let}\; f(x) = E \;\texttt{in}\; E'$" statement in functional programming. In particular, they share the same static scoping discipline. The major difference comes from asynchrony in our model, meaning that we must explicitly create and send continuations.

For a given CPS, we encode λ-terms as processes that can be triggered, and we compare their respective behaviour. With minor adaptations, we obtain results of adequacy similar to those for the π-calculus[17]: The terms and their translations converge or diverge accordingly. Our purpose here is to illuminate the tight connection between functions and join-definitions, which makes our encodings simpler than [17, 7]. Our syntax for the λ-calculus is as usual:

$$T \;\stackrel{\text{def}}{=}\; x \mid \lambda x.T \mid TT$$

Call-by-name: in this reduction strategy, λ-terms are reduced in leftmostorder and no reduction may occur under a λ. Our encoding is:

$$\begin{array}{rcl}
[\![x]\!]_v & \stackrel{\text{def}}{=} & x\langle v\rangle \\
[\![\lambda x.T]\!]_v & \stackrel{\text{def}}{=} & \texttt{def}\ \kappa\langle x,w\rangle = [\![T]\!]_w\ \texttt{in}\ v\langle\kappa\rangle \\
[\![TU]\!]_v & \stackrel{\text{def}}{=} & \texttt{def}\ x\langle u\rangle = [\![U]\!]_u \\
 & & \texttt{in def}\ w\langle\kappa\rangle = \kappa\langle x,v\rangle\ \texttt{in}\ [\![T]\!]_w
\end{array}$$

Intuitively, the process $[\![T]\!]_v$ sends its value on v, a value is a process that serves evaluation requests sent on κ, and requests supply two names: x to send requests for the value of the argument, and w to eventually return a value when evaluation converges.

The image of the translation is exactly the *deterministic subset* of the join-calculus, defined as the set of processes that contain no parallel composition, and neither join-pattern nor "$\wedge$" in definitions. As expected, reductions for processes in this subset are entirely sequential.

Parallel call-by-value: the λ-term (TU) can be reduced as soon as both T and U have been reduced to values, thus allowing the function and the argument to be evaluated in parallel. Again, no reduction may occur under a λ. Using a larger subset of the join-calculus, we encode this confluent but non-deterministic reduction strategy:

$$\begin{array}{rcl}
[\![x]\!]_v & \stackrel{\text{def}}{=} & v\langle x\rangle \\
[\![\lambda x.T]\!]_v & \stackrel{\text{def}}{=} & \texttt{def}\ \kappa\langle x,w\rangle = [\![T]\!]_w\ \texttt{in}\ v\langle\kappa\rangle \\
[\![TU]\!]_v & \stackrel{\text{def}}{=} & \texttt{def}\ t\langle\kappa\rangle|u\langle w\rangle = \kappa\langle w,v\rangle\ \texttt{in}\ [\![T]\!]_t|[\![U]\!]_u
\end{array}$$

Again, the encoding $[\![T]\!]_v$ sends its value on v and a value is a process that serves evaluation requests sent on κ, but evaluation requests now supply the value of the parameter along with a name for the value of the term.

The image of the translation now uses parallel composition to capture the non-determinism of the strategy. The symmetry between the evaluation of the function and of the argument is apparent, backed by the two symmetries, on the fork of evaluation requests and on the join of their results.

4.2 A language with sequencing

In our basic model, synchronization happens only as molecules are consumed, and this suffices to express control flow. In practice however, the resulting programs would contain many explicit continuations and would be difficult to understand. Instead, we make the sequential control apparent: we fix a CPS, and provide it as syntactic sugar in the language. To this end, the new grammar extends the syntax in two steps:

- Names are split in two families: *synchronous* and *asynchronous*
- Processes can consist of series of instructions $\{I^*\}$ that are executed sequentially.

P	$\overset{\text{def}}{=}$	$x(\widetilde{v})$	asynchronous message
		$\{I^*\}$	sequence of instructions
		$P\|P$	parallel composition
I	$\overset{\text{def}}{=}$	$\texttt{def}\ J = P\ [\texttt{and}\ J' = P']^*$	recursive definition
		$\texttt{let}\ \widetilde{v} = V$	named values
		$\texttt{run}\ P$	asynchronous process
		$\texttt{do}\ f(\widetilde{V})$	synchronous call
		$\texttt{if}\ V\ \texttt{then}\ I\ [\texttt{else}\ I]$	conditional
		$\texttt{return}\ \widetilde{V}\ \texttt{to}\ f$	implicit continuation
V	$\overset{\text{def}}{=}$	v	value (name,...)
		$f(\widetilde{V})$	synchronous call
J	$\overset{\text{def}}{=}$	$x(\widetilde{v})$	asynchronous message
		$f(\widetilde{v})$	synchronous message
		$J\|J$	join of several messages

As in the calculus, *Asynchronous names* x are defined and used for asynchronous messages; *Synchronous names* f are names that implicitly transmit a continuation in every message. We extend the sort discipline to distinguish names consistently. Whenever a message is sent to a synchronous name, a continuation channel is defined as the remaining part of the current instruction sequence, and the continuation is added to the message. Whenever such a message is received as part of a join pattern, the continuation is bound in the corresponding guarded process, and may be used to send back results using the `return` instruction. Briefly, `let` binds names from synchronous calls; `do` does the same when the result is a synchronisation signal (); `run` asynchronously forks a process; `return` (asynchronously) sends results back on the continuation that was received on f. Finally, any value may contain nested synchronous calls. The formal translation is omitted.

While the underlying model is the same, this smoothly merges in a declarative style some non-deterministic programming in a functional framework. For definitions with only one message in their pattern, as is particular for continuations, the substitution lemma holds, as the instantiated body can be substituted for the calling message, as in any functional language.

In our examples, synchronous names are capitalised; for instance the *mkcell* example is the compilation of the program

```
def MKCELL(v0) =
{ def GET()  | s(v)  = s(v)| {return v to GET}
  and SET(u) | s(v)  = s(u)| {return to SET}
  run s(v0)
  return (GET,set) to MKCELL }
```

The second example elaborates on the print spooler of the overview, in an imperative style. Now, the user select a printer and a format, and files are pre-processed accordingly before printing. The channels PRINT and ENSCRIPT are synchronous calls to the library. At run-time, the files letter and note are transcripted and laser-printed, then the current printer is changed, then the file drawing is printed in colour.

```
{ def NEWPRINTER(PRINT,format) | current(_,_) =
        { run current(PRINT,format)
          return to NEWPRINTER }
  and JOB(file) | current(PRINT,format)) =
        { run current(PRINT,format)
          do  PRINT(ENSCRIPT(file,format))
          return to JOB }
  run current(LASER,ps)
  do  JOB(letter)
  do  JOB(note)
  do  NEWPRINTER(colour,pscolour)
  do  JOB(drawing) }
```

While this style mostly comes from the design of PICT [24], the functional syntax for emissions and the static definition of receptions make it more direct and allow a finer control. The main drawback is that whenever a PICT channel is actually used with several emitters and several receptors in parallel, it must be compiled into a join-definition (see example 4); fortunately, this is uncommon in programming examples. Our approach also offers more declarativeness than object-based languages, since there is no need to mutate systematically the receptor.

Concerning the implementation, the set of rules that comes from a definition is independent from any other definition. Taking advantage of asynchrony, these rules are managed locally by queues for messages, and by an automaton that matches them with join patterns and forks accordingly the guarded processes. To this end, well-known compilation techniques are available [15]. Besides, the embedding of large functional-style definitions can be made reasonably efficient using tail-recursion-like optimizations. Finally, concrete values and built-in functions can easily be added: The behaviour of their reserved names is given by specific δ-rules that describe the consumption of their messages, and are implemented as low-level function calls.

4.3 Concurrent objects and synchronization patterns

Our model provides the essential features of objects, as is already the case for the π-calculus [28]. First, we consider primitive objects that are already present in the language: Using message-passing and pattern-matching in our definitions, we encode objects as servers that receive requests to execute their methods.

The design of a full-fledged object-oriented language would require some more encoding: For instance, inheritance (or cloning) is not primitive. We sketch some features to support more general objects with dynamic definitions and inheritance.

4.3.1 Primitive objects

Objects are created in definitions, whose port names may be either returned and made public, or kept private in the body of their definition. In that sense, our cell example is a simple primitive object. We identify names and methods, definitions and active concurrent objects. The current state of an object can be split into several components held on internal messages, according to the critical sections. Besides, the interface may feature several states with different synchronization capabilities. The declarative pattern-matching on join messages is much more expressive than the serialization of method calls: It contains the expressiveness of coloured Petri nets, and can even be dynamically expanded.

We illustrate the combination of concurrency and synchronization on the example of priority queues (figure 2):

```
def MK_PRIORITY_QUEUE() =
{ def EMPTY() | none() =
      { run none()
        return TRUE to EMPTY }
  and EMPTY() | some(x,E,A,R) =
      { run some(x,E,A,R)
        return FALSE to EMPTY }
  and ADD(x)  | none() =
      { return to ADD
        let E,A,R = MK_PRIORITY_QUEUE() }
        run some(x,E,A,R) }
  and ADD(x)  | some(y,E,A,R) =
      { return to ADD
        do  A(MAX (x,y))
        run some(MIN(x,y),E,A,R) }
  and REMOVE()| some(x,E,A,R) =
      { return x to REMOVE
        if I() then run none()
               else run some(R(),E,A,R) }
  run none()
  return EMPTY,ADD,REMOVE to NEW_PRIORITY_QUEUE }
```

Our priority queue features three synchronous methods `EMPTY`, `ADD`, `REMOVE` with the expected meaning; `REMOVE` retrieves the smallest value, or blocks until a value is available. There are two internal states, `none()` when empty, and `some(x,E,A,R)` when containing the smallest value `x` in its head and another priority queue with methods `E,A,R` in its tail. Statically, we can check that there is always exactly one state message available for each definition. Values can be concurrently tested, added, and removed; in particular, a new `some` message is released after at most one comparison when a new value is added, while the update propagates toward the tail in parallel. When the tail is eventually reached, a new, empty priority queue is created using the recursive definition `MK_PRIORITY_QUEUE`, which returns three fresh methods on an empty priority queue to be stored in the last-but-one `some` message.

4.3.2 Class-based objects and inheritance

Our primitive objects lack dynamicity, because the lexical scope of their definitions forbids overloading or cloning. It is well-known that inheritance and synchronization for concurrent objects do not merge gracefully; in our case, we can recover ad-hoc dynamicity using indirections. In our spooler example, a dynamic method to print files is implemented by two static methods, job for invocation and newprinter for overriding, while the current name associated with the method is kept in the internal state currentprinter. Likewise, one can substitute state overwriting for method overriding in many cases, and mix freely static and dynamic components within the same objects.

While our solution seems better than the traditional object-as-server encoding, it requires more than some local syntactic sugar. An alternative approach consists in complementing the join-calculus with new features, e.g. with more general records, to obtain richer primitive objects.

5 The join-calculus

The join-calculus is simply the set of molecules of the reflexive CHAM. In this section, we study in more details its properties as a process calculus: we first give another, equivalent definition of a variant of the join-calculus. then we briefly discuss observation, and we identify observational equivalence as a barbed bisimulation congruence; finally, we use this tool to precisely reduce the join-calculus to its essential features.

Our reflexive chemical machine entirely defines the syntax (molecules as processes), the structural congruence ($\rightleftharpoons$), and the reduction relation ($\rightleftharpoons^* \longrightarrow \rightleftharpoons^*$). Any chemical solution can be cooled down into a single process, wrapping all the reactions in a big definition header. Thus, join-calculus processes provide another presentation of our model as a first-order rewriting system modulo structural equivalence. This more syntactic approach is especially useful to compare our model to other calculi (several subsets of the join-calculus in this section, and the π-calculus in the next one).

5.1 The join-calculus as a process calculus

The core (recursive) join-calculus is a restriction of the full calculus with simpler definitions, join patterns and messages. Its syntax is given by the grammar:

$$P \overset{\text{def}}{=} x\langle u\rangle \quad | \quad P_1|P_2 \quad | \quad \texttt{def } x\langle u\rangle|y\langle v\rangle \triangleright P_1 \texttt{ in } P_2$$

As before, the scope of u, v is P_1, whereas the scope of x, y extends to the whole definition. The structural congruence $\equiv$ is the smallest relation such that for all processes P, Q, R, S, for all definitions D, D' such that $dv(D), dv(D')$ contain only fresh names,

$$\begin{array}{rcl} P|Q & \equiv & Q|P \\ (P|Q)|R & \equiv & P|(Q|R) \\ P|\ \texttt{def}\ D\ \texttt{in}\ Q & \equiv & \texttt{def}\, D\ \texttt{in}\ P|Q \\ \texttt{def}\, D\ \texttt{in}\ \texttt{def}\, D'\ \texttt{in}\ P & \equiv & \texttt{def}\, D'\ \texttt{in}\ \texttt{def}\, D\ \texttt{in}\ P \end{array}$$

$$\begin{array}{rcl} P \equiv_\alpha Q & \Longrightarrow & P \equiv Q \\ P \equiv Q & \Longrightarrow & P|R \equiv Q|R \\ R \equiv S, P \equiv Q & \Longrightarrow & \texttt{def}\, J \triangleright R\ \texttt{in}\ P \equiv \texttt{def}\, J \triangleright S\ \texttt{in}\ Q \end{array}$$

We now define the reduction relation as the τ-transitions of a labelled transition system $\xrightarrow{\delta}$, where δ ranges over $\{D\} \cup \{\tau\}$. Our transition relation is the smallest relation such that for every definition $D = x\langle u\rangle|y\langle v\rangle \triangleright R$,

$$x\langle s\rangle|y\langle t\rangle \xrightarrow{D} R\{^s/_u, {}^t/_v\}$$

and for every transition $P \xrightarrow{\delta} P'$,

$$\begin{array}{rcll} P|Q & \xrightarrow{\delta} & P'|Q & \\ \texttt{def}\, D\ \texttt{in}\ P & \xrightarrow{\delta} & \texttt{def}\, D\ \texttt{in}\ P' & \text{if } fv(D) \cap dv(\delta) = \emptyset \\ \texttt{def}\, \delta\ \texttt{in}\ P & \xrightarrow{\tau} & \texttt{def}\, \delta\ \texttt{in}\ P' & \text{if } \delta \neq \tau \\ Q & \xrightarrow{\delta} & Q' & \text{if } P \equiv P' \text{ and } Q \equiv Q' \end{array}$$

Lemma 1 *The structural congruence $\equiv$ is the smallest congruence that contains all pair of processes P, Q such that $\vdash P \rightleftharpoons^* \vdash Q$. The reduction relation $\xrightarrow{\tau}$ contains exactly the pairs of processes P, Q up to $\equiv$ such that $\vdash P \longrightarrow \vdash Q$*

5.2 Observation

While the observation of concurrent processes is difficult in general, the join-calculus benefits from the experience gained from CCS and from the π-calculus. After an informal discussion of observation criteria, we introduce the equivalence among processes as the largest congruence with a few suitable properties, thus following the approach proposed in [12, 13] for the ν-calculus. This provides an accurate basis for comparisons with other calculi.

5.2.1 What is observable?

The only way for a process to communicate with the outside is to export some names in messages on its free names, and to wait for an answer from an enclosing definition. We distinguish processes accordingly: To each free name x, we associate an *asynchronous, output-only barb* $\Downarrow_x$, which tests the ability of processes to emit anything on x. In the following, $\longrightarrow^*$ stands for any sequence of $\longrightarrow$ and $\rightleftharpoons$.

$$P \Downarrow_x \overset{\text{def}}{=} x \in fv(P)\ \wedge\ \exists \widetilde{v}, \mathcal{R}, \mathcal{M},\ \emptyset \vdash P \longrightarrow^* \mathcal{R} \vdash \mathcal{M}, x\langle\widetilde{v}\rangle$$

5.2.2 Observational congruence

The congruence between processes is the largest equivalence relation $\approx$ that is a refinement of the output barbs $\Downarrow_x$, that is weak-reduction-closed, and that is a congruence for definitions and parallel compositions: $\forall P, Q$, if $P \approx Q$, then

1. $\forall x \in \mathcal{N},\ P \Downarrow_x$ implies $Q \Downarrow_x$
2. $P \longrightarrow^* P'$ implies $\exists Q', Q \longrightarrow^* Q'$ and $P' \approx Q'$
3. $\forall D,\ \texttt{def}\, D \texttt{ in } P \ \approx\ \texttt{def}\, D \texttt{ in } Q$
4. $\forall R,\ R|P \ \approx\ R|Q$

In most proofs, we also need a finer, auxiliary *expansion* relation, and we apply the *bisimulation up to expansion* technique [25]: the expansion between processes is the largest relation $\preceq$ that verifies properties like 1–4, and such that $\forall P, Q$, if $P \preceq Q$, then

$$Q \longrightarrow Q' \text{ implies } P' \preceq Q' \text{ or } \exists P', P \longrightarrow P' \preceq Q'$$

For example we have:

$$\begin{array}{rcll}
fv(P) = \emptyset \Longrightarrow P & \approx & 0 & (1) \\
P \equiv Q \Longrightarrow P & \approx & Q & (2) \\
x\langle u\rangle & \not\approx & y\langle u\rangle & (3) \\
\texttt{def}\, z\langle t\rangle \triangleright t\langle u\rangle \texttt{ in } z\langle x\rangle & \not\approx & \texttt{def}\, z\langle t\rangle \triangleright t\langle u\rangle \texttt{ in } z\langle y\rangle & (4) \\
z\langle x\rangle & \not\approx & z\langle y\rangle & (5) \\
\texttt{def}\, u\langle z\rangle \triangleright v\langle z\rangle \texttt{ in } x\langle u\rangle & \approx & x\langle v\rangle & (6)
\end{array}$$

In (1) no process has any barb, and reductions are simulated by no reduction on the other side; in (3) and (4), the two processes don't have the same barbs; in (5) the two names x and y can be distinguished in contexts as in (4); in (6), two distinct names are sent on x, but their behaviour is the same in every context (although an internal reduction is needed to relay values from u to v).

Despite technical differences in their definitions, $\approx$ is also the congruence over all contexts that is obtained from the weak, barbed bisimulation whose barbs are $\Downarrow_x$, as defined for the π-calculus in [20]. Such barbed congruences can be defined for many process calculi, independently of their syntaxes, and we take advantage of this common framework to obtain precise results.

5.2.3 Full abstraction

In all the following, we assess the relative expressive powers of miscellaneous calculi from the existence of fully-abstract encodings between them.

Definition 1 *Let $\mathcal{P}_1, \mathcal{P}_2$ be two process calculi, with respective equivalences $\approx_1 \subset \mathcal{P}_1 \times \mathcal{P}_1$, $\approx_2 \subset \mathcal{P}_2 \times \mathcal{P}_2$.*

$\mathcal{P}_2$ *is* more expressive *than* $\mathcal{P}_1$ *when there is a fully abstract encoding* $[\![\ ]\!]_{1\to 2}$ *from* $\mathcal{P}_1$ *to* $\mathcal{P}_2$*: for all* P, Q *in* $\mathcal{P}_1$*, we have*

$$P \approx_1 Q \iff [\![P]\!]_{1\to 2} \approx_2 [\![Q]\!]_{1\to 2}$$

$\mathcal{P}_1$ *and* $\mathcal{P}_2$ *have the* same expressive power *when each one is more expressive than the other.*

We use observational congruence as the reference equivalence for each process calculus, meaning that our full abstraction results are up to observation in any context. This seems to be the finest results one could expect between different process calculi.

5.3 Internal encodings

The reflexive CHAM model corresponds to a join-calculus that is convenient as the kernel of a programming language. However, it is possible to reduce it further to simpler primitives. To this end, we successively remove recursive scope, definitions with several clauses $D \wedge D$, join-patterns with more than two messages, and messages with several transmitted values. We replace them by internal encodings, which we prove to be fully abstract. Our purpose here is to isolate the essential features of the join-calculus, and to give some useful examples. Of course, all the derived features would be taken as primitives in a realistic implementation.

Theorem 1 *The core join-calculus has the same expressive power than the full join-calculus up to congruence; in particular, there is a fully-abstract encoding* $[\![\ ]\!]_0$ *from the full calculus to the core calculus: for all processes* Q, R *of the full join-calculus,*

$$Q \approx R \iff [\![Q]\!]_0 \approx [\![R]\!]_0$$

The actual proof consists of successive internal encodings of redundant features; in each part of this section, we explain a stage of the encoding. We omit the proofs. Please note that the following encodings have been chosen for their accuracy with regards to observation, and for their simplicity of exposition; as a result, they may use busy-waiting, and introduce infinite sequences of internal reductions; they are not meant to be used in practice to implement the features that we remove from the join-calculus. Indeed, we plan to implement efficiently the full calculus directly from its reflexive machine specification.

5.3.1 Recursive binding

The non-recursive variant of the join-calculus is defined by restricting the scope of defined variables in `def` D `in` P to P only, so that guarded processes inside of D cannot refer to them. To get rid of the recursive usage of names in definitions, our encoding simply shift the binding variables from definition to reception. Another name ρ is defined to hold formerly recursive names, and a message containing the recursive names is always available on it. In particular the received variable ρ_1 is always bound to ρ each time a molecule is received.

Lemma 2 *Let $\widetilde{x}$ be the vector of variables $fv(Q) \cap dv(J)$, and ρ, ρ_1 be fresh variables. We have:*

$$\begin{array}{lllllll} & \texttt{def} & J & \triangleright & Q & \texttt{in} & P \\ \preceq & \texttt{def} & J|\rho\langle\widetilde{x},\rho_1\rangle & \triangleright & Q|\rho_1\langle\widetilde{x},\rho_1\rangle & \texttt{in} & P|\rho\langle\widetilde{x},\rho\rangle \end{array}$$

5.3.2 Complex definitions

We compile every complex definition with n-way join patterns and/or multiple clauses connected by $\wedge$ into several simpler definitions with only one pattern that joins at most two atoms. For that purpose, we implement an invisible layer between the emitters and the guarded processes of the definition, that makes explicit the automaton that matches messages and patterns.

For clarity, we use the syntactic sugar developped for our language to present our encoding, except use the $x\langle v\rangle$ notation to indicate asyncronous names, rather than capitalization as in Section 4. We encode the definition $D = J_1 \triangleright P_1 \wedge \ldots J_n \triangleright P_n$. Up to α-conversion, we may assume that patterns J_k joins messages of the form $x_i\langle\widetilde{v_i}\rangle$. Then, the translation of $\texttt{def}\, D \texttt{ in } Q$ is:

$$\left\{\begin{array}{l}
\texttt{def } get()|set\langle\tilde{s},\tilde{v}\rangle = \texttt{return } \tilde{s},\tilde{v} \texttt{ to } get \\
\texttt{def } x_i\langle\tilde{u}\rangle = \left\{\begin{array}{l} \texttt{let } \tilde{s},\tilde{v} = get() \\ \texttt{run } set\langle\tilde{s}\cup\{i\},\tilde{v}\{^{\tilde{u}}/_{\tilde{v_i}}\}\rangle \\ \texttt{if } i \in \tilde{s} \texttt{ then run } x_i\langle v_i\rangle \end{array}\right\} \\
\ldots \\
\texttt{def } p_k\langle\rangle = \left\{\begin{array}{l} \texttt{let } \tilde{s},\tilde{v} = get() \\ \texttt{if } \tilde{s_k} \subset \tilde{v} \\ \texttt{then run } P_k|set\langle\tilde{s}-\tilde{s_k},\tilde{v}\rangle \\ \texttt{else run } set\langle\tilde{s},\tilde{v}\rangle \\ \texttt{run } p_k() \end{array}\right\} \\
\texttt{run } p_k\langle\rangle \\
\ldots \\
\texttt{run } set\langle\emptyset,\tilde{0}\rangle \\
\texttt{run } Q
\end{array}\right\}$$

The translation consists of a simple two-way-join definition that matches *internal actions* to an *internal state* $\langle\widetilde{s},\widetilde{v}\rangle$ that "caches" the current pending messages on each of the defined names x_i of D: $\widetilde{s}$ represents the set of names of available messages, and $\widetilde{v}$ contains one of the pending values for these names, if any.

For each J_k, the auxiliary process definition p_k repeatedly checks whether the current state s contains all the defined variables s_k of J_k, and triggers the guarded process P_k when successful.

For each x_i, the new definition inserts values of messages in the current state. Notice that if another message is already present, it is removed and re-sent on x_i; this makes sure that the choice of messages that are present in the cache $\widetilde{v}$ can freely be reconsidered until they are actually sent to a P_k.

Lemma 3 *If DQ is the translation defined above, then*

$$\texttt{def}\, D \texttt{ in } Q \preceq DQ$$

Hence, if $[\![P]\!]$ is defined as the join-term obtained from P by applying the above translation to all compound definitions in P, then $[\![\ ]\!]$ is fully abstract.

5.3.3 Polyadic messages

As in the π-calculus, we communicate tuples of names on auxiliary private names; we first describe the protocol for pairs: the process $O_{x(u,v)}$ sends the pair u, v on x; the context $I^t_{(u,v)}[P]$ extracts a pair u, v from a "pair" name p, then executes P. On the sender's side, an internal state w holds the next value to be returned to rw. Again, we use the syntactic sugar of the language to hide continuations:

$$O_{x(u,v)} \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{def } r()|w\langle z\rangle = \texttt{return } z \texttt{ to } r \\ \texttt{run } w\langle u\rangle \\ \texttt{def } rw() = \left\{ \begin{array}{l} \texttt{let } z = r() \\ \texttt{run } w\langle v\rangle \\ \texttt{return } z \texttt{ to } rw \end{array} \right\} \\ \texttt{run } x\langle rw\rangle \end{array} \right\}$$

$$I^t_{(u,v)}[P] \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \texttt{let } u = t() \\ \texttt{let } v = t() \\ \texttt{run } P \end{array} \right\}$$

The translation of well-sorted polyadic processes is defined inductively on processes; after a first encoding of tuples as nested pairs, we only have to describe the translations for dyadic messages and definitions:

$$[\![x\langle u,v\rangle]\!] \stackrel{\text{def}}{=} O_{x(u,v)}$$

$$[\![\ \texttt{def } x\langle u,v\rangle|y\langle w,z\rangle \triangleright P \texttt{ in } Q]\!]$$
$$\stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \texttt{def} & x'(r)|y'(t) = I^r_{(u,v)}[I^t_{(w,z)}[[\![P]\!]]] \\ \texttt{def} & x(r) = I^r_{(u,v)}[O_{x'(u,v)}] \\ \texttt{def} & y(t) = I^t_{(w,z)}[O_{y'(w,z)}] \\ \texttt{run} & [\![Q]\!] \end{array} \right\}$$

This encoding may appear redundant, as pairs are encoded and decoded twice! However, this ensures that only valid pairs are involved in the actual join-definition. With only one level of encodings, some contexts that do not comply with our protocol may interfere, as is the case in the following example, where $P \approx Q$ but $\mathcal{C}[\![P]\!]' \not\approx \mathcal{C}[\![Q]\!]'$.

$$\begin{array}{rcl} P & \stackrel{\text{def}}{=} & \texttt{def } x\langle u,v\rangle|y\langle w,z\rangle \triangleright b\langle b\rangle \texttt{ in } z\langle x\rangle|x\langle a,a\rangle|y\langle a,a\rangle \\ Q & \stackrel{\text{def}}{=} & \texttt{def } x\langle u,v\rangle|y\langle w,z\rangle \triangleright b\langle b\rangle \texttt{ in } z\langle x\rangle|b\langle b\rangle \\ \mathcal{C}[\ ] & \stackrel{\text{def}}{=} & \texttt{def } z\langle x\rangle \triangleright (\texttt{def } t(\kappa) \triangleright 0 \texttt{ in } x\langle t\rangle) \texttt{ in } [\ ] \end{array}$$

Lemma 4 *The encoding $[\![\ ]\!]$ is fully abstract. The core monadic and the core polyadic variants of the join-calculus have the same expressive power.*

6 A comparison with the π-calculus

Despite their syntactic differences, the join-calculus can be considered as an offspring of the π-calculus, in the asynchronous branch of the family. The latter was introduced independently in [7] as the (mini) asynchronous π-calculus, and in [10] as the ν-calculus. Both authors suppress the guards on emission, and compare the result to the original π-calculus. Going further in that direction, the join-calculus is an asynchronous π-calculus with the strong restrictions:

- the three binders (scope restriction, reception, replicated reception) are syntactically merged in a single construct: the definition;
- communication occurs only on defined names;
- for every defined name, there is exactly one, replicated reception.

There are several reasons to be interested in a formal comparison between the two calculi: the π-calculus has been thoroughly studied; it is a reference calculus, and many results relate other formalisms or implementations to it [3, 17, 26, 28]. Therefore, it is appealing to "translate" such results automatically to the join-calculus. On the other hand, some issues are best addressed in the join-calculus, as for instance locality, implementation purposes, and explicit distribution. Besides, this also provides a deep insight into what is common and what is different in the join-calculus and in the π-calculus.

Both encodings that are used to get our most precise results are complex, but their underlying ideas are simple. In particular, much simpler encodings can be obtained in less general settings; for instance, programs written in PICT and programs in the language presented in section 4 would use very similar implementation techniques.

Using the results of the previous section, we consider the recursive, polyadic join-calculus with at most two-way-join definitions as the target calculus to encode the π-calculus, and its monadic variant for the reverse encoding.

We first recall the definition of the *asynchronous π-calculus*; then we encode the π-calculus in the join-calculus. The first, naive encoding replaces each channel of the π-calculus by a two-way definition; however, some more work is needed to achieve full abstraction. We present our approach based on "firewalls" in detail, but we defer the presentation of the proof to annex A. In the same manner, we then encode the join-calculus in the π-calculus using the straightforward translation of definitions into scope-restriction and replicated reception. Using the same approach, we also need to refine the encoding. The sketch of the proof can be found in annex B.

In all our encodings, we will assume that every name that is introduced in the translation rules is a fresh name that does not appear elsewhere in the terms. This may involve some α-conversion.

6.1 The asynchronous π-calculus

To study this relationship, we precisely compare the join-calculus to the asynchronous π-calculus. We use the syntax of Milner in [18]. Without loss of generality, we allow only monadic messages, and replicated input instead of more general recursion.

$$P \stackrel{\text{def}}{=} P|Q \mid \nu u.P \mid \overline{x}u \mid x(u).P \mid !x(u).P$$

Following the observational approach of [10, 11, 13], the congruence $\approx_\pi$ is defined for the π-calculus as the asynchronous barbed congruence whose barbs are the emissions on free channels.

Theorem 2 *The join-calculus and the summation-free asynchronous π-calculus have the same expressive power, up to their weak output-only barbed congruences.*

6.2 Asynchrony, Relays and Equators

Our encodings essentially rely on the properties of the asynchronous reduction-based π-calculus as discussed in [11, 12], and on similar properties of the join-calculus.

In both calculi, it is not possible to observe the reception of a message; for instance we have $x(u).\overline{x}u \approx_\pi 0$, and it is not possible either to distinguish between two different names that have the same external behaviour. We illustrate the latter with a definition of *equators* between names:

$$M^\pi_{x,y} \stackrel{\text{def}}{=} !x(u).\overline{y}u|!y(v).\overline{x}v$$

This process repeatedly receives values from x and forwards them to y and vice-versa, so that no matter which name x or y is used to send a value, it can always be made available for reception on the other name in one internal reduction:

Lemma 5 *For all π-processes P, Q*

$$P\{^x/_y\} \approx_\pi Q\{^x/_y\} \quad \textit{implies} \quad M^\pi_{x,y}|P \approx_\pi M^\pi_{x,y}|Q$$

6.3 Encoding the π-calculus

6.3.1 Naive structural definition

To each channel x of the π-calculus, we associate two names x_o for output, x_i for input, and an enclosing definition that matches output and input . The emitter simply sends values on x_o; the receiver defines a name for its continuation, and sends it as a reception offer on x_i:

$$\begin{aligned}
[\![P|Q]\!]_\pi &\stackrel{\text{def}}{=} [\![P]\!]_\pi|[\![Q]\!]_\pi \\
[\![\nu x.P]\!]_\pi &\stackrel{\text{def}}{=} \texttt{def}\; x_o\langle v_o, v_i\rangle|x_i\langle\kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle \;\texttt{in}\; [\![P]\!]_\pi \\
[\![\overline{x}v]\!]_\pi &\stackrel{\text{def}}{=} x_o\langle v_o, v_i\rangle \\
[\![x(v).P]\!]_\pi &\stackrel{\text{def}}{=} \texttt{def}\; \kappa\langle v_o, v_i\rangle \triangleright [\![P]\!]_\pi \;\texttt{in}\; x_i\langle\kappa\rangle \\
[\![!x(v).P]\!]_\pi &\stackrel{\text{def}}{=} \texttt{def}\; \kappa\langle v_o, v_i\rangle \triangleright x_i\langle\kappa\rangle|[\![P]\!]_\pi \;\texttt{in}\; x_i\langle\kappa\rangle
\end{aligned}$$

For example, we translate the following π-process and its reduction

$$\nu x.(\overline{x}a|\overline{x}b|x(u).\overline{y}u) \quad \rightarrow \quad \nu x.(\overline{x}a|\overline{y}b)$$

to the join-process and the series of reductions

$$\begin{array}{rll} & \texttt{def} & x_o\langle v_o, v_i\rangle | x_i\langle\kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle \\ & \texttt{in} & x_o\langle a_o, a_i\rangle | x_o\langle a_o, a_i\rangle \\ & & \mid \texttt{def } \kappa\langle u_o, u_i\rangle \triangleright y_o\langle u_o, u_i\rangle \texttt{ in } x_i\langle\kappa\rangle \\ & \texttt{def} & \kappa\langle u_o, u_i\rangle \triangleright y_o\langle u_o, u_i\rangle \\ \rightarrow\rightarrow & \texttt{in} & \texttt{def}\, x_o\langle v_o, v_i\rangle | x_i\langle\kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle \\ & \texttt{in} & x_o\langle a_o, a_i\rangle | y_o\langle b_o, b_i\rangle \end{array}$$

In the same manner, any reduction on a bound name in the π-calculus can be simulated by a join-reduction followed by a deterministic reduction in the join-calculus, and conversely any reduction in a join-calculus translation belongs to one of these two cases, and can be simulated in at most one reduction in the π-calculus.

6.3.2 Full abstraction of the encoding

Unfortunately, the previous encoding does not reflect the behaviour of processes of the π-calculus when placed in an arbitrary join-calculus context: the protocol relies on the presence of specific definitions for every free name, while the context may define them in some other way.

For example, the translation $[\![\overline{x}a|\overline{x}b|x(u).\overline{y}u]\!]_\pi$ cannot reduce anymore, because there is no englobing νx. Worse, $[\![x(u).\overline{x}u]\!]_\pi$ exhibits a barb on x_i that reveals the presence of an input for x, and allows a context to distinguish this process from 0. And because of mobility, it would not be enough to supply a correct definition of x_o, x_i for every translated free name, since a context would still be able to "forge" a message $x_o\langle z, t\rangle$ from some of its own names z, t with arbitrary definitions.

To protect the translation from hostile contexts, the names resulting from the free channels of the π-term must set-up a "firewall" that enforces the protocol. We refine our first idea: each channel x is now represented as several pairs x_o, x_i from the naive encoding that cannot be distinguished from the outside. Two pairs are merged by repeatedly communicating their pending messages to one another. New pairs are defined at run-time according to the following secure protocol:

- Whenever a pair of names is received from the outside, the firewall defines a new, correct *proxy pair*, merges it to the external pair, and transmits the new pair instead.

- Whenever a pair of names is sent to the outside, a new firewall is inserted to set up proxies for future messages on this pair.

As a result, the translation and the context never exchange names from a syntactic point of view. We use the following contexts to build the firewall on top of the naive translation:

$$\begin{array}{rcl}
\mathcal{P}_x[\;] & \stackrel{\text{def}}{=} & \texttt{def}\, x_l\langle v_o, v_i\rangle | x_i\langle\kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle \\
& & \texttt{in def}\ x_o\langle v_o, v_i\rangle \triangleright p\langle v_o, v_i, x_l\rangle\ \texttt{in}\ [\;] \\
\mathcal{E}_x[\;] & \stackrel{\text{def}}{=} & \mathcal{P}_x[x_e\langle x_o, x_i\rangle | [\;]] \\
\mathcal{M}[\;] & \stackrel{\text{def}}{=} & \texttt{def}\, p(x_o, x_i, \kappa) \triangleright \mathcal{P}_y[\kappa\langle y_o, y_i\rangle | [\![M^\pi_{x,y}]\!]_\pi]\ \texttt{in}\ [\;]
\end{array}$$

For every free name x, $\mathcal{P}_x$ encodes the creation of a new proxy for its output. $\mathcal{E}_x$ does the same, and also exports the proxy on a conventional free name x_e. Finally, $\mathcal{M}$ recursively defines the proxy creator p for the whole translation.

Notation: Whenever a context defined with a name index appears without this index, it stands for the application of the context for at least all the free variables in the π-term, and for a definition for p. For instance, $\mathcal{E}[[\![\overline{x}y]\!]_\pi] \stackrel{\text{def}}{=} \mathcal{M}[\mathcal{E}_x[\mathcal{E}_y[[\![\overline{x}y]\!]_\pi]]]$.

Theorem 3 *For all processes Q, R in the π-calculus,*

$$Q \approx_\pi R \Longleftrightarrow \mathcal{E}[[\![Q]\!]_\pi] \approx \mathcal{E}[[\![R]\!]_\pi]$$

Note that $\mathcal{E}$ catches all the free variables of $P|Q$. In the proof, we also give an auxiliary encoding that is strictly compositional.

6.4 Encoding the join-calculus

The reverse translation is simpler, because the join-calculus is somehow the π-calculus with restrictions on communication patterns. However, a careful encoding is needed to prevent contexts of the π-calculus from reading messages from the names they receive from the translation.

6.4.1 Structural definition

$$\begin{array}{rcl}
[\![Q|R]\!]_j & \stackrel{\text{def}}{=} & [\![Q]\!]_j | [\![R]\!]_j \\
[\![x\langle v\rangle]\!]_j & \stackrel{\text{def}}{=} & \overline{x}v \\
[\![\texttt{def}\ x\langle u\rangle | y\langle v\rangle \triangleright Q\ \texttt{in}\ R]\!]_j & \stackrel{\text{def}}{=} & \nu xy.(!x(u).y(v).[\![Q]\!]_j | [\![R]\!]_j)
\end{array}$$

Reductions in π-calculus translations correspond exactly to the reception of messages in join-patterns. In the translation, we loose the symmetry between x and y and the atomicity of their join-reduction, but it does not matter as scope restriction and $[\![\;]\!]_j$ guarantee that these details cannot be observed.

Again, the translation reveals too much about the source process, as a context of the π-calculus could start reading values on names bound in the translation of definitions. Indeed, if we were translating the join-calculus into an asynchronous π-calculus extended with a type system with polarities [22], we could specify write-only types for every channel that is communicated inside of the translation, and the (typed) previous encoding would already be fully-abstract.

6.4.2 Full abstraction

To obtain our second full abstraction result, we also need to build a firewall. The interface recursively sets up one-way relays for every name that crosses the boundary. It is built from the following terms, with the same convention on indices.

$$\begin{array}{rcl} R_{xy} & \stackrel{\text{def}}{=} & !x(v).\nu v_e.(\overline{r} v_e v | \overline{y} v_e) \\ \mathcal{R}[\,] & \stackrel{\text{def}}{=} & \nu r.!r(x, x_e).R_{xx_e} | [\,]) \\ \mathcal{E}^\pi_x[\,] & \stackrel{\text{def}}{=} & \nu x.(R_{xx_e} | [\,]) \end{array}$$

$\mathcal{R}$ is a global definition for the translation, which sets up one-way relays R_{xx_e} or R_{xx_j} from the first to the second of its argument. When a relay forwards a message, it also sets up a relay going in the reverse direction for the transmitted value. There is no syntactic scope extrusion of the translations of definitions, and their synonym can always receive messages. We use the same notation convention as before: $\mathcal{E}^\pi[\![P]\!]_j$ stands for the application of $\mathcal{R}$ followed by applications of $\mathcal{E}^\pi_x$ for at least all free variables of x of P.

Theorem 4 *For all processes Q, R in the join-calculus,*

$$Q \approx R \iff \mathcal{E}^\pi[[\![Q]\!]_j] \ \approx_\pi \ \mathcal{E}^\pi[[\![R]\!]_j]$$

7 Future work

Many interesting issues on the join-calculus are outside the scope of this paper. They include actual implementation techniques, type systems and in particular linear types for the names that represent continuations, as in [22, 14, 27], extensions of the calculus with records for a better support of object-oriented programming. Observation and equivalences also deserve a more detailed treatment, as well as a comparison with their counterparts in the asynchronous π-calculus.

To conclude, we briefly mention our current usage of the reflexive CHAM and of the join-calculus in a programming language design where resources and environments are explicitly distributed, while the details of the network and its connectivity remain hidden. In a practical distributed setting where some sites may fail, the atomicity of each interaction must be specified accurately [3], in a way that can be implemented locally. The join-calculus relieves us of many difficult issues: As synchronization can only happen on definitions, it is sufficient to require each definition to be annotated with some location, that is shared by all its names and guarded processes. Likewise, the actual allocation of resources for a definition such as waiting queues, automaton, and closures, happens locally as the definition is activated using the chemical rule (str-def). In that setting, messages are forwarded to their definition asynchronously, then handled locally. We currently study extensions of the reflexive CHAM and of the language that provide explicit control of the localization of definitions on several sites, and possibly their imperative migration from one site to another. This would make intensive interaction more local, and would protect it from local failure. A distributed prototype is under way, to assess the feasibility and the interest of a distributed implementation of process calculi.

Acknowledgments

This works started from a seminar on "distributing the π-calculus" held in 1995 at INRIA, with the participation of Damien Doligez, Florent Guillaume, Jean-Jacques Lévy, Luc Maranget, and Didier Remy. We also thank Roberto Amadio, Gerard Boudol, Xavier Leroy, Benjamin Pierce, Davide Sangiorgi, and David Turner for fruitful discussions and comments. We used Paul Taylor's macros for typesetting diagrams.

References

[1] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. Technical report, UIUC, 1993.

[2] G. A. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[3] R. Amadio and S. Prasad. Localities and failures. *Foundations of Software Technology and Theoretical Computer Science*, 14, 1994.

[4] J.-M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, pages 212–229, Nov. 1991. Published as Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, number 11.

[5] J.-P. Banâtre, M. Banâtre, and F. Ployette. Distributed system structuring using multi-functions. Rapport de Recherche 694, INRIA Rennes, June 1987.

[6] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[7] G. Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.

[8] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, Jan. 1995. A preliminary version appeared in Proceedings of the 22nd ACM Symposium on Principles of Programming Languages.

[9] A. Giacalone, P. Mishra, and S. Prasad. FACILE: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989. Also in TAPSOFT '89, ed. J. Diaz and F. Orejas, pp. 184-209, Springer-Verlag, Lecture Notes in Computer Science 352 (1989).

[10] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 133–147, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[11] K. Honda and M. Tokoro. On asynchronous communication semantics. In P. W. M. Tokoro and O. Nierstrasz, editors, *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 21–51. Springer-Verlag, 1992.

[12] K. Honda and N. Yoshida. On reduction-based process semantics. *Foundations of Software Technology and Theoretical Computer Science*, 13, 1993.

[13] K. Honda and N. Yoshida. Combinatory representation of mobile processes. In *Proceedings POPL '94*, 1994.

[14] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *Proceedings POPL '96*, 1996.

[15] L. Maranget. Two techniques for compiling lazy pattern matching. Research report 2385, INRIA, 1994.

[16] R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.

[17] R. Milner. Functions as processes. In *Automata, Languages and Programming 17th Int. Coll.*, volume 443 of *LNCS*, pages 167–180. Springer Verlag, July 1990.

[18] R. Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification.* Springer Verlag, 1993.

[19] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, pages 1–40 & 41–77, Sept. 1992.

[20] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proceedings ICALP '92*, LNCS 623, pages 685–695, Vienna, 1992. Springer-Verlag.

[21] B. C. Pierce, D. Rémy, and D. N. Turner. A typed higher-order programming language based on the pi-calculus. In *Workshop on Type Theory and its Application to Computer Systems, Kyoto University*, July 1993.

[22] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 1995. To appear. A summary was presented at LICS '93.

[23] B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In T. Ito and A. Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan (Nov. 1994)*, number 907 in Lecture Notes in Computer Science, pages 187–215. Springer-Verlag, Apr. 1995.

[24] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical report in preparation, 1995.

[25] D. Sangiorgi and R. Milner. The problem of "weak bisimulation up to". In W. R. Cleaveland, editor, *Proceedings of CONCUR'92*, LNCS 630, pages 32–46. Springer-Verlag, 1992.

[26] G. Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 50–72, München, Germany, 7–9 Sept. 1994. Springer-Verlag.

[27] D. N. Turner. *The π-calculus: Types, polymorphism and implementation.* PhD thesis, LFCS, University of Edinburgh, 1995. In preparation.

[28] D. Walker. Objects in the pi-calculus. *Information and Computation*, 116(2):253–271, 1995.

A Sketch of the proofs of section 6

Notation: In all the diagrams that follow, we use the usual conventions : for all relations on plain lines, there exists relations on dotted lines, and stars denotes the reflexive-transitive closure of a relation.

A.1 $\mathcal{E}[\![\ ]\!]_\pi$ is fully-abstract

A.1.1 Combining translations and contexts

We first prove the direct implication by studying how a translation can interact with an arbitrary join-calculus context. This is performed on an auxiliary translation that is very similar to $[\![\ ]\!]_\pi$:

Definition 2 *The translation $[\![\ ]\!]$ maps π-processes to join-processes using the same structural definition that for $[\![\ ]\!]_\pi$, except for scope restriction (so that output always leads to the creation of a new proxy pair), and for unguarded outputs (where the definition of x_o is unfolded):*

$$\begin{aligned} [\![\nu x.P]\!] &\stackrel{\text{def}}{=} \mathcal{P}_x[[\![P]\!]] \\ [\![\overline{x}v]\!] &\stackrel{\text{def}}{=} \mathcal{P}_z\left[x_l\langle z_o, z_i\rangle | [\![M^\pi_{z,v}]\!]_\pi\right] \quad \textit{(when unguarded)} \end{aligned}$$

Definition 3 *Hybrid terms are terms of the join-calculus that are structurally equivalent to some $\mathcal{P}\left[E|[\![Q]\!]\right]$, where E is a join-calculus process, Q is a π-calculus process, and $\mathcal{P}$ is an header of definitions such that for all free names x in Q, $\mathcal{P}_x$ appears in $\mathcal{P}$, and such that every message on x_l channels matches some $x_l\langle v_o, v_i\rangle$ where $\mathcal{P}_v$ appears in $\mathcal{P}$.*

In particular, the processes $\mathcal{E}\left[[\![Q]\!]\right]$ that appear in the theorem are hybrid terms. We now study reductions inside of hybrid terms. These reductions can be:

1. reductions that use the join-definition of some x_l, x_i, which correspond to reductions in the π-calculus;

2. reductions that manipulate pairs of synonyms, or trigger continuations κ, which are induced by the encoding;

3. reductions inside of E, which are independent of the translation.

We are mostly interested in the first family of reductions. To get rid of the details of the encoding, we first define two auxiliary expansions to relate hybrid terms that differ only because some deterministic reduction hasn't been performed yet, or because some extra synonyms have been introduced for pairs x_o, x_i. Then we use the weak bisimulation up-to expansion technique [25].

Lemma 6 *Let $\rightarrow_{det}$ be the relation on join-processes that contains all pairs of deterministic reductions; $\succeq_{\mathrm{det}} \overset{\mathrm{def}}{=} (\rightarrow_{det})^*$; $\preceq_{\mathrm{det}} \overset{\mathrm{def}}{=} \succeq_{\mathrm{det}}^{-1}$. Then $\preceq_{\mathrm{det}}$ is a barbed expansion.*

Lemma 7 *Let $\rightsquigarrow_{\mathrm{merge}}$ relates hybrid terms with one additional pair of synonyms on the left-hand-side:*

$$\begin{array}{rcl} \mathcal{P}\mathcal{P}_{x,y}\left[E|[\![Q]\!]|[\![M^{\pi}_{x,y}]\!]_{\pi}\right] & \rightsquigarrow_{\mathrm{merge}} & \mathcal{P}\mathcal{P}_{x}\left[E|[\![Q]\!]\right]\{^{x}/_{y}\} \\ \preceq_{\mathrm{merge}} & \overset{\mathrm{def}}{=} & (\rightsquigarrow^{*}_{\mathrm{merge}})^{-1}. \end{array}$$

Then $\preceq_{\mathrm{merge}}$ is a barbed expansion.

Lemma 8 $\mathcal{E}\left[[\![Q]\!]_{\pi}\right] \approx \mathcal{E}\left[[\![Q]\!]\right]$

Lemma 9 *The reductions in the π-calculus can be mimicked on their translations:*

$$\textit{if } Q \rightarrow^{*} Q', \quad \textit{then} \quad \mathcal{P}\left[[\![Q]\!]\right] \rightarrow^{*} (\succeq_{\mathrm{det}}\succeq_{\mathrm{merge}})^{*}\mathcal{P}\left[[\![Q']\!]\right]$$

Lemma 10 *For all pair of hybrid terms $\mathcal{P}\left[E|[\![Q]\!]\right]$, $\mathcal{P}\left[E|[\![R]\!]\right]$,*

$$\textit{if } Q \approx_{\pi} R, \quad \textit{then} \quad \mathcal{P}\left[E|[\![Q]\!]\right] \approx \mathcal{P}\left[E|[\![R]\!]\right]$$

Proof: Let the relation $\mathcal{B}$ contains all the pairs of hybrid terms that are obtained from congruent π-processes:

$$\forall Q \approx_{\pi} R, \quad \mathcal{P}\left[E|[\![Q]\!]\right] \quad \mathcal{B} \quad \mathcal{P}\left[E|[\![R]\!]\right]$$

We establish that $\mathcal{B} \subset \approx$. To this end, we distinguish among the reactions that may happen on the left-hand-side hybrid term, and in each case simulate it on the right-hand-side. We consider four cases:

Reduction outside of the translation: they are the same on both sides.

Reduction inside of the translation: a communication occurs between a reception offer and an output; it corresponds to a communication in the original π-term, except that as the value is received, a new proxy is created, and that several deterministic reductions may be necessary to reach an hybrid term. On the right side, we use $Q \approx_\pi R$ to obtain a sequence of reductions $R \rightarrow^* R'$, and we mimic them in the translation.

$$\begin{tikzcd}
\mathcal{P}[E|[\![Q]\!]] \arrow[rrr, dash, "\mathcal{B}"] \arrow[d] & & & \mathcal{P}[E|[\![R]\!]] \arrow[d, dashed, "*"] \\
\cdot \arrow[r, dashed, dash, "\succeq"] & \mathcal{P}[E|[\![Q']\!]] \arrow[r, dashed, dash, "\mathcal{B}"] & \mathcal{P}[E|[\![R']\!]] \arrow[r, dashed, dash, "\preceq"] & \cdot
\end{tikzcd}$$

Intrusion of a pair of channels from the environment into the translation: The receiving π-term Q is of the form $\nu\widetilde{u}.(Q'|x(z).Q'')$, while there is a correct pending value $x_l\langle z_o, z_i\rangle$ in the join-calculus and a definition to match them. We first use an auxiliary commutative diagram to push the emission under the translation. Using our lemma, we can then build reductions for the right term, and with a few deterministic reductions on the bottom left, we get a new pair in $\mathcal{B}$. On both sides, we use our expansions on hybrid terms to switch $x_l\langle z_o, z_i\rangle$ and $[\![\overline{x}z]\!]$.

$$\begin{tikzcd}
\overline{x}z|\nu\widetilde{u}.(Q'|x(z).Q'') \arrow[r, dash, "\approx_\pi"] \arrow[d] & \overline{x}z|R \arrow[d, dashed, "*"'] \\
\nu\widetilde{u}.(Q'|Q'') \arrow[r, dashed, dash, "\approx_\pi"] & R'
\end{tikzcd}$$

$$\begin{tikzcd}
\mathcal{P}[E|x_l\langle z_o, z_i\rangle|[\![Q]\!]] \arrow[rr, dash, "\mathcal{B}"] \arrow[d] & & \mathcal{P}[E|x_l\langle z_o, z_i\rangle|[\![R]\!]] \arrow[d, dashed, "*"'] \\
\cdot \arrow[r, dashed, dash, "\succeq"] & \mathcal{P}[E|[\![\nu\widetilde{u}.(Q'|Q'')]\!]] \arrow[r, dashed, dash, "\mathcal{B}\ \preceq\ \succeq"] & \cdot
\end{tikzcd}$$

Extrusion of a pair of channels from the translation to the environment: the emitting π-term is of the form $Q = \nu u.(Q'|\overline{x}y)$, where possibly $u = y$. If l, u are fresh names, we get a similar emission for R by applying the congruence $Q \approx_\pi R$ to the context $\mathcal{O}[\,] \stackrel{\text{def}}{=} \overline{l}u|x(y).l(u).M^\pi_{yz}|[\,]$:

$$\begin{tikzcd}
\mathcal{O}[\nu u.(Q'|\overline{x}y)] \arrow[r, dash, "\approx_\pi"] \arrow[d, "2"] & \mathcal{O}[R] \arrow[d, dashed, "*"] \\
\nu u.(Q'|M^\pi_{yz}) \arrow[r, dashed, dash, "\approx_\pi"] & \nu u'.(R'|M^\pi_{y'z})
\end{tikzcd}$$

In the diagram, the reductions on the right can be reordered as internal reductions in R, followed by the two reductions on x, l with $\mathcal{O}$. From the first ones, we build the corresponding reductions from the translation on the right to a term that is an expansion of $\nu u'.(R'|M^{\pi}_{y'z})$.

$$\begin{tikzcd}
\mathcal{P}\left[E\left[x_i\langle\kappa\rangle\right]|\llbracket\nu u.(Q'|\overline{x}y)\rrbracket\right] \arrow[r, dash, "\mathcal{B}"] \arrow[d] & \mathcal{P}\left[E\left[x_i\langle\kappa\rangle\right]|\llbracket R\rrbracket\right] \arrow[d, dashed, "*"'] \\
\mathcal{P}\left[E\left[\kappa\langle z_o, z_i\rangle\right]|\llbracket\nu u.(Q'|M^{\pi}_{y,z})\rrbracket\right] \arrow[r, dash, dashed, "\mathcal{B}\ \preceq"] & {}
\end{tikzcd}$$

From the previous diagrams, and as $\succeq\mathcal{B}\approx$ is a congruence and respects the barbs, we obtain by definition of $\approx$ that $(\succeq\mathcal{B}\approx)\subset\approx$, and in particular $\mathcal{B}\subset\approx$ $\square$

A.1.2 Correctness

We need yet another translation that is fully compositional: each term is wrapped in a protective context, at each step of the structural definition; conversely, the names of each subterm must be made synonym for the names in the current term:

Definition 4 *The translation* $\llbracket\ \rrbracket$ *maps* π*-processes to join-calculus processes:*

$$\begin{aligned}
\llbracket P|Q\rrbracket &\stackrel{\text{def}}{=} \mathcal{E}\left[\mathcal{I}[\llbracket P\rrbracket]|\mathcal{I}[\llbracket Q\rrbracket]\right]\\
\llbracket \nu x.P\rrbracket &\stackrel{\text{def}}{=} \mathcal{N}_x[\llbracket P\rrbracket]\\
\llbracket \overline{x}v\rrbracket &\stackrel{\text{def}}{=} \mathcal{E}\left[x_o\langle v_o, v_i\rangle\right]\\
\llbracket x(v).P\rrbracket &\stackrel{\text{def}}{=} \mathcal{E}\left[\mathtt{def}\ \kappa\langle v_o, v_i\rangle = \mathcal{I}[\llbracket P\rrbracket]\ \mathtt{in}\ x_i\langle\kappa\rangle\right]\\
\llbracket !x(v).P\rrbracket &\stackrel{\text{def}}{=} \mathcal{E}\left[\mathtt{def}\ \kappa\langle v_o, v_i\rangle = x_i\langle\kappa\rangle|\mathcal{I}[\llbracket P\rrbracket]\ \mathtt{in}\ x_i\langle\kappa\rangle\right]
\end{aligned}$$

with the following definitions (and with the convention on indices)

$$\begin{aligned}
\mathcal{I}_x[\ ] &\stackrel{\text{def}}{=} \mathtt{def}\ x_e\langle y_o, y_i\rangle = \llbracket M^{\pi}_{x,y}\rrbracket_{\pi}\ \mathtt{in}\ [\ ]\\
\mathcal{N}_x[\ ] &\stackrel{\text{def}}{=} \mathtt{def}\ x_e\langle y_o, y_i\rangle = 0\ \mathtt{in}\ [\ ]
\end{aligned}$$

$\mathcal{I}$ catches exported synonyms for channels in the scope of the context. $\mathcal{N}$ prevents extension, thus providing locality.

Lemma 11 $\forall Q,\ \llbracket Q\rrbracket\approx\mathcal{E}\llbracket Q\rrbracket$

Proof: In each case of the structural induction, we use variants of the relation $\mathcal{E}_x\mathcal{I}_x\mathcal{E}_x[P]\ \succeq_{\text{det}}\succeq_{\text{merge}}\ \mathcal{E}_x[P]$. We present two significant cases:

$$\begin{aligned}
\llbracket P|Q\rrbracket &\stackrel{\text{def}}{=} \mathcal{E}\left[\mathcal{I}[\llbracket P\rrbracket]|\mathcal{I}[\llbracket Q\rrbracket]\right]\\
&\approx \mathcal{E}\left[\mathcal{I}\mathcal{E}[\llbracket P\rrbracket]|\mathcal{I}\mathcal{E}[\llbracket Q\rrbracket]\right]\\
&\approx \mathcal{E}\llbracket P|Q\rrbracket
\end{aligned}$$

$$\begin{aligned}
\llbracket \nu x.P\rrbracket &\stackrel{\text{def}}{=} \mathcal{N}_x[\llbracket P\rrbracket]\\
&\approx \mathcal{E}\mathcal{N}_x\mathcal{E}_x\left[\llbracket P\rrbracket\right]\\
&\approx \mathcal{E}\mathcal{P}_x\left[\llbracket P\rrbracket\right]\\
&\approx \mathcal{E}\left[\llbracket \nu x.P\rrbracket\right]
\end{aligned}$$

□

Lemma 12 *if* $\mathcal{E}[\![Q]\!] \approx \mathcal{E}[\![R]\!]$ *then* $Q \approx_\pi R$

Proof: $\{(Q,R), \mathcal{E}[\![Q]\!] \approx \mathcal{E}[\![R]\!]\}$ is a barbed congruence in the π-calculus: The congruence follows from the previous result and the congruence property of $\approx$.

$$\mathcal{E}[\![\mathcal{C}\,[Q]\,]\!] \approx [\![\mathcal{C}\,[Q]\,]\!] = ([\![\mathcal{C}]\!])\,[[\![Q]\!]] \approx ([\![\mathcal{C}]\!])\,[\mathcal{E}[\![Q]\!]]$$

The asynchronous barbs are the same: They can be individually tested in simple contexts. The bisimulation is obtained from previous lemmas. □

A.2 $\mathcal{E}^\pi[\![\ ]\!]_j$ is fully-abstract

In this part, we use conventions for names: In the π-calculus, z_j is a free variable of the translation, that corresponds to the external name z. In the join-calculus, we introduce for each name x another name, $\widehat{x}$, that may appear at most once in a process, and only as the port name of an unguarded message $\widehat{x}(y)$; we note $\widehat{P}$ a process that may contain these messages. Such messages will keep track of internal names y that have been exported to the context; to this end, we adapt $[\![\ ]\!]_j$ to translate them into incoming relays.

$$[\![\widehat{x}(y)]\!]_j \quad \stackrel{\text{def}}{=} \quad R_{xy_j}$$

Definition 5 *A hybrid term is a term of the π-calculus that is structurally equivalent to*

$$\mathcal{C}\left[\mathcal{E}^\pi_{\tilde{z}}[\![\widehat{P}]\!]_j\right]$$

where $\mathcal{C}$ is any context of the π-calculusof the form $\nu\tilde{w}(R|[\ ])$, and where $\widehat{P}$ is a join-process with possibly some unguarded messages $\widetilde{\widehat{x}(x)}$, such that its free variables are in $\{\tilde{z}_j, \widetilde{\widehat{x}}\}$.

Lemma 13

$$\mathcal{C}\left[\nu x_j y_j (R_{x_j z}|R_{y_j z}|[\![\widehat{P}]\!]_j)\right] \quad \succeq \quad \mathcal{C}\left[\nu z_j (R_{x_j z}|[\![\widehat{P}]\!]_j\{^{x_j}/_{y_j}\})\right]$$

Lemma 14 *Let $\preceq_{J-\pi}$ be the largest expansion between the π-calculus and the join-calculus that respects barbs. Then for all join-process P we have $P \preceq_{J-\pi} [\![P]\!]_j$.*

Lemma 15 *The relation that contains all pairs of hybrid terms whose extended join-processes are congruent is a barbed congruence in the π-calculus.*

Proof: The congruence property is obvious; the bisimulation requires a case analysis, which will also establishes that barbs are preserved. We study in more details the interactions between the translation and its π-context, and the set-up of new relays. Four kinds of reductions may occur:

External communication: apply the same one on the other side.

Intrusion of an external message is received on an incoming relay: the messages is withdrawn from the π-calculus context, and committed to an internal usage; except from the first step which prevents the input of the message in the context, the following steps are deterministic, and lead to a new hybrid term.

$$\mathcal{C}\left[\overline{x}u|\mathcal{E}^{\pi}_{\tilde{z}}[\![\widehat{P}[\widehat{x}(x)]]\!]_j\right] \quad \to\to^*_{det} \quad \mathcal{C}\left[\mathcal{E}^{\pi}_{u,\tilde{z}}[\![\widehat{P}[x\langle u\rangle|\widehat{x}(x)]]\!]_j\right]$$

To obtain a bisimilar hybrid term on the other side, we use the join-calculus context:

$$\mathcal{O}_i[\,] \overset{\text{def}}{=} \texttt{def}\,\widehat{y}(x) \triangleright \widehat{x}(x) \ \texttt{in def}\,\widehat{x}(x) \triangleright x\langle u\rangle|\widehat{y}(x) \ \texttt{in}\ [\,]$$

Both join-processes have a barb on $\widehat{x}$ that is necessarily a single, unguarded message. Consuming $\widehat{x}$ on both sides leads to a pair of congruent processes. We then discard the useless definition of $\widehat{x}(x)$, and wrap both processes as a new pair of related hybrid terms.

Internal reduction on the translation of a definition: using $\approx$ in the join-calculus, we obtain a sequence of reductions on the left side, and mimic it in the translation. Some deterministic reductions may be needed on both sides to reach a hybrid term.

Extrusion by internal reduction on an export relay: in a few deterministic reductions, messages on translations of free variables are exported to their public π-calculus name:

$$\mathcal{C}\left[\mathcal{E}^{\pi}_{\tilde{z}}[\![\widehat{P}[z\langle x\rangle]]\!]_j\right] \quad \to^*_{det} \quad \mathcal{C}\left[\nu x(\overline{z}x|\mathcal{E}^{\pi}_{\tilde{z}}[\![\widehat{P}[\widehat{x}(x)]]\!]_j\right]$$

We use the following context $\mathcal{O}_e$ to obtain an adequate sequence of reductions on the right side: the first emission on z_j is handled in a special way, while the next ones are silently transmitted.

$$\mathcal{O}_e[\,] \overset{\text{def}}{=} \begin{array}{l} \texttt{def}\, e() \triangleright 0 \ \texttt{in def}\, a\langle u\rangle|l\langle\rangle \triangleright u\langle\rangle \\ \texttt{in def}\ z'\langle x\rangle \triangleright z\langle x\rangle \\ \texttt{in def}\ z\langle x\rangle|c\langle\kappa\rangle \triangleright \kappa\langle x\rangle|c\langle z'\rangle \\ \texttt{in def}\ \kappa_1\langle x\rangle \triangleright \widehat{x}(x)|a\langle e\rangle \\ \texttt{in}\ c\langle\kappa_1\rangle|a\langle d\rangle|l\langle\rangle\ [\,] \end{array}$$

$$\begin{tikzcd}
& \mathcal{O}_e\left[\widehat{P}[z\langle x\rangle]\right] \arrow[r, dash, "\approx"] \arrow[d, "3"'] & \mathcal{O}_e\left[\widehat{Q}\right] \arrow[d, dashed, "*"'] & \\
\widehat{P}[\widehat{x}u] \arrow[r, dash, "\approx"] & \cdot \arrow[r, dash, dashed, "\approx"] & \cdot \arrow[r, dash, "\approx"] & \widehat{Q'}
\end{tikzcd}$$

$\widehat{Q'}$ has no barb on c; we can reorder the reductions on the right to defer interaction with the context $\mathcal{O}_e$, to obtain a sequence of reductions to mimic in the translation. $\square$

Lemma 16 *for all P, Q in the join-calculus,*

$$P \approx Q \quad \Rightarrow \quad \mathcal{E}^{\pi}[\![P]\!]_j \approx_{\pi} \mathcal{E}^{\pi}[\![Q]\!]_j$$

Definition 6 *We use the compositional encoding $[\![\ ]\!]_j$ where $D = x\langle u\rangle | y\langle v\rangle \triangleright Q$ to obtain the second half of the theorem:*

$$\begin{array}{rcl} [\![Q|R]\!]_j & \stackrel{\text{def}}{=} & [\![Q]\!]_j | [\![R]\!]_j \\ [\![x\langle v\rangle]\!]_j & \stackrel{\text{def}}{=} & \mathcal{E}^{\pi}_{x,v}[\overline{x_j}(v_j)] \\ [\![\ \mathtt{def}\ D\ \mathtt{in}\ R]\!]_j & \stackrel{\text{def}}{=} & \nu xy.((!x(u).y(v).[\![Q]\!]_j)|[\![R]\!]_j) \end{array}$$

Lemma 17 *For every join-process Q with free variables $\widetilde{z}$,*

$$[\![Q]\!]_j \quad \approx \quad \mathcal{E}^{\pi}_{\widetilde{z}}[\![Q]\!]_j$$

We conclude using the same argument that for the reverse translation.