Concurrent Pattern Calculus

Thomas Given-Wilson[1], Daniele Gorla[2], and Barry Jay[1]

[1] Centre for Quantum Computation and Intelligent Systems &
School of Software, University of Technology, Sydney
[2] Dip. di Informatica, Univ. di Roma "La Sapienza"

Abstract. Concurrent pattern calculus drives interaction between processes by unifying patterns, just as sequential pattern calculus drives computation by matching a pattern against a data structure. By generalising from pattern matching to 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 calculus.

1 Introduction

The π-calculus [13] 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* [11,9] supports even more computations than λ-calculus since pattern-matching functions are commonly intensional with respect to their arguments [10]. For example, the pattern $x\ y$ can decompose any data structure in (static) pattern calculus by matching against the internal structure. Hence it is natural to wonder what a concurrent pattern calculus might look like. In fact it turns out rather well.

This paper adapts the pattern matching mechanism of the pure pattern calculus [11,9] to a concurrent process language that supports 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 can be combined into patterns; their *unification* triggers a two-way, or symmetric, flow of information, as represented by the sole interaction rule

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

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

Its support for structure and symmetry of interaction makes its pattern matching more expressive than several representative approaches in the literature. For example, checking equality of channel names, as in π-calculus [13], can be viewed as a trivial form of pattern matching. This can be generalised to match tuples of names, as in polyadic π-calculus [12], fusion calculus [15] and Linda [4]. Spi calculus [6] has an even richer collection of patterns, for equality of terms, pairs of terms, numbers (zero and successors) and encryptions.

C.S. Calude and V. Sassone (Eds.): TCS 2010, IFIP AICT 323, pp. 244–258, 2010.
© IFIP International Federation for Information Processing 2010

More formally, π-calculus, Spi calculus and Linda can all be encoded into CPC but CPC cannot be encoded in any of them. Although the patterns of fusion calculus are relatively simple, the peculiarities of name fusion ensure that there are no encodings of fusion calculus into CPC or conversely, of CPC into fusion calculus.

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 (e.g. [14,16]). Really it is a question of deciding how granular one wishes to be. CPC may 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 CPC to other process calculi. Section 5 concludes and considers future work. Most proofs are omitted from this paper but can be found on-line [5].

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 ways. 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 symmetry by fusing names together but has no intensional patterns. On the other hand, Spi calculus has intensional patterns, e.g. for natural numbers, and can check equality of terms (i.e. patterns), 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* and unifying them (symmetrically) rather than matching them (asymmetrically). This supports equality testing and bi-directional input and output in a single step. Although the increased expressive power makes it harder to protect private information, 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 $p, p_1, q, q_1, \ldots$) and the *processes* (meta-variables $P, P', P_1, Q, Q', Q_1 \ldots$).

The patterns have the following forms

$$\begin{array}{llll} \textit{Patterns} & p ::= & x & \text{variable name} \\ & & \ulcorner x \urcorner & \text{protected name} \\ & & \lambda x & \text{binding name} \\ & & p \bullet p & \text{compound.} \end{array}$$

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. A compound combines two patterns into a single one.

Given a pattern p the sets of: *variables names*, denoted $\mathsf{vn}(p)$; *protected names*, denoted $\mathsf{pn}(p)$; and *binding names*, denoted $\mathsf{bn}(p)$, are as expected with the union being taken for compounds. The *free names* of a pattern p, written $\mathsf{fn}(p)$, is the union of the variable names and protected names of p. A pattern is *well formed* if each binding name appears exactly once. All patterns appearing in the rest of the paper are assumed to be well formed.

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.

Protection can be extended to a communicable pattern p by defining

$$\ulcorner x \urcorner = \ulcorner x \urcorner \qquad\qquad \ulcorner p \bullet q \urcorner = \ulcorner p \urcorner \bullet \ulcorner q \urcorner \, .$$

A *substitution* σ (also denoted $\sigma_1, \rho, \rho_1, \ldots$) is defined as a partial function from names to communicable patterns. These are applied to patterns in the obvious manner on the understanding that

$$\sigma \ulcorner x \urcorner = \ulcorner \sigma x \urcorner \quad \text{if } x \text{ is in the domain of } \sigma.$$

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:

$$\begin{array}{rll} \left.\begin{array}{r} \{x \| x\} \\ \{x \| \ulcorner x \urcorner\} \\ \{\ulcorner x \urcorner \| x\} \\ \{\ulcorner x \urcorner \| \ulcorner x \urcorner\} \end{array}\right\} & = \mathsf{Some}\ (\{\}, \{\}) & \\ \{\lambda x \| q\} & = \mathsf{Some}\ (\{q/x\}, \{\}) & \text{if } q \text{ is communicable} \\ \{p \| \lambda x\} & = \mathsf{Some}\ (\{\}, \{p/x\}) & \text{if } p \text{ is communicable} \\ \{p_1 \bullet p_2 \| q_1 \bullet q_2\} & = \mathsf{Some}\ ((\sigma_1 \cup \sigma_2), (\rho_1 \cup \rho_2)) & \left\{\begin{array}{l} \{p_1 \| q_1\} = \mathsf{Some}\ (\sigma_1, \rho_1) \\ \{p_2 \| q_2\} = \mathsf{Some}\ (\sigma_2, \rho_2) \end{array}\right. \\ \{p \| q\} & = \text{undefined} & \text{otherwise.} \end{array}$$

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 the unification of patterns p and q is defined then any protected name of p is a free name of q.*

The processes of CPC are given by

$$\begin{array}{lrll} \textit{Processes} & P & ::= & \mathbf{0} & \text{zero} \\ & & & P|P & \text{parallel composition} \\ & & & !P & \text{replication} \\ & & & (\nu x)P & \text{restriction} \\ & & & p \rightarrow P & \text{case.} \end{array}$$

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 $\mathsf{fn}(P)$, are defined as usual for all the traditional primitives and

$$\mathsf{fn}(p \rightarrow P) \quad = \quad \mathsf{fn}(p) \cup (\mathsf{fn}(P) \backslash \mathsf{bn}(p))$$

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

The general *structural equivalence relation* $\equiv$ is defined just as in π-calculus [12], with α-conversion defined in the usual manner.

The application of a substitution to a process is defined in the usual manner, to avoid name capture.

2.2 Operational Semantics

CPC has one *interaction rule* given by

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

It states that if the unification of two patterns p and q is defined and generates $\mathsf{Some}\ (\sigma, \rho)$, 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 in the usual manner. The reflexive, transitive closure of $\longmapsto$ is denoted $\Longmapsto$. The examples and theorems developed later in the paper rely on control of interaction, as now defined.

Definition 1. *The processes P and Q* do not interact *if, whenever $P|Q \Longmapsto R$, then there are processes P' and Q' such that $P \Longmapsto P'$, $Q \Longmapsto Q'$ and $R \equiv P'|Q'$.*

Lemma 2. *A process of the form $p \to P$ with a protected name n in the pattern can only interact with a process Q containing n among its the free names.*

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 demonstrates discovery, the second introduces a registrar to validate traders, the third extends the second with protected names to ensure privacy.

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

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

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

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

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 place in a parallel composition and discover each other by matching on s

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

The next phase is to exchange billing information for share certificates, as in

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

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 atomically to complete a transaction. However, there is no way to determine if a process is a trustworthy trader.

Solution 2. Now add a registrar that keeps track of registered traders. Traders offer their identity to potential partners and the registrar confirms if the identity belongs to a valid trader. The buyer is now

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

The first pattern now swaps the buyer's identity i_B for the seller's, bound to j. The buyer then consults the registrar using the identifier n_B to validate j, if valid the exchange continues as before.

Now define the seller symmetrically by

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

Also define the registrar Reg_2 with identifiers n_B and n_S to communicate with the buyer and seller, respectively, by

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

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 yields the following interaction

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

The share information s allows the buyer and seller to discover 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

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

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

$$(\nu n)(n \bullet b \bullet \lambda x \to B(x) \mid n \bullet \lambda y \bullet c \to S(y)) \quad \longmapsto \quad (\nu n)(B(c) \mid S(b)) \; .$$

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

$$\mathsf{Prom} = \lambda z_1 \bullet \lambda z_2 \bullet a \to P(z_1, z_2) \; .$$

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 interfere with the traders trying to complete the exchange phase of a transaction. For example,

$$\begin{aligned} &(\nu n)(n \bullet b \bullet \lambda x \to B(x) \mid n \bullet \lambda y \bullet c \to S(y)) \mid \mathsf{Prom} \\ \longmapsto\; &(\nu n)(B(a) \mid n \bullet \lambda y \bullet c \to S(y) \mid P(n, b)) \end{aligned}$$

where the promiscuous process has stolen the identifier n and 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. The vulnerability of Solution 2 can be repaired by using protected names. The buyer, seller and registrar can be repaired to

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

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:

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

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 (Lemma 2).

The solution could be extended further: although the share information is treated as a variable name in the example, it could be represented as a compound structure with a company code, number of shares and price per share, e.g. $\mathsf{ABC} \bullet 100 \bullet \$0.38$. This format allows discovery based on partial share information, for example: specify a company code and price, but not the number of shares $\mathsf{ABC} \bullet \lambda v \bullet \$0.38$; 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 trader 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 $\mathsf{accountnumber}$ and $\mathsf{accountname}$, could be $(\ulcorner\mathsf{accountnumber}\urcorner \bullet \lambda u) \bullet (\ulcorner\mathsf{accountname}\urcorner \bullet \lambda v)$. Thus, any pattern that successfully matches must be identically structured and tagged. Indeed, this could be developed further to account for XML and web services such as in `PiDuce` [3].

4 Comparison with Other Process Calculi

This section exploits the techniques developed in [7,8] to formally asses the expressive power of CPC w.r.t. π-calculus, Linda, Fusion and Spi calculus. After briefly recalling these models and some basic material from [8], the relation to CPC is formalised. First, let each model, including CPC, be augmented with a reserved process '$\surd$', used to signal successful termination.

4.1 Some Process Calculi

π-calculus [13,12]. The π-calculus processes given by the following grammar:

$$P \ ::= \ \mathbf{0} \ \mid \ \surd \ \mid \ \overline{a}\langle b\rangle.P \ \mid \ a(x).P \ \mid \ (\nu n)P \ \mid \ P|Q \ \mid \ !P$$

and the only reduction rule is

$$\overline{a}\langle b\rangle.P \mid a(x).Q \quad \longmapsto \quad P \mid Q\{b/x\} \ .$$

Linda [4]. Consider an instance of Linda formulated to follow CPC's syntax.

Processes are defined as:

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

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

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

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

$$\textsc{Match}(\ ;\ ) = \{\} \qquad \textsc{Match}(\ulcorner b\urcorner; b) = \{\} \qquad \textsc{Match}(\lambda x; b) = \{b/x\}$$

$$\frac{\textsc{Match}(t; b) = \sigma_1 \qquad \textsc{Match}(\widetilde{t}; \widetilde{b}) = \sigma_2}{\textsc{Match}(t, \widetilde{t}\,;\, b, \widetilde{b}) = \sigma_1 \uplus \sigma_2}$$

where $\widetilde{e}$ 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 is given by:

$$\langle \widetilde{b} \rangle \mid (\widetilde{t}).P \longmapsto \sigma P \qquad \text{if } \textsc{Match}(\widetilde{t}; \widetilde{b}) = \sigma \ .$$

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

Fusion [15]. Following the the presentation in [17], processes are defined as:

$$P \ ::= \ \mathbf{0} \ \mid \ P|P \ \mid \ (\nu x)P \ \mid \ !P \ \mid \ \overline{u}\langle \widetilde{x} \rangle.P \ \mid \ u(\widetilde{x}).P \ .$$

The interaction rule for Fusion is

$$(\nu \widetilde{u})(\overline{u}\langle \widetilde{x} \rangle.P \mid u(\widetilde{y}).Q \mid R) \longmapsto \sigma P \mid \sigma Q \mid \sigma R \quad \begin{array}{l} \text{with } \mathsf{dom}(\sigma) \cup \mathsf{ran}(\sigma) \subseteq \{\widetilde{x}, \widetilde{y}\} \\ \text{and } \widetilde{u} = \mathsf{dom}(\sigma) \setminus \mathsf{ran}(\sigma) \text{ and} \\ \sigma(v) = \sigma(w) \text{ iff } (v, w) \in E(\widetilde{x} = \widetilde{y}) \end{array}$$

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

Spi calculus [6]. This language is unusual as names are now generalised to *terms* of the form

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

They are rather similar to the patterns of CPC in that they may have internal structure. Of particular interest are the pair, successor and encryption that may be bound to a name and then decomposed later by an intensional reduction.

Concerning the operational semantics, we consider a slightly modified version of Spi calculus where interaction is generalised to

$$\overline{M}\langle N \rangle.P \mid M(x).Q \quad \longmapsto \quad P \mid \{N/x\}Q$$

where M is any term of the Spi calculus.

4.2 Valid Encodings and Their Properties

An *encoding* of a language $\mathcal{L}_1$ into another language $\mathcal{L}_2$ is a pair $(\llbracket \cdot \rrbracket, \varphi_{\llbracket \rrbracket})$ where $\llbracket \cdot \rrbracket$ translates every $\mathcal{L}_1$-process into an $\mathcal{L}_2$-process and $\varphi_{\llbracket \rrbracket}$ 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, the translation may fix some names to play a precise rôle or it may translate a single name into a tuple of names. This can be obtained by exploiting $\varphi_{\llbracket \rrbracket}$ (details in [8]).

Now consider only encodings that satisfy the following properties, that are justified and discussed at length in [8]. Let a *k-ary context* $\mathcal{C}(\__1; \ldots; \__k)$ be a

term where k occurrences of $\mathbf{0}$ are linearly replaced by the holes $\{\__1; \ldots; \__k\}$ (every hole must occur once and only once). Moreover, denote with $\longmapsto_i$ and $\Longmapsto_i$ the relations $\longmapsto$ and $\Longmapsto$ in language $\mathcal{L}_i$; denote with $\longmapsto_i^\omega$ 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, let $P \Downarrow_i$ mean that there exists P' such that $P \Longmapsto_i P'$ and $P' \equiv P'' \mid \surd$, for some P''. Finally, to simplify reading, 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* $(\llbracket \cdot \rrbracket, \varphi_{\llbracket\rrbracket})$ *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* $\mathcal{C}^N_{\mathsf{op}}(\__1; \ldots; \__k)$ *such that, for all $S_1, \ldots, S_k$ with* $\mathsf{fn}(S_1, \ldots, S_k) = N$, *it holds that* $\llbracket \mathsf{op}(S_1, \ldots, S_k) \rrbracket = \mathcal{C}^N_{\mathsf{op}}(\llbracket S_1 \rrbracket; \ldots; \llbracket S_k \rrbracket)$.
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 $\varphi_{\llbracket\rrbracket}(\sigma(a)) = \sigma'(\varphi_{\llbracket\rrbracket}(a))$ for every name a.
3. Operational correspondence:
   - *for all $S \Longmapsto_1 S'$, it holds that $\llbracket S \rrbracket \Longmapsto_2 \simeq_2 \llbracket S' \rrbracket$;*
   - *for all $\llbracket S \rrbracket \Longmapsto_2 T$, there exists S' such that $S \Longmapsto_1 S'$ and $T \Longmapsto_2 \simeq_2 \llbracket S' \rrbracket$.*
4. Divergence reflection: *for every S such that $\llbracket S \rrbracket \longmapsto_2^\omega$, it holds that $S \longmapsto_1^\omega$.*
5. Success sensitiveness: *for every S, it holds that $S \Downarrow_1$ if and only if $\llbracket S \rrbracket \Downarrow_2$.*

[8] 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 $\mathcal{L}_1$ and $\mathcal{L}_2$ satisfying certain conditions. Here, these languages will be limited to CPC and those introduced in Section 4.1. Further, the valid encodings considered will be assumed to be *semi-homomorphic*, i.e. where the interpretation of parallel composition is via a context of the form $(\nu \widetilde{n})(\__1 \mid \__2 \mid R)$, for some $\widetilde{n}$ and R that only depend on the free names of the translated processes.

Proposition 1 (from [8]). *Let $\llbracket \cdot \rrbracket$ be a valid encoding; then, $S \not\longmapsto_1$ implies that $\llbracket S \rrbracket \not\longmapsto_2$.*

Theorem 1 (from [8]). *Assume that there exists S such that $S \not\longmapsto_1$, $S \not\Downarrow_1$ and $S \mid S \Downarrow_1$; moreover, assume that every T that does not reduce is such that $T \mid T \not\longmapsto_2$. Then, there cannot exist any semi-homomorphic valid encoding of $\mathcal{L}_1$ into $\mathcal{L}_2$.*

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

Theorem 2 (from [8]). *If $\textsc{Md}(\mathcal{L}_1) > \textsc{Md}(\mathcal{L}_2)$, then there exists no valid encoding of $\mathcal{L}_1$ into $\mathcal{L}_2$.*

4.3 CPC vs. π-Calculus and Linda

A hierarchy of process calculi with different communication primitives is obtained in [7] by combining four features: synchronism (synchronous vs asynchronous), arity (monadic vs polyadic data exchange), communication medium (channels vs shared dataspaces), and the presence of a form of pattern matching (that checks the arity of the tuple of names and equality of some specific names). This hierarchy is built upon a very similar notion of encoding to that presented in Definition 2 and, in particular, it is proved that Linda [4] (called $L_{\text{A,P,D,PM}}$ in [7]) is more expressive than monadic/polyadic π-calculus [13,12] (called $L_{\text{S,M,C,NO}}$ and $L_{\text{S,P,C,NO}}$, respectively, in [7]). Thus, it suffices to show that CPC is more expressive than $L_{\text{A,P,D,PM}}$ (this is the language called Linda in Section 4.1).

First notice that CPC cannot be encoded into $L_{\text{A,P,D,PM}}$: this is a corollary of Theorem 1. Indeed, consider the self-matching CPC process $x \to \surd$: alone it cannot reduce and cannot report success but, reports success in parallel with itself. On the contrary, it is easy to prove that every $L_{\text{A,P,D,PM}}$-process that reduces if put in parallel with itself is such that it reduces in isolation.

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

$$[\![\langle \widetilde{b} \rangle]\!] \stackrel{\text{def}}{=} \mathsf{pat\text{–}d}(\widetilde{b}) \to \mathbf{0} \qquad [\![(\widetilde{t}).P]\!] \stackrel{\text{def}}{=} \mathsf{pat\text{–}t}(\widetilde{t}) \to [\![P]\!]$$

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

$$\mathsf{pat\text{–}d}(\,) \stackrel{\text{def}}{=} \lambda x \qquad \mathsf{pat\text{–}d}(b,\widetilde{b}) \stackrel{\text{def}}{=} \lambda x \bullet b \bullet \mathsf{pat\text{–}d}(\widetilde{b}) \qquad \text{for } x \notin \mathsf{bn}(\mathsf{pat\text{–}d}(\widetilde{b}))$$
$$\mathsf{pat\text{–}t}(\,) \stackrel{\text{def}}{=} \mathsf{in} \qquad \mathsf{pat\text{–}t}(t,\widetilde{t}) \stackrel{\text{def}}{=} \mathsf{in} \bullet t \bullet \mathsf{pat\text{–}t}(\widetilde{t})$$

where in is any name (a symbolic name is used for clarity but no result relies upon this). Moreover, the function $\mathsf{pat\text{–}d}(\cdot)$ associates a bound variable to every name in the sequence; this fact ensures that a pattern that translates a datum and a pattern that translates a template match only if they have the same length (this is a feature of $L_{\text{A,P,D,PM}}$'s pattern matching but not of CPC's). It is worth noting that the simpler translation $[\![\langle b_1,\ldots,b_n \rangle]\!] \stackrel{\text{def}}{=} b_1 \bullet \ldots \bullet b_n \to \mathbf{0}$ would not work: the $L_{\text{A,P,D,PM}}$-process $\langle b \rangle \mid \langle b \rangle$ does not reduce, whereas such an encoding $(b \to \mathbf{0} \mid b \to \mathbf{0})$ does. This fact would contradict Proposition 1.

Next is to prove that this encoding is valid. This is an easy corollary of the following lemma, stating a strict correspondence between $L_{\text{A,P,D,PM}}$'s pattern matching and CPC's one (on patterns arising from the translation).

Lemma 3. $\textsc{Match}(\widetilde{t};\widetilde{b}) = \sigma$ *if and only if* $\{\mathsf{pat\text{–}t}(\widetilde{t}) \| \mathsf{pat\text{–}d}(\widetilde{b})\} = \mathsf{Some}(\sigma, \{\mathsf{in}/x_0,\ldots,\mathsf{in}/x_n\})$, *where* $\{x_0,\ldots,x_n\} = \mathsf{bn}(\mathsf{pat\text{–}d}(\widetilde{b}))$ *and* σ *maps names to names.*

4.4 CPC vs. Fusion

Fusion calculus and CPC are unrelated in that there exists no valid encoding from one into the other. The impossibility for a valid encoding of CPC into Fusion is ensured by Theorem 2: 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 any number of name equalities can be checked atomically in a single CPC interaction. The converse separation result is ensured by the following theorem.

Theorem 3. *There exists no valid encoding of Fusion into CPC.*

Proof:(Sketch) The idea is to show that any interaction in Fusion can be rendered only by having: (1) two parallel processes performing an input and an output on the same channel, and (2) a restriction enclosing them to allow application of name fusions. Thus, to yield a reduction three entities have to mutually cooperate; this ternary interaction cannot be rendered in CPC, and this can be used to prove that no valid encoding can exist (see the technical report [5] for full details). □

4.5 CPC vs. Spi

That CPC cannot be encoded into Spi calculus is a corollary of Theorem 1 and identical to the technique used in Section 4.3. The self-matching CPC process $x \to \surd$ cannot be encoded into Spi.

The remainder of this section develops an encoding of Spi calculus into CPC. The terms can be encoded as patterns using the reserved names pair, encr, 0 and suc by

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

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.

The processes of the Spi calculus are

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

The nil process, parallel composition, replication and restriction are all familiar. The input $M(x).P$ and output $M\langle N\rangle.P$ are generalised to allow terms in the place of channel names and output arguments. The match $[M \ is \ N]P$ determines equality of M and N. The splitting $let\ (x,y) = M \ in \ P$ decomposes pairs. The decryption case $case\ M\ of\ \{x\}_N : P$ decrypts M binding the encrypted message

to x. The integer case *case* M *of* $0 : P$ *suc*$(x) : Q$ branches according to the number. Note that the last four can all get stuck if M is an incompatible term. Further, the last three are intensional, i.e. they depend on the internal structure of M.

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

$$\begin{aligned}
&[\![M(x).P]\!] \stackrel{\text{def}}{=} \mathsf{in} \bullet \ulcorner [\![M]\!] \urcorner \bullet \lambda x \to [\![P]\!] \\
&[\![\overline{M}\langle N \rangle.P]\!] \stackrel{\text{def}}{=} \lambda x \bullet \ulcorner [\![M]\!] \urcorner \bullet ([\![N]\!]) \to [\![P]\!] \quad x \notin \mathsf{fn}([\![P]\!], [\![M]\!], [\![N]\!]) \,.
\end{aligned}$$

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 $[\![M]\!]$ 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. As in the Spi calculus, the encodings will reduce only after a successful matching and will be stuck otherwise. The encodings are

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

The match $[M \ \mathit{is}\ N]P$ only reduces to P if $M = N$, thus the encoding creates two patterns using $[\![M]\!]$ and $[\![N]\!]$ with one reducing to $[\![P]\!]$. The pair splitting $\mathit{let}\ (x,y) = M\ \mathit{in}\ P$ encoding creates a case with a pattern that matches a tagged pair and binds the components to x and y in $[\![P]\!]$. This is put in parallel with another case that has $[\![M]\!]$ in the pattern. The decryption case $\mathit{case}\ M\ \mathit{of}\ \{x\}_N : P$ checks whether M is a message encoded with key $[\![N]\!]$ and retrieves the value encrypted by binding it to x in the continuation. Lastly the integer case $\mathit{case}\ M\ \mathit{of}\ 0 : P\ \mathit{suc}(x) : Q$ 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 $[\![P]\!]$, or suc and binding x in $[\![Q]\!]$. The term to be compared M is as in the others.

Theorem 4. *The encoding of Spi calculus into CPC is valid.*

To conclude, notice that the criteria for a valid encoding does not imply full abstraction of the encoding (actually, they were defined in [7,8] as an alternative

to full abstraction). This means that the encoding of equivalent Spi calculus processes can be distinguished by contexts in CPC that do not result from the encoding of any Spi calculus context. Indeed, while this encoding allows Spi calculus to be modelled in CPC, it does *not* entail that cryptography can be properly rendered. Consider the pattern $\mathsf{encr} \bullet \lambda x \bullet \lambda y$ 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 uses patterns to represent input, output and tests for equality, whose interaction is driven by unification that allows a two-way flow of information. This symmetric information exchange provides a concise model of trade in the information age. This is illustrated by the example of traders who can discover each other in the open and then close the deal in private.

CPC supports valid encodings of many popular concurrent calculi such as π-calculus, Spi calculus and Linda as its patterns describe more structures. However, these three calculi do not support valid encodings of CPC because, among other things, they are insufficiently symmetric. On the other hand, while fusion calculus is completely symmetric, it has an incompatible approach to interaction.

Future work may proceed in several directions. Just as pattern calculus expands upon the expressive power of sequential programming, CPC expands the expressive power of concurrent programming. The consequences of this remain to be developed. Possibilities applications include web services based upon symmetric information exchange. As first step is to implement the calculus, perhaps by augmenting the programming language **bondi** [2] that was built to implement pattern calculus.

Concurrent pattern calculus supports a generous class of patterns whose interaction is fully symmetric. The implications of this increased expressive power are worthy of further investigation.

Acknowledgements. Thanks to Eugenio Moggi and the reviewers for their helpful comments on drafts of this paper.

References

1. Barendregt, H.P.: The Lambda Calculus. Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics. Elsevier Science Publishers B.V., Amsterdam (1985)
2. bondi programming language, `http://www-staff.it.uts.edu.au/~cbj/bondi`
3. Brown, A.L., Laneve, C., Meredith, L.G.: Piduce: A process calculus with native XML datatypes. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) EPEW/WS-EM 2005. LNCS, vol. 3670, pp. 18–34. Springer, Heidelberg (2005)

4. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems 7(1), 80–112 (1985)
5. Given-Wilson, T., Gorla, D., Jay, B.: Concurrent pattern calculus, long version (2010), http://www.progsoc.uts.edu.au/~sanguinev/files/cpc-long.pdf
6. Gordon, A., Abadi, M.: A calculus for cryptographic protocols: The spi calculus. In: 4th ACM Conference on Computer and Communications Security, pp. 36–47 (1997)
7. Gorla, D.: Comparing communication primitives via their relative expressive power. Information and Computation 206(8), 931–952 (2008)
8. Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 492–507. Springer, Heidelberg (2008)
9. Jay, B.: Pattern Calculus: Computing with Functions and Data Structures. Springer, Heidelberg (2009)
10. Jay, B., Given-Wilson, T.: A combinatory account of internal structure (2010), http://www-staff.it.uts.edu.au/~cbj/Publications/factorisation.pdf
11. Jay, B., Kesner, D.: First-class patterns. Journal of Functional Programming 19(2), 34pages (2009)
12. Milner, R.: The polyadic π-calculus: A tutorial. In: Bauer, F.L., Brauer, W., Schwichtenberg, H. (eds.) Logic and Algebra of Specification. NATO ASI., vol. 94. Springer, Heidelberg (1993)
13. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I/II. Information and Computation 100, 1–77 (1992)
14. Nicola, R.D., Ferrari, G., Pugliese, R.: KLAIM: A kernel language for agents interaction and mobility. IEEE Transactions on Software Engineering 24(5), 315–330 (1998)
15. Parrow, J., Victor, B.: The fusion calculus: Expressiveness and symmetry in mobile processes. In: Proc. of LICS, pp. 176–185. IEEE Computer Society, Los Alamitos (1998)
16. Picco, G., Murphy, A., Roman, G.-C.: Lime: Linda Meets Mobility. In: Garlan, D. (ed.) Proc. of the 21st Int. Conference on Software Engineering (ICSE'99), pp. 368–377. ACM Press, New York (1999)
17. Wischik, L., Gardner, P.: Explicit fusions. Theor. Comput. Sci. 340(3), 606–630 (2005)