

Concurrent Pattern Calculus

Thomas Given-Wilson¹, Daniele Gorla², and Barry Jay¹

¹ University of Technology, Sydney,

² Dip. di Informatica, Univ. di Roma “La Sapienza”

Abstract. Concurrent pattern calculus drives interaction between processes by comparing data structures, just as sequential pattern calculus drives computation. By generalising from pattern matching to pattern unification, interaction becomes symmetrical, with information flowing in both directions. This provides a natural language for describing any form of exchange or trade. Many popular process calculi can be encoded in concurrent pattern calculi.

1 Introduction

The π -calculus [11] holds a pivotal position among process calculi as it is the simplest that is able to support computation as represented by λ -calculus [1]. However, *pattern calculus* [9, 7] supports even more computations than λ -calculus, once one generalises from numbers to general data structures [8]. Hence it is natural to wonder what a concurrent pattern calculus might look like. In fact it turns out rather well.

We take (a slight enhancement of) the pattern matching mechanism of the pure pattern calculus [9, 7] and build up on top of it a concurrent process language, by adding the standard constructs of parallel composition, name restriction and replication. This yields a *concurrent pattern calculus* (CPC) where the usual prefixes for input and output along agreed channels can be combined into patterns; their unification triggers a two-way flow of information, as represented by the sole reduction rule

$$(p \rightarrow P \mid q \rightarrow Q) \quad \longmapsto \quad \sigma P \mid \rho Q$$

where σ and ρ are the substitutions resulting from the unification of p and q .

Of course, this is not the first proposal for equipping a concurrent language with structured forms of interactions. However, we shall show that CPC’s approach turns out to be more expressive than several well-known proposals. To this aim, let us consider some examples of pattern-matching appearing in the literature of process calculi. The π -calculus [11] matches the channel name with $m(x).P \mid n\langle y \rangle.Q$ reducing when $m = n$. A structural approach is used in the polyadic π -calculus [10] to match the number of names as well as the channel name, so that $m(x_1, \dots, x_i).P \mid n\langle y_1, \dots, y_j \rangle.Q$ will only reduce when $m = n$ and $i = j$. The fusion calculus [13] also matches the channel name and arity of the inputs and outputs but fuses names, instead of replacing input variables with

output names. In the spi calculus [4] pattern-matching occurs in communication (the same as the π -calculus) but also in reductions: the match $[M \text{ is } N]P$, when $M = N$; pair splitting $\text{let } (x, y) = M \text{ in } P$, when M is a pair; decryption case $M \text{ of } \{x\}_N : P$, to P when $M = \{M'\}_N$ for some M' ; and integer case $M \text{ of } 0 : P \text{ suc}(x) : Q$, to P when $M = 0$ or Q when $M = \text{suc}(N)$ for some N . Pattern matching in Linda [3] checks for sameness of arity and of some specific names when matching a datum against a template (used to select data from a shared dataspace).

In CPC, information flows in both directions, from p to Q and from q to P . Thus, its pattern matching is more symmetrical than communication in π -calculus, or the matching supported by spi calculus or Linda. Not incidentally, CPC can express all these forms of interactions: we show that π -calculus, spi calculus and Linda can be encoded into CPC. In contrast, CPC cannot be encoded (while respecting some reasonable properties) in any of the calculi mentioned above, including fusion calculus. Also, CPC's symmetry is different from the mechanism of name fusions supported by the fusion calculus: indeed, we also show that CPC is not able to properly simulate name fusions.

A natural objection to CPC is that the unification is too complex to be an atomic operation. In particular, any limit to the size of communicated messages could be violated by some match. Also, one cannot, in practice, implement a simultaneous exchange of information, so that pattern unification must be implemented in terms of simpler primitives.

This objection is similar to those made against λ -calculus, whose substitution is not atomic either. Even more, the pattern matching of Linda suffers from the same problems (it cannot be implemented as an atomic action), but there are many existing programming environments based on it ([12, 14], just to cite a few representative samples). Really it is a question of deciding how granular one wishes to be. We believe that CPC will prove to be a convenient specification language since, if symmetry between processes is to be taken seriously, there must always be some give and take, some exchange of information. This is most obvious in the world of trade, where negotiation is paramount, and the mechanics of settlement are secondary.

To this end, our major example supports a simple negotiation. Buyer and seller must *discover* their compatibility in an open environment, establish trust (through a third party) and then communicate privately.

The structure of the paper is as follows. Section 2 introduces symmetric matching through a concurrent pattern calculus. Section 3 develops a share trading example. Section 4 formalises the relation of concurrent pattern calculus to other process calculi. Section 5 concludes and considers future work.

2 Concurrent pattern calculus

This section presents a *concurrent pattern calculus* (CPC) that uses symmetric pattern matching as the basis of communication. Both symmetry and pattern matching appear in existing models of concurrency, but in more limited way.

For example, π -calculus requires a sender and receiver to share a channel, so that the presence of the channel is symmetric but information flow is in one direction only. Fusion calculus achieves this by fusing names together but has no patterns. On the other hand, spi calculus has patterns for natural numbers, and can check equality of terms (i.e. patterns or data structures) but does not perform matching in general, or support much symmetry.

The expressiveness of CPC comes from extending the traditional names to a class of *patterns*. The symmetry comes from allowing two patterns to be symmetrically matched or unified, to perform equality testing, input and output in a single step. The increased expressive power makes it harder to protect private information, but this can be managed by allowing some names (and patterns) to be protected, in the sense that they can be matched but not shared.

2.1 Syntax

The CPC has two syntactic classes, the *patterns* (meta-variables p, p_1, q, q_1, \dots) and the *processes* (meta-variables $P, P', P_1, Q, Q', Q_1, \dots$).

The patterns have the following forms

| | | |
|-----------------|-------------------------|----------------|
| <i>Patterns</i> | $p ::= x$ | variable name |
| | $\ulcorner x \urcorner$ | protected name |
| | λx | binding name |
| | $p \bullet p$ | compound. |

Variable names x are available for equality, output and substitution. Protected names $\ulcorner x \urcorner$ are only available for equality and substitution. Binding names λx are available for input only. The compound combines two patterns into a single one.

Given a pattern p the sets of: *variables names*, denoted $\text{vn}(p)$; *protected names*, denoted $\text{pn}(p)$; and *binding names*, denoted $\text{bn}(p)$, are as expected with the union being taken for compounds. The *free names* of a pattern p , written $\text{fn}(p)$, is the union of the variable names and protected names of p . A pattern is *well formed* if each binding name appears exactly once; we are only going to consider well-formed patterns.

As the protected names serve to test for equality and the binding names represent input, neither should be able to be communicated to another process. Thus, a pattern is *communicable* if it contains no protected or binding names.

A substitution σ (also denoted $\sigma_1, \rho, \rho_1, \dots$) is defined as a partial function from names to communicable patterns. The domain of σ is denoted $\text{dom}(\sigma)$ and the range $\text{ran}(\sigma)$. The set of free names of σ , written $\text{fn}(\sigma)$, is given by the union of the sets of the form $\text{fn}(\sigma x) \cup \{x\}$ where $x \in \text{dom}(\sigma)$. A set of names φ *avoids* σ if $\varphi \cap \text{fn}(\sigma) = \{\}$.

The *application* of a substitution σ to a pattern is defined as

$$\sigma x = \begin{cases} \sigma x & x \in \text{dom}(\sigma) \\ x & x \notin \text{dom}(\sigma) \end{cases} \quad \sigma \ulcorner x \urcorner = \begin{cases} \ulcorner \sigma x \urcorner & x \in \text{dom}(\sigma) \\ \ulcorner x \urcorner & x \notin \text{dom}(\sigma) \end{cases}$$

$$\sigma(\lambda x) = \lambda x \quad \sigma(p \bullet q) = (\sigma p) \bullet (\sigma q) .$$

As substitutions may replace a protected name with a communicable data structure, we define the following function on the latter

$$\ulcorner x \urcorner = \ulcorner x \urcorner \quad \ulcorner (p \bullet q) \urcorner = (\ulcorner p \urcorner) \bullet (\ulcorner q \urcorner) .$$

For the later development of α -conversion, given a name substitution σ we define the *renaming substitution* $\widehat{\sigma}$ on binding names as follows:

$$\widehat{\sigma}(\lambda x) = \begin{cases} \lambda y & x \in \text{dom}(\sigma) \text{ and } \sigma(x) = y \\ \lambda x & x \notin \text{dom}(\sigma) \end{cases}$$

$$\widehat{\sigma}x = x \quad \widehat{\sigma}\ulcorner x \urcorner = \ulcorner x \urcorner \quad \widehat{\sigma}(p \bullet q) = (\widehat{\sigma}p) \bullet (\widehat{\sigma}q) .$$

The *symmetric matching* or *unification* $\{p\|q\}$ of two patterns p and q attempts to unify p and q by generating substitutions upon their binding names. When defined, the result is some pair of substitutions whose domains are the binding names of p and of q . The rules to generate the substitutions are:

$$\left. \begin{array}{l} \{x\|x\} \\ \{x\|\ulcorner x \urcorner\} \\ \{\ulcorner x \urcorner\|x\} \\ \{\ulcorner x \urcorner\|\ulcorner x \urcorner\} \end{array} \right\} = \text{Some} (\{\}, \{\})$$

$$\begin{array}{ll} \{\lambda x\|q\} & = \text{Some} (\{q/x\}, \{\}) \quad \text{if } q \text{ is communicable} \\ \{p\|\lambda x\} & = \text{Some} (\{\}, \{p/x\}) \quad \text{if } p \text{ is communicable} \end{array}$$

$$\{p_1 \bullet p_2\|q_1 \bullet q_2\} = \text{Some} ((\sigma_1 \cup \sigma_2), (\rho_1 \cup \rho_2)) \begin{cases} \{p_1\|q_1\} = \text{Some} (\sigma_1, \rho_1) \\ \{p_2\|q_2\} = \text{Some} (\sigma_2, \rho_2) \end{cases}$$

$$\{p\|q\} = \text{undefined} \quad \text{otherwise.}$$

A name matches against itself when both instances are either variable or protected. That a protected name $\ulcorner x \urcorner$ unifies with the variable name x means that a process that protects a name may communicate with one that does not. A binding name λx binds any communicable pattern p by generating a substitution $\{p/x\}$. If both patterns are compounds and there is some matching for their respective components, then take the union of the substitutions. Otherwise the patterns cannot be unified and the matching is undefined.

Lemma 1. *If a pattern p has a protected name n and there is a pattern q where the matching of p and q is defined, then n must be in the free names of q .*

Proof: Without loss of generality q is not a binding name λx as p contains a protected name. Now proceed by induction on the structure of p :

- If p is a name then it follows that p is $\ulcorner n \urcorner$. Then q must be either n or $\ulcorner n \urcorner$ for the matching to be defined, therefore n is in the free names of q .
- If p is of the form $p_1 \bullet p_2$ then q must be of the form $q_1 \bullet q_2$. If $n \in \text{pn}(p_1)$ then proceed by induction on p_1 and q_1 , otherwise $n \in \text{pn}(p_2)$ and so proceed by induction on p_2 and q_2 . □

The processes of CPC are given by

| | | |
|--------------------------|-------------------|----------------------|
| <i>Processes</i> $P ::=$ | $\mathbf{0}$ | zero |
| | $P P$ | parallel composition |
| | $!P$ | replication |
| | $(\nu x)P$ | restriction |
| | $p \rightarrow P$ | case. |

The zero, parallel composition, replication and restriction are all familiar. The traditional input and output primitives are replaced by the case $p \rightarrow P$ that has a pattern p and a *body* P . The pattern of a case may be considered as a form of prefix, as commonly used for input or output.

The free names of processes, denoted $\text{fn}(P)$, are familiar for all the traditional primitives. The only interesting process form is the case

$$\text{fn}(p \rightarrow P) = \text{fn}(p) \cup (\text{fn}(P) \setminus \text{bn}(p))$$

since the binding names of the pattern bind their free occurrences in the body.

2.2 Operational Semantics

Renaming is handled through α -conversion, $=_\alpha$, that is the congruence relation generated by the following axioms

$$\begin{aligned} (\nu x)P &=_\alpha (\nu y)(\{y/x\}P) & y &\notin \text{fn}(P) \\ p \rightarrow P &=_\alpha (\{y/x\}p) \rightarrow (\{y/x\}P) & x &\in \text{bn}(p), y \notin \text{fn}(P) \cup \text{bn}(p). \end{aligned}$$

Proposition 1. *For every substitution σ and process P , there is an α -equivalent process P' such that $\sigma(P')$ is defined. If P_1 and P_2 are α -equivalent terms, then $\text{fn}(P_1) = \text{fn}(P_2)$ and, if $Q_1 = \sigma(P_1)$ and $Q_2 = \sigma(P_2)$ are both defined, then $Q_1 =_\alpha Q_2$.*

Proof: The proof is by straightforward induction. □

The general *structural equivalence relation* \equiv is defined in the usual way: it includes $=_\alpha$ and its defining axioms are the standard π -calculus ones [10]:

$$\begin{aligned} P | \mathbf{0} &\equiv P & P | Q &\equiv Q | P & P | (Q | R) &\equiv (P | Q) | R & (\nu n)\mathbf{0} &\equiv \mathbf{0} \\ (\nu n)(\nu m)P &\equiv (\nu m)(\nu n)P & !P &\equiv P | !P & P | (\nu n)Q &\equiv (\nu n)(P | Q) & \text{if } n &\notin \text{fn}(P) \end{aligned}$$

CPC has one *interaction rule* given by

$$(p \rightarrow P | q \rightarrow Q) \longmapsto (\sigma P) | (\rho Q) \quad \text{if } \{p\|q\} = \text{Some}(\sigma, \rho).$$

It states that if the unification of two patterns p and q is defined and generates $\text{Some}(\sigma, \rho)$, then apply the substitutions σ and ρ to the bodies P and Q , respectively. If the matching of p and q is undefined then no interaction occurs.

The interaction rule is then closed under parallel composition, restriction and structural equivalence to yield the reduction relation:

$$\frac{P \mapsto P'}{P|Q \mapsto P'|Q} \quad \frac{P \mapsto P'}{(\nu n)P \mapsto (\nu n)P'} \quad \frac{P \equiv Q \quad Q \mapsto Q' \quad Q' \equiv P'}{P \mapsto P'}$$

Definition 1. *The processes P and Q do not interact if, when ever there is a reduction $P|Q \mapsto R$, then there are processes P' and Q' such that $P \mapsto P'$, $Q \mapsto Q'$ and $R \equiv P'|Q'$.*

The remainder of this section formalises the property that protected names can be used to represent channels. Although the implementation of channel names through pattern-matching has been formalised before [5, 6], here the results shall be generalised to account for symmetric matching and the larger class of patterns.

First we show that if a case has a protected name n in the pattern then it will only interact with other cases that also have n in their pattern.

Lemma 2. *A process of the form $p \rightarrow P$ with a protected name n in the pattern, will only interact with a process Q if n is in the free names of Q .*

Proof: The proof is by induction on the reduction of Q :

- If Q is of the form $(q_1 \rightarrow Q_1)|Q_2$ and $\{p||q_1\}$ is defined then by Lemma 1 n is in the free names of Q .
- If $Q \mapsto Q'$ then proceed by induction on Q' . □

This result can be used to show that a message is *protected* when communicated between cases that simulate a channel. The following theorem is an adaption of “security” from the spi calculus [4, p. 38].

Theorem 1. *Given the processes*

$$A(m) = \ulcorner n \urcorner \bullet m \rightarrow F(m) \quad B = \ulcorner n \urcorner \bullet \lambda x \rightarrow G(x) \quad Inst(m) = (\nu n)(A(m) | B)$$

then the process $Inst(m)$ protects m if $F(m)$ and $G(m)$ protect m .

Proof: Consider the process

$$Inst(m)|R$$

where R does not contain m . Now proceed by induction on the reduction relation

- If $Inst(m)|R \mapsto ((\nu n)(F(m)|G(m)))|R$, then m is protected as $F(m)$ and $G(m)$ protect m .
- If $Inst(m)|R \mapsto Inst(m)|R'$, then proceed by induction.

Observe that if we take the structurally equivalent $(\nu n)((A(m)|B)|R)$ then n must not be in the free names of R . Therefore R cannot interact with A or B by Lemma 2 and m is protected. □

3 Trade

This section uses the example of share trading to explore the potential of CPC. The scenario is that two potential traders, a buyer and a seller, wish to engage in trade. To complete a transaction the traders need to progress through two stages: *discovering* each other and *exchanging* information. Both traders begin with a pattern for their desired transaction.

The discovery phase can be characterised as a pattern-unification problem, where traders' patterns are used to find a compatible partner.

The exchange phase occurs when a buyer and seller have agreed upon a transaction. Now each trader wishes to exchange information in a single interaction, preventing any incomplete trades from occurring.

The rest of this section explores three solutions to completing a transaction. The first is simple and focuses on how discovery can be achieved. The second introduces a registrar to validate the traders' credentials. The third extends the second with protected names to ensure privacy of communication.

Solution 1. Let us consider two traders, a buyer and a seller. The buyer Buy_1 with bank account b and desired shares s can be given by

$$\text{Buy}_1 = s \bullet \lambda m \rightarrow m \bullet b \bullet \lambda x \rightarrow B(x) .$$

The first pattern $s \bullet \lambda m$ is used to match with a compatible seller using share information s , and to input a name m to be used as a channel to exchange bank account information b for share certificates bound to x . The transaction successfully concludes with $B(x)$.

The seller Sell_1 with share certificates c and desired share sale s is given by

$$\text{Sell}_1 = (\nu n)s \bullet n \rightarrow n \bullet \lambda y \bullet c \rightarrow S(y) .$$

The seller creates a channel name n and then tries to find a buyer for the shares described in s , offering n to the buyer to continue the transaction. The channel is then used to exchange billing information, bound to y , for the share certificates c . The seller then concludes with the successfully completed transaction as $S(y)$.

The discovery phase succeeds when the traders are put in a parallel composition and match on the share information s

$$\begin{aligned} \text{Buy}_1 | \text{Sell}_1 &\equiv (\nu n)(s \bullet \lambda m \rightarrow m \bullet b \bullet \lambda x \rightarrow B(x) \mid s \bullet n \rightarrow n \bullet \lambda y \bullet c \rightarrow S(y)) \\ &\mapsto (\nu n)(n \bullet b \bullet \lambda x \rightarrow B(x) \mid n \bullet \lambda y \bullet c \rightarrow S(y)) . \end{aligned}$$

Here the share information s allows the traders to discover each other.

Once the traders have discovered each other, the next phase is to exchange the billing information for the share certificates. This trade of information occurs in the following interaction

$$(\nu n)(n \bullet b \bullet \lambda x \rightarrow B(x) \mid n \bullet \lambda y \bullet c \rightarrow S(y)) \mapsto (\nu n)(B(c) \mid S(b)).$$

The transaction concludes with the buyer having the share certificates c and the seller having the billing account b .

This solution allows the traders to discover each other and exchange information to complete a transaction. However, there is no way to determine if a process is trustworthy: anyone could claim to be a trader.

Solution 2. Consider the situation where a registrar keeps track of registered traders. The traders can offer their identity to potential partners and the registrar will confirm if the identity is for a valid trader. The buyer Buy_2 can now be represented by

$$\text{Buy}_2 = s \bullet i_B \bullet \lambda j \rightarrow n_B \bullet j \bullet \lambda m \rightarrow m \bullet b \bullet \lambda x \rightarrow B(x) .$$

The first pattern now swaps the buyer's identity i_B for the seller's identity bound to j . The buyer then asks the registrar using the identifier n_B to validate j and provide an identifier for the exchange phase. If that succeeds the exchange continues as before.

The seller Sell_2 is defined symmetrically by

$$\text{Sell}_2 = s \bullet \lambda j \bullet i_S \rightarrow n_S \bullet j \bullet \lambda m \rightarrow m \bullet \lambda y \bullet c \rightarrow S(y) .$$

The registrar Reg_2 has identifiers n_B and n_S to communicate with the buyer and seller, respectively, and is given by

$$\text{Reg}_2 = (\nu n)(n_B \bullet \ulcorner i_S \urcorner \bullet n \rightarrow \mathbf{0} \mid n_S \bullet \ulcorner i_B \urcorner \bullet n \rightarrow \mathbf{0}) .$$

The registrar creates a new identifier n to provide to traders who have been validated; then it makes the identifier available to known traders who attempt to validate another known trader. Although rather simple, the registrar can easily be extended to support a multitude of traders.

Running these processes in parallel we have the following interaction

$$\begin{aligned} & \text{Buy}_2 \mid \text{Sell}_2 \mid \text{Reg}_2 \\ \equiv & (\nu n)(s \bullet i_B \bullet \lambda j \rightarrow n_B \bullet j \bullet \lambda m \rightarrow m \bullet b \bullet \lambda x \rightarrow B(x) \mid n_B \bullet \ulcorner i_S \urcorner \bullet n \rightarrow \mathbf{0} \\ & \mid s \bullet \lambda j \bullet i_S \rightarrow n_S \bullet j \bullet \lambda m \rightarrow m \bullet \lambda y \bullet c \rightarrow S(y) \mid n_S \bullet \ulcorner i_B \urcorner \bullet n \rightarrow \mathbf{0}) \\ \mapsto & (\nu n)(n_B \bullet i_S \bullet \lambda m \rightarrow m \bullet b \bullet \lambda x \rightarrow B(x) \mid n_B \bullet \ulcorner i_S \urcorner \bullet n \rightarrow \mathbf{0} \\ & \mid n_S \bullet i_B \bullet \lambda m \rightarrow m \bullet \lambda y \bullet c \rightarrow S(y) \mid n_S \bullet \ulcorner i_B \urcorner \bullet n \rightarrow \mathbf{0}) . \end{aligned}$$

The share information s allows the buyer and seller to discovery each other and swap identities i_B and i_S . The next two interactions involve the buyer and seller validating each other's identity and inputting the identifier to complete the transaction

$$\begin{aligned} & (\nu n)(n_B \bullet i_S \bullet \lambda m \rightarrow m \bullet b \bullet \lambda x \rightarrow B(x) \mid n_B \bullet \ulcorner i_S \urcorner \bullet n \rightarrow \mathbf{0} \\ & \mid n_S \bullet i_B \bullet \lambda m \rightarrow m \bullet \lambda y \bullet c \rightarrow S(y) \mid n_S \bullet \ulcorner i_B \urcorner \bullet n \rightarrow \mathbf{0}) \\ \mapsto & (\nu n)(n \bullet b \bullet \lambda x \rightarrow B(x) \\ & \mid n_S \bullet i_B \bullet \lambda m \rightarrow m \bullet \lambda y \bullet c \rightarrow S(y) \mid n_S \bullet \ulcorner i_B \urcorner \bullet n \rightarrow \mathbf{0}) \\ \mapsto & (\nu n)(n \bullet b \bullet \lambda x \rightarrow B(x) \mid n \bullet \lambda y \bullet c \rightarrow S(y)) . \end{aligned}$$

Now that the traders have validated each other they can continue with the exchange step from before

$$(\nu n)(n \bullet b \bullet \lambda x \rightarrow B(x) \mid n \bullet \lambda y \bullet c \rightarrow S(y)) \quad \mapsto \quad (\nu n)(B(c) \mid S(b)) .$$

The traders exchange information and successfully complete with $B(c)$ and $S(b)$.

Although this solution satisfies the desire to validate that traders are legitimate, the freedom of matching allows for malicious processes to interfere. Consider the promiscuous process **Prom** given by

$$\text{Prom} = \lambda z_1 \bullet \lambda z_2 \bullet a \rightarrow P(z_1, z_2) .$$

This process is willing to match any other process that will swap two pieces of information for some arbitrary name a . Such a process could match with the traders trying to complete the exchange phase of a transaction

$$\begin{aligned} & (\nu n)(n \bullet b \bullet \lambda x \rightarrow B(x) \mid n \bullet \lambda y \bullet c \rightarrow S(y)) \mid \text{Prom} \\ \mapsto & (\nu n)(B(a) \mid n \bullet \lambda y \bullet c \rightarrow S(y) \mid P(n, b)) \end{aligned}$$

where the promiscuous process has stolen the identifier n and, more concerningly, the bank account information b . The unfortunate buyer is left with some useless information a and the seller is waiting to complete the transaction.

Solution 3. Here we repair the vulnerability of Solution 2 by using protected names to prevent promiscuous processes from being able to jump into any transaction.

The buyer, seller and registrar can be repaired to

$$\begin{aligned} \text{Buy}_3 &= s \bullet i_B \bullet \lambda j \rightarrow \ulcorner n_B \urcorner \bullet j \bullet \lambda m \rightarrow \ulcorner m \urcorner \bullet b \bullet \lambda x \rightarrow B(x) \\ \text{Sell}_3 &= s \bullet \lambda j \bullet i_S \rightarrow \ulcorner n_S \urcorner \bullet j \bullet \lambda m \rightarrow \ulcorner m \urcorner \bullet \lambda y \bullet c \rightarrow S(y) \\ \text{Reg}_3 &= (\nu n)(\ulcorner n_B \urcorner \bullet \ulcorner i_S \urcorner \bullet n \rightarrow \mathbf{0} \mid \ulcorner n_S \urcorner \bullet \ulcorner i_B \urcorner \bullet n \rightarrow \mathbf{0}) . \end{aligned}$$

Now all communication between the buyer, seller and registrar use protected identifiers: $\ulcorner n_B \urcorner$, $\ulcorner n_S \urcorner$ and $\ulcorner m \urcorner$. Thus all that remains is to ensure appropriate restrictions:

$$(\nu n_B)(\nu n_S)(\text{Buy}_3 \mid \text{Sell}_3 \mid \text{Reg}_3) .$$

Therefore other processes can only interact with the traders during the discovery phase, which will not lead to a successful transaction. The registrar will only interact with the traders as all the registrar's patterns have protected names known only to the registrar and a trader.

The solution could be extended further. Although treated as a simple variable name in the example, the share information could be made more realistic by being represented as a compound structure with a company code, number of shares and price per share, e.g. $\text{ABC} \bullet 100 \bullet \0.38 . This format allows discovery based on partial share information. Consider two examples: specify a company

code and price, but not the number of shares $ABC \bullet \lambda v \bullet \0.38 ; or specify only the price and accept any company or number of shares $\lambda u \bullet \lambda v \bullet \0.38 . The seller could also offer similarly partial share information, although this may be a very risky business strategy! Observe that either the buyer or seller can protect any component of the pattern if they wish to ensure that the other party exactly meets that criterion.

Another possibility is to allow for some checking of the integrity of the patterns being communicated. Given some standard language for the representation of data, such as XML, this could be checked by the matching. For example, a valid bank account may be required to have: an account number and account name. Thus a pattern to input only valid bank accounts, binding the account number to u , the name to v and using standardised tags `accountnumber` and `accountname`, could be $(\ulcorner \text{accountnumber} \urcorner \bullet \lambda u) \bullet (\ulcorner \text{accountname} \urcorner \bullet \lambda v)$. Thus any pattern that successfully matches must be identically structured and tagged.

4 Comparison with other process calculi

In this section, we exploit the techniques developed in [5,6] to formally assess the expressive power of CPC w.r.t. well-known process calculi. To this aim, we first recall some basic material from [6].

First of all, let us add to each calculus a reserved process \checkmark , used to signal successful termination. Moreover, let a k -ary context $\mathcal{C}_{(-1; \dots; -k)}$ be a term where k occurrences of $\mathbf{0}$ are linearly replaced by the holes $\{-1; \dots; -k\}$ (every hole must occur once and only once).

An *encoding* of a language \mathcal{L}_1 into another language \mathcal{L}_2 is a pair $(\llbracket \cdot \rrbracket, \varphi_{\llbracket \cdot \rrbracket})$ where $\llbracket \cdot \rrbracket$ translates every \mathcal{L}_1 -process into an \mathcal{L}_2 -process and $\varphi_{\llbracket \cdot \rrbracket}$ maps every source name into a k -tuple of (target) names, for $k > 0$. The translation $\llbracket \cdot \rrbracket$ turns every source term into a target term; in doing this, it is possible that the translation fixes some names to play a precise rôle or it can translate a single name into a tuple of names. This can be obtained by exploiting $\varphi_{\llbracket \cdot \rrbracket}$ (for more details, see [6]).

We now consider only encodings that satisfy the following properties, that are justified and discussed at length in [6]. Here, we denote with \mapsto_i the reduction relation in language \mathcal{L}_i and with \Longrightarrow_i its reflexive and transitive closure; we denote with \mapsto_i^ω an infinite sequence of reductions in \mathcal{L}_i . Moreover, we let \simeq_i denote the reference behavioural equivalence for language \mathcal{L}_i . Also, we let $P \Downarrow_i$ mean that there exists P' such that $P \Longrightarrow_i P'$ and $P' \equiv P'' \mid \checkmark$, for some P'' . Finally, to simplify reading, we let S range over processes of the source language (viz., \mathcal{L}_1) and T range over processes of the target language (viz., \mathcal{L}_2).

Definition 2 (Valid Encoding). *An encoding $(\llbracket \cdot \rrbracket, \varphi_{\llbracket \cdot \rrbracket})$ is valid if it satisfies the following five properties:*

1. *Compositionality: for every k -ary operator op of \mathcal{L}_1 and for every subset of names N , there exists a k -ary context $\mathcal{C}_{\text{op}}^N(-1; \dots; -k)$ such that, for all S_1, \dots, S_k with $\text{fn}(S_1, \dots, S_k) = N$, it holds that $\llbracket \text{op}(S_1, \dots, S_k) \rrbracket = \mathcal{C}_{\text{op}}^N(\llbracket S_1 \rrbracket; \dots; \llbracket S_k \rrbracket)$.*

2. Name invariance: for every S and name substitution σ , it holds that

$$\llbracket \sigma S \rrbracket \begin{cases} = \sigma' \llbracket S \rrbracket & \text{if } \sigma \text{ is injective} \\ \simeq_2 \sigma' \llbracket S \rrbracket & \text{otherwise} \end{cases}$$

where σ' is such that $\varphi_{\llbracket \cdot \rrbracket}(\sigma(a)) = \sigma'(\varphi_{\llbracket \cdot \rrbracket}(a))$ for every name a .

3. Operational correspondence:

– for all $S \Longrightarrow_1 S'$, it holds that $\llbracket S \rrbracket \Longrightarrow_2 \simeq_2 \llbracket S' \rrbracket$;

– for all $\llbracket S \rrbracket \Longrightarrow_2 T$, there exists S' such that $S \Longrightarrow_1 S'$ and $T \Longrightarrow_2 \simeq_2 \llbracket S' \rrbracket$.

4. Divergence reflection: for every S such that $\llbracket S \rrbracket \mapsto_2^\omega$, it holds that $S \mapsto_1^\omega$.

5. Success sensitiveness: for every S , it holds that $S \Downarrow_1$ if and only if $\llbracket S \rrbracket \Downarrow_2$.

[6] contains some results concerning valid encodings. In particular, it shows some proof-techniques for showing separation results, i.e. for proving that no valid encoding can exist between a pair of languages satisfying certain conditions.

Proposition 2 (from [6]). *Let $\llbracket \cdot \rrbracket$ be a valid encoding; then, $S \not\mapsto_1$ implies that $\llbracket S \rrbracket \not\mapsto_2$.*

Theorem 2 (from [6]). *Assume that there exists S such that $S \not\mapsto_1$, $S \not\Downarrow_1$ and $S \mid S \Downarrow_1$; moreover, assume that every T that does not reduce is such that $T \mid T \not\mapsto_2$. Then, there cannot exist any valid encoding of \mathcal{L}_1 into \mathcal{L}_2 .*

To state the following proof-technique, let us define the *matching degree* of a language \mathcal{L} , written $\text{MD}(\mathcal{L})$, as the least upper bound on the number of names that must be matched to yield a reduction in \mathcal{L} .

Theorem 3 (from [6]). *If $\text{MD}(\mathcal{L}_1) > \text{MD}(\mathcal{L}_2)$, then there exists no valid encoding of \mathcal{L}_1 into \mathcal{L}_2 .*

We now exploit these techniques to compare CPC with well-known process calculi and languages. In particular, we shall say that \mathcal{L}_1 is *more expressive* than \mathcal{L}_2 if there exists a valid encoding of the latter into the former but not vice versa.

4.1 π -calculus and Linda

A hierarchy of process calculi with different communication primitives is obtained [5] by combining four features: synchronism (synchronous vs asynchronous), arity (monadic vs polyadic data exchange), communication medium (channels vs shared dataspace) and the presence of a form of pattern-matching (that checks for (i) the same length of the output message and of the input parameters, and (ii) for equality of some specific names). In particular, it covers monadic/polyadic π -calculus [11, 10] and Linda [3]. The hierarchy is built by using a notion of encoding very similar to valid encodings presented in Definition 2 and, in particular, it is proved that Linda (called $L_{A,P,D,PM}$ in *loc.cit.*) is more expressive than monadic/polyadic π -calculus (called $L_{S,M,C,NO}$ and $L_{S,P,C,NO}$ in *loc.cit.*).

For our purposes, it suffices to show that CPC is more expressive than $L_{A,P,D,PM}$; to this aim, let us recall the definition of $L_{A,P,D,PM}$, whose syntax is slightly modified here w.r.t. [5] to follow CPC's syntax. Processes are defined as:

$$P ::= \mathbf{0} \mid \surd \mid \langle b_1, \dots, b_n \rangle \mid (t_1, \dots, t_n).P \mid (\nu n)P \mid P|Q \mid !P$$

where b ranges over names and t denotes a template field, defined by the following syntax:

$$t ::= \lambda x \mid \ulcorner b \urcorner$$

We assume that input variables occurring in templates are all distinct: this assumption rules out template (x, x) , but accepts $(x, \ulcorner b \urcorner, \ulcorner b \urcorner)$. Templates are used to implement Linda's pattern matching, defined as follows:

$$\text{MATCH}(\ ;) = \{\} \quad \text{MATCH}(\ulcorner b \urcorner; b) = \{\} \quad \text{MATCH}(\lambda x; b) = \{b/x\}$$

$$\frac{\text{MATCH}(t; b) = \sigma_1 \quad \text{MATCH}(\tilde{t}; \tilde{b}) = \sigma_2}{\text{MATCH}(t, \tilde{t}; b, \tilde{b}) = \sigma_1 \uplus \sigma_2}$$

where \tilde{e} denotes a (possibly empty) sequence of entities of kind e (names or template fields, in our case) and ' \uplus ' denotes the union of partial functions with disjoint domains. The interaction rule for $L_{A,P,D,PM}$ is given by

$$\langle \tilde{b} \rangle \mid (\tilde{t}).P \longmapsto \sigma P \quad \text{if } \text{MATCH}(\tilde{t}; \tilde{b}) = \sigma$$

The reduction relation is obtained by closing this interaction rule by parallel, restriction and the same structural equivalence relation defined for CPC.

We first show that CPC cannot be encoded into $L_{A,P,D,PM}$; this is a corollary of Theorem 2. Indeed, consider the CPC process $x \rightarrow \surd$: it cannot reduce and cannot report success but, if put in parallel with itself, it reports success. On the contrary, it is easy to prove that every $L_{A,P,D,PM}$ -process that reduces if put in parallel with itself is such that it reduces in isolation.

We now have to show a valid encoding of $L_{A,P,D,PM}$ into CPC. The encoding is a homomorphism w.r.t. to all operators, with the only two following exceptions:

$$\llbracket \langle \tilde{b} \rangle \rrbracket \stackrel{\text{def}}{=} \text{pat-d}(\tilde{b}) \rightarrow \mathbf{0} \quad \llbracket (\tilde{t}).P \rrbracket \stackrel{\text{def}}{=} \text{pat-t}(\tilde{t}) \rightarrow \llbracket P \rrbracket$$

Functions $\text{pat-d}(\cdot)$ and $\text{pat-t}(\cdot)$ are used to translate data and templates into CPC patterns; they are defined as follows:

$$\begin{aligned} \text{pat-d}(\) &\stackrel{\text{def}}{=} \lambda x & \text{pat-d}(b, \tilde{b}) &\stackrel{\text{def}}{=} \lambda x \bullet b \bullet \text{pat-d}(\tilde{b}) & \text{for } x \notin \text{bn}(\text{pat-d}(\tilde{b})) \\ \text{pat-t}(\) &\stackrel{\text{def}}{=} \text{in} & \text{pat-t}(t, \tilde{t}) &\stackrel{\text{def}}{=} \text{in} \bullet t \bullet \text{pat-t}(\tilde{t}) \end{aligned}$$

where in is any name (for the sake of clarity, we have used a symbolic name, but this is not necessary for proving any result). It is worth noting that the simpler translation

$$\llbracket \langle b_1, \dots, b_n \rangle \rrbracket \stackrel{\text{def}}{=} b_1 \bullet \dots \bullet b_n \rightarrow \mathbf{0}$$

would not work: the $L_{A,P,D,PM}$ -process $\langle b \rangle \mid \langle b \rangle$ does not reduce, whereas its encoding (viz., $b \rightarrow \mathbf{0} \mid b \rightarrow \mathbf{0}$) does. This fact would contradict Proposition 2. Moreover, function $\text{pat-d}(\cdot)$ associates a bound variable to every name in the sequence; this fact ensures that a pattern that translates a template and a pattern that translates a datum match only if they have the same length (this is a feature of $L_{A,P,D,PM}$'s pattern matching but not of CPC's one).

We now have to prove that this encoding is valid. To this aim, the main result is Theorem 4; this is an easy corollary of the following two lemmata, the first one being the most interesting and stating a strict correspondence between $L_{A,P,D,PM}$'s pattern matching and CPC's one (on patterns arising from the translation).

Lemma 3. $\text{MATCH}(\tilde{t}; \tilde{b}) = \sigma$ if and only if $\{\text{pat-t}(\tilde{t}) \parallel \text{pat-d}(\tilde{b})\} = \text{Some}(\sigma \cup \{\text{in}/x_0, \dots, \text{in}/x_n\}; \{\})$, where $\{x_0, \dots, x_n\} = \text{bn}(\text{pat-d}(\tilde{b}))$ and σ maps names to names.

Proof: The proof is by induction on the length of \tilde{t} . For the base case, we have that \tilde{t} is the empty sequence of template fields; thus, $\text{pat-t}(\tilde{t}) = \text{in}$.

\Rightarrow : by definition of function MATCH , it must be that \tilde{b} is the empty sequence and that σ is the empty substitution. Thus, $\text{pat-d}(\tilde{b}) = \lambda x$ and the thesis easily follows.

\Leftarrow : let us assume by contradiction that \tilde{b} is not the empty sequence. In this case, $\text{pat-d}(\tilde{b}) = \lambda x_0 \bullet \dots \bullet \lambda x_n$, for some $n > 0$. By definition of pattern matching in CPC, $\text{pat-d}(\tilde{b})$ and $\text{pat-t}(\tilde{t})$ cannot match, and this would contradict the hypothesis. Thus, it must be that \tilde{b} is the empty sequence and we easily conclude.

For the inductive step, we have that $\tilde{t} = t, \tilde{t}'$; thus, $\text{pat-t}(\tilde{t}) = \text{in} \bullet t \bullet \text{pat-t}(\tilde{t}')$.

\Rightarrow : by definition of function MATCH , it must be that $\tilde{b} = b, \tilde{b}'$, $\text{MATCH}(t, b) = \sigma_1$, $\text{MATCH}(\tilde{t}', \tilde{b}') = \sigma_2$ and $\sigma = \sigma_1 \uplus \sigma_2$. Then $\{\text{pat-t}(\tilde{t}') \parallel \text{pat-d}(\tilde{b}')\} = \text{Some}(\sigma_2 \cup \{\text{in}/x_1, \dots, \text{in}/x_n\}; \{\})$, where $\{x_1, \dots, x_n\} = \text{bn}(\text{pat-d}(\tilde{b}'))$ by the induction hypothesis. Let us consider two subcases, according to the kind of template field t :

- $t = \ulcorner b \urcorner$: in this case $\sigma_1 = \{\}$; thus, $\sigma = \sigma_2$ and $\{\text{pat-t}(\tilde{t}) \parallel \text{pat-d}(\tilde{b})\} = \text{Some}(\sigma \cup \{\text{in}/x_0, \dots, \text{in}/x_n\}; \{\})$.

- $t = \lambda x$: in this case $\sigma_1 = \{b/x\}$ and $x \notin \text{dom}(\sigma_2)$; thus, $\text{pat-t}(\tilde{t})$ is a pattern in CPC and $\{\text{pat-t}(\tilde{t}) \parallel \text{pat-d}(\tilde{b})\} = \text{Some}(\{\text{in}/x_0\} \cup \sigma_1 \cup \sigma_2 \cup \{\text{in}/x_1, \dots, \text{in}/x_n\}; \{\}) = \text{Some}(\sigma \cup \{\text{in}/x_0, \dots, \text{in}/x_n\}; \{\})$.

\Leftarrow : by contradiction, assume that \tilde{b} is the empty sequence; thus, $\text{pat-d}(\tilde{b}) = \lambda x$ and it cannot match against $\text{pat-t}(\tilde{t})$ (notice that, by definition of function $\text{pat-t}(\cdot)$, we have that $\text{pat-t}(\tilde{t})$ is communicable if and only if \tilde{t} is the empty sequence). Hence, $\tilde{b} = b, \tilde{b}'$ and so $\text{pat-d}(\tilde{b}) = \lambda x \bullet b \bullet \text{pat-d}(\tilde{b}')$. By definition of pattern matching in CPC, we have that $\{\text{pat-t}(\tilde{t}') \parallel \text{pat-d}(\tilde{b}')\} = \text{Some}(\{\text{in}/x_0\} \cup \sigma_1 \cup \sigma'; \{\})$, where

$\{t\|b\} = \text{Some}(\sigma_1; \{\})$, $\{\text{pat-t}(\tilde{t})\|\text{pat-d}(\tilde{b}')\} = \text{Some}(\sigma'; \{\})$ and $\sigma \cup \{\text{in}/x_0, \dots, \text{in}/x_n\} = \{\text{in}/x_0\} \cup \sigma_1 \cup \sigma'$. The latter fact entails that $\sigma_1 \cup \sigma' = \sigma \cup \{\text{in}/x_1, \dots, \text{in}/x_n\}$. Let us consider two subcases, according to the kind of template field t :

- $t = \ulcorner b \urcorner$: in this case $\sigma_1 = \{\}$ and so $\sigma' = \sigma \cup \{\text{in}/x_1, \dots, \text{in}/x_n\}$. By induction hypothesis, $\text{MATCH}(\tilde{t}'; \tilde{b}') = \sigma$, and so $\text{MATCH}(\tilde{t}; \tilde{b}) = \sigma$.
- $t = \lambda x$: in this case $\sigma_1 = \{b/x\}$ and $\sigma' = \sigma_2 \cup \{\text{in}/x_1, \dots, \text{in}/x_n\}$, where $\sigma_2 = \sigma|_{\text{dom}(\sigma) \setminus \{x\}}$; thus, $x \notin \text{dom}(\sigma_2)$ and so $\sigma = \sigma_1 \uplus \sigma_2$. By induction hypothesis, $\text{MATCH}(\tilde{t}'; \tilde{b}') = \sigma_2$ and, by definition of MATCH , $\text{MATCH}(t; b) = \sigma_1$; thus, $\text{MATCH}(\tilde{t}; \tilde{b}) = \sigma$. □

Lemma 4. $P \equiv Q$ if and only if $\llbracket P \rrbracket \equiv \llbracket Q \rrbracket$.

Proof: Trivial, from the fact that \equiv acts only on operators that $\llbracket \cdot \rrbracket$ translates homomorphically. □

Theorem 4.

- If $P \mapsto P'$ then $\llbracket P \rrbracket \mapsto \llbracket P' \rrbracket$;
- if $\llbracket P \rrbracket \mapsto Q$ then $Q = \llbracket P' \rrbracket$ for some P' such that $P \mapsto P'$.

Proof: Both parts can be easily proved by a straightforward induction on judgments $P \mapsto P'$ and $\llbracket P \rrbracket \mapsto Q$, respectively. In both cases, the base step is the most interesting one and it trivially follows from Lemma 3; the inductive cases where the last rule used is the structural one rely on Lemma 4. □

Corollary 1. *The encoding of $L_{A,P,D,PM}$ into CPC is valid.*

Proof: Compositionality and name invariance hold by construction. Operational correspondence and divergence reflection easily follow from Theorem 4. Success sensitiveness can be proved as follows: $P \Downarrow$ means that there exists P' and $k \geq 0$ such that $P \mapsto^k P' \equiv P'' \mid \surd$; by exploiting Theorem 4 k times and Lemma 4, we obtain that $\llbracket P \rrbracket \mapsto^k \llbracket P' \rrbracket \equiv \llbracket P'' \rrbracket \mid \surd$, i.e. that $\llbracket P \rrbracket \Downarrow$. The converse implication can be proved similarly. □

4.2 Fusion

We now present the Fusion calculus [13] by following the presentation given in [15]. Processes are defined as

$$P ::= \mathbf{0} \mid P|P \mid (\nu x)P \mid !P \mid \bar{u}(\tilde{x}).P \mid u(\tilde{x}).P$$

The interaction axiom for Fusion is

$$(\nu \tilde{u})(\bar{u}(\tilde{x}).P \mid u(\tilde{y}).Q \mid R) \mapsto \sigma P \mid \sigma Q \mid \sigma R \quad \text{with } \text{dom}(\sigma) \cup \text{ran}(\sigma) \subseteq \{\tilde{x}, \tilde{y}\} \\ \text{and } \tilde{u} = \text{dom}(\sigma) \setminus \text{ran}(\sigma) \text{ and} \\ \sigma(v) = \sigma(w) \text{ iff } (v, w) \in E(\tilde{x} = \tilde{y})$$

where $E(\tilde{x} = \tilde{y})$ is the least equivalence relation on names generated by the equalities $\tilde{x} = \tilde{y}$ (that is defined whenever $|\tilde{x}| = |\tilde{y}|$). Fusion's reduction relation is obtained by closing the interaction axiom under parallel, restriction and the structural equivalence presented for CPC.

We now prove that Fusion and CPC are unrelated, i.e. that there exists no valid encoding of one into the other. The impossibility for a valid encoding of CPC into Fusion is ensured by Theorem 3: the matching degree of Fusion is 1 (only the channel name is checked for equality in any interaction); by contrast, the matching degree of CPC is ∞ , since we can atomically check for any number of name equalities within a single CPC interaction. The converse separation result is ensured by the following theorem.

Theorem 5. *There exists no valid encoding of Fusion into CPC.*

Proof: By contradiction, assume that there exists a valid encoding $\llbracket \cdot \rrbracket$ of Fusion into CPC. Consider the Fusion process $P \stackrel{\text{def}}{=} (\nu x)(\bar{u}\langle x \mid u(y).\sqrt{\ } \rangle)$, for x, y and u pairwise distinct. By success sensitiveness, $P \Downarrow$ entails that $\llbracket P \rrbracket \Downarrow$; we first want to prove that the latter fact can only happen after a reduction of $\llbracket P \rrbracket$, i.e. that every occurrence of $\sqrt{\ }$ in $\llbracket P \rrbracket$ falls underneath some prefix. By compositionality, $\llbracket P \rrbracket \stackrel{\text{def}}{=}} \mathcal{C}_{(\nu x)}^{\{u,x,y\}}(\mathcal{C}_{|}^{\{u,x,y\}}(\llbracket \bar{u}\langle x \rangle \rrbracket; \llbracket u(y).\sqrt{\ } \rrbracket))$. If $\llbracket P \rrbracket$ had a top-level unguarded occurrence of $\sqrt{\ }$, then such an occurrence could be in $\mathcal{C}_{(\nu x)}^{\{u,x,y\}}(-)$, in $\mathcal{C}_{|}^{\{u,x,y\}}(-_1; -_2)$, in $\llbracket \bar{u}\langle x \rangle \rrbracket$ or in $\llbracket u(y).\sqrt{\ } \rrbracket$; in any case, we would also have that at least one of $\llbracket (\nu x)(\bar{u}\langle x \mid y(u).\sqrt{\ } \rangle) \rrbracket$ or $\llbracket (\nu x)(\bar{x}\langle u \mid u(y).\sqrt{\ } \rangle) \rrbracket$ would report success, whereas both $(\nu x)(\bar{u}\langle x \mid y(u).\sqrt{\ } \rangle) \not\Downarrow$ and $(\nu x)(\bar{x}\langle u \mid u(y).\sqrt{\ } \rangle) \not\Downarrow$, against success sensitiveness of $\llbracket \cdot \rrbracket$. Thus, the only possibility for $\llbracket P \rrbracket$ to report success is to perform some reduction steps (at least one) and then exhibit a top-level unguarded occurrence of $\sqrt{\ }$.

We have thus obtained that $\llbracket P \rrbracket$ must reduce. We now prove that every possible reduction leads to contradict validity of $\llbracket \cdot \rrbracket$; this suffices to conclude. There are five possibilities for any reduction $\llbracket P \rrbracket \mapsto$:

1. either $\mathcal{C}_{(\nu x)}^{\{u,x,y\}} \mapsto$, or $\mathcal{C}_{|}^{\{u,x,y\}} \mapsto$, or $\llbracket \bar{u}\langle x \rangle \rrbracket \mapsto$ or $\llbracket u(y).\sqrt{\ } \rrbracket \mapsto$. In any of these cases, we would have that at least one of $\llbracket (\nu x)(\bar{u}\langle x \mid y(u).\sqrt{\ } \rangle) \rrbracket$ or $\llbracket (\nu x)(\bar{x}\langle u \mid u(y).\sqrt{\ } \rangle) \rrbracket$ would reduce, whereas both $(\nu x)(\bar{u}\langle x \mid y(u).\sqrt{\ } \rangle) \not\mapsto$ and $(\nu x)(\bar{x}\langle u \mid u(y).\sqrt{\ } \rangle) \not\mapsto$, against Proposition 2 (that must hold whenever $\llbracket \cdot \rrbracket$ is valid).
2. the reduction is generated by an interaction between $\mathcal{C}_{(\nu x)}^{\{u,x,y\}}$ and $\mathcal{C}_{|}^{\{u,x,y\}}$. Like before, $\llbracket (\nu x)(\bar{u}\langle x \mid y(u).\sqrt{\ } \rangle) \rrbracket \mapsto$ whereas $(\nu x)(\bar{u}\langle x \mid y(u).\sqrt{\ } \rangle) \not\mapsto$, against Proposition 2.
3. the reduction is generated by an interaction between $\mathcal{C}_{\text{op}}^{\{u,x,y\}}$ and $\llbracket \bar{u}\langle x \rangle \rrbracket$, for $\text{op} \in \{(\nu x), |\}$. Like case 2.
4. the reduction is generated by an interaction between $\mathcal{C}_{\text{op}}^{\{u,x,y\}}$ and $\llbracket u(y).\sqrt{\ } \rrbracket$, for $\text{op} \in \{(\nu x), |\}$. We now would have that $\llbracket (\nu x)(\bar{x}\langle u \mid u(y).\sqrt{\ } \rangle) \rrbracket \mapsto$ whereas $(\nu x)(\bar{x}\langle u \mid u(y).\sqrt{\ } \rangle) \not\mapsto$, against Proposition 2.

5. the reduction is generated by an interaction between $\llbracket \bar{u}(x) \rrbracket$ and $\llbracket u(y).\sqrt{} \rrbracket$. In this case, we would have that $\llbracket \bar{u}(x) \mid u(y).\sqrt{} \rrbracket \mapsto$ whereas $\bar{u}(x) \mid u(y).\sqrt{} \not\mapsto$: indeed, the interaction rule of Fusion imposes that at least one between x and y must be restricted to yield the interaction. \square

4.3 Spi

We now explore the relation between the spi calculus [4] and CPC. The spi calculus is unusual as the names covered by previous discussion are now generalised to *terms* of the form

$$M, N ::= n \mid x \mid (M, N) \mid 0 \mid \text{suc}(M) \mid \{M\}_N$$

These are rather like the patterns of CPC. Of particular interest is the pair, that combines terms and allows the construction of arbitrary structured data, and the encryption construct. Pairing is distinct from the polyadic data exchange discussed previously, as compound messages may be bound to a single name and then decomposed later. Similarly, the encryption requires they key to be known to gain access to the encrypted term.

The processes of the spi calculus are

$$\begin{aligned} P, Q ::= & 0 \mid P|Q \mid !P \mid (\nu m)P \mid M(x).P \mid \bar{M}\langle N \rangle.P \\ & \mid [M \text{ is } N]P \mid \text{let } (x, y) = M \text{ in } P \\ & \mid \text{case } M \text{ of } \{x\}_N : P \mid \text{case } M \text{ of } 0 : P \text{ suc}(x) : Q. \end{aligned}$$

The nil process, parallel composition, replication and restriction are all familiar from π -calculus [10]. The input $M(x).P$ and output $\bar{M}\langle N \rangle.P$ are generalised to allow terms in the place of channel names and output arguments. The match $[M \text{ is } N]P$ determines equality of M and N . The splitting $\text{let } (x, y) = M \text{ in } P$ decomposes pairs. The decryption case $\text{case } M \text{ of } \{x\}_N : P$ decrypts M and binds the encrypted message to x in the continuation. The integer case $\text{case } M \text{ of } 0 : P \text{ suc}(x) : Q$ branches according to the number. The last three can all get stuck if they have the wrong value for M .

In this section we shall focus on the novel aspects of spi calculus, the familiar aspects already covered in [5, 6] and Section 4.1. However, we will consider a slightly modified version of spi calculus where “interaction” is generalised to allow any term be used as a channel for communicating.

$$\bar{M}\langle N \rangle.P \mid M(x).Q \mapsto P \mid \{N/x\}Q \quad (1)$$

$$[M \text{ is } M]P \mapsto P \quad (2)$$

$$\text{let } (x, y) = (M, N) \text{ in } P \mapsto \{M/x, N/y\}P \quad (3)$$

$$\text{case } \{M\}_N \text{ of } \{x\}_N : P \mapsto \{M/x\}P \quad (4)$$

$$\text{case } 0 \text{ of } 0 : P \text{ suc}(x) : Q \mapsto P \quad (5)$$

$$\text{case } \text{suc}(M) \text{ of } 0 : P \text{ suc}(x) : Q \mapsto \{M/x\}Q \quad (6)$$

The reduction relation is obtained by closing the axioms under parallel composition, restriction and the same structural equivalence as the one defined for CPC.

We first show that CPC cannot be encoded into spi calculus; this is a corollary of Theorem 2 and identical to the technique used in Section 4.1. Consider again the CPC process $x \rightarrow \surd$: it cannot reduce and cannot report success but, if put in parallel with itself, it reports success. It is easy to prove that every spi calculus process that reduces if put in parallel with itself is such that it reduces in isolation.

We shall now develop an encoding of the spi calculus into CPC. The terms can be encoded as data structures using the reserved the names `pair`, `encr`, `0` and `suc` as follows

$$\begin{array}{ll} \llbracket n \rrbracket \stackrel{\text{def}}{=} n & \llbracket (M, N) \rrbracket \stackrel{\text{def}}{=} \text{pair} \bullet \llbracket M \rrbracket \bullet \llbracket N \rrbracket \\ \llbracket x \rrbracket \stackrel{\text{def}}{=} x & \llbracket \{M\}_N \rrbracket \stackrel{\text{def}}{=} \text{encr} \bullet \llbracket M \rrbracket \bullet \llbracket N \rrbracket \\ \llbracket 0 \rrbracket \stackrel{\text{def}}{=} 0 & \llbracket \text{suc}(M) \rrbracket \stackrel{\text{def}}{=} \text{suc} \bullet \llbracket M \rrbracket . \end{array}$$

The tagging is used for safety, as otherwise there are potential pathologies in the translation: without tags, the representation of a natural number could be confused with a pair or an encryption.

For processes, the encoding of the familiar forms are as expected. The input and output both encode as cases:

$$\begin{array}{l} \llbracket M(x).P \rrbracket \stackrel{\text{def}}{=} \text{in} \bullet \ulcorner \llbracket M \rrbracket \urcorner \bullet \lambda x \rightarrow \llbracket P \rrbracket \\ \llbracket \overline{M}\langle N \rangle.P \rrbracket \stackrel{\text{def}}{=} \lambda x \bullet \ulcorner \llbracket M \rrbracket \urcorner \bullet (\llbracket N \rrbracket) \rightarrow \llbracket P \rrbracket \quad x \notin \text{fn}(\llbracket P \rrbracket) . M, \llbracket N \rrbracket) . \end{array}$$

The encoding of the input $M(x).P$ has a pattern offering the reserved name in compounded with the encoding of the term M and the binding of the name x . The encoding of the output $\overline{M}\langle N \rangle.P$ has a pattern binding the fresh name x compounded with the encoding of M and the encoding of N . The reserved name in (input) and fresh name x (output) are used to ensure that encoded inputs will only match with encoded outputs. Observe that in both processes forms $\llbracket M \rrbracket$ contains no binding names, and so is communicable.

The four remaining process forms all require pattern-matching and so translate to cases in parallel. In each encoding a fresh name n is used to prevent interaction with other processes, see Lemma 2. Note that, as in the spi calculus, the encodings will reduce only after a successful matching and will be stuck

otherwise. The encodings are

$$\begin{aligned}
\llbracket [M \text{ is } N]P \rrbracket &\stackrel{\text{def}}{=} (\nu n)(\ulcorner n \urcorner \bullet \llbracket M \rrbracket \rightarrow \llbracket P \rrbracket \mid \ulcorner n \urcorner \bullet \llbracket N \rrbracket \rightarrow \mathbf{0}) \\
\llbracket \text{let } (x, y) = M \text{ in } P \rrbracket &\stackrel{\text{def}}{=} (\nu n)(\ulcorner n \urcorner \bullet (\text{pair} \bullet \lambda x \bullet \lambda y) \rightarrow \llbracket P \rrbracket \\
&\quad \mid \ulcorner n \urcorner \bullet \llbracket M \rrbracket \rightarrow \mathbf{0}) \\
\llbracket \text{case } M \text{ of } \{x\}_N : P \rrbracket &\stackrel{\text{def}}{=} (\nu n)(\ulcorner n \urcorner \bullet (\text{encr} \bullet \lambda x \bullet \llbracket N \rrbracket) \rightarrow \llbracket P \rrbracket \\
&\quad \mid \ulcorner n \urcorner \bullet \llbracket M \rrbracket \rightarrow \mathbf{0}) \\
\llbracket \text{case } M \text{ of } 0 : P \text{ suc}(x) : Q \rrbracket &\stackrel{\text{def}}{=} (\nu n)(\ulcorner n \urcorner \bullet 0 \rightarrow \llbracket P \rrbracket \\
&\quad \mid \ulcorner n \urcorner \bullet (\text{suc} \bullet \lambda x) \rightarrow \llbracket Q \rrbracket \\
&\quad \mid \ulcorner n \urcorner \bullet \llbracket M \rrbracket \rightarrow \mathbf{0}) .
\end{aligned}$$

where we always assume that name n is fresh w.r.t. the following encoded terms and processes.

The “match” $[M \text{ is } N]P$ only reduces to P if $M = N$, thus the encoding creates two protected patterns using $\llbracket M \rrbracket$ and $\llbracket N \rrbracket$ with one of them reducing to $\llbracket P \rrbracket$ and the other to $\mathbf{0}$. The “pair splitting” $\text{let } (x, y) = M \text{ in } P$ encoding creates a case with a pattern that matches a tagged pair and binds the components to x and y in $\llbracket P \rrbracket$. This is put in parallel with another case that has $\llbracket M \rrbracket$ in the pattern and reduces to $\mathbf{0}$. The “decryption case” $\text{case } M \text{ of } \{x\}_N : P$ checks whether M is a message encoded with key $\llbracket N \rrbracket$ and retrieves the value encrypted by binding it to x in the continuation. Lastly the “integer case” $\text{case } M \text{ of } 0 : P \text{ suc}(x) : Q$ translation creates a case for each of the zero and the successor possibilities. These cases match the tag and the reserved names 0, reducing to $\llbracket P \rrbracket$, or suc and binding x in $\llbracket Q \rrbracket$. The term to be compared M is used as the pattern of a third case that reduces to $\mathbf{0}$.

Theorem 6. *The encoding of Spi into CPC is valid.*

Proof: Like for corollary 1, it suffices to prove that

1. If $P \mapsto P'$ then $\llbracket P \rrbracket \mapsto \simeq_2 \llbracket P' \rrbracket$;
2. if $\llbracket P \rrbracket \mapsto Q$ then $Q \simeq_2 \llbracket P' \rrbracket$ for some P' such that $P \mapsto P'$.

where \simeq_2 is any equivalence for CPC that equates $\mathbf{0}$ and any process of the form $(\nu n)\ulcorner n \urcorner \bullet \dots \rightarrow P$. Even though we still have not started the business of defining behavioural equivalences for CPC, we claim that any ‘reasonable’ equivalence for it would work: indeed, thanks to Lemma 2, we know that every process of kind $(\nu n)\ulcorner n \urcorner \bullet \dots \rightarrow P$ cannot interact with any other process, nor reduce; thus, it behaves like $\mathbf{0}$.

The first claim can be easily proved by a straightforward induction on judgment $P \mapsto P'$. The base case is proved by reasoning on the Spi axiom used to infer the reduction. Every case is trivial. Some care must only be spent on (5) and (6); let us consider the latter (the former is simpler). In this case,

$P = \text{case } \text{suc}(M) \text{ of } 0 : P_1 \text{ suc}(x) : P_2$ and $P' = \{M/x\}P_2$. Then,

$$\begin{aligned} \llbracket P \rrbracket &\stackrel{\text{def}}{=} (\nu n)(\ulcorner n \urcorner \bullet (\text{num} \bullet \ulcorner 0 \urcorner) \rightarrow \llbracket P_1 \rrbracket \\ &\quad | \ulcorner n \urcorner \bullet (\text{num} \bullet (\ulcorner \text{suc} \urcorner \bullet \lambda x)) \rightarrow \llbracket P_2 \rrbracket \\ &\quad | \ulcorner n \urcorner \bullet (\text{num} \bullet (\text{suc} \bullet \llbracket M \rrbracket)) \rightarrow \mathbf{0} \text{ .} \end{aligned}$$

and it can only reduce to

$$\{\llbracket M \rrbracket/x\}\llbracket P_2 \rrbracket \mid (\nu n)\ulcorner n \urcorner \bullet (\text{num} \bullet \ulcorner 0 \urcorner) \rightarrow \llbracket P_1 \rrbracket$$

By a straightforward induction on the structure of P_2 it is easy to prove that $\{\llbracket M \rrbracket/x\}\llbracket P_2 \rrbracket = \llbracket \{M/x\}P_2 \rrbracket$. Thus, $\llbracket P \rrbracket \mapsto \llbracket \{M/x\}P_2 \rrbracket \mid (\nu n)\ulcorner n \urcorner \bullet (\text{num} \bullet \ulcorner 0 \urcorner) \rightarrow \llbracket P_1 \rrbracket \simeq_2 \llbracket P' \rrbracket$. The inductive case is straightforward, with the structural case relying on a Lemma that is formally identical (and can be identically proved) to Lemma 4.

The second part can be proved by induction on judgment $\llbracket P \rrbracket \mapsto Q$. There is just one base case, i.e. when $\llbracket P \rrbracket = p \rightarrow Q_1 \mid q \rightarrow Q_2$, $Q = \sigma Q_1 \mid \rho Q_2$ and $\{p\|q\} = \text{Some } (\sigma, \rho)$. By definition of the encoding, it can only be that $p = \text{in} \bullet \llbracket M \rrbracket \bullet \lambda x$, $Q_1 = \llbracket P_1 \rrbracket$, $q = \lambda x \bullet \llbracket M \rrbracket \bullet (\llbracket N \rrbracket)$ and $Q_2 = \llbracket P_2 \rrbracket$, for some P_1, P_2, M and N . This means that $P = M(x).P_1 \mid \overline{M}\langle N \rangle.P_2$ and that $Q = \{\llbracket N \rrbracket/x\}\llbracket P_1 \rrbracket \mid \llbracket P_2 \rrbracket = \llbracket \{N/x\}P_1 \mid P_2 \rrbracket$. To conclude, it suffices to take $P' = \{N/x\}P_1 \mid P_2$. For the inductive case, let us consider two possibilities:

1. The inference of $\llbracket P \rrbracket \mapsto Q$ ends with an application of the rule for parallel composition or for structural equivalence: this case can be proved by a straightforward induction.
2. The inference of $\llbracket P \rrbracket \mapsto Q$ ends with an application of the rule for restriction; thus, $\llbracket P \rrbracket = (\nu n)Q'$, with $Q' \mapsto Q''$ and $Q = (\nu n)Q''$. If $Q' = \llbracket P'' \rrbracket$, for some P'' , we can apply a straightforward induction. Otherwise, we have the following four possibilities:
 - (a) $Q' = \ulcorner n \urcorner \bullet \ulcorner \llbracket M \rrbracket \urcorner \rightarrow \llbracket P_1 \rrbracket \mid \ulcorner n \urcorner \bullet \ulcorner \llbracket N \rrbracket \urcorner \rightarrow \mathbf{0}$ and, hence, $Q'' = \llbracket P_1 \rrbracket$. By definition of the encoding, $P = [M \text{ is } N]P_1$. Notice that the reduction $Q' \mapsto Q''$ can happen only if $\llbracket M \rrbracket$ and $\llbracket N \rrbracket$ match; by construction of the encoding of Spi-terms, this can happen only if $M = N$ and, hence, $P \mapsto P_1$. The thesis follows by letting $P' = P_1$, since n is a fresh name and so $Q = (\nu n)\llbracket P_1 \rrbracket \equiv \llbracket P_1 \rrbracket$.
 - (b) $Q' = \ulcorner n \urcorner \bullet (\ulcorner \text{pair} \urcorner \bullet (\lambda x \bullet \lambda y)) \rightarrow \llbracket P_1 \rrbracket \mid \ulcorner n \urcorner \bullet (\text{pair} \bullet (\llbracket M \rrbracket \bullet \llbracket N \rrbracket)) \rightarrow \mathbf{0}$ and, hence, $Q'' = \{\llbracket M \rrbracket/x, \llbracket N \rrbracket/y\}\llbracket P_1 \rrbracket$. This case is similar to the previous one, by letting P be *let* $(x, y) = (M, N)$ *in* P_1 .
 - (c) $Q' = \ulcorner n \urcorner \bullet (\ulcorner \text{encr} \urcorner \bullet (\lambda x \bullet \llbracket N \rrbracket)) \rightarrow \llbracket P_1 \rrbracket \mid \ulcorner n \urcorner \bullet (\text{encr} \bullet (\llbracket M \rrbracket \bullet \llbracket N \rrbracket)) \rightarrow \mathbf{0}$ and, hence, $Q'' = \{\llbracket M \rrbracket/x\}\llbracket P_1 \rrbracket$. This case is similar to the previous one, by letting P be *case* $\{M\}_N$ *of* $\{x\}_N : P_1$.
 - (d) $Q' = \ulcorner n \urcorner \bullet (\text{num} \bullet \ulcorner 0 \urcorner) \rightarrow \llbracket P_1 \rrbracket \mid \ulcorner n \urcorner \bullet (\text{num} \bullet (\ulcorner \text{suc} \urcorner \bullet \lambda x)) \rightarrow \llbracket P_2 \rrbracket \mid \ulcorner n \urcorner \bullet \llbracket M \rrbracket \rightarrow \mathbf{0}$. Hence, $P = \text{case } M \text{ of } 0 : P_1 \text{ suc}(x) : P_2$. According to the kind of $\llbracket M \rrbracket$, we have two sub-cases (notice that, since $Q' \mapsto Q''$, no other possibility is allowed for $\llbracket M \rrbracket$):

- i. $\llbracket M \rrbracket = \text{num} \bullet 0$: in this case, $Q'' = \llbracket P_1 \rrbracket \mid \ulcorner n \urcorner \bullet (\text{num} \bullet (\ulcorner \text{succ} \urcorner \bullet \lambda x)) \rightarrow \llbracket P_2 \rrbracket$ and so $Q = (\nu n)Q'' \equiv \llbracket P_1 \rrbracket \mid (\nu n)\ulcorner n \urcorner \bullet (\text{num} \bullet (\ulcorner \text{succ} \urcorner \bullet \lambda x)) \rightarrow \llbracket P_2 \rrbracket \simeq_2 \llbracket P_1 \rrbracket$. In this case, $M = 0$ and so $P \mapsto P_1$; to conclude, it suffices to let P' be P_1 .
- ii. $\llbracket M \rrbracket = \text{num} \bullet (\text{succ} \bullet \llbracket M' \rrbracket)$, for some M' : in this case, $Q'' = \{\llbracket M' \rrbracket/x\}\llbracket P_2 \rrbracket \mid \ulcorner n \urcorner \bullet (\text{num} \bullet \ulcorner 0 \urcorner) \rightarrow \llbracket P_1 \rrbracket$ and so $Q = (\nu n)Q'' \equiv \{\llbracket M' \rrbracket/x\}\llbracket P_2 \rrbracket \mid (\nu n)\ulcorner n \urcorner \bullet (\text{num} \bullet 0) \rightarrow \llbracket P_1 \rrbracket \simeq_2 \{\llbracket M' \rrbracket/x\}\llbracket P_2 \rrbracket$. In this case, $M = \text{succ}(M')$ and so $P \mapsto \{M'/x\}P_2$; to conclude, it suffices to let P' be $\{M'/x\}P_2$. □

To conclude, notice that this encoding allows spi calculus to be modelled in CPC. This fact does not entail that cryptography can be properly rendered in CPC. Consider the pattern $\text{encr} \bullet \lambda x \bullet \lambda y$ that could match the encoding of an encrypted term to bind the message and key, so that CPC can break any encryption! One solution is to simply add this encryption to CPC, a topic for future work.

5 Conclusions and future work

The concurrent pattern calculus generalises several of the most popular process calculi by generalising from prefixes to patterns whose unification yields a symmetric exchange of information. This increased expressive power makes it easy to model information exchange, which underpins trade in general. Further, we have proven that it is not captured by other calculi such as π -calculus or Linda.

Future work aims to implement CPC as a vehicle for supporting web services that can exploit structured data as well as interaction. One approach would be to augment the programming language **bondi** [2] which already supports pattern calculus.

Acknowledgements. Thanks to Eugenio Moggi for his helpful comments on a previous draft of this paper.

References

1. H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science Publishers B.V., 1985. BAR h 85:1 1.Ex.
2. bondi programming language. www-staff.it.uts.edu.au/~cbj/bondi.
3. D. Gelernter. Generative communication in LINDA. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
4. A. Gordon and M. Abadi. A calculus for cryptographic protocols: The spi calculus. *4th ACM Conference on Computer and Communications Security*, pages 36 – 47, 1997.
5. D. Gorla. Comparing communication primitives via their relative expressive power. *Information and Computation*, 206(8):931–952, 2008.

6. D. Gorla. Towards a unified approach to encodability and separation results for process calculi. In F. van Breugel and M. Chechik, editors, *Proc. of 19th International Conference on Concurrency Theory (CONCUR'08)*, number 5201 in LNCS, pages 492–507. Springer, 2008.
7. B. Jay. *Pattern Calculus: Computing with Functions and Data Structures*. Springer, 2009.
8. B. Jay and T. Given-Wilson. A combinatory account of internal structure, 2009. <http://www-staff.it.uts.edu.au/~cbj/Publications/factorisation.pdf>.
9. B. Jay and D. Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):34 pages, 2009.
10. R. Milner. The polyadic π -calculus: A tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Series F*. NATO ASI, Springer, 1993.
11. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, 1992.
12. R. D. Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
13. J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proc. of LICS*, pages 176–185. IEEE Computer Society, 1998.
14. G. Picco, A. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conference on Software Engineering (ICSE'99)*, pages 368–377. ACM Press, 1999.
15. L. Wischik and P. Gardner. Explicit fusions. *Theor. Comput. Sci.*, 340(3):606–630, 2005.