On Distributability in Process Calculi*

Kirstin Peters[1], Uwe Nestmann[1], and Ursula Goltz[2]

[1] TU Berlin, Germany
[2] TU Braunschweig, Germany

Abstract. We present a novel approach to compare process calculi and their synchronisation mechanisms by using synchronisation patterns and explicitly considering the degree of distributability. For this, we propose a new quality criterion that (1) measures the preservation of distributability and (2) allows us to derive two synchronisation patterns that separate several variants of pi-like calculi. Precisely, we prove that there is no good and distributability-preserving encoding from the synchronous pi-calculus with mixed choice into its fragment with only separate choice, and neither from the asynchronous pi-calculus (without choice) into the join-calculus.

1 Introduction

The pi-calculus is a well-known and frequently used process calculus to model concurrent systems. Therein, intuitively, the *degree of distributability* corresponds to the amount of parallel components that can act independently. Practical experience has shown that it is not possible to implement every pi-calculus term—not even every asynchronous one—in an asynchronous setting while preserving its degree of distributability. To overcome these problems, the join-calculus was introduced as a model of distributed computation [12]. It employs a *locality* principle by ensuring that there is always exactly one immobile receiver for each communication channel. More precisely, for every name, exactly one receiver is defined at the time of the name's creation, and communication occurs only on so-defined channels [7].

Most of the existing approaches that analyse the distributability of concurrent systems use special formalisms often equipped with an explicit notion of location, e.g. [2] in Petri nets or the distributed pi-calculus [9]. In contrast to these approaches, we analyse (similarly to [17,25]) the potential of a formalism to describe distributed systems without an explicit allocation of locations to processes. Instead, we abstract from a particular distribution and consider *distributability* and, thus, all possible explicitly-located variants of a calculus. We do so, because we consider the expressive power of languages, not just individual terms. Moreover, we obtain results for a larger number of process calculi.

In order to measure whether an encoding respects the degree of distribution, usually the homomorphic translation of the parallel operator, i.e., $[\![P \mid Q]\!] =$

* Supported by the DFG (German Research Foundation), grants NE-1505/2-1 and GO-671/6-1.

M. Felleisen and P. Gardner (Eds.): ESOP 2013, LNCS 7792, pp. 310–329, 2013.

$[\![P]\!] \mid [\![Q]\!]$, is used as a criterion (see e.g. [17,5,11]). Such an encoding naturally preserves the parallel structure of terms and, thus (at least for process calculi such as CSP or the pi-calculus), the degree of distribution. However, the opposite is not true. In [19], the first two authors present an encoding that preserves the degree of distribution although it does not translate the parallel operator homomorphically. In this sense, the homomorphic translation of the parallel operator is too strict—at least for separation results. It rightly forbids the introduction of coordinators that reduce the degree of distribution. But it also forbids protocols that handle communications of parallel components without sequentialising them or reducing the degree of distribution in another sense. Moreover, the homomorphic translation of the parallel operator is not always suited to reason about distribution in process calculi as, for example, the join-calculus: there, it is not always possible to separate distributable subterms by means of a parallel operator (see the discussion in Section 3). To overcome this problem, [19] presents a first formulation of a new criterion to more succinctly measure the preservation of distributability in process calculi like the pi-calculus. We generalise this criterion to reason about arbitrary process calculi. Moreover, we show that the distributability of processes implies also distributability of executions. This leads to a new proof method for separation results.

As a result, we obtain a difference between the distributability of the asynchronous pi-calculus (π_{a}) and the join-calculus (J), elucidated by the non-existence of a good and distributability-preserving encoding from π_{a} into J. Interestingly, the difference between these two calculi is captured by a synchronisation pattern that was already used in [25] when studying the distributability of Petri nets. Moreover, we shed more light on the difference between the synchronous pi-calculus with mixed choice (π_{m}) and its fragment with only separate choice (π_{s}) already considered in [17,8,19] by capturing this difference within a novel synchronisation pattern. Hence, these calculi, although they all have the same abstract expressive power [7,16,19], embody different levels of synchronisation.

Overview. We start with some general definitions on process calculi in §2. In §3, we propose a new criterion to reason about the preservation of distributability. §4 then introduces the first synchronisation pattern and separates π_{a} and J. A second synchronisation pattern and separation between π_{m} and π_{s} is presented in §5. We conclude with §6. Proofs and additional material can be found in [20].

2 Process Calculi

Within this paper we compare different variants of the pi-calculus and the join-calculus as they are described e.g. in [14,13] and [7], respectively. We provide a short introduction into process calculi in general and these variants in particular.

Assume a countably-infinite set $\mathcal{N}$, whose elements are called *names*. We use lower case letters $a, b, c, \ldots, a', a_1, \ldots$ to range over names. Moreover, let $\tau \notin \mathcal{N}$ and $\overline{\mathcal{N}} = \{ \overline{n} \mid n \in \mathcal{N} \}$ be the set of co-names (used in the pi-calculus). A *process calculus* is a language $\mathcal{L} = \langle \mathcal{P}, \longmapsto \rangle$ that consists of a set of process terms $\mathcal{P}$ (its syntax) and a relation $\longmapsto : \mathcal{P} \times \mathcal{P}$ on process terms (its semantics). We often

refer to process terms also simply as processes or as terms and use upper case letters $P, Q, R, \ldots, P', P_1, \ldots$ to range over them.

The *syntax* is usually defined by a context-free grammar defining operators, i.e., functions $\mathrm{op} : \mathcal{N}^n \times \mathcal{P}^m \to \mathcal{P}$. An operator of arity 0, i.e., $m = 0$, is a *constant*. The arguments that are again process terms are called *subterms* of P.

Definition 1 (Subterms). *Let $\langle \mathcal{P}, \longmapsto \rangle$ be a process calculus and $P \in \mathcal{P}$. The set of* subterms *of $P = \mathrm{op}(x_1, \ldots, x_n, P_1, \ldots, P_m)$ is defined recursively as $\{ P \} \cup \{ P' \mid \exists i \in \{ 1, \ldots, m \} \,.\, P'$ is a subterm of $P_i \}$.*

Hence every term is a subterm of itself and constants have no further subterms. We require that each process calculus defines at least the *empty process* as constant and the *parallel operator* as binary operator. Moreover, we add the special constant $\checkmark$ to each process calculus. Its purpose is to denote *success* (or *successful termination*) which allows us to compare the abstract behaviour of terms in different process calculi as described in Section 2.1. Another typical operator is the restriction of scopes of names. A *scope* defines an area in which a particular name is known and can be used. For several reasons, it can be useful to restrict the scope of a name. For instance to forbid interaction between two processes or with an unknown and, hence, potentially untrusted environment. Names whose scope is restricted such that they cannot be used from outside the scope are denoted as *bound names*. The remaining names are called *free names*. Accordingly, we assume three sets—the sets of names $\mathsf{n}(P)$ and its subsets of free names $\mathsf{fn}(P)$ and bound names $\mathsf{bn}(P)$—with each term P. In the case of bound names, their syntactical representation as lower case letters serves as a place holder for any fresh name, i.e., any name that does not occur elsewhere in the term. To avoid name capture or clashes, i.e., to avoid confusion between free and bound names or different bound names, bound names can be mapped to fresh names by *α-conversion*. We write $P \equiv_\alpha Q$ if P and Q differ only by α-conversion.

We use σ, σ', σ_1, ... to range over substitutions. A substitution is a finite mapping from names to names defined by a set $\{ {}^{y_1}/_{x_1}, \ldots, {}^{y_n}/_{x_n} \}$ of renamings, where the $x_1, \ldots, x_n$ are pairwise distinct. The application of a substitution on a term $\{ {}^{y_1}/_{x_1}, \ldots, {}^{y_n}/_{x_n} \}(P)$ is defined as the result of simultaneously replacing all free occurrences of x_i by y_i for $i \in \{ 1, \ldots, n \}$, possibly applying α-conversion to avoid capture or name clashes. For all names $\mathcal{N} \setminus \{ x_1, \ldots, x_n \}$ the substitution behaves as the identity mapping. We sometimes omit the parentheses, i.e., $\sigma(P) = \sigma P$. We naturally extend substitutions to co-names, i.e., $\forall \overline{n} \in \overline{\mathcal{N}} \,.\, \sigma(\overline{n}) = \overline{\sigma(n)}$ for all substitutions σ.

To reason about environments of process terms, we use functions on process terms called contexts. More precisely, a *context* $\mathcal{C}([\cdot]_1, \ldots, [\cdot]_n) : \mathcal{P}^n \to \mathcal{P}$ with n holes is a function from n process terms into a process term, i.e., given $P_1, \ldots, P_n \in \mathcal{P}$, the term $\mathcal{C}(P_1, \ldots, P_n)$ is the result of inserting $P_1, \ldots, P_n$ in that order into the n holes of $\mathcal{C}$.

We consider three variants of the pi-calculus—the full pi-calculus π_m including mixed choice, its subcalculus π_s with only separate choice, and the asynchronous pi-calculus π_a—, and the join-calculus J. Their process terms are given by the sets $\mathcal{P}_\mathrm{m}$, $\mathcal{P}_\mathrm{s}$, $\mathcal{P}_\mathrm{a}$, and $\mathcal{P}_\mathrm{J}$, respectively.

Definition 2 (Syntax). *The sets of process terms are given by*

$$\begin{aligned}
\mathcal{P}_{\mathrm{m}} &::= P_1 \mid P_2 \;\mid\; \checkmark \;\mid\; (\nu n)\, P \;\mid\; !P \;\mid\; \textstyle\sum_{i\in I} \pi_i.P_i \\
&\quad\; \pi ::= \overline{y}\langle z\rangle \;\mid\; y(x) \;\mid\; \tau \\
\mathcal{P}_{\mathrm{s}} &::= P_1 \mid P_2 \;\mid\; \checkmark \;\mid\; (\nu n)\, P \;\mid\; !P \;\mid\; \textstyle\sum_{i\in I} \pi_i^O.P_i \;\mid\; \sum_{i\in I} \pi_i^I.P_i \\
&\quad\; \pi^O ::= \overline{y}\langle z\rangle \;\mid\; \tau \quad \textit{and} \quad \pi^I ::= y(x) \;\mid\; \tau \\
\mathcal{P}_{\mathrm{a}} &::= \mathsf{0} \;\mid\; P_1 \mid P_2 \;\mid\; \checkmark \;\mid\; (\nu n)\, P \;\mid\; !P \;\mid\; \overline{y}\langle z\rangle \;\mid\; y(x).P \;\mid\; \tau.P \\
\mathcal{P}_{\mathrm{J}} &::= \mathsf{0} \;\mid\; P_1 \mid P_2 \;\mid\; \checkmark \;\mid\; y\,\langle z\rangle \;\mid\; \mathsf{def}\ D\ \mathsf{in}\ P \\
&\quad\; J ::= y\,(x) \;\mid\; J_1 \mid J_2 \quad \textit{and} \quad D ::= J \triangleright P \;\mid\; D_1 \wedge D_2
\end{aligned}$$

for some names $n, x, y, z \in \mathcal{N}$ *and a finite index set* I.

The interpretation of the defined terms is as usual. In all languages the *empty process* is denoted by $\mathsf{0}$ and $P_1 \mid P_2$ defines *parallel composition*. Within the pi-calculi *restriction* $(\nu n)\, P$ restricts the scope of the name n to the definition of P and $!P$ denotes *replication*. The process term $\sum_{i\in I} \pi_i.P_i$ represents *finite guarded choice*; as usual, the sum $\sum_{i\in\{1,\ldots,n\}} \pi_i.P_i$ is sometimes written as $\pi_1.P_1 + \ldots + \pi_n.P_n$ and $\mathsf{0}$ abbreviates the empty sum, i.e., where $I = \emptyset$. The input prefix $y(x)$ is used to describe the ability of receiving the value x over link y and, analogously, the output prefix $\overline{y}\langle z\rangle$ describes the ability to send a value z over link y. The prefix τ describes the ability to perform an internal, not observable action. The choice operators of π_{m} and π_{s} require that all branches of a choice are guarded by one of these prefixes. We omit the match prefix, because it does not influence the results.

In $\mathcal{P}_{\mathrm{s}}$ within a single choice term either there are no input or no output guards, i.e., we have input- and output-guarded choice, but no mixed choice. Apart from that, $\mathcal{P}_{\mathrm{m}}$ and $\mathcal{P}_{\mathrm{s}}$ define the same processes. π_{m} and π_{s} represent synchronous variants of the pi-calculus. Asynchronous variants were introduced independently by [10] and [3]. In asynchronous communication, a process has no chance to directly determine (without a hint by another process) whether a value sent by it was already received or not. Hence, output actions are not allowed to guard a process different from $\mathsf{0}$. Also, the interpretation of output guards within a choice construct is delicate. We use the standard variant of π_{a}, where choice is not allowed at all. Since $\mathcal{P}_{\mathrm{a}}$ has no choice, we include $\mathsf{0}$ as a primitive.

In $\mathcal{P}_{\mathrm{J}}$ the operator $y\,\langle z\rangle$ describes an output prefix similar to $\mathcal{P}_{\mathrm{a}}$. A *definition* $\mathsf{def}\ D\ \mathsf{in}\ P$ defines a new receiver on fresh names, where D consists of one or several elementary definitions $J \triangleright P$ connected by $\wedge$, J potentially joins several reception patterns $y\,(x)$ connected by $|$, and P is a process. Note that $\mathsf{def}\ D\ \mathsf{in}\ P$ unifies the concepts of restriction, input prefix, and replication of the pi-calculus. Moreover, [7] define the *core join-calculus* cJ as a subcalculus of J that restricts definitions to the form $\mathsf{def}\ y_1\,(x_1) \mid y_2\,(x_2) \triangleright P_1\ \mathsf{in}\ P_2$, i.e., in cJ definitions consist of a single elementary definition of exactly two reception patterns.

As usual, the continuation $\mathsf{0}$ is often omitted, so e.g. $y(x).\mathsf{0}$ becomes $y(x)$. In addition, for simplicity in the presentation of examples, we sometimes omit an action's object when it does not effectively contribute to the behaviour of a term, e.g. $y(x)\,.\mathsf{0}$ is written as $y.\mathsf{0}$ or just y, and $\mathsf{def}\ y\,(x) \triangleright \mathsf{0}\ \mathsf{in}\ y\,\langle z\rangle$ is abbreviated as $\mathsf{def}\ y \triangleright \mathsf{0}\ \mathsf{in}\ y$. Moreover, let $(\nu\tilde{x})\, P$ abbreviate the term $(\nu x_1) \ldots (\nu x_n)\, P$.

The definitions of free and bound names are completely standard, i.e., names are bound by restriction and as parameter of input and $\mathsf{n}(P) = \mathsf{fn}(P) \cup \mathsf{bn}(P)$ for all P. In the join-calculus the definition $\mathsf{def}\, D \,\mathsf{in}\, P$ binds for all elementary definitions $J_i \triangleright P_i$ in D and all join pattern $y_{i,j}\,(x_{i,j})$ in J_i the *received variables* $x_{i,j}$ in the corresponding P_i and the *defined variables* $y_{i,j}$ in P. By convention, the received variables of composed join patterns have to be pairwise distinct.

To compare process terms, process calculi usually come with different well-studied equivalence relations (see [23] for an overview). A special kind of equivalence with great importance to reason about processes are *congruences*, i.e., the closure of an equivalence with respect to contexts. Process calculi usually come with a special congruence $\equiv \subseteq \mathcal{P} \times \mathcal{P}$ called *structural congruence*. Its main purpose is to equate syntactically different process terms that model quasi-identical behaviour. In the pi-calculus structural congruence is usually provided by a set of equivalence equations. For the above variants we have:
$P \equiv Q$ if $P \equiv_\alpha Q$ $\quad P \mid \mathsf{0} \equiv P$ $\quad P \mid Q \equiv Q \mid P$ $\quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$ $\quad !P \equiv P \mid !P$
$(\nu n)\,\mathsf{0} \equiv \mathsf{0}$ $\quad (\nu n)\,(\nu m)\,P \equiv (\nu m)\,(\nu n)\,P$ $\quad P \mid (\nu n)\,Q \equiv (\nu n)\,(P \mid Q)$ if $n \notin \mathsf{fn}(P)$
The entanglement of input prefix and restriction within the definition operator of the join-calculus limits the flexibility of relations defined by sets of equivalence equations. Instead structural congruence is given by an extension of the chemical approach in [1] by the heating and cooling rules. They operate on so-called solutions $\mathcal{R} \vdash \mathcal{M}$, where $\mathcal{R}$ and $\mathcal{M}$ are multisets. We have (1) $\vdash P \mid Q \rightleftharpoons \vdash P, Q$, (2) $D \wedge E \vdash \rightleftharpoons D, E \vdash$, and (3) $\vdash \mathsf{def}\, D \,\mathsf{in}\, P \rightleftharpoons \sigma_{dv}(D) \vdash \sigma_{dv}(P)$, where only elements—separated by commas—that participate in the rule are mentioned and σ_{dv} instantiates the defined variables in D to distinct fresh names. Then $P \equiv Q$ if P and Q differ only by applications of the $\rightleftharpoons$-rules, i.e., if $\vdash P \rightleftharpoons \vdash Q$.

We assume that the *semantics* is given as an *operational semantics* consisting of inference rules defined on the operators of the language [22]. For many process calculi, the semantics is provided in two forms, as *reduction semantics* and as *labelled semantics*. We assume that at least the reduction semantics $\longmapsto$ is given as part of the definition, because its treatment is easier in the context of encodings. A single application of the reduction semantics is called a *(reduction) step* and is written as $P \longmapsto P'$. If $P \longmapsto P'$ we say P' is a *derivative* of P. Moreover, let $P \longmapsto$ (or $P \not\longmapsto$) denote the existence (absence) of a step from P, i.e., $P \longmapsto \triangleq \exists P' \in \mathcal{P} \,.\, P \longmapsto P'$ and $P \not\longmapsto \triangleq \neg(P \longmapsto)$, and let $\Longmapsto$ denote the reflexive and transitive closure of $\longmapsto$. A sequence of reduction steps is called a *reduction*. We write $P \longmapsto^\omega$ if P has an infinite sequence of steps. We also use *execution* to refer to a reduction starting from a particular term. A *maximal execution* of a process P is a reduction starting from P that cannot be further extended, i.e., that is either infinite or of the form $P \Longmapsto P' \not\longmapsto$.

The semantics of the above variants of the pi-calculus is given by the axioms
$(\ldots + \tau.P + \ldots) \longmapsto P \qquad (\ldots + y(x)\,.P + \ldots) \mid (\ldots + \overline{y}\langle z\rangle\,.Q + \ldots) \longmapsto \{\,{}^{z}/_{x}\,\}\,P \mid Q$
for π_{m} and π_{s}, the axioms

$$\tau.P \longmapsto P \qquad y(x)\,.P \mid \overline{y}\langle z\rangle \longmapsto \{\,{}^{z}/_{x}\,\}\,P$$

for π_{a}, and the three rules

$$\frac{P \longmapsto P'}{P \mid Q \longmapsto P' \mid Q} \qquad \frac{P \longmapsto P'}{(\nu n)\,P \longmapsto (\nu n)\,P'} \qquad \frac{P \equiv Q \qquad Q \longmapsto Q' \quad Q' \equiv P'}{P \longmapsto P'}$$

that hold for all three variants π_{m}, π_{s}, and π_{a}. The operational semantics of J is given by the heating and cooling rules (see structural congruence) and the reduction rule $J \triangleright P \vdash \sigma_{rv}(J) \longmapsto J \triangleright P \vdash \sigma_{rv}(P)$, where σ_{rv} substitutes the transmitted names for the distinct received variables.

We distinguish between *dynamic* and *static* operators. Intuitively, dynamic operators define terms that can perform steps, while static operators define connections between terms and side conditions on the reductions of their respective subterms. Moreover, we denote the parts of a term that are removed in reduction steps as *capabilities*. Usually, the reduction of dynamic operators is described by the axioms of the reduction semantics, while the remaining inference rules and the structural congruence describe the interplay with static operators. Accordingly, the dynamic operators of the above calculi are prefix and choice, because these operators are removed in the axioms of the respective reductions semantics, while $\mathsf{0}$, $\checkmark$, parallel composition, restriction, and replication are static operators. Note that we consider the definition operator of the join-calculus as dynamic, because e.g. a reduction of $\mathsf{def}\ J \triangleright P' \ \mathsf{in}\ P$ copies the elementary definition $J \triangleright P'$ and removes J if P contains the required outputs.

Furthermore, we distinguish between operators that allow for reductions of their subterms and those that require to be reduced first. We denote an operator as *guard* if at least one of its subterms cannot be used to perform a step before the guard itself is reduced. Its subterm(s) that cannot perform steps before the guard is reduced are denoted as *guarded* subterms. The other subterms, if there are any, as well as the subterms of operators that are not guards are denoted as *unguarded* subterms. Guards model sequential behaviour. To our intuition a purely sequential component cannot be cut into pieces to occupy different locations. Hence guarded subterms are not distributable until their guards are removed. However, there are process calculi, as the join-calculus, where a single operator combines different needs and guards only some of its subterms. Section 3 explains how we deal with such operators in the definition of distributability.

The capabilities of the pi-calculus are the prefixes, where the capability of a choice is the conjunction of the prefixes of all its branches—considered as single capability. Prefixes and thus also choice are guards, and all their subterms are guarded. The capabilities of the join-calculus are outputs and (compositions of) reception pattern, where the capability of a definition $\mathsf{def}\ D\ \mathsf{in}\ P$ is the conjunction of all compositions of reception patterns in D. In $\mathsf{def}\ (J_1 \triangleright P_1) \wedge \ldots \wedge (J_n \triangleright P_n)\ \mathsf{in}\ P$ the subterms $P_1, \ldots, P_n$ are guarded while P is an unguarded subterm. Reception patterns are matched against outputs in order to instantiate and unguard an instance of a guarded subterm. Note that the distinction into static and dynamic operators, guards, and capabilities are decisions made with the design of a process calculus. We use guards and capabilities to define distributability in Section 3. Hence, we require that all process calculi explicitly distinguish their guards, guarded subterms, and capabilities.

Replication or recursion can be provided by dynamic or static operators, e.g. $\mathsf{def}\ D\ \mathsf{in}\ P$ in J is a dynamic and $!P$ in π_{m} a static operator. Also the semantics can be given by a reduction rule or a rule of structural congruence. In both cases,

recursion or replication distinguishes itself from other operators by the fact that (one of) its subterms can be copied within rules of structural congruence or by reduction rules while the operator itself is usually never removed during reductions. We call such operators and capabilities *recurrent*.

In order to formalise the identification of sequential components, we assume for each process calculus a so-called *labelling* on the capabilities of processes. The labelling has to ensure that (1) each capability has a label (2) no label occurs more than once in a labelled term, (3) a label disappears only when the corresponding capability is reduced in a reduction step, and (4), once it has disappeared, it will not appear in the execution any more. A labelling method that satisfies these conditions for processes of the pi-calculus is presented in [4] (cf. [20]). Note that such a labelling can be derived from the syntax tree of processes. We require that, once the labelling of a term is fixed, the labels are preserved by the rules of structural congruence as well as by the reduction semantics of the respective calculus. Because of recurrent operators, new subterms with fresh labels for their capabilities may arise from applications of structural congruence or reduction rules. Since we need the labels only to distinguish syntactically similar components of a term, and to track them alongside reductions, we do not restrict the domain of the labels nor the method used to obtain them as long as the resulting labelling satisfies the above properties for all terms and all their derivatives in the respective calculus. Due to space constraints, and in order not to clutter the development with the details of labelling, we prefer to argue at the corresponding informal level. More precisely, we assume that **all** processes in the following are implicitly labelled. Remember that we need these labels only to distinguish between syntactical equivalent capabilities, e.g. to distinguish between the left and the right $\overline{y}$ in $\overline{y} \mid \overline{y}$.

2.1 Encodings and Quality Criteria

Let $\mathcal{L}_{\mathrm{S}} = \langle\, \mathcal{P}_{\mathrm{S}}, \longmapsto_{\mathrm{S}} \,\rangle$ and $\mathcal{L}_{\mathrm{T}} = \langle\, \mathcal{P}_{\mathrm{T}}, \longmapsto_{\mathrm{T}} \,\rangle$ be two process calculi, denoted as *source* and *target language*. An *encoding* from $\mathcal{L}_{\mathrm{S}}$ into $\mathcal{L}_{\mathrm{T}}$ is a function $[\![\cdot]\!] : \mathcal{P}_{\mathrm{S}} \to \mathcal{P}_{\mathrm{T}}$. Encodings often translate single source term steps into a sequence or pomset of target term steps. We call such a sequence or pomset an *emulation* of the corresponding source term step.

To analyse the quality of encodings and to rule out trivial or meaningless encodings, they are augmented with a set of quality criteria. In order to provide a general framework, Gorla in [8] suggests five criteria well suited for language comparison. Accordingly, we consider an encoding to be "good", if it satisfies the following conditions:

(1) *Compositionality*: The translation of an operator **op** is the same for all occurrences of that operator in a term, i.e., it can be captured by a context.
(2) *Name Invariance*: The encoding does not depend on particular names.
(3) *Operational Correspondence*: Every computation of a source term can be emulated by its translation, i.e., $S \Longmapsto_{\mathrm{S}} S'$ implies $[\![S]\!] \Longmapsto_{\mathrm{T}} \asymp [\![S']\!]$ (completeness), and every computation of a target term corresponds to some computation of the corresponding source term (soundness).

(4) *Divergence Reflection*: The encoding does not introduce divergence.
(5) *Success Sensitiveness*: A source term and its encoding answer the tests for success in exactly the same way, i.e., $S \Downarrow_{\checkmark}$ iff $[\![S]\!] \Downarrow_{\checkmark}$.

Note that the second criterion is not necessary to derive the separation results of this paper. Also note that a behavioural equivalence $\asymp$ on the target language is assumed for the definition of name invariance and operational correspondence. Its purpose is to describe the abstract behaviour of a target process, where abstract refers to the behaviour of the source term. By [8] the equivalence $\asymp$ is often defined in the form of a barbed equivalence (as described e.g. in [15]) or can be derived directly from the reduction semantics and is often a congruence, at least with respect to parallel composition. We require only that $\asymp$ is a weak reduction bisimulation, i.e., for all $T_1, T_2 \in \mathcal{P}_{\mathrm{T}}$ such that $T_1 \asymp T_2$, for all $T_1 \Longrightarrow_{\mathrm{T}} T_1'$ there exists a T_2' such that $T_2 \Longrightarrow_{\mathrm{T}} T_2'$ and $T_1' \asymp T_2'$.

We choose may-testing to instantiate the test for success in success sensitiveness, i.e., $P \Downarrow_{\checkmark}$, if it is reducible to a process containing a top-level unguarded occurrence of $\checkmark$. However, as we claim, this choice is not crucial. We have $\mathsf{n}(\checkmark) = \mathsf{fn}(\checkmark) = \mathsf{bn}(\checkmark) = \emptyset$. Moreover, we write $P \Downarrow_{\checkmark!}$, if P reaches success in every finite maximal execution. Note that success sensitiveness only links the behaviours of source terms and their literal translations, but not of their derivatives. To do so, Gorla relates success sensitiveness and operational correspondence by requiring that the equivalence on the target language never relates two processes with different success behaviours, i.e., $P \Downarrow_{\checkmark}$ and $Q \not\Downarrow_{\checkmark}$ implies $P \not\asymp Q$.

3 Distributability

Within this section, we discuss and fix the notions of distributability and preservation of distributability in the context of process calculi. Intuitively, a distribution of a process means the extraction (or: separation) of its (sequential) components and their association to different locations. However, we do not consider locations explicitly; we just focus on the possible division of a process term into components. Accordingly, a process P is *distributable into* $P_1, \ldots, P_n$, if we find some distribution that extracts $P_1, \ldots, P_n$ from within P onto different locations. Preservation of distributability then means that the target term is at least as distributable as the source term.

3.1 Distributable Processes

The most important operator to implement distributability is the parallel operator. Indeed we consider distributability as a special case of parallel composition with a stricter notion of independence, which becomes visible if we compare calculi. So, first of all, two subterms are distributable if they are parallel.

Unfortunately, the converse of that statement—two subterms are not distributable if they are not parallel—is usually not true. The main reason for this is scoping of names. Consider for example the term $(\nu x)\,(P \mid Q)$ in the pi-calculus. Although the outermost operator is not the parallel operator, the processes

P and Q are nonetheless distributable. More precisely, for all considered variants of the pi-calculus, two subterms are distributable if they are (modulo $\equiv$) composed in parallel under some restrictions; see the notion of *standard form* of the pi-calculus [13]. Hence, (1) we consider distributability modulo structural congruence, and (2) we allow to remove toplevel restrictions and parallel operators to separate the distributable components.

In the case of the join-calculus, the situation is worse. Again, the problematic operator is responsible for scoping of names. But in the case of the join-calculus scoping is realised by definitions that at the same time represent the input capabilities of the calculus. Consider the term $R = \mathsf{def}\ a \triangleright 0\ \mathsf{in}\ (\mathsf{def}\ b \triangleright c\langle a\rangle\ \mathsf{in}\ (a \mid b))$. It is constructed of two nested definitions. Intuitively, it represents the combination of the two processes $\mathsf{def}\ a \triangleright 0\ \mathsf{in}\ a$ and $\mathsf{def}\ b \triangleright c\langle a\rangle\ \mathsf{in}\ b$ but, because of $c\langle a\rangle$, we cannot get rid of the nesting of the definitions—not even modulo structural congruence. The best we can achieve is $R \equiv \mathsf{def}\ a \triangleright 0\ \mathsf{in}\ ((\mathsf{def}\ b \triangleright c\langle a\rangle\ \mathsf{in}\ b) \mid a)$. Note that $\mathsf{def}\ b \triangleright c\langle a\rangle\ \mathsf{in}\ b$ is not guarded within R. Because of that, the cooling and heating rules, which model structural congruence of the join-calculus, allow us to derive $\vdash R \rightleftharpoons b \triangleright c\langle a\rangle \vdash \mathsf{def}\ a \triangleright 0\ \mathsf{in}\ a, b$ as well as $\vdash R \rightleftharpoons a \triangleright 0 \vdash \mathsf{def}\ b \triangleright c\langle a\rangle\ \mathsf{in}\ b, a$. This reason is enough for us to consider $\mathsf{def}\ a \triangleright 0\ \mathsf{in}\ a$ and $\mathsf{def}\ b \triangleright c\langle a\rangle\ \mathsf{in}\ b$ as distributable within R. Formally, each J-term J is distributable into the terms $J_1, \ldots, J_n \in \mathrm{J}$ if, for all $1 \leq i \leq n$, there exists some multisets $\mathcal{R}, \mathcal{M}$ such that $\vdash J \rightleftharpoons \mathcal{R} \vdash J_i, \mathcal{M}$ and there are no two capabilities in $J_1, \ldots, J_n$ with the same label. Note that we can define structural congruence for all process calculi by a chemical abstract machine, but that this kind of special consideration is only necessary because definitions in the join-calculus are guards that have unguarded subterms. Hence, we assume that, (at least) for all process calculi that contain a guard with unguarded subterms, structural congruence is given by a chemical abstract machine.

Note that this example on the join-calculus illuminates that we consider distributability as an irreversible predicate. There is no possibility to restore from a given set of distributable components the original process term, because by the separation of the components we irreversibly loose their original connections. Thus, we cannot beyond doubt conclude that the terms $\mathsf{def}\ a \triangleright 0\ \mathsf{in}\ a$ and $\mathsf{def}\ b \triangleright c\langle a\rangle\ \mathsf{in}\ b$ originally belong to R. Similarly, we cannot conclude that the terms P and Q were originally subterms of the pi-calculus term $(\nu x)\,(P \mid Q)$, because we lost the information about the restriction. However, these lost information, i.e., the connections between distributable components in the original term, are already captured by the other criteria on the quality of an encoding.

Another important observation is that, because of $!P \equiv P \mid !P$, different copies of a recursive term are distributable in the pi-calculus, whereas there is no such $\equiv$-rule for definitions in the join-calculus. This reflects a fundamental design decision in the join-calculus, namely that the receptors of a given channel are forced to reside at the same location [7,12]. Note that this design decision marks the main difference between the join-calculus and the asynchronous pi-calculus. Accordingly, we require that this design decision is made explicit within the structural congruence of the calculus. A recurrent operator is called

distributable if such a $\equiv$-rule is provided and, otherwise, as not distributable, i.e., $!P$ is distributable but J-term definitions are not distributable.

Definition 3 (Distributability). *Let $\langle \mathcal{P}, \longmapsto \rangle$ be a process calculus, $\equiv$ be its structural congruence, and $P \in \mathcal{P}$. P is* distributable *into $P_1, \ldots, P_n \in \mathcal{P}$ if there exists $P' \in \mathcal{P}$ with $P' \equiv P$ such that*

1. *for all $1 \leq i \leq n$, P_i contains at least one capability or constant different from $\mathsf{0}$ and P_i is an unguarded subterm of P' or, in case $\equiv$ is given by a chemical approach, $\vdash P' \rightleftharpoons \mathcal{R} \vdash P_i, \mathcal{M}$ for some multisets $\mathcal{R}, \mathcal{M}$,*
2. *in $P_1, \ldots, P_n$ there are no two occurrences of the same capability, i.e., no label occurs twice, and*
3. *each guarded subterm and each constant (different from $\mathsf{0}$) of P' is a subterm of at least one of the terms $P_1, \ldots, P_n$.*

The degree of distributability *of P is the maximal number of distributable subterms of P.*

Hence, we can split a process into its sequential components or larger subterms, e.g. each term is distributable into itself. This allows us to analyse the behaviour of distributable subterms. Note that we do not allow to distribute the empty process, because otherwise usually every process is distributable into infinitely many empty processes. The same holds for subterms not containing any capability or constant different from $\mathsf{0}$, as e.g. in the term $\mathsf{0} \mid \mathsf{0}$. Of course, $!P$ is distributable into arbitrary many copies of P (and one $!P$). However, since none of the later counterexamples contains replication, this decision is not crucial.

Hence a pi-term P is distributable into $P_1, \ldots, P_n$ if $P \equiv (\nu \tilde{a})\,(P_1 \mid \ldots \mid P_n)$. The $\mathcal{P}_\mathrm{J}$-term $\mathsf{def}\ a \triangleright \mathsf{0}\ \mathsf{in}\ (\mathsf{def}\ b \triangleright c\,\langle a \rangle\ \mathsf{in}\ (a \mid b))$ is distributable into $\mathsf{def}\ a \triangleright \mathsf{0}\ \mathsf{in}\ a$ and $\mathsf{def}\ b \triangleright c\,\langle a \rangle\ \mathsf{in}\ b$, but e.g. also into $\mathsf{def}\ a \triangleright \mathsf{0}\ \mathsf{in}\ \mathsf{0}$, $\mathsf{def}\ b \triangleright c\,\langle a \rangle\ \mathsf{in}\ \mathsf{0}$, a, and b, because $\vdash \mathsf{def}\ a \triangleright \mathsf{0}\ \mathsf{in}\ (\mathsf{def}\ b \triangleright c\,\langle a \rangle\ \mathsf{in}\ (a \mid b)) \rightleftharpoons \mathsf{def}\ a\ \mathsf{in}\ \mathsf{0}, \mathsf{def}\ b\ \mathsf{in}\ c\,\langle a \rangle \vdash a \mid b \rightleftharpoons \mathsf{def}\ a\ \mathsf{in}\ \mathsf{0}, \mathsf{def}\ b\ \mathsf{in}\ c\,\langle a \rangle \vdash a, b \rightleftharpoons \vdash \mathsf{def}\ a \triangleright \mathsf{0}\ \mathsf{in}\ \mathsf{0}, \mathsf{def}\ b \triangleright c\,\langle a \rangle\ \mathsf{in}\ \mathsf{0}, a, b$.

3.2 Preservation of Distributability

Note that an encoding can always trivially ensure that the encoding has at least as much distributable components by introducing new subterms without any behaviour. Hence, it does not suffice to reason only about the degree of distributability, i.e., about the number of distributable components. Instead we require that the encodings of distributable source term parts and their corresponding parts in the encoding are related by $\asymp$. By doing so we relate the definition of the preservation of distributability to operational completeness, i.e., a semantical criterion that ensures the preservation of the behaviour of the source term (part). We require that each target term part has to be able to emulate at least all behaviour of the respective source part. As a side effect we require that whenever a part of a source term can solve a task independently of the other parts—i.e., it can reduce on its own—then the respective part of its encoding must also be able to emulate this reduction independently of the rest of the encoded term. This reflects the intuition that distribution adds some additional requirements on the independence of parallel terms.

Definition 4 (Preservation of Distributability). *An encoding* $[\![\cdot]\!] : \mathcal{P}_{\mathrm{S}} \to \mathcal{P}_{\mathrm{T}}$ preserves distributability *if for every* $S \in \mathcal{P}_{\mathrm{S}}$ *and for all terms* $S_1, \ldots, S_n \in \mathcal{P}_{\mathrm{S}}$ *that are distributable within* S *there are some* $T_1, \ldots, T_n \in \mathcal{P}_{\mathrm{T}}$ *that are distributable within* $[\![S]\!]$ *such that* $T_i \asymp [\![S_i]\!]$ *for all* $1 \leq i \leq n$.

In essence, this requirement is a distributability-enhanced adaptation of operational completeness. It respects both the intuition on distribution as separation on different locations—an encoded source term is at least as distributable as the source term itself—as well as the intuition on distribution as independence of processes and their executions—implemented by $T_i \asymp [\![S_i]\!]$.

To ensure that the new criterion is not in conflict with the framework of Gorla, it suffices to show the existence of encodings that satisfy all six criteria. Such encodings are presented in [16] and [19]. Moreover, [19] shows that in case of the pi-calculus every good encoding that translates the parallel operator and restriction homomorphically and preserves structural congruence also preserves distributability. Not surprisingly, the most crucial requirement here is the homomorphic translation of the parallel operator. However, this holds only in case of process calculi as the pi-calculus, where distributable terms can be separated modulo $\equiv$ by parallel operators.

Thus, the (semantic) criterion formalised in Definition 4 can be considered to be at most as hard as the (syntactic) criterion on the homomorphic translation of the parallel operator. To see that it is not an equivalent requirement, but indeed strictly weaker, [19] refers to an encoding from π_{m} (without replication) into π^2_{a}, the asynchronous pi-calculus augmented with a two-level polyadic synchronisation by Carbone and Maffeis [5]. This encoding is good and preserves distributability but it does not translate the parallel operator homomorphically. Moreover, [5] proves that there is no good encoding from π_{m} into π^2_{a} that translates the parallel operator homomorphically; this separation result does not rely on replication, i.e., it also implies that there is no such encoding from π_{m} without replication into π^2_{a}.

3.3 Distributable Reductions

As discussed above, the criterion in Definition 4 requires not only the preservation of the distributability of processes but also the preservation of the distributability of steps or executions of the respective distributable processes. In order to obtain an alternative way to prove the preservation of distributability, we make this intuition explicit. More precisely, we show that an operationally complete encoding that preserves distributability always also preserves the distributability between sequences of source term steps. To do so, we define first what it means for two steps or executions to be distributable.

If a single process—of an arbitrary process calculus—can perform two different steps, i.e., steps on capabilities with different labels, then we call these steps alternative to each other. Two alternative steps can either be in conflict or not; in the latter case, it is possible to perform both of them in parallel, according to some assumed step semantics.

Definition 5 (Distributable Steps). *Let* $\langle \mathcal{P}, \longmapsto \rangle$ *be a process calculus and* $P \in \mathcal{P}$ *a process. Two alternative steps of* P *are in* conflict*, if performing one step disables the other step, i.e., if both reduce the same not recurrent capability. Otherwise they are* parallel*. Two parallel steps of* P *are* distributable*, if each recurrent capability reduced by both steps is distributable, else the steps are* local*.*

Remember that the "same" means "with the same label", i.e., in $\overline{y} \mid y.P_1 \mid y.P_2$ the two steps on y are in conflict but $\overline{y} \mid y.P_1 \mid y.P_2 \mid \overline{y}$ and $\overline{y} \mid !y.P_1 \mid \overline{y}$ can both perform two parallel steps on y. Moreover, the reductions on channel a and b are parallel in $\overline{a} \mid \overline{b} \mid a.P_1 \mid b.P_2$, but they are in conflict in $\overline{a} \mid \overline{b} \mid a.P_1 + b.P_2$, because choice counts as a single capability which is reduced in both steps.

Also note that in contrast to parallel steps, distributable steps can reduce the same recurrent capability only if it is distributable. In many process calculi such as π_{a}, two steps are distributable iff they are parallel, because all recurrent capabilities are distributable. However, there are also process calculi as J in which these notions indeed refer to quite different situations. Thus, for the comparison with these calculi, their intuitive distinction is useful.

In the join-calculus, two alternative steps that reduce the same definition but do not compete for some output, as e.g. the reduction of $x\langle u \rangle$ and $x\langle v \rangle$ in $\mathsf{def}\, x(z) \triangleright y\langle z \rangle\ \mathsf{in}\ (x\langle u \rangle \mid x\langle v \rangle)$, can be considered as *parallel* steps; they do not compete for the input capability, because it is recurrent. However, we can *not* consider these two steps as distributable, as this would imply that the definition itself is distributable which—by design—is not intended in J: there is always exactly one receiver for each defined name [7].

Next we define parallel and distributable sequences of steps.

Definition 6 (Distributable Executions). *Let* $\langle \mathcal{P}, \longmapsto \rangle$ *be a process calculus,* $P \in \mathcal{P}$*, and let* A *and* B *denote two executions of* P*.* A *and* B *are in* conflict*, if a step of* A *and a step of* B *are in conflict, else* A *and* B *are* parallel*.*

Two parallel sequences of steps A *and* B *are* distributable*, if each pair of a step of* A *and a step of* B *is distributable.*

In π_{a}, two sequences of steps A and B of a process P are parallel iff $P \equiv (\nu\tilde{x})(P_1 \mid P_2)$ such that P_1 can perform A while P_2 can perform B, i.e., if $A : P \longmapsto P_{A,1} \longmapsto \ldots \longmapsto P_{A,n}$ and $B : P \longmapsto P_{B,1} \longmapsto \ldots \longmapsto P_{B,m}$ then, for all $1 \leq i \leq n$ and all $1 \leq j \leq m$, there exists $P'_{A,i}, P'_{B,j} \in \mathcal{P}$ such that $P_{A,i} \equiv (\nu\tilde{x})\left(P'_{A,i} \mid P_2\right)$ and $P_{B,j} \equiv (\nu\tilde{x})\left(P_1 \mid P'_{B,j}\right)$. Again, two sequences of steps are distributable iff they are parallel. Unfortunately, in the join-calculus two processes able to perform parallel sequences of steps cannot always be separated by a parallel operator in this way; even if they do not reduce the same definition. The reason is again the restriction caused by definitions. In the term $\mathsf{def}\, a \triangleright P_1\ \mathsf{in}\ (\mathsf{def}\, b \triangleright c\langle a \rangle\ \mathsf{in}\ (a \mid b))$ the reduction of a is independent of the reduction of b. Hence, these two steps are parallel and even distributable. But, because of $c\langle a \rangle$, we cannot get rid of the nesting of these two definitions.

Although the definitions of distributable processes in Definition 3 and distributable executions in Definition 6 are quite different, they are closely related. Two executions of a term P are distributable iff P is distributable into two

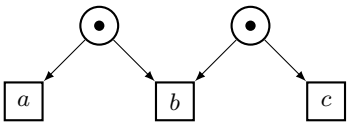

Fig. 1. A fully reachable pure M in Petri nets

subterms such that each performs one of these executions. Hence, an operationally complete encoding is distributability-preserving only if it preserves the distributability of sequences of source term steps. The proofs of this and the following results can be found in [20].

Lemma 1 (Distributability-Preservation). *An operationally complete encoding* $[\![\cdot]\!] : \mathcal{P}_S \to \mathcal{P}_T$ *that preserves distributability also preserves distributability of executions, i.e., for all source terms* $S \in \mathcal{P}_S$ *and all sets of pairwise distributable executions of* S*, there exists an emulation of each execution in this set such that all these emulations are pairwise distributable in* $[\![S]\!]$*.*

4 Separation by the Synchronisation Pattern M

[24] analyses the possibility to implement a (synchronous) Petri net specification within an asynchronous setting. They find a semi-structural property called M that distinguishes distributable Petri nets from those nets that may only under additional assumptions on the underlying system structure be implemented in a fully asynchronous and distributed setting.

An M, as visualised in Figure 1, describes a Petri net that consists of two parallel transitions and one transition that is in conflict with both of the former. In other words, it describes a situation where either two parts of the net can proceed independently or they synchronise to perform a single transition together. We denote such descriptions of special situations of synchronisation as *synchronisation pattern*. [24,25] states that a Petri net specification can be implemented in an asynchronous, fully distributed setting iff it does not contain a fully reachable pure M. Accordingly, they denote such Petri nets as distributable. They also present a description of a fully reachable pure M as a property of a step transition system which allows us to directly use this pattern to reason about process calculi.

A first analysis shows that we find the M also in the asynchronous pi-calculus (see Example 1 below). This reflects earlier observations in [12]: it is not possible to implement the pi-calculus and even its asynchronous fragment in an asynchronous and fully distributed setting. To overcome these problems the join-calculus was introduced as a model of distributed computation [7,12]. Mutual encodings between the (core) join-calculus and the asynchronous pi-calculus have shown that they have the same expressive power [7]. Here, we show a difference with respect to the degree of distributability. Hence, we explain what exactly distinguishes both calculi. It turns out that this distinction is well described by

the synchronisation pattern M, i.e., what distinguishes the asynchronous pi-calculus and the join-calculus is the ability to express conflicts between distributable steps. This lack in expressiveness in turn allows fully distributed implementations of the join-calculus.

4.1 The Synchronisation Pattern M

If we compare the asynchronous pi-calculus and the join-calculus, the most obvious difference is that in J any channel can appear only once in input position. As a consequence, two conflicting steps in the join-calculus can only compete for different output messages but not for different input capabilities, as it is the case in π_{a}. Repeating this argument, all steps of a chain of conflicting steps in the join-calculus are tied to the same definition, i.e., are not distributable.

Lemma 2. *For all $P \in \mathcal{P}_{\mathrm{J}}$ and all lists $S = [s_1, \ldots, s_n]$ of steps of P such that for all $1 \leq i < n$ the step s_i is in conflict with the step s_{i+1}, all steps in S are pairwise local and reduce the same definition.*

In contrast, in π_{a}, it is very easy to find such a list of conflicting steps of which some are distributable, by combining conflicts on outputs and inputs.

Example 1. Consider $P = \overline{y}\langle u\rangle \mid y(x).P_1 \mid \overline{y}\langle v\rangle \mid y(x).P_2$ with $P \in \mathcal{P}_{\mathrm{a}}$. P can perform four different alternative steps modulo structural congruence:

$$P \longmapsto \{\, {}^{u}/_{x} \,\} P_1 \mid \overline{y}\langle v\rangle \mid y(x).P_2 \qquad (s_1)$$

$$P \longmapsto y(x).P_1 \mid \overline{y}\langle v\rangle \mid \{\, {}^{u}/_{x} \,\} P_2 \qquad (s_2)$$

$$P \longmapsto \overline{y}\langle u\rangle \mid y(x).P_1 \mid \{\, {}^{v}/_{x} \,\} P_2 \qquad (s_3)$$

$$P \longmapsto \overline{y}\langle u\rangle \mid \{\, {}^{v}/_{x} \,\} P_1 \mid y(x).P_2 \qquad (s_4)$$

The step s_1 is in conflict with step s_2, since both compete for the first output $\overline{y}\langle u\rangle$. Similarly, step s_2 and s_3 compete for the second input $y(x).P_2$, and step s_3 and step s_4 compete for the second output, i.e., P has a chain $S = [s_1, \ldots, s_4]$ of conflicting steps. But s_1 and s_3 as well as s_2 and s_4 are distributable in P.

Thus, the ability to express distributable conflicts separates the asynchronous pi-calculus from the join-calculus. However, the preservation of distributability in Definition 4 does not require to preserve the distributability of conflicts but only of processes and their executions. On the other side, the structure used in [24] to identify distributable Petri nets strongly relies on the notion of conflict. More precisely, an M arises from the combination of two parallel steps and a third step that is in conflict with both of the former.

Definition 7 (Synchronisation Pattern M). *Let $\langle\, \mathcal{P}, \longmapsto \,\rangle$ be a process calculus and $P \in \mathcal{P}$ such that:*

1. *P can perform at least three alternative reduction steps $a : P \longmapsto P_a$, $b : P \longmapsto P_b$, and $c : P \longmapsto P_c$ such that P_a, P_b, and P_c are pairwise different.*
2. *Moreover, the steps a and c are parallel in P.*
3. *But b is in conflict with both a and c.*

In this case, we denote the process P as M*. If the steps a and c are distributable in P, then we call the* M non-local*. Otherwise, the* M *is called* local*.*

We observe, that the P of Example 1 represents a *non-local* M in π_{a}, because we can choose the step s_1 as a, s_2 as b, and s_3 as c. In contrast, the term $Q = \mathsf{def}\, x\,(z) \mid y\,(z') \triangleright z\,\langle z'\rangle \;\mathsf{in}\; (x\,\langle u\rangle \mid x\,\langle v\rangle \mid y\,\langle u\rangle \mid y\,\langle v\rangle)$ is a *local* M in the (core) join-calculus. Indeed, all M in the join-calculus are local, because, by Lemma 2, the step b forces its conflicting counterparts to reduce the same definition.

Lemma 3. *All* M *in the join-calculus are local.*

Thus, the asynchronous pi-calculus and the join-calculus do also differ by the ability to express a non-local M. As described in [24], a language that cannot express a non-local M can be considered as distributable. Accordingly, as intended by its design, the join-calculus is distributable. We show that the pi-calculus is not distributable—not even in its asynchronous and choice-free fragment.

4.2 Distributability of the Pi-calculus

To show that the examined difference forbids distributability-preserving encodings, we have to show that it is not possible to express the abstract behaviour of all non-local M in the join-calculus with respect to our requirements on good and distributability-preserving encodings. We use the M of Example 1 as running counterexample S. In the framework of Gorla, source terms and their encodings are compared by their ability to reach success. To distinguish the conflicting step $b = s_2$ from the parallel steps $a = s_1$ and $c = s_3$, we instantiate P_1 with $\overline{x}$, P_2 with $\overline{x} \mid \overline{x}$, and place the observer $O = u.v.v.\checkmark$ in parallel to P. Hence,

$$S = (\overline{y}\langle u\rangle \mid y(x)\,.\overline{x}) \mid (\overline{y}\langle v\rangle \mid y(x)\,.\,(\overline{x} \mid \overline{x}) \mid u.v.v.\checkmark) \tag{E1}$$

reaches success iff S performs both of the distributable steps a and c. Note that any good encoding that preserves distributability has to translate E1 such that the emulations of the steps a and c are again distributable. However, the encoding can translate these two steps into sequences of steps, which allows to emulate the conflicts with the emulation of b by two different distributable steps. We show that every distributability-preserving encoding has to distribute b and, afterwards, that this distribution of b violates the criteria of a good encoding.

Lemma 4. *Every encoding $[\![\cdot]\!] : \mathcal{P}_{\mathrm{a}} \to \mathcal{P}_{\mathrm{J}}$ that is good (except for compositionality) and distributability-preserving has to split up the conflict in S given by E1 of b with a and c such that there exists a maximal execution in $[\![S]\!]$ in which a is emulated but not c, and vice versa.*

Lemma 4 describes a partial deadlock. If the emulation of b and with it the conflicts with the emulation of a and c are distributed, the encoded term can make the wrong decision and, thus, result in one successful emulation (of a or c) but two deadlocked emulation attempts of the respective other two steps. Since there is no maximal execution of E1 with a but not c (or vice versa), such an

encoding cannot be considered as a good encoding. In the setting used so far, we cannot observe the difference in the abstract behaviour of E1 and $[\![\text{E1}]\!]$.

One reason is the weak requirements on $\asymp$. A success respecting bisimulation, in its simplest case, cannot distinguish between more than three different cases: success is not reachable, success is always reachable, and success is reachable in some but not all maximal executions. To prove non-existence of distribution-preserving encodings it suffices to require that $\asymp$ is not trivial, e.g. by requiring that it distinguishes more than two observables. In this case, we have to modify E1, i.e., choose a suitable instantiation of P_1, P_2, and the observer, such that $[\![S_a]\!]$, $[\![S_b]\!]$, $[\![S_c]\!]$, and $[\![S_{ac}]\!]$ are pairwise distinguished by $\asymp$, where S_{ac} is the result of performing a and c in S. Then, the maximal execution that emulates a but not c contradicts operational correspondence. Note that in this case we do not need compositionality at all.

Another way is to make use of compositionality. Remember that the best known encoding from the asynchronous pi-calculus into the join-calculus in [7] is not compositional, but consists of an inner, compositional encoding surrounded by a fixed context—the implementation of so-called firewalls—that is parametrised on the free names of the source term. Actually, it is this surrounding context that reduces the degree of distributability, because different steps on the same channel name have to synchronise on a firewall. The following result captures this and similar encodings.

Theorem 1. *There is no good and distributability-preserving encoding from* π_{a} *into* J. *There is no distributability-preserving encoding from* π_{a} *into* J *that is good except for compositionality but consists of an inner compositional encoding surrounded by a fixed context parametrised on the free names of the source term.*

4.3 Distributability in Other Calculi

Above, first an absolute result, i.e., a result that refers to the properties of a single language, is derived in Lemma 2. It clarifies which property distinguishes the source and the target language, i.e., the reason why the target language does not contain the synchronisation pattern M. Then, the existence of the M in the source language is shown by an example, which is subsequently used as counterexample. Lemma 4 uses properties of the target language—basically the absolute result in Lemma 2—to show that any encoding has to split the conflict in the counterexample. Finally, Theorem 1 reasons about some properties of the source language to show that the split of the conflict in the encoded counterexample violates the criteria of a good encoding. This argumentation provides a guideline for similar considerations in other languages.

Note that the synchronisation pattern does not only describe the difference between two languages as an abstraction of a particular situation of synchronisation but it also serves as an abstract description of the properties of the counterexample. This allows us to separate more clearly between the argumentation for the source and the target language in the above proofs. Hence, to change the source language it usually suffices to find an example with the properties required by the synchronisation pattern. In case of the target language

we have to revise the absolute result and Lemma 4, i.e., we have to show why the new target language can not express the synchronisation pattern modulo the criteria required on an encoding. As example, we exhibit a separation between two simple variants of CSP in [20,18]. The splitting of arguments on the source and the target languages simplifies also the comparison of multiple languages, because not every pair has to be checked.

5 Another Synchronisation Pattern

In the last section we compare different process calculi by their ability to express the synchronisation pattern M. We learn that the different synchronisation mechanisms of the calculi lead to differences in the expressive power with respect to specific kinds of conflicts. By [17,8,21,19], we also know that the restriction in the choice operator leads to a separation result between π_{m} and π_{s}. However, in [17] and [8] the homomorphic translation of the parallel operator was used to derive this separation result and in [21,19] the proof was unsatisfactory, because it reveals not much intuition on why the counterexamples lead to the difference. In order to provide more intuition on this separation result and on the difference in the expressive power of π_{m} and π_{s} with respect to conflicts, we show that the calculi can be distinguished by a new synchronisation pattern similar to the M. Not surprisingly, the new pattern combines again conflicting and distributable steps. Interestingly, it reflects a well-known standard problem in the area of distributed systems, namely the problem of the dining philosophers [6].

We start with a simple observation on the asynchronous pi-calculus. Without choice each reduction step reduces exactly one output and one input. So all conflicts in π_{a} are on steps on the same link. With separate choice a single step can reduce more than a single out- or input. But if we consider steps between two distributable subprocesses then each reduction step reduces only outputs in one subprocess and only inputs in the other. As a consequence, a chain of conflicting steps can build an M by alternating input and output capabilities as visualised in Example 1. But, by this method, no circle of odd length can be constructed as it is represented by the synchronisation pattern $\star$.

Definition 8 (Synchronisation Pattern $\star$). *Let $\langle\, \mathcal{P}, \longmapsto \,\rangle$ be a process calculus and $P \in \mathcal{P}$ such that:*

1. *P can perform at least five alternative reduction steps $i : P \longmapsto P_i$ for $i \in \{\, a, b, c, d, e \,\}$ such that the P_i are pairwise different.*
2. *Moreover, the steps a, b, c, d, and e form a circle such that a is in conflict with b, b is in conflict with c, c is in conflict with d, d is in conflict with e, and e is in conflict with a. Finally,*
3. *every pair of steps in $\{\, a, b, c, d, e \,\}$ that is not in conflict is parallel in P.*

In this case, we denote the process P as $\star$. The synchronisation pattern $\star$ is visualised by the Petri net in Figure 2. If all pairs of parallel steps in $\{\, a, b, c, d, e \,\}$ are distributable in P, then we call the $\star$ non-local. *Otherwise, it is called* local.

Note that in the pi-calculus every $\star$ and every M is non-local. To see the connection with the dining philosophers problem, consider the places in Figure 2 as the

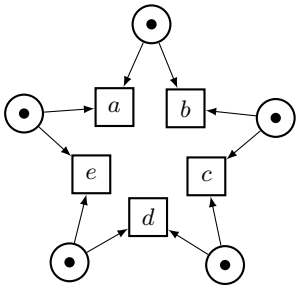

Fig. 2. The Synchronisation Pattern $\star$ in Petri nets

chopsticks of the philosophers, i.e., as resources, and the transitions as eating operations, i.e., as steps consuming resources. Each step needs mutually exclusive access to two resources and each resource is shared among two subprocesses. If both resources are allocated simultaneously, eventually exactly two steps are performed. As shown in the following, a fully distributable implementation of that pattern requires the expressive power of mixed choice.

By Example 1 we know that π_{s} can express distributable conflicts, but π_{s} cannot express a circle of such conflicts that is of odd degree greater than four as it is depicted by $\star$. Note that smaller circles do not have parallel, i.e., distributable, steps. Hence, $\star$ represents the smallest example of the problematic structure but separation can principally be proved for any such structure of odd degree and at least five steps. The main argument is that π_{s} can build chains of conflicts by alternating conflicts between output and input capabilities, but without mixed choice no cycle of odd degree can be obtained this way.

Lemma 5. *There is no $\star$ in π_{s}.*

In contrast to π_{s}, π_{m} can express the synchronisation pattern $\star$ as the example

$$S = \overline{a} + b.S_1 \mid \overline{b} + c.S_2 \mid \overline{c} + d.S_3 \mid \overline{d} + e.S_4 \mid \overline{e} + a.S_5 \tag{E3}$$

shows. We use this example as counterexample. Similar to Section 4.2, we show that each encoding of the counterexample requires that at least one conflict has to be distributed and that this violates the requirements on a good encoding.

Theorem 2. *No good encoding from π_{m} into π_{s} preserves distributability.*

Note that we could derive the same result if, as in Section 4.2, we allow for a not compositional encoding that consists of an inner compositional encoding surrounded by a fixed context parametrised on the free names of the source term. Moreover, since the synchronisation pattern $\star$ includes the pattern M—more precisely it consists of three cyclic overlapping M—separation results derived on these two patterns (with respect to the same quality criteria) automatically lead to a lattice. Here, by Theorem 1 and Theorem 2, no good encoding from π_{m} into J preserves distributability.

Also note, that the E3 is in fact a CCS-term. Hence, we can apply the same line of argument to show separation between the corresponding variants of CCS.

Moreover, we can show that there is no good and distributability-preserving encoding from π_a into CCS with mixed choice.

6 Conclusion

As main contributions, we (1) propose a new criterion to reason about the degree of distribution which is better suited than the common homomorphic translation of the parallel operator. Then, (2) we present a new separation result that clarifies the difference between the asynchronous pi-calculus and the join-calculus. Moreover, we (3) show that the proof method of this result is in general well suited to reason about the expressive power of synchronisation mechanisms by discussing how it can be transferred with little effort to compare other source and target languages (cf. [20,18]). And (4) we present two generally formulated synchronisation patterns that expose the power of different synchronisation mechanisms in the pi-calculus family but can be used in a similar manner to reason about and to classify synchronisation mechanisms in other process calculi.

Note that [16] presents a good encoding from π_s into π_a that translates the parallel operator homomorphically, i.e., that preserves distributability. Moreover, [7,19] present good (but not distributability-preserving) encodings between J and π_a, and from π_m into π_a. Combining these positive results and the new separation results on the two synchronisation patterns, we obtain a hierarchy of distributability between pi-like calculi. The synchronous pi-calculus (π_m), the asynchronous pi-calculus (π_a), and the join-calculus (J) all have the same abstract expressive power, but there exists no good and distributability-preserving encoding from π_m into π_a, and neither from π_a into J.

Of course we do not believe that these two patterns already capture all kinds of synchronisation mechanisms in process calculi. In further research we want to analyse e.g. what kind of synchronisation patterns are expressed by polyadic synchronisation in [5] or by the synchronisation mechanisms described in [11].

In case of separation results, a natural next step to improve the results is to go back to particular distributions in terms, in order to examine the problematic set of distributed terms in the source language. This way a positive result for a sublanguage of the source language can be derived. An exhaustive analysis may even lead to an exact borderline between distributable and not distributable languages. Note that the results in [25] go in this direction for the area of Petri nets. This kind of consideration is beyond the scope of this paper but another interesting topic of further research.

References

1. Berry, G., Boudol, G.: The Chemical Abstract Machine. In: Proc. of POPL. SIGPLAN-SIGACT, pp. 81–94 (1990)
2. Best, E., Darondeau, P.: Petri Net Distributability. In: Clarke, E., Virbitskaite, I., Voronkov, A. (eds.) PSI 2011. LNCS, vol. 7162, pp. 1–18. Springer, Heidelberg (2012)

3. Boudol, G.: Asynchrony and the π-calculus (note). Note, INRIA (1992)
4. Cacciagrano, D., Corradini, F., Palamidessi, C.: Explicit fairness in testing semantics. Logical Methods in Computer Science 5(2), 1–27 (2009)
5. Carbone, M., Maffeis, S.: On the Expressive Power of Polyadic Synchronisation in π-Calculus. Nordic Journal of Computing 10(2), 70–98 (2003)
6. Dijkstra, E.W.: Hierarchical Ordering of Sequential Processes. Acta Informatica 1(2), 115–138 (1971)
7. Fournet, C., Gonthier, G.: The Reflexive CHAM and the Join-Calculus. In: Proc. of POPL. SIGPLAN-SIGACT, pp. 372–385 (1996)
8. Gorla, D.: Towards a Unified Approach to Encodability and Separation Results for Process Calculi. Information and Computation 208(9), 1031–1053 (2010)
9. Hennessy, M.: A Distributed Pi-Calculus. Cambridge University Press (2007)
10. Honda, K., Tokoro, M.: An Object Calculus for Asynchronous Communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)
11. Laneve, C., Vitale, A.: The Expressive Power of Synchronizations. In: Proc. of LICS, pp. 382–391 (2010)
12. Lévy, J.-J.: Some Results in the Join-Calculus. In: Ito, T., Abadi, M. (eds.) TACS 1997. LNCS, vol. 1281, pp. 233–249. Springer, Heidelberg (1997)
13. Milner, R.: Communicating and Mobile Systems: The π-Calculus. Cambridge University Press, New York (1999)
14. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Part I and II. Information and Computation 100(1), 1–77 (1992)
15. Milner, R., Sangiorgi, D.: Barbed Bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
16. Nestmann, U.: What is a "Good" Encoding of Guarded Choice? Information and Computation 156(1-2), 287–319 (2000)
17. Palamidessi, C.: Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus. Mathematical Structures in Computer Science 13(5), 685–719 (2003)
18. Peters, K.: Translational Expressiveness. PhD thesis, TU Berlin (2012), http://nbn-resolving.de/urn:nbn:de:kobv:83-opus-37495
19. Peters, K., Nestmann, U.: Is It a "Good" Encoding of Mixed Choice? In: Birkedal, L. (ed.) FOSSACS 2012. LNCS, vol. 7213, pp. 210–224. Springer, Heidelberg (2012)
20. Peters, K., Nestmann, U., Goltz, U.: On Distributability in Process Calculi (Appendix). Technical Report, TU Berlin (2013), http://www.mtv.tu-berlin.de/fileadmin/a3435/pubs/distProcCal.pdf
21. Peters, K., Schicke-Uffmann, J.-W., Nestmann, U.: Synchrony vs Causality in the Asynchronous Pi-Calculus. In: Proc. of EXPRESS. EPTCS, vol. 64, pp. 89–103 (2011)
22. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming 60, 17–140 (2004); (An earlier version of this paper was published as technical report at Aarhus University in 1981)
23. van Glabbeek, R.: The Linear Time – Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes. Handbook of Process Algebra, 3–99 (2001)
24. van Glabbeek, R., Goltz, U., Schicke, J.-W.: On Synchronous and Asynchronous Interaction in Distributed Systems. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 16–35. Springer, Heidelberg (2008)
25. van Glabbeek, R., Goltz, U., Schicke-Uffmann, J.-W.: On Distributability of Petri Nets. In: Birkedal, L. (ed.) FOSSACS 2012. LNCS, vol. 7213, pp. 331–345. Springer, Heidelberg (2012)