Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

On Linear Time and Congruence in Channel-passing Calculi

Frédéric PESCHANSKI
University of Tokyo, Bunkyo-ku Hongo 7-3-1, 113-0033 Tokyo, Japan.
also with *LIP6, 8, rue du Capitaine Scott, 75015 Paris, France.*

Abstract. Process algebras such as CSP or the Pi-calculus are theories to reason about concurrent software. The Pi-calculus also introduces channel passing to address specific issues in mobility. Despite their similarity, the languages expose salient divergences at the formal level. CSP is built upon trace semantics while labeled transition systems and bisimulation are the privileged tools to discuss the Pi-calculus semantics. In this paper, we try to bring closer both approaches at the theoretical level by showing that proper trace semantics can be built upon the Pi-calculus. Moreover, by introducing locations, we obtain the same discriminative power for both the trace and bisimulation equivalences, in the particular case of early semantics. In a second part, we propose to develop the semantics of a slightly modified language directly in terms of traces. This language retains the full expressive power of the Pi-calculus and most notably supports channel passing. Interestingly, the resulting equivalence, obtained from late semantics, exhibits a nice congruence property over process expressions.

1 Introduction

CSP [1] and the Pi-calculus [2] are two well-developed theories to reason about concurrent software. Both approaches expose similar concepts, notably *interleaving semantics* and *synchronous communication*. They also diverge in some important areas. First of all, CSP is arguably richer and of higher-level than Pi. It provides various forms of parallel, choice and sequential constructs; it is also open to a rich set of datatypes. The Pi-calculus, on the other hand, only provides minimalistic forms of parallel and choice constructs; and it only supports the datatype of *names*. However, the Pi-calculus offers an interesting form of mobility through name/channel passing which is not present in CSP.

The divergence between the two language amplifies at the level of the underlying semantics. The Pi-calculus semantics are (generally) proposed through *labeled transition systems* whereas CSP mostly builds on top of *trace semantics*. It is our opinion that the latter, denoting set-based equivalences, are easier to deal with than the former, relying on more complex *bisimulation* equivalences. It is often stated, however, that *"Bisimulation equivalence discriminates more processes than trace semantics"* [3]. In this paper we show that mobile calculi close to the Pi-calculus can be analyzed *with full precision* using techniques similar to the ones developed within the CSP framework, namely *trace equivalence*. We hope that this preliminary step could help (re)conciliating the two approaches that have a lot in common. Strikingly enough, mobile channel types are now implemented in many CSP-based programming languages and implementations [4, 5, 6].

In section 2, we present a minimal language of concurrent sequential processes with a channel-passing form of mobility. The language is almost identical to the Pi-calculus. It can also be seen as a subset of CSP extended by channel passing. We further propose to characterize the operational semantics of this language, in two complementary ways. First, in section 3, we show that by inserting *locations* at well-chosen positions in terms (mostly

Table 1: A common syntax for mobile calculi

<*expr*> $::=$	0	inert
	| $($<*expr*>$)$	precedence
	| <*prefix*>$.$<*expr*>	sequence
	| $\nu(x)$ <*expr*>	restriction
	| $[x = y]$ <*expr*>	match
	| $[x \neq y]$ <*expr*>	mismatch
	| <*expr*> $+$ <*expr*>	choice
	| <*expr*> $\|\|$ <*expr*>	parallel
	| $*$<*expr*>	replication
<*prefix*> $::=$	$c!y$	output
	| $c?(x)$	input
	| τ	silent

around parallel and choice sub-terms), we can provide a linear-time form of Pi-calculus that retains all the features of its branching-time counterpart. We then prove that the LTS-based semantics can be equated to CSP-like trace semantics thus allowing reasoning based on both these complementary approaches. As a second contribution, developed in section 4, we take the counterpoint of developing CSP-like trace semantics directly, without having to construct a LTS first, on a slightly modified version of the language. Interestingly, this complementary development allows us to characterize a trace-based equivalence that enjoys a congruence property on process expressions which is not present in the "traditional" semantics for the Pi-calculus. A panorama of related work, a conclusion and a selection of references follow.

2 A Concurrent Language with Channel Passing

Several theories for concurrency have been developed during the past twenty years. Some of these theories, often designed as *process calculi* or *process algebras*, focus on *compositional semantics*. In such approaches, the behavior of high-level components are explained in terms of their lower-level sub-components and the way they interact. Successful members of this compositional family include the *Communicating Sequential Processes* (CSP) work introduced by Hoare [1], the *Calculus of Communicating Systems* (CCS) by Milner, further developed in its mobile counterpart the *Pi-calculus* [2], and the *Algebra of Communicating Processes* by Bergstra and Klop [7]. Empirically, one can say that all these languages allow to express the behavior of concurrent and sequential processes that can communicate with each other. In this section we give the minimalistic syntax of such a process calculus, which is almost identical to the Pi-calculus but with an additional – and intentional – CSP "feel".

In Table 1 we give the BNF syntax of the proposed language. Concurrent processes are expressed as terms separated by the parallel construct (noted here $||$ as in the CSP). The following example is a canonical communication system:

$$c!e.0 \mid\mid c?(x).P(x)$$

Following the process algebra terminology, we can distinguish the *prefix* and the *continuation* of both processes. For example, the prefix of the left-side process is $c!e$ (for *emitting the value* e *on the channel* c) and its continuation is 0 (for *terminating the process*). The prefix of the right-side process is $c?(x)$ which denotes the reception of a value through the channel c. The value is bound to name x in the continuation $P(x)$. So the rewrite we expect from such a term may be informally noted as follows:

$$c!e.0 \parallel c?(x).P(x) \rightarrow 0 \parallel P(e)$$

The 0 (inert) process of the Pi-calculus indicates a process termination. Since there is no sequential composition operator, it is not possible to synchronize on termination as in CSP. That is why trailing 0's may generally be omitted in terms, which might seems unfamiliar for CSP experts. Note that synchronization on termination can be encoded in the Pi-calculus, for example by introducing dedicated end channels.

The Pi-calculus also provides channel mobility, which we can illustrate on the following example:

$$c!e.0 \parallel c?(x).x!f.0 \parallel e?(y).Q(y)$$

The value e we pass through the channel c from the left-side process to the one in the middle is bound to x in the destination, which is then used as a channel. The expected rewrites for the previous example are as follows:

$$\begin{aligned} c!e.0 \parallel c?(x).x!f.0 \parallel e?(y).Q(y) &\rightarrow 0 \parallel e!f.0 \parallel e?(y).Q(y) \\ &\rightarrow 0 \parallel 0 \parallel Q(f) \end{aligned}$$

One may also employ *restriction* (ν construct) to encapsulate names/channels. Consider the following variant of the previous example:

$$\nu(\mathbf{e})(\mathbf{c!e.0}) \parallel c?(x).x!f.0 \parallel e?(y).Q(y)$$

Here, the name e is restricted to the scope of the left process. This means that the name e in the right process, which is a free name, is different from the bound occurrence of e in the left process. The rewrite we expect is as follows:

$$\nu(\mathbf{e})(\mathbf{c!e.0}) \parallel c?(x).x!f.0 \parallel e?(y).Q(y) \rightarrow \nu(\mathbf{e})\,(\mathbf{0} \parallel \mathbf{e!f.0}) \parallel e?(y).Q(y)$$

The highlighted scope modification is called a *scope extrusion*, it will be discussed in the next section. The difference with the previous example is that no more rewrite is possible since we cannot extend further the scope of e to the right-side process without renaming either the restricted e (inside the ν) or the free one (outside). Unlike CSP, the proposed language does not provide a recursion operator. But such an operator can be encoded using *replication* ($*$ operator) and communication (see [2]). Moreover, only the datatype of *names* (sometimes considered as channels) is available. Names can solely be tested for (in)equality using the *match* $[x = y]$ and *mismatch* $[x \neq y]$ operators.

As such, the proposed language, which is equivalent to the Pi-calculus, can be seen as a severe restriction if compared to the CSP language. Except for communication prefixes and generalized forms of parallel and choice operators, most of CSP is not supported. However, channel passing itself gives almost all its expressiveness to the Pi-calculus, as largely exemplified in the literature (starting from [2]).

3 Linear-time Channel Passing

Bisimulation and *trace equivalences* are often opposed in the literature. Even in introductory textbooks (such as [3]), the former is generally considered as a *finer* equivalence (*i.e.* it discriminates more processes) than the latter.

Consider the following process expressions:

$$a!e.c!f + b!e.d!g \text{ and } a!e.c!f + a!e.d!g$$

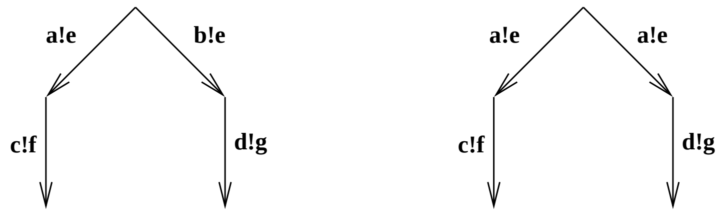

Figure 1: Branching vs linear-time tree structures

These expressions can be represented as tree structures (cf. figure 1). The left tree represents the behavior of the left process that can first perform either an output on a or an output on b, non-deterministically. If $a!e$ occurs, then $c!f$ follows, or if $b!e$ occurs, then $d!g$ follows. This *linear-time* behavior can be fully and naturally characterized with both trace and bisimulation equivalences. This is not true for the right-side process expression which exhibits *branching-time* semantics. In that case, the first action that can occur is always an output of a name e on a. Then, either $c!f$ or $d!g$ can follow. But the *choice* among which of these should occur is performed at the time $a!e$ occurs. Trace equivalence will equate such process to a less precise $a!e.(c!f + d!g)$. Intuitively, you cannot discriminate the "left" $a!e$ and the "right" $a!e$. Bisimulation equivalence, in contrast, properly discriminates here.

3.1 A Linearized Language with Locations

Trace equivalence can be extended with *stable failures* and *divergences* in order to obtain a more precise equivalence [1]. However, the resulting model does not integrate channel passing. In this paper, we thus propose a complementary approach which consists in changing slightly the language of section 2 so that trace semantics may be used even in the case of branching-time behaviors (such as the "splitting" action $a!e$ in our example).

Table 2: The syntax enriched with locations

<*expr*> $::=$	$[0]@l$	inert
	| $[($<*expr*>$)]@l$	precedence
	| $[$<*prefix*>.<*expr*>$]@l$	sequence
	| $\nu(x)$ $[$<*expr*>$]@l$	restriction
	| $[x = y]$ $[$<*expr*>$]@l$	match
	| $[x \neq y]$ $[$<*expr*>$]@l$	mismatch
	| $[$<*expr*>$]@l_1 + [$ <*expr*>$]@l_2$	choice ($l_1 \neq l_2$)
	| $[$<*expr*>$]@l_1 \parallel [$<*expr*>$]@l_2$	parallel($l_1 \neq l_2$)
	| $[*$<*expr*>$]@l$	replication
<*prefix*> $::=$	$c!y$	output
	| $c?(x)$	input
	| τ	silent

Our objective is to "linearize" the language and its semantics using *locations*. These are simple positional informations inserted into terms. The updated syntax is presented in Table 2. In this variant, a process P must be written $[P]@l$ which means intuitively that P "resides at" location l. Two processes P and Q, if composed in parallel, must reside at locations that are *different*. For instance, we write:

$$[P]@l_1 \parallel [Q]@l_2 \text{ with } l_1 \neq l_2$$

The only operation available on locations is to test their (in)equality, $l_1 = l_2$ or $l_1 \neq l_2$. It is important to note that locations are here abstract entities that do not only help at separating concurrent processes. In fact, locations must also be used to disambiguate choice alternatives. Suppose we decorate the branching trees with locations, as depicted on figure 2. In the localized syntax, the two processes may be written as follows:

$$[a!e.[c!f]@l_3]@l_1 + [b!e.[d!g]@l_4]@l_2 \text{ and } [a!e.[c!f]@l_3]@l_1 + [a!e.[d!g]@l_4]@l_2$$

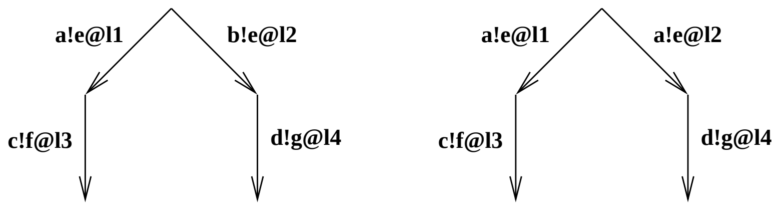

Figure 2: Linearized tree structures with locations

On the right-side of figure 2 we can see that locations solve the branching-time vs. linear-time issue. Now the choice of which among the "left" or "right" $a!e$ should occur is captured by the two distinct traces $a!e@l_1$ and $a!e@l_2$.

3.2 LTS-based Operational Semantics

We propose in this section the operational semantics of our channel-passing language using labeled transition systems. The presentation is brief since it is not the purpose of this paper to discuss precisely the LTS-based semantics, which are fully exposed in various documents (*e.g.* [8]). It is common to develop two levels in the semantics: *(i)* a *structural congruence* noted $\equiv$ (for now also noted $\equiv_{(2)}$) which equates "trivially" equivalent processes, and *(ii)* a set of *inference rules*, presented in the structural operational semantics style, from which the behavior of a process expression can be derived. We suppose the existence of two functions over processes $fn([P]@l)$ and $bn([P]@l)$ respectively denoting the *free names* and *bound names* of P. Names can be bound in either input prefix or restriction scopes. Note that locations are not considered as names and are thus "invisible" for these functions.

The structural congruence is defined by the axioms of Table 3. First, by rule (1), terms are alpha-convertible as in the lambda-calculus. Rules (2), (3) and (4) allow to simplify expressions by removing unnecessary locations. Parallel and choice operators expose common abelian monoid laws (commutativity, associativity and unit), as expressed by rules (5) to (10). Rules (11) and (12) are elimination rules for the match and mismatch operators. Rule (13) gives the semantics for replicated processes. Note that a fresh location must be introduced here. The most interesting rules are the last ones: they "implement" the combination of channel passing and restriction. For example, rule (14) states that a restricted name (a name in the scope of a ν) can see its scope *extruded* (if we read the equivalence from right-to-left) or conversely *intruded* within parallel constructs if there is no name conflict. More precisely, if read right-to-left, the rule states that a name x with restricted scope $[Q]@l_2$ can see its scope extruded to $[P]@l_1 \parallel [Q]@l_2$ if x is not a free name of P. The remaining rules (15) to (18) describe similar interactions with restricted names.

In the operational semantics defined in Table 4, the $(Struct)$ rule establishes the connection with the structural congruence. The (In), (Out) and (Tau) rules relate the prefixes of the languages to labels in the transition systems. The possible labels (or actions) are $a?u@l$ (input), $a!x@l$ (output), $a!\nu x@l$ (*bound output*) and $\tau@L$ (silent step, with possible multiple locations). Note that the labels are also localized, as suggested previously. We use the *early*

Table 3: Definition of structural congruence

(1) $[P]@l \equiv_{(2)} [Q]@l$ if P and Q are variants of alpha-conversion

(2) $[[P]@l_2]@l_1 \equiv_{(2)} [P]@l_2$
(3) $[[P]@l_2 \parallel [Q]@l_3]@l_1 \equiv_{(2)} [P]@l_2 \parallel [Q]@l_3$
(4) $[[P]@l_2 + [Q]@l_3]@l_1 \equiv_{(2)} [P]@l_2 + [Q]@l_3$

(5) $[P]@l_1 \parallel [Q]@l_2 \equiv_{(2)} [Q]@l_2 \parallel [P]@l_1$
(6) $([P]@l_1 \parallel [Q]@l_2) \parallel [R]@l_3 \equiv_{(2)} [P]@l_1 \parallel ([Q]@l_2 \parallel [R]@l_3)$
(7) $[P]@l_1 \parallel [0]@l_2 \equiv_{(2)} [P]@l_1$
(8) $[P]@l_1 + [Q]@l_2 \equiv_{(2)} [Q]@l_2 + [P]@l_1$
(9) $([P]@l_1 + [Q]@l_2) + [R]@l_3 \equiv_{(2)} [P]@l_1 + ([Q]@l_2 + [R]@l_3)$
(10) $[P]@l_1 + [0]@l_2 \equiv_{(2)} [P]@l_1$

(11) $[x = y][P]@l \equiv [P]@l$ if $x = y$, $[0]@l$ either
(12) $[x \neq y][P]@l \equiv [P]@l$ if $x \neq y$, $[0]@l$ either

(13) $[*P]@l_1 \equiv_{(2)} [*P]@l_1 \parallel [P]@l_2$ with $l_1 \neq l_2$

(14) $\nu(x)([P]@l_1 \parallel [Q]@l_2) \equiv [P]@l_1 \parallel \nu(x)[Q]@l_2$ if $x \notin fn(P)$
(15) $\nu(x)([P]@l_1 + [Q]@l_2) \equiv [P]@l_1 + \nu(x)[Q]@l_2$ if $x \notin fn(P)$
(16) $\nu(x)[u = v][P]@l \equiv [u = v]\nu(x)[P]@l$ if $x \neq u$ and $x \neq v$
(17) $\nu(x)[u \neq v][P]@l \equiv [u \neq v]\nu(x)[P]@l$ if $x \neq u$ and $x \neq v$
(18) $\nu(x)\nu(y)[P]@l \equiv \nu(y)\nu(x)[P]@l$

Table 4: Definition of transition rules

$$\frac{[P]@l \equiv [P']@l \quad [P]@l \xrightarrow{\alpha} [Q]@l' \quad [Q]@l' \equiv [Q']@l'}{[P']@l \xrightarrow{\alpha} [Q']@l'} \ (Struct)$$

$$\frac{[P]@l_1 \xrightarrow{\alpha} [P']@l'_1}{[P]@l_1 + [Q]@l_2 \xrightarrow{\alpha} [P']@l'_1} \ (Sum)$$

$$\frac{[P]@l_1 \xrightarrow{\alpha} [P']@l'_1 \quad bn(\alpha) \cap fn(Q) = \emptyset}{[P]@l_1 \parallel [Q]@l_2 \xrightarrow{\alpha} [P']@l'_1 \parallel [Q]@l_2} \ (Par)$$

$$\frac{}{[c!y.P]@l \xrightarrow{c!y@l} [P]@l} \ (Out) \quad \frac{}{[c?(x).P]@l \xrightarrow{c?u@l} [P]@l\{u/x\}} \ (In) \quad \frac{}{[\tau.P]@l \xrightarrow{\tau@l} [P]@l} \ (Tau)$$

$$\frac{[P]@l_1 \xrightarrow{a?u@l_1} [P']@l_1 \quad [Q]@l_2 \xrightarrow{a!u@l_2} [Q']@l_2}{[P]@l_1 \parallel [Q]@l_2 \xrightarrow{\tau@\{l_1,l_2\}} [P']@l_1 \parallel [Q']@l_2} \ (Com)$$

$$\frac{[P]@l_1 \xrightarrow{\alpha} [P']@l'_1 \quad x \notin \alpha}{\nu(x)[P]@l \xrightarrow{\alpha} \nu(x)[P']@l'_1} \ (Res) \qquad \frac{[P]@l_1 \xrightarrow{a!x@l_1} [P']@l_1 \quad a \neq x}{\nu(x)[P]@l \xrightarrow{a!\nu x@l_1} [P']@l_1} \ (Open)$$

semantics that perform substitutions directly through instantiations of the (In) inference rule. This rule relates a single input prefix $a?(x)$ to an infinite number of transitions/labels $a?u$ for every possible name u that can be received. The *late semantics* delay substitutions in instantiations of rule (Com). But in the early case, it only matches pairs of input and output actions to infer silent steps. The (Res) rule explains that restriction is preserved by transitions that do not refer to the restricted name. Finally, the $(Open)$ rule implements the communication of a restricted channel as a bound output action.

3.3 Behavioral Equivalences

The labeled transition systems defined previously denote a "natural" bisimulation equivalence that can be stated as follows:

Definition 1 *The* ***(strong early) bisimulation equivalence*** *of two processes* $[P]@l$ *and* $[Q]@l$ *is noted* $[P]@l \dot{\sim} [Q]@l$. *This is the* largest symmetric *relation such that:*

if $[P]@l \xrightarrow{\alpha} [P']@l'$, *then* $\exists Q'$ *such as* $[Q]@l \xrightarrow{\alpha} [Q']@l'$ *and* $[P']@l' \dot{\sim} [Q']@l'$

Intuitively, this relation equates the tree-like structure of process behaviors by ensuring that both equivalent terms covers the same possible transition paths. In comparison with more "traditional" definitions for *early bisimulation* (cf. [8]), our variant only differs by the introduction of locations. Only co-located process may be equated here.

As illustrated in [2] and many other works, such equivalence relations can be used in various ways to derive semantic laws about the language. However, the CSP community developed an arguably simpler equivalence in which processes are considered as equivalent if they expose the same traces, which is a natural equality on sets of sequences. Thanks to locations, we can reformulate the equivalence for the language proposed in section 2 in similar terms. We first need to define the notion of *trace*:

Definition 2 *The (LTS-based)* ***trace*** *of a process* $[P]@l$, *noted* $tr([P]@l)$, *is obtained inductively by the following rules:*

(i) $tr([0]@l) = \{\langle\rangle\}$
(ii) if $[P]@l \xrightarrow{\alpha} [P']@l'$ *then* $\langle\alpha\rangle.tr([P']@l') \subseteq tr([P]@l)$

We also define the ***trace prefixing*** *(. operator) on traces with the following rules:*

(i) $\langle\alpha\rangle.\{\langle\rangle\} = \langle\rangle.\{\langle\alpha\rangle\} = \{\langle\alpha\rangle\}$
(ii) $\langle\alpha\rangle.\{\langle\beta\rangle\} = \langle\alpha, \beta\rangle$
(iii) $\langle\alpha\rangle.A \cup B = \langle\alpha\rangle.A \cup \langle\alpha\rangle.B$

As discussed previously, the main issue with such connection between labeled transitions and traces is that one may not distinguish traces corresponding to different transitions sharing the same label. However, thanks to locations, we have the following important lemma:

Lemma 1 ***(Linear time)***
if $[P]@l \xrightarrow{\alpha} [P']@l'$ *and* $[P]@l \xrightarrow{\beta} [P'']@l''$ *with* $[P']@l' \not\dot{\sim} [P'']@l''$ *then* $\alpha \neq \beta$

The proof sketch for this lemma is as follows. First, only the choice and parallel operator can lead to multiple transitions for similar actions. This non-determinism stems from the interaction of the commutativity axiom for $+$ and $\|$, reflected in the semantics by the $(Struct)$ rule of Table 4, with the (Par), (Com) and (Sum) rules. We have to consider the cases of expression P corresponding to either $[Q]@l_1 \parallel [R]@l_2$ or $[S]@l_3 + [T]@l_4$. By definition of the localized syntax, we have $l_1 \neq l_2$ and $l_3 \neq l_4$. And from distinct locations we may only form distinct labels (*e.g.* $a?u@l_1 \neq a?u@l_2$) so that $\alpha \neq \beta$ □

We may now deduce an important reformulation for the equivalence of processes, under the form of the following theorem:

Theorem 1 ***(Trace and bisimulation equivalences coincide)***
$[P]@l \dot{\sim} [Q]@l \iff tr([P]@l) = tr([Q]@l)$

This theorem is a trivial consequence of lemma 1 on definitions 1 and 2 □

Note that this result does not translate to late semantics because of the more complex bisimulation equivalence they denote. However, we can formulate similar results on *observational* variants of the semantics, which consist in disabling the silent steps (the τ actions) when comparing processes. Given our two formulations for process equivalence, we have two ways to define such observational variant. In the Pi-calculus, *weak transitions* are introduced; they are noted $[P]@l \overset{\alpha}{\Longrightarrow} [P']@l'$. These transitions are transitive closures on τ steps around non-silent actions, which may be noted $[P]@l(\xrightarrow{\tau@L})^*[P_1]@l_1 \xrightarrow{\alpha} [P_2]@l_2(\xrightarrow{\tau@L'})^*[P']@l'$. From these weak semantics we can define a *weak (early) bisimulation equivalence*. But we can also propose an observational variant directly based on the trace-equivalence of theorem 1. We define a function otr of *observational traces* over processes as:

$$otr([P]@l) = \{S \in tr([P]@l) \mid \forall L,\ \tau@L \notin S\}$$

This is the set of traces $tr([P]@l)$ in which we filter out all silent steps. We could also provide a model closer to the CSP trace model by making otr closed over prefixes, which we note otr^* and define as follows:

$$\forall \mathcal{S}.\mathcal{T} \subseteq otr([P]@l),\ S \subseteq otr^*([P]@l)$$

We may finally introduce a *refinement operator* on processes, noted $\sqsubseteq$, as follows:

$$[P']@l \sqsubseteq [P]@l \iff otr^*([P']@l) \subseteq otr^*([P]@l)$$

Refinement techniques are largely exploited in trace-based semantics such as CSP. It is a very practical tool to reason in a top-down manner from specifications to actual implementations. Refinement is harder to define in terms of labeled transition systems [9]. This illustrates the advantage of the proposed model in which both semantics coincide.

4 Congruent Channel-passing

The equivalences defined in the previous sections raise important issues we propose to address now. First, there is some inconvenience in the fact that we defined trace equivalence upon LTS for process expressions. One has to build the transition system before being able to apply the trace model. Also, the equivalence relations of section 3 only coincide in early semantics, in which input prefixes denote impractical infinite branching. We would like to obtain similar results with late semantics. But more importantly, both the equivalences are *not preserved by input prefixes*. Consider the following example:

$$[d!e]@l_1 \parallel [c?(x)]@l_2 \dot{\sim} [d!e.[c?(x)]@l_2]@l_1 + [c?(x).[d!e]@l_1]@l_2 \quad (1)$$

This corresponds to the *interleaving semantics* as implemented by the semantics proposed in the previous section. This equivalence is true for bisimulation equivalence (see [8]) and also for trace-equivalence since both match. However, we can also prove that:

$$[b?(d).([d!e]@l_1 \parallel [c?(x)]@l_2)]@l \not\dot{\sim} [b?(d).([d!e.[c?(x)]@l_2]@l_1 + [c?(x).[d!e]@l_1]@l_2)]@l$$

Table 5: The revised syntax

<*expr*> $::=$	$[0]@l$	inert
	| $[($<*expr*>$)]@l$	precedence
	| $[$<*prefix*>.<*expr*>$]@l$	sequence
	| $[$<*expr*>$]@l_1 + [$ <*expr*>$]@l_2$	choice ($l_1 \neq l_2$)
	| $[$<*expr*>$]@l_1 \parallel [$<*expr*>$]@l_2$	parallel ($l_1 \neq l_2$)
	| $[*$<*expr*>$]@l$	replication
<*prefix*> $::=$	$c!y$	output
	| $c?(x)$	input
	| τ	silent
	| $\nu(c)$	restriction
	| $[x = y]$	match
	| $[x \neq y]$	mismatch

Here, the name d is bound through the input prefix $b?(d)$ and the equivalence is infirmed for the substitution $\{c/d\}$ (remember that input prefixes denote all the possible substitutions for d in early semantics). This tells that the equivalence relation is not a congruence on process expression. The proposed language is thus not truly compositional, at least with the proposed equivalences.

4.1 A Syntax with Substitutables and Fresh Occurrences

Let us consider again the issue raised by example (1) in the previous section. The problem is that we do not know in advance if the name d is susceptible to be captured by a binder (either an input prefix or a restriction) in some context. In the variant of the semantics we define in this section, if we write $d?(x)$ or $d!y$, then both the names d and y are *not substitutable* at all. In that case, the equivalence becomes trivially a congruence but by the same occasion, we lose the ability to communicate or restrict names and channels ! To circumvent this limitation, we modify the language syntax in two ways, as described on Table 5.

First, we remove the syntax rules for restriction and match/mismatch (in Table 2) and introduce dedicated prefixes. We do this because we need specific rules for these constructs. The second modification involves two new kinds of names that are defined as follows:

Definition 3 *The* ***substitutable*** *of a name x, noted λx, is a name that can be bound by either an input prefix $d?(x)$ (for any channel d) or a restriction $\nu(x)$. A* ***fresh occurrence*** *of x, noted νx (or νy provided that νy is fresh), is a name used for substitution of λx in a restriction scope $\nu(x)$. Moreover, νx must be a fresh name, different from any other name.*

In the new syntax, the prefix $d!x$ expresses an output on a channel d that cannot be received or restricted (*i.e.* it cannot be bound) in any context. More precisely, if the term appears in a context where the name d is bound, then d remains a free occurrence. On the other hand, in the prefix $\lambda d!x$, if d is bound (*e.g.* in either $c?(d).\lambda d!x$ or $\nu(d).\lambda d!x$), then λd may be accordingly substituted in the sub-term.

The *freshness property* of a fresh occurrence νx states in particular that distinct restrictions must result in substitutions by distinct *and unique* names. For example, two restrictions on the same name x, both noted $\nu(x)$, will lead to substitutions by different names, such as νx_1 and νx_2 with $\nu x_1 \neq \nu x_2$. A sufficient condition is to consider fresh occurrences as globally unique names (which is a reasonable assumption in practice).

Table 6: The trace semantics of the Pi-calculus with substitutables and fresh names

$$
\begin{array}{ll}
(1) & tr([P]@l) = tr([Q]@l) \text{ if } [P]@l \equiv_{(2)} [Q]@l \\
\\
(2) & tr([0]@l) = \langle\rangle \\
(3) & tr([\tau.P]@l) = \langle\tau@l\rangle.tr([P]@l) \\
(4) & tr([\xi?(x).P]@l) = \langle\xi?(x)@l\rangle.tr([P]@l) \\
(5) & tr([\xi!\gamma.P]@l) = \langle\xi!\gamma@l\rangle.tr([P]@l) \\
(6) & tr([\nu(x).P]@l) = tr([P]@l)\{\nu x/\lambda x\} \text{ with } \nu x \text{ fresh} \\
(7) & tr([[\xi = \gamma].P]@l) = tr([P]@l) \text{ if } \xi = \gamma, \langle\rangle \text{ either} \\
(8) & tr([[\xi \neq \gamma].P]@l) = tr([P]@l) \text{ if } \xi \neq \gamma, \langle\rangle \text{ either} \\
\\
(9) & tr([P]@l_1 + [Q]@l_2) = tr([P]@l_1) \cup tr([Q]@l_2) \\
(10) & tr([P]@l_1 \parallel [Q]@l_2) = tr([P]@l_1) \otimes tr([Q]@l_2)
\end{array}
$$

4.2 Revised Operational Semantics

We may now define the operational semantics for the extension of the language proposed in the previous section. We define these semantics *directly* in term of traces, through the function tr over processes as defined inductively on Table 6.

The first rule (1) relates the trace function tr to a structural congruence noted $\equiv_{(2)}$. This is the equivalence $\equiv$ (cf. Table 3) in which we remove all rules related to the "old" restriction and match/mismatch constructs (*i.e.* we remove the rules (11),(12), and (14) to (18) in Table 3). The rules (2) to (8) give the semantics of the language prefixes. The occurrences ξ and γ of names in these rules can represent either regular names x, substitutables λx, or fresh occurrences νx. This is to reduce the number of cases where the difference among name categories does not intervene. Of particular interest are rules (4) and (6) respectively for input prefix and restriction. In contrast to the LTS trace semantics given previously, the traces for input prefixes does not account for all (infinite) possible substitutions. The rule for restriction $\nu(x)$ involves the substitution of λx by a fresh occurrence νx. The substitution over process expressions trivially extends to traces, we thus have for any substitution σ, $(tr([P]@l))\sigma = tr(([P]@l)\sigma)$. We finally propose a rule for the choice operator, which is the union of traces for both the branches, and a rule for parallel which is obtained through *trace product and interleaving* defined as follows:

Definition 4 *The **trace product** of two process traces $\mathcal{T}_1$ and $\mathcal{T}_2$ is noted $\mathcal{T}_1 \otimes \mathcal{T}_2$. It produces the **trace interleaving**, noted $\mathcal{T}_1 \oplus \mathcal{T}_2$, in which communication steps are correctly substituted. It is defined inductively as follows:*

$\mathcal{T}_1 \otimes \{\langle\rangle\} = \mathcal{T}_1$ and $\{\langle\rangle\} \otimes \mathcal{T}_2 = \mathcal{T}_2$ and $\{\langle\rangle\} \otimes \{\langle\rangle\} = \{\langle\rangle\}$ or
$\mathcal{T} = \mathcal{T}_1 \otimes \mathcal{T}_2$ with $\forall\langle\alpha\rangle.\mathcal{T}_1' \subseteq \mathcal{T}_1$ and $\forall\langle\beta\rangle.\mathcal{T}_2' \subseteq \mathcal{T}_2$ then
(i) if $\alpha = c!\gamma@l_1$ or $\alpha = \lambda d!\gamma@l_1$ and $\beta = c?(x)@l_2$ or $\beta = \lambda e?(x)@l_2$
then $\langle\tau@\{l_1,l_2\}\rangle.(\mathcal{T}_1' \otimes \mathcal{T}_2'\{\gamma/\lambda x\}) \cup (\mathcal{T}_1 \oplus \mathcal{T}_2) \subseteq \mathcal{T}$
(ii) if $\alpha = \nu c!\gamma@l_1$ and $\beta = \nu c?(x)@l_2$
then $\langle\tau@\{l_1,l_2\}\rangle.(\mathcal{T}_1' \otimes \mathcal{T}_2'\{\gamma/\lambda x\}) \subseteq \mathcal{T}$
(iii) or $\mathcal{T}_1 \oplus \mathcal{T}_2 = \langle\alpha\rangle.(\mathcal{T}_1' \otimes \mathcal{T}_2) \cup \langle\beta\rangle.(\mathcal{T}_1 \otimes \mathcal{T}_2') \subseteq \mathcal{T}$

The definition above is relatively complex since it mixes trace interleaving (case *(iii)*) and communications with substitution. Let us illustrate these semantics on some basic examples, starting with the canonical communication system written as follows:

$$[c!e]@l_1 \parallel [c?(x).P(\lambda x)]@l_2$$

Note that we now use a substitutable λx to denote the possible substitution of the bound name x. In contrast to the language and semantics of section 2, the plain name x cannot be substituted anymore. The semantics of the previous example are traces defined as follows:

$$\mathcal{T} = tr([c!e]@l_1 \parallel [c?(x).P(\lambda x)]@l_2) = tr([c!e]@l_1) \otimes tr([c?(x).P(\lambda x)]@l_2)$$

From the rules of Table 6 we have:

$$\left[\begin{array}{l} tr([c!e]@l_1) = \langle c!e@l_1 \rangle \\ tr([c?(x).P(\lambda x)]@l_2) = \langle c?(x)@l_2 \rangle.tr([P(\lambda x)]@l_2) \end{array} \right.$$

These traces can only be matched by case *(i)* in the definition of $\otimes$, from which we infer:

$$\begin{array}{l} \mathcal{T} = \langle \tau@\{l_1,l_2\}\rangle.\,(\{\langle\rangle\} \otimes tr([P(\lambda x)]@l_2)\{e/\lambda x\}) \\ \qquad \cup \{\langle c!e@l_1\rangle\} \oplus (\langle c?(x)@l_2\rangle.tr([P(\lambda x)]@l_2)) \end{array}$$

This may be simplified by applying the interleaving operator $\oplus$ as follows:

$$\begin{array}{l} \mathcal{T} = \langle \tau@\{l_1,l_2\}\rangle.\,(\{\langle\rangle\} \otimes tr([P(\lambda x)]@l_2)\{e/\lambda x\}) \\ \qquad \cup \langle c!e@l_1, c?(x)@l_2\rangle.tr([P(\lambda x)]@l_2) \cup \langle c?(x)@l_2, c!e@l_1\rangle.tr([P(\lambda x)]@l_2) \end{array}$$

Finally, we end up with the following semantics:

$$\begin{array}{rl} \mathcal{T} = & \langle \tau@\{l_1,l_2\}\rangle.tr([P(e)]@l_2) \\ & \cup \langle c!e@l_1, c?(x)@l_2\rangle.tr([P(\lambda x)]@l_2) \\ & \cup \langle c?(x)@l_2, c!e@l_1\rangle.tr([P(\lambda x)]@l_2) \end{array}$$

Intuitively, the meaning of the program is that either the communication depicted occurs or we take into account the fact that someone else could resolve the communication on channel c through composition. Consider now the variant with a restriction on c as follows:

$$[\nu(c).\,([\lambda c!e]@l_1 \parallel [\lambda c?(x).(\lambda x)]@l_2)]@l$$

We use the substitutable λc because c itself cannot be restricted since it cannot be substituted. From rule (6) of Table 6, we derive the trace semantics $\mathcal{T}$ of this example as follows:

$$\begin{array}{l} \mathcal{T} = tr([\lambda c!e]@l_1 \parallel [\lambda c?(x).P(\lambda x)]@l_2)\{\nu c/\lambda c\} \text{ with } \nu c \text{ fresh} \\ \quad = tr([\nu c!e]@l_1 \parallel [\nu c?(x).P(\lambda x)]@l_2) \end{array}$$

We are now in the case *(ii)* of the definition for the trace product, from which we obtain:

$$\mathcal{T} = \langle \tau@\{l_1,l_2\}\rangle.\,(\{\langle\rangle\} \otimes tr([P(\lambda x)]@l_2)\{e/\lambda x\})$$

And finally: $\mathcal{T} = \langle \tau@\{l_1,l_2\}\rangle.tr([P(e)]@l_2)$

We finally illustrate channel passing using the following example:

$$[c!e]@l_1 \parallel [c?(x).\lambda x!f]@l_2 \parallel [e?(y).P(\lambda y)]@l_3$$

The associativity of $\parallel$ allows us to treat the left or right composition in any order. Moreover, there are many possible executions in which the two communications we are interested in do not occur. We will thus only give the sub-traces $\mathcal{T}'$ of this expression in which the communication occur. For the left-side process which is an instance of the first example with $P = \lambda x!f$, we are left with the sequence : $\langle \tau@\{l_1,l_2\}, e!f@l_2\rangle$. We see here that the passed name e is used as a channel in a subsequent trace. When combined with the traces of the right-side process, we can deduce the trace $\langle \tau@\{l_1,l_2\}, \tau@\{l_2,l_3\}\rangle.tr([P(f)]@l_3)$.

4.3 Revised Trace-equivalence

As illustrated in the previous section, the proposed language, if not equivalent, retains the major features of the Pi-calculus such as channel-passing. Moreover, we integrate locations and as such preserve the results of section 3. Thereupon, we can deduce from the previous semantics, directly expressed as traces, a precise *trace equivalence* as follows:

Definition 5 *The* ***trace equivalence*** *of two processes* $[P]@l$ *and* $[Q]@l$, *which is noted* $[P]@l \sim [Q]@l$, *is defined as the equality of the implied traces:*
$[P]@l \sim [Q]@l \iff tr([P]@l) = tr([Q]@l)$

In comparison to the equivalence proposed in section 3, this revised version is based on late semantics. Moreover, we can state a very important lemma about the revised equivalence:

Lemma 2 ***($\sim$ is preserved by input prefixes)***
if $[P]@l \sim [Q]@l$ *then* $\forall\xi, \forall\gamma,\ \xi?(\gamma).[P]@l \sim \xi?(\gamma).[Q]@l$

We have $[P]@l \sim [Q]@l \iff tr([P]@l) = tr([Q]@l)$ by definition of trace equivalence. From the semantics for input prefixes, as defined by rule (4) in Table 6, we also have $tr(\xi?(\gamma).[P]@l) = \langle\xi?(\gamma)@l\rangle.tr([P]@l)$, which trivially equates $\langle\xi?(\gamma)@l\rangle.tr([Q]@l)$ □

Now we can move on to the generalization of the congruence property of the proposed trace equivalence. We first need to establish the following lemma:

Lemma 3 ***($\otimes$ left-preserves equality)***
Let $\mathcal{T}_1$, $\mathcal{T}_2$ *and* $\mathcal{T}_3$ *be traces, we then have* $\mathcal{T}_1 = \mathcal{T}_2 \Longrightarrow \mathcal{T}_1 \otimes \mathcal{T}_3 = \mathcal{T}_2 \otimes \mathcal{T}_3$.

The proof sketch for this lemma is by structural induction on traces, following the scheme for the definition of $\otimes$ (cf. definition 4). For the base case, in which $\mathcal{T}_1 = \mathcal{T}_2 = \langle\rangle$, the lemma is trivially verified since we have by definition $\langle\rangle \otimes \mathcal{T}_3 = \mathcal{T}_3$. Now we are left with the three inductive cases *(i)*, *(ii)* and *(iii)* of definition 4. In each case, we only have to look at the left trace since we only prove the preservation of equality on the left side of the $\otimes$ operator (the right-side case is handled through commutativity of $\|$). In the inductive cases, we define $\mathcal{T}_i = \bigcup_{\alpha,\mathcal{T}_i'}\langle\alpha\rangle.\mathcal{T}_i'$ $(i \in \{1,2,3\})$. By the hypothesis of induction, we have $\forall\mathcal{U},\ \mathcal{T}_1' \otimes \mathcal{U} = \mathcal{T}_2' \otimes \mathcal{U}$. So, in particular, $\forall\gamma, \lambda x,\ \mathcal{T}_1' \otimes \mathcal{T}_3'\{\gamma/\lambda x\} = \mathcal{T}_2' \otimes \mathcal{T}_3'\{\gamma/\lambda x\}$. Finally, equality on sets is trivially preserved by both the union and prefixing relations (and also by $\oplus$ which is a composition of union and prefixing) so that we have $\langle\tau@\{l_1,l_2\}\rangle.((\mathcal{T}_1' \otimes \mathcal{T}_3'\{\gamma/\lambda x\}) \cup (\mathcal{T}_1 \oplus \mathcal{T}_3)) = \langle\tau@\{l_1,l_2\}\rangle.((\mathcal{T}_2' \otimes \mathcal{T}_3'\{\gamma/\lambda x\}) \cup (\mathcal{T}_2 \oplus \mathcal{T}_3))$. This covers the three inductive cases and thus concludes the proof □

Theorem 2 ***($\sim$ is a congruence on processes)***
For all terms $\mathcal{E}$ *in which* $[P]@l$ *appears, if* $[P]@l \sim [Q]@l$, *then* $\mathcal{E} \sim \mathcal{E}\{[P]@l/[Q]@l\}$

The proof for this theorem is by structural induction on the language constructors. First, structural congruence is a congruence by definition. Moreover, we can generalize lemma 2 for input, output and silent prefixes. For such a prefix α and a process P, we have the same basic fact that $tr([\alpha.P]@l) = \langle\alpha@l\rangle.tr([P]@l)$. Match and mismatch may reduce to the empty trace, but this does not impact the congruence property. The restriction semantics also involve substitutions by fresh (and free) names. Let σ and σ' be two substitutions on the same domain. Through alpha-conversion, we have $tr([P]@l) = tr([Q]@l) \Longrightarrow tr([P]@l)\sigma = tr([Q]@l)\sigma'$ which is enough to cover the congruence property of the restriction prefixes.

Now consider the case of the choice operator. Suppose two processes P and P' such as $[P]@l_1 \sim [P']@l_1$, which means $tr([P]@l_1) = tr([P']@l_1)$. The semantics of $[P]@l_1+[Q]@l_2$ are $tr([P]@l_1 + [Q]@l_2) = tr([P]@l_1) \cup tr([Q]@l_2)$. From simple properties of set union, we have $tr([P]@l_1) \cup tr([Q]@l_2) = tr([P']@l_1) \cup tr([Q]@l_2)$ since $tr([P]@l_1) = tr([P']@l_1)$. So we have $[P]@l_1 + [Q]@l_2 \sim [P']@l_1 + [Q]@l_2$. We can follow a similar scheme to prove the congruence property for the parallel operator. But for this we have to prove that if $tr([P]@l_1) = tr([P'@l_1])$ then $tr([P]@l_1) \otimes tr([Q]@l_2) = tr([P']@l_1) \otimes tr([Q]@l_2)$. This is true by lemma 3. □

The simple but fundamental corollary of theorem 2 is that unlike most Pi-calculus variants, the language we propose in this paper is truly *compositional*. This removes the burden of reasoning based on contextual informations, unavoidable in most behavioral equivalences on Pi-calculus processes [8].

5 Related Work

To our knowledge, there exist few works that aim at bringing CSP and the Pi-calculus closer at the formal level. Yet, many CSP-based programming languages and implementations such as Kroc/Linux [4], Icarus [5] or JCSP Network Edition [6] introduce channel-passing extensions. These works do not discuss, though, the influence of the new constructs on the formal semantics of the language. We hope that our propositions could be used as foundations for such mobile variants.

Testing theories can be used to compare trace-based and bisimulation-based equivalences for both static and channel-passing calculi. It is usual to associate bisimulation and trace models with respectively *may* and *must testing equivalences* [10]. In this work, we show that for a particular language, a variant of the pi-calculus with explicit locations and early semantics, both definitions coincide. There also exist transition-based models for CSP, for example in [11]. But channel passing is not discussed.

Various process algebras and programming language semantics have been recast and compared in the *unifying theories of programming* by Hoare and Jifeng [12]. Given a single denotational tool, namely relations and associated operators, most programming language concepts can be expressed and compared in this unifying framework. In contrast, our work focuses on operational semantics and discusses slight extensions making the departing worlds of bisimulation and trace equivalence coincide. Interestingly enough, trace equivalence on its own seems sufficient to characterize the category of language proposed in the paper. In the light of Hoare and Jifeng's unification work, it is probable that more profound coincidences exist. The fact that channel-passing may be expressed in more denotational terms, however, remains an open (and intriguing) question.

In this paper, we define a trace model that is an extension of a subset of the traditional CSP semantics (as defined in [1]) without prefix closure. Moreover, we integrate the silent steps to model the generalized choice operator of the Pi-calculus. In order to adapt the very practical refinement techniques of CSP, we must exhibit the prefix closure of the traces in which silent steps are filtered out. It is thankfully very easy to derive such semantics, noted otr^* in the paper. Stable failures and divergences are important tools we have not yet taken into account in the proposed model. The reason is that we capture the branching-time semantics and thus provide a precise equivalence we would like to investigate furthermore.

Explicit locations are introduced for a variant of the CCS language in [13] to discuss fairness models. Locations in this work are much more precise than ours. An order relation is introduced whereas we only rely on (in)equality. We may also remark that in both approaches, the frontier between safety and liveness properties is not as strongly delimited as usual. Moreover, the expressive power of localized variants does not seem to suffer much from the addition of locations. While this is not needed for the results presented, we may relax the semantics by proposing a weakest form of equivalence in which processes need not to be co-located in order to be compared. An interesting weaker variant is to only maintain (in)equality of locations among traces. For example the trace $\{\langle \alpha @ l_1, \beta @ l_2 \rangle\}$ with $l_1 \neq l_2$ would be equivalent to $\{\langle \alpha @ l_3, \beta @ l_4 \rangle\}$ *iff* $l_3 \neq l_4$.

There are many discussions on the basic syntax for Pi-calculus terms, and there exist in fact many variants of the language. In this paper, most of the language constructors are prefixes. In [8], the fact that most constructs of the Pi-calculus can be expressed in purely operational ways is discussed. For example, the restriction operator can be seen as an action through dedicated open and close rules. But it is then difficult to produce normal forms of terms, which is an important step for most proofs on congruence properties. It is notable that normalization is less prominent when the semantics are expressed in terms of traces. In fact, we do not rely on any normal form in our proposition.

From all characterizations of Pi-calculus semantics, our proposition is probably closest to the *open bisimulation* by Sangiorgi [14]. This is to our knowledge the only fully compositional characterization of the Pi-calculus semantics. To achieve this, open bisimulation integrates the quantification over all substitutions in its definition. In our proposition, the quantification occurs directly in the semantics, because substitutable channels can always be used for communication. This leads to an arguably simpler characterization of behavioral equivalence. Moreover, the mismatch operator remains available whereas no simple definition of open bisimulation with mismatch is known [8].

6 Conclusion

In this paper, we tried to illustrate that there were not always such *well delimited* frontiers between process algebras that are often considered as intrinsically dissimilar, for example CSP on one side and the Pi-calculus on the other side. As a matter of fact, we think that the activity of bringing things closer, by opposition of exhibiting differences, should be also of primary importance.

The difference between linear-time/branching-time semantics, the may/must testing dichotomy, or even the lack of precision of trace equivalence if compared to bisimulation, all these disappear with the surprisingly simple adjunction of *locations*. This does not seem to involve unbearable loss in terms of expressive power; though this issue should be further investigated.

Likewise, the often accepted idea that a channel passing language retaining all the features of the Pi-calculus may not be fully compositional seems, at least, unsatisfactory. Our proposition, once again, only involves a thin extension, namely the *substitutable* names, in order to obtain the most expected congruence property. Additionally, locations and substitutables may be treated separately. A motivating future work would be to define a congruent form of Pi-calculus, characterized through labeled transition systems and bisimulation exclusively.

Our main motivation, however, is to bring closer the CSP semantics and the Pi-calculus language. And there is a lot more to investigate in that direction. First, we should provide higher-level abstractions, most notably a richer set of datatypes. Distribution and explicit manipulation of locations ought to be explored as well. Termination of process is also somewhat neglected by Pi-calculus experts. On the semantic side, further experiments with the CSP-like refinement model are needed. For now, the translation to the proposed channel-passing calculus has been but roughly sketched. Later on, we wish that the proposed model could be used as foundations for practical languages and tools for the development of mobile systems with an emphasis on safety and assisted verification.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[2] Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[3] Wan Fokkink. *Introduction to Process Algebra*. EATCS. Springer, 2000.

[4] F.R.M.Barnes and P.H.Welch. Prioritised Dynamic Communicating and Mobile Processes. *IEE Proceedings-Software*, 150(2):121–136, April 2003.

[5] David May and Henk Muller. Copying, moving and borrowing semantics. In *Communicating Process Architectures (CPA'2001)*. IOS Press, 2001.

[6] Quickstone Technologies. JCSP Network Edition. `http://www.quickstone.com/xcsp/jcspnetworkedition`.

[7] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.

[8] Joachim Parrow. *Handbook of Process Algebra*, chapter An Introduction to the Pi-calculus, pages 479–543. Elsevier, 2001.

[9] C. J. Frige and J. J. Zic. A notion of (weak) refinement for CCS. In *Fourth Australasian Refinement Workshop (ARW'95)*, April 1995.

[10] Michele Boreale and Rocco De Nicola. Testing equivalence for mobile processes. *Inf. Comput.*, 120(2):279–303, 1995.

[11] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[12] C. A. R. Hoare and He Jifeng. *Unification Theories of Programming*. Prentice Hall, 1998.

[13] Gerardo Costa and Colin Stirling. Weak and strong fairness in CCS. *Inf. Comput.*, 73(3):207–244, 1987.

[14] D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996.