Process, Systems and Tests: Three Layers in Concurrent Computation

Clément Aubert

School of Computer and Cyber Sciences, Augusta University, Georgia, USA
caubert@augusta.edu

Daniele Varacca

LACL, Université Paris-Est Créteil, France
daniele.varacca@u-pec.fr

Abstract

In this position paper, we would like to offer a new template to study process algebras for concurrent computation. We believe our template will clarify the distinction that is too often left implicit between user and programmer, and that it enlightens pre-existing issues that have been running across process algebras as diverse as the calculus of communicating systems (CCS), the π-calculus—also in its distributed version—or mobile ambients. Our distinction starts by subdividing the notion of process itself in three conceptually separated entities, and shapes future improvements—both technically and organizationally—as well as it captures recent and diverse progresses in process algebras.

While the role of what can be observed and the subtleties in the definitions of congruences have been intensively studied, the fact that *not all the comparisons serve the same purpose and should not be made in the same context* is curiously left over, or at least not formally discussed. We argue that this blind spot comes from the under-specification of contexts—environments in which the comparison takes place—that supposedly 'stay the same' no matter the nature of the process, who is testing it, or for what. We illustrate our statement with the 'usual' concurrent languages, but also back it up with λ-calculus and existing implementations of concurrent languages as well.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Semantics and reasoning

Keywords and phrases Concurrency, Process Semantics, Equivalences, Process Algebra, Testing Equivalences

Funding The authors were partly supported by the ANR-14-CE25-0005 ELICA & the ANR-11-INSE-0007 REVER when planting the first seeds of this proposal.

1 Introduction

A Foreword on the Nature of This Proposal

In this paper, we would like to discuss without technicality, but precisely,[1] a question that we believe is too often left implicit in the design of process algebras: *what is a context?*, or, more precisely, *what notions of context should be used?*.

In the study of process calculi, contextual relations play a central role. The basic idea is that to study the behaviour of a 'process', one needs to make it interact with different possible 'environments', and observe the outcomes. Environments are often represented in the syntax in the form of contexts surrounding the term. But the role played by contexts is, in our opinion, never entirely clear, as it actually serves multiple purposes and is modified 'on the fly' to fit changing needs, as we will expose in this paper. We propose a way of clarifying

[1] We privileged diverse and heterogeneous sources, to which we refer precisely, but it is hard to document *the absence* of an answer in the literature.

the notions of context, and show that having co-existing notions of contexts retrospectively legitimates and explains recurring choices, and supports a rigorous guideline to separate the development of a process from its testing.

Maybe in the mind of most of the experts in the formal study of concurrent systems is our proposal too obvious to even being discussed. However, if this is the case, we believe that it is 'folklore' and has never been written, and that since we were not at *that* 'seminar at Columbia in 1976',[2] we are to remain in darkness.

We believe the Computer Science Logic conference is the ideal place to reach the principal actors in the field, and to either being proven wrong or—hopefully—impact some researchers and encourage them to adopt our frame to clarify their reflections and future developments. We believe in any case that ignoring or downplaying the distinctions we would like to stress may have repeatedly caused confusions in the past—at least in the authors' minds—, and risk to continue doing so if not addressed.

Plan

In the following, we would like to start by having a quick look at the 'godfather' of multiple concurrent calculi, the λ-calculus (Sect. 2), which will help us to shape a distinction between *processes*, *systems*, and *tests* (Sect. 3). With this notion in mind, we will be able to revisit the definitions of contextual relations (Sect. 4), and illustrate some of the exception to the apparent monolithic status of contexts (Sect. 5). Sect. 6 constitutes the core of our proposal, and will hopefully serve as a guideline to define future concurrent languages and to revisit the existing ones. We illustrate some of the benefits by showing how numerous and heterogeneous process algebras can fit our frame (Sect. 7). We finally sketch how this line of research could be pushed further to produce interesting new results (Sect. 8), and conclude with some remarks on context closure (Sect. 9.1), the so-called 'context lemma' (Sect. 9.2), and the history of this paper (Sect. 9.3).

2 A Foreword on λ-Calculus

Theoretical languages for concurrent computation often take λ-calculus as a model or a comparison basis[3]: one wish concurrent computation could have a language as mature and as stable as this functional language,[4] and 'achieve the same economy' [44, p. 86]. And indeed, pure λ-calculus (i.e. without types or additions like probabilistic sum [26] or quantum capacities [58, 62]) is a reasonnable [4], Turing-complete and elegant language, that requires only a couple of operators (literally: application and abstraction), one reduction rule (β-reduction) and one equivalence relation (α-equivalence) to produce a rich and meaningful theory. This calculus is sometimes seen as an idealized target language for functional programming languages.

Since most terms do not reduce as they are, to study their behaviour, one needs first to make them interact with an 'environment', represented by the notion of 'context'. However, even in such a simple set-up, the notion of context is troublesome. Contexts are generally defined as 'term[s] with some holes' [7, p. 29, 2.1.18], that we prefer to call *slots* and that

[2] To re-use in our setting Paul Taylor's witty comment published at `http://math.andrej.com/2012/09/28/substitution-is-pullback/`.

[3] That the 'λ-calculus is to sequential programs what the π-calculus is to concurrent programs' is a common trope [19, 63], and other process algebras share similar lineages.

[4] This common belief actually needs some revision [3], but that's not our point here.

we denote $[\square]$. Under this apparent simplicity, a lot of care is needed when manipulating them, as having multiple slots or not being careful when defining what is meant by 'filling a slot' can lead to e.g. losing confluence [10, pp. 40–41, Example 2.2.1], and as those issues persist even in the presence of a typing system [31]. Furthermore, definitions and theorems that use contexts frequently impose some restrictions on the contexts considered, to exclude e.g. contexts like $(\lambda x.y)[\square]$ that simply 'throw away' the term put in the slot in one step of β-reduction. Following those considerations, contexts often come in two flavors, depending on the nature of the term under consideration:

If the term is closed (i.e. without free variables), then a context is essentially a series of arguments to feed the term. This observation is used when defining e.g. *solvable terms* [7, p. 171, 8.3.1 and p. 416, 16.2.1].

If the term is open (i.e. with free variables), then a context is a *Böhm transformation* [7, p. 246, 10.3.3], which is equivalent [7, p. 246, 10.3.4] to a series of abstractions followed by a series of applications, and sometimes called 'head context'.

Adding features to the λ-calculus certainly does not restore the supposed purity or unicity of the concept of context, but actually distances it even further from being simply 'a term with a slot'. For instance, contexts are narrowed down to term context [62, p. 1126] and surface context [26, pp. 4, 10] for respectively quantum and probabilistic λ-calculus, to 'tame' the conceptual power of contexts. In resource sensitive extensions of the λ-calculus, the quest for full abstraction even led to more drastic separation, as λ-terms were split between terms and tests [14], a separation that was later on naturally extended to contexts [13, p. 73, Figure 2.4].

All this variety was introduced after the 2000's formal studies of contexts was undertaken [10, 11, 31], which led to the observation that treating contexts 'merely as a notation [...] hinders any formal reasoning[, while treating them] as first-class objects [allows] to gain control over variable capturing and, more generally, "communication" between a context and expressions to be put into its holes' [11, p. 29]. It seems ironic that λ-calculists took inspiration from a concurrent language to split their syntax in two right at its core [14, p. 97], or to study formally the *communication* between a context and its expression, while concurrent languages sometimes tried to keep the 'purity' and indistinguishability of their contexts.[5]

As it is the case for the λ-calculus, to study concurrent calculi like CCS or the π-calculus one has to represent an interaction with an environment by means of a notion of context. But the status of contexts in concurrent calculi is even more unsettling when one note that, while contexts are of interest mainly for open terms in lambda calculus, *all* terms need a pertinent notion of context in concurrent systems to be tested and observed. Our contribution starts by questioning this distinction between open and closed terms in process calculi, and will advocate for the emergence of multiple notions of contexts.

3 Processes, Systems and Tests

As in the λ-calculus, most concurrent calculi make a distinction between open and closed terms. For instance, the distributed π-calculus [32] implements a distinction between closed

[5] I.e., 'a context *is* a term, period' seems to have been the motto, with some exceptions—sometimes acknowledged, sometimes not—that we will discuss mainly in Sect. 5.

terms (called processes [32, p. 14]) and open terms, based on binding operators (input and recursion).

Most of the time, and since the origin of the calculus of communicating systems, the theory starts by considering only programs—'closed behaviour expression[s], i.e. ones with no free variable' [42, p. 73]—when comparing terms, as—exactly like in λ-calculus—they correspond to self-sufficient, well-rounded programs: it is generally agreed upon that open terms should not be released 'into the wild', as they are not able to remain in control of their internal variables, to prevent e.g. undesirable or uncontrolled interferences. Additionally, closed terms are also the only ones to have a *reduction semantics*, which means that they can evolve without interacting with the environment.

However, in concurrent calculi, the central notions of binders and of variables have been changing, and still seem today sometimes 'up in the air'. For instance, in the original CCS, restriction was not a binder [42, p. 68], and by 'refusing to admit channels as entities distinct from agents' [43, p. 16] and defining two different notions of scopes [43, p. 18], everything was set-up to produce a long and recurring confusion as to what a 'closed' term meant in CCS. In the original definition of π-calculus [46, 47], there is no notion of closed terms, as every (input) binding on a channel introduce a new and free occurrence of a variable. However, the language they build upon—ECCS [25]—made this distinction clear.

Once again in an attempt to mimic the 'economy' [44, p. 86] of λ-calculus, but also taking inspiration from the claimed 'monotheism' of the actor model [33], different notions such as values, variables, or channels have been united under the common terminology of 'names'. However, it seems that a distinction between those notions always needs to be carefully re-introduced when discussing technically the language [5, p. 258, Remark 493] or possible implementations [28]. Finally, let us note that extensions of π-calculus can sometimes have different binders, as e.g. output binders are binding in the private π-calculus [52, p. 113].

In the λ-calculus, being closed is what makes a term 'ready to be executed in an external environment'. But in concurrent calculi, being a closed term is often not enough, as it is routine to exclude e.g. terms with un-guarded operators like sum [22, p. 416] or recursion [43, p. 166]. In our opinion, the right distinction is not about binders of free variables, but about the role played by the syntactic objects in the theory. As 'being closed' is **1.** not always well-defined, or at least changing, **2.** sometimes not the only condition, we would like to use the slightly more generic adjectives *incomplete* and *complete*.

Once a notion of 'being complete' is defined, process algebras generally study terms by comparing one another, using equivalences or preorders. To obtain those, one generally studies the behaviour of terms, by completing them if needed, and then by executing them and observing the outcome. In order to execute a complete term, one must often put it in a larger environment, where some standard observations can be performed. Often, the environment is essentially made of another system composed in parallel with the one studied. The observations are generally carried out thanks to predicates on the execution ('terminates', 'emitted the barb a', etc.) and constitute the outcome needed to study the behaviour of the term. Because the environment is generally tweaked to improve the likeliness to observe such or such behaviour, we would like to think of them as tests that the observed systems has to pass.

With this informal discussion in mind, we now propose a terminology that we would like to use for the rest of the paper:

Processes are 'partial' programs, still under development, written by the programmer; they are sometimes called 'open terms', and correspond to *incomplete terms*. They would be called code fragments in standard programming.

Systems are 'configured processes', ready to be executed in any external environment: they are sometimes called 'closed terms', and correspond to *complete terms*. They would be functions shipped with a library in standard programming.

Tests are defined using contexts, and aims at executing and testing systems. They would be `main` functions calling a library or an API in standard programming.

Our terminology is close to the one used e.g. in aDpi [32, Chapter 5] or mobile ambients [40, Table 1], which use a distinction between processes and systems.

In another expressive analogy, one could see processes as 'source code', systems as 'compiled code', while tests would correspond to 'operating systems', i.e. platforms where compiled code can be executed and tested.

In the literature of process algebra, the term 'process' is commonly used to denote these three layers, which may generate confusion. We believe this usage comes from a strong desire to keep the three layers uniform, using the same name, operators and rules: but this principle is actually constantly dented (Sect. 5), for reasons we will expose below.

4 Contextual Relations

Comparing terms is the core of the study of concurrent languages, and it is made using relations, equivalences, pre-congruences and congruences. Generally, and similarly to what is done in λ-calculus, a comparison is deemed of interest only if its results are valid in every possible context. Formally, it is generally said that an equivalence relation $\mathcal{R}$ is a congruence if it is closed by context, i.e. if for all P, Q (open or closed) terms, $(P, Q) \in \mathcal{R}$ implies that *for all context* $C[\square]$, $(C[P], C[Q]) \in \mathcal{R}$ holds.[6] However, we argue that there are two different perspectives in the use of congruence, depending on whether one would like to know

1. if a process can be substituted for another, in any possible completion,
2. if a system will behave as another in any environment

Roughly speaking, **1.** is to be understood from a programmer's point of view (i.e. '*can I replace this piece of code by this other one and still obtain the same behavior?*'). The usage at **2.** should be understood from the point of view of the users in an external environment, or, in a security setting, of an attacker (i.e. '*will they be able to tell whenever a program or the other is running?*'). This is exemplified by barbed congruences [45, Definition 8][38, Definition 2.1.4], which closes by context a reduction-closed relation used to observe 'barbs'. This congruence is often taken to be *the* 'reference behavioural equivalence' [38, p. 4], as it observes the interface of processes, i.e. on which channels they can interact over the time and in parallel. If the user is considered as an attacker, this usage should be understood as a security test in a hostile environment, and used to test security protocols. Systems are generally tested here, but we would like to argue that *processes need to have congruent comparison tools as well, and that those may require a different notion of context.*

Of course, in both cases, the definition of what the can be *observed* is fundamental, and we come back to this aspect in Sect. 6. We would like to first stress that in most process algebras, those two perspectives are already present, but the distinction is rarely, if at all, made explicit. It is indeed harder to shape this distinction as operators used to build the

[6] In some cases, the additional requirement that terms in the relation needs to be similar up to uniform substitution is added [34], and sometimes [53, p. 516, Definition 2], only the closure by substitution—seen as a particular kind of context—is required.

terms and the ones used to build the contexts used to observe the terms are generally the same—at least in principle, as we will see in the next section that the motto '*A context is a term with a slot*' is often only a *façade*. The aim of our work is to illustrate that the set of operators to construct terms and the one used to test them often are the same only *by accident*, and not by design, and to illustrate some of the benefits that would result from a conceptual clarification of this issue.

5 Context Have a Pre-Existing Condition

We would like in this section to discuss the 'pre-existing' condition of contexts, to illustrate that *they have been modified and altered with no clear explanation nor justification* in multiple places. Let us consider five different treatments of contexts before trying to make general statements.

In the Calculus of Communicating Systems, notions as central as contextual bisimulation [5, pp. 223-224, Definition 421] and barbed equivalence [5, p. 224, Definition 424] considers only *static* contexts [5, p. 223, Definition 420], which are composed only of parallel composition with arbitrary term and restriction. As the author of those notes puts it himself, 'the rules of the bisimulation game may be hard to justify [and] contextual bisimulation [...] is more natural' [5, p. 227], but there is no justification—other than technical, i.e. because they 'they persist after a transition' [5, p. 223]—as to *why* should only some contexts being considered in contextual equivalences.

In the π-calculus, contexts are defined liberally [22, p. 19, Definition 1.2.1], but still excludes contexts like e.g. $[\square] + 0$ right from the beginning. Congruences and structural congruences [22, p. 19, Definitions 1.2.2, 1.2.3] are then defined using this notion of context, as e.g. strong barbed congruence [22, p. 59, Definition 2.1.17]. Other notions, like strong barbed equivalence [22, p. 62, Definition 2.1.20], are shown to be a non-input congruence [22, p. 63, Lemma 2.1.24], which is a notion relying on contexts that prevent the slot from occurring under an input prefix [22, p. 62, Definition 2.1.22]. In other words, two notions of contexts and of congruences co-exist generally in π-calculus, but '[i]t is difficult to give rational arguments as to why one of these relations is more reasonable than the other.' [32, p. 245]

In the distributed π-calculus, contexts are restricted right from the beginning to particular operators [32, Defintion 2.6]. Then, relations are contextual if they are preserved by static contexts [32, Defintion 2.6], which contains only parallel composition with arbitrary terms and name binding.[7] Static operators are deemed 'sufficient for our purpose' [32, p. 37] and static contexts only are considered '[t]o keep life simple' [32, p. 38], but no further justification is given.

In the semantic theories for processes, one difficulty is that the class of formal theories restricted to 'reduction contexts' [34, p. 448] still fall short on providing a satisfactory 'formulation of semantic theories for processes which does not rely on the notion of observables or convergence'. Hence, the authors have to furthermore restrict the class of terms to *insensitive* terms [34, p. 450] to obtain a notion of *generic reduction* [34, p. 451] that allows a satisfactory definition of what a sound theory is [34, p. 452]. Insensitive terms are essentially the collection of terms that do not interact with contexts [34,

[7] Such contexts have varying name, e.g. 'configuration context' [36, p. 375]. They have been studied under the name *harness* in the ambient calculus [30, p. 372].

p. 451, Proposition 3.15], an analogue to the so-called *genericity Lemma* [7, p. 374, Proposition 14.3.24] from λ-calculuus. Here, contexts are restricted by duality: insensitive terms are terms that will *not* interact with the context in which they are placed, and that need to be equated by sound theories.

Accross calculi, a notion of 'closing context'—that emerged from λ-calculus [5, p. 85]— can be found in typed versions of the π-calculus [22, p. 479], in mobile ambient [63, p. 134], or in the fusion calculus [39, p. 6]. Aslo known as 'completing context' [18, p. 466], those contexts—that we prefer to name *instantiating contexts*—are parametric in a term, the idea being that such context would 'close'—we prefer to say 'complete'—the term under study.

Let us try to extract some general principles from this short survey.

It seems to us that contexts are **1.** in appearance given access to the same operators than terms, **2.** sometimes deemed to be 'un-reasonable', without always a clear justification, **3.** shrunken by need, to bypass some of the difficulties they raise, or to preserve some notions, **4.** sometimes picked by the term itself.

Additionally, in all those cases, contexts are given access only to a strict subsets of operators, or restricted to contexts with particular behavior, but *never extended*. If we consider that contexts are the main tool to test the equivalence of systems, then why should the testing facilities always have access to fewer tools than the programmer? What reason is there not to *extend* the set of tools, of contexts, or simply take it to be orthogonal? The method we sketch below allows and actually encourages such nuances, and would acknowledge the restrictions we just discussed.

6 Acknowledging Contexts

We argue that concurrent languages would benefit from being articulated as follows right from their conception:

I. Define processes The first step is to select a set of operators called *construction operators*. The programmer will use those to write terms, and they should be expressive, easy to combine, and with light constraints. To ease their usage, a 'meta-syntax' can be used, something that is generally represented by the structural equivalence.[8]

II. Define deployment criteria The programmer should define how can a process become a system: it is a series of restrictions that can include condition on the binding of variables, the presence or absence of some construction operators at top-level, and even the addition of *deployment operators*, marking that the process is ready to be deployed in an external environment. Having a set of operators for systems that restrict,[9] expand or intersect with the set of construction operators is perfectly acceptable, and should include instruction on how to obtain a system from a process, or how to compose them.

III. Define tests The last step requires to define **1.** a set of observables, i.e. a function from systems to a subset of a set of atomic proposition (like 'emits barb a', 'terminates',

[8] Structural equivalence generally uses the most liberal notion of context to justify a syntactic manipulation, and would benefit in simply being postulated and accepted as a syntactic sugar that can be used 'anywhere'.

[9] Even if it may seem weird to *remove* operators before deploying a process, we believe that this is generally what happen when one suddenly decide that recursion or sum should be guarded when terms are compared.

'contains recursion operator', etc.), **2.** a notion of context, that should come with its own set of *testing operators* and reduction rules.

Tests would be key in defining notions of congruence, that would be 'reduction-closed', 'observational' 'contextually-closed' relations. Note that we propose a refined version of how a concurrent language is generally defined along two axis: **1.** every step allows the introduction of novel operators, **2.** multiple notions of systems or tests can and should co-exist in the same process algebra, one being targeted to e.g. programmers, and another for e.g. users.

7 Addressing Existing Issues

In the literature, processes and systems often have the same structure as tests and are—at least on the surface of it—not distinguished from what they are supposed to test.

Our frame captures and clarifies some of the choices, debates, improvements and explanations that have been proposed in process algebras, as we would like to stress below. Indeed, we believe some of the most fundamental-yet-unspoken questions—that have been disturbing the monolithicity of process algebras—can be treated in our setting, that can also offer a new take on recent improvements.

Clarifying the confusion between observations and contexts At its origin, the barb was a predicate [45, p. 690], whose definition was purely syntactic. Probably inspired by the notion of observer for testing equivalences [23, p. 91], an alternative definition was made in terms of parallel composition with a tester process [38, p. 10, Definition 2.1.3]. This example perfectly illustrates how the set of observables and the notion of context are inter-dependent, and that tests should always come with a definition of observable *and* a notion of context.

Justifying the special treatment made to 'silent' transition It is routine to define relations (often called 'weak') that ignore silent (a.k.a. τ) transitions, seen as 'internal'. This sort of transitions was dubbed 'unobservable internal activity' [32, p. 6] and sometimes opposed to 'externally observable actions' [20, p. 230]. While we agree that '[t]his abstraction from internal differences is essential for any tractable theory of processes' [43, p. 3], we would also like to stress that *both can and should be accommodated*, and that *'internal' transition should be treated as invisible* to the user*, but should still be accessible* to the programmer.

Sometimes is asked the question 'to what extent should one identify processes differing only in their internal or silent actions?' [9, p. 6], but the question is treated as a property of the process algebra,[10] and not as something that can *internally* be tuned as needed, and in that particular example, this distinction is later on *simply discarded* [9, p. 6]! We argue that the answer to that question is '*it depends who is asking!*'.

Admiting the co-existence of multiple comparisions The discussion on τ-transitions resonates with a long debate on which notion of behavioral relation is the most 'reasonable', and still recently, a textbook can conclude a brief overview of this issue by 'hop[ing] that [they] have provided enough information to [their] readers so that they can draw their own conclusions on this long-standing debate.' [20, p. 160] We firmly believe that the best conclusion is that different relations match different needs, and that there is no 'one size fits all' relation for the need of programmers and users. Of course, comparing multiple

[10] More precisely, as a property of concurrency semantics.

relations is an interesting and needed task [27, 61], but one should also state clearly that multiple comparison tools can and should co-exist, and such vision will be encapsulated by the division we are proposing.

Embracing a feared distinction The distinction between our notions of processes and systems is rampant in the literature, but too often feared, as if it was a parenthesis that needed to be closed to restore some supposedly needed purity and uniformity of the syntax. A good example is probably given by mobile ambients [40]. The authors start with a two-level syntax that distinguishes between processes and systems [40, p. 966]. Processes have access to strictly more constructors than systems [40, p. 967, Table 1], that are supposed to hide the threads of computation [40, p. 965]. A notion of *system context* is then introduced—as a restriction of arbitrary contexts—and discussed, and two different ways for relations to be preserved by context are defined [40, p. 969, Definiton 2.2].

The authors even extend further the syntax for processes with a special $\circ$ operator [40, p. 971, Definition 3.1], and note that the equivalences studied will not consider this additional constructor: we can see at work the distinction we sketched, were operators are added and removed based on different needs, and where the language needs not to be monolithic. The authors furthermore introduce two different reduction barbed congruences [40, p. 969, Definition 2.4]—one for systems, and one for processes, with different notions of contexts—but later on prove that they coincide on systems [40, p 989, Theorem 6.10]. It seems to us that the distinction between processes and systems was essentially introduced for technical reasons, but that re-unifying the syntax—or at least prove that the systems do not do significantly more than the processes— was a clear goal right from the start. We believe this distinction could have been embraced in a framework similar to the one we sketched: while retaining the interesting results already proven, maintaining this two-level syntax would allow to make a clearer distinction between the user's and the programmer's roles and interests, and assert that, sometimes, systems can and *should* do more than processes, and can be compared using different tools.

Keeping on extending contexts We are not the first one to argue that constructors can and should be added to calculii to access better discriminatory power, but without necessarily changing the 'original' language. The mismatch operator, for instance, has a similar feeling: it is needed to provide 'reasonable' testing equivalences [12, p. 280], and is considered across languages [2, p. 24] to provide finer-grained equivalences. For technical reasons [22, p. 13], this operator is generally not part of the 'core' of π-calculus, but is added *by need* to obtain better equivalences: we defend a liberal use of this fruitful technics, by making a clear separation between the construction operators—added for their expressivity—and the testing operators—that are here to improve the testing capacities.

Seeing conservative extensions as different completion strategies We could consider conservative extensions of processes algebras as different completion strategies for the same construction operators. For instance, reversible [37] or timed [65] extensions of CCS could be seen as different completion strategies—different conditions for a process to become a system—for the same class of processes, inspired from the usual CCS syntax [5, Chapter 28.1]. Those completion strategies would be suited for different needs, as one could e.g. complete a CSS process as a RCCS [16] system to test for relations such as hereditary history-preserving bisimulation [6], and then complete it with time markers as a safety-critical system. This would correspond to having multiple compilation, or deployment, strategies, based on the need, similar to 'debug' and 'real-time'[11] versions of the same piece of software.

Modelling 'real-life' experience Of course, one always have to be vigilant when claiming that an abstract set-up is a better model of 'real-life usage' than another, as **1.** since the purpose is to benefit from the power of abstraction, one always want to have some distance with implementation meanders, **2.** it is sometimes tempting to tweak reality to have it better fit the model. However, we believe that our frame would account for common aspects of software development in a useful way, as for instance **1.** Every compiled language is *de facto* embodying a distinction between complete (i.e. compiled, closed) programs and incomplete (i.e. open) source code. **2.** Every object-oriented language makes a strong distinction between *private* and *public* parts of their classes, making a system-wide distinction between the programmers' and the users' needs and tools. **3.** It is better to account for different usages and phases of development that are standard in software development.

Getting finer-grained typing disciplines The development of typing systems for concurrent languages is a notoriously difficult topic. Some results in π-calculus have been solidified [22, Part III], but diverse difficulties remain. To name a few, the co-existence of multiple systems for e.g. session types [60], the difficulty to tie them precisely to other type systems as Linear Logic [15], or the doubts about how to match, exactly, the 'proof-as-program' paradigm in a concurrent setting [8], makes this area of research active and diverse. The ultimate goal seems to find a typing system that would accommodate different uses and scenarios that are not necessarily comparable. Using our proposal, one could imagine easing this process by developing two different typing systems, one aimed at programmer—to track errors and ease the composition of processes– and one aimed at users—to track security leaks or perform user-input validation. Once again, the distinction between batteries of tests could also allow to develop a type system for processes only, and to erase the information when completing the process— similar to Java's arrays of parameterized types [51, pp. 253–258], that checks the typing discipline at compilation time, but not at run-time—or reciprocally to type only systems.

While this series of examples and references illustrates how our proposal could clarify pre-existing distinctions, we would like to stress that **1.** nothing prevents from collapsing our distinction when it is not needed, **2.** *additional progresses* could be made using it, and we would like to suggest possible future directions in the next section.

8 Exploiting Context Awareness

We would like to sketch below some possible exploitations of our frame that we believe could benefit the study and expressivity of some popular concurrent languages.

For CCS, we sketch below two possible improvements, the second being related to security.

Testing for auto-concurrency Auto-concurrency (a.k.a. auto-parallelism) is when a system have two different transitions—leading to different states—labeled with the same action [49, p. 391, Definition 5]. Systems with auto-concurrency do not fare well with

[11] In the spirit of Debian's DebugPackage which enables easy generation of stack traces for any package, or of the `CONFIG_PREEMPT_RT` patch that converts a kernel into a real-time micro-kernel: both uses the same source code as their 'casual' versions.

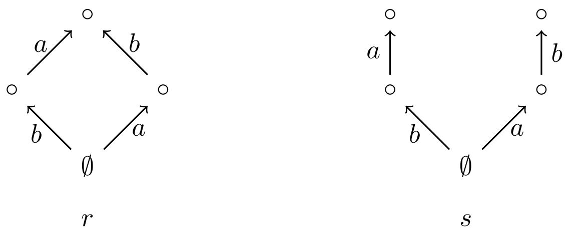

Figure 1 The trouble with auto-concurrency

back and forth bisimulation, and are sometimes excluded as non-valid terms [24, p. 155] or simply not considered in particular models [50, p. 531].

Consider for instance the labeled configuration structures (a.k.a. stable family [64, Section 3.1]) r and s of Fig. 1: non-interleaving models of concurrency [56] distinguishes between r and s, as a 'true concurrency model' would do. But back and forth bisimulations cannot discriminate between r and s if $a = b$.

While not being able to distinguish between those two terms may make sense from an 'external' point of view, we argue that a programmer should have access to an internal test that could answer the question '*Can this process perform two barbs with the same label at the same time?*'. Such an observation would allow to distinguish between e.g. $!a.P \mid !a.P$ and $!a.P$, that are generally taken to be bisimilar, and would re-integrate auto-concurrent systems—that are, after all, fairly useful—in the realm of comparable systems.

Representing man-in-the-middle attack One could imagine adding to the testing operators an operator $\nabla a[\square]$, which would forbid the process that is composed with it to act silently on a. This novel operator would add the possibility, for the environment, to 'spy' on a determined channel, as if the environment was controlling (at least part of) the router of the tested system. One could then reduce 'normally' below the $\nabla a.[\square]$ operator if the channel is still secure:

$$\nabla a(b.Q \mid \bar{b}P \mid R) \to^{\tau} \nabla a(Q \mid P \mid R) \qquad \text{(If } a \neq b\text{)}$$

But in the case where $a = b$, the environment would decide if it forwards the communication, prevents it, or alters it. Adding this operator to the set of testing operators would for instance open up the possibility of interpreting $\nu a([\square])$ as a 'secure' operation, and would open up the possibility of studying relations $\sim$ that could include e.g.

$$\nabla a(\nu a(P|Q) \sim \nabla a(\nu b(P[a/b]|Q[a/b])) \qquad \text{(For } b \notin \text{fn}(P) \cup \text{fn}(Q)\text{)}$$
$$\nu a(\nabla a(P|Q)) \sim \nabla a(P|Q) \qquad \text{(Uselessness of securising a hacked communication)}$$

In π-calculus, all tests are instantiating contexts (in the sense that the term tested needs to be either already closed, or to be closed by the context), and all instantiating contexts use only construction operators. This situation can be depicted as Situation A in Fig. 2 We believe the picture could be much more general, with tests having potentially access to *more constructors*, and not needing to be instantiating—in the sense that completion can be different from closedness—, so that we could move to Situation B in Fig. 2. While we believe this remark applies to most of the process algebras we have discussed so far, it is particularly salient in π-calculus, where the match and mismatch operators have been used 'to internalize a lot of meta theory' [29, p. 57], standing 'inside' the 'Construction operators' circle while most authors seem to agree that they would prefer not to add it to

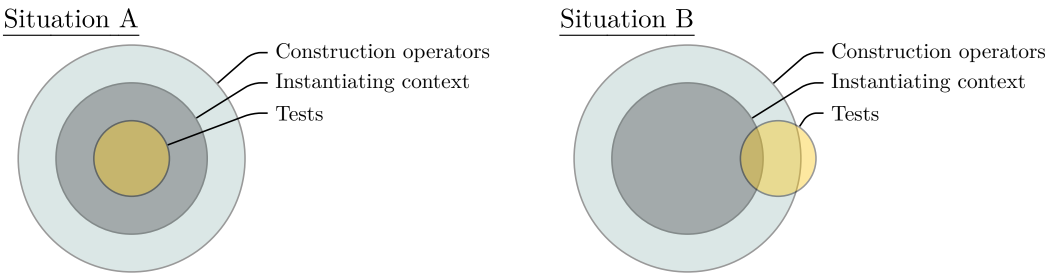

Figure 2 Opening up the testing capacities of π-calculus

the internals of the language.[12] It should also be noted that the mismatch operator—in its 'intuitionistic' version—furthermore 'tried to escape the realm of instantiating contexts' by being tightly connected [35] to *quasi-open bissimilarities* [21, p. 300, Definition 6], which is a subtle variation on how substitutions can be applied by context to the terms being tested.

Having a notion of 'being complete' not requiring to being closed could be useful when representing distributed programming, where 'one often wants to send a piece of code to a remote site and execute it there. [...] [T]his feature will greatly enhance the expressive power of distributed programming[by] send[ing] an open term and to make the necessary binding at the remote site.' [31, p. 250] We believe that maintaining the possibility of testing 'partially closed'—but still complete—terms would enable a more theoretical understanding of distributed programming and remote compilation.

Distributed π-calculus, in our opinion, could explore the possible differences between two parallelisms: between threads in the same process—in the Unix sense—and between units of computation. Such a distinction could be rephrased thanks to two parallel operators, one on processes and the other on systems. Such a distinction would allow to observationally distinguish between e.g. the execution of a program with multiple threads on a dual-core computer with the execution of two programs on two single-core computers.

For cryptographic protocols, we could imagine representing encryption of data as a special context $\mathcal{E}[\Box]$ that would transform a process P into an encrypted system $\mathcal{E}[P]$, and make it un-executable unless 'plugged' in an environment $\mathcal{D}[\Box]$ that could decrypt it. This could allow the applied π-calculus [1] to become more expressive and to be treated as a decoration of the pure π-calculus more effectively. This could also, as the authors wish, make 'the formalization of attackers as contexts [...] continue to play a role in the analysis of security protocols' [1, p. 35].

9 Concluding Remarks

We would like to conclude by offering a new light on the technical issue of the definition of barbed congruences, by stressing *why* we believe context lemmas may not be such the 'grail' that it sometimes seems to be, and come back to our motivations.

---

[12] To be more precise: while 'most occurences of matching can be encoded by parallel composition [...,] mismatching cannot be encoded in the original π-calculus' [54, p. 526], which makes it somehow suspicious.

9.1 Playing With Contexts

The interesting question of *when* to use contexts when comparing terms [22, pp. 116–117, Section 2.4.4] raises a technical question that we believe our analysis puts under a different perspective. Essentially, the question is whether the congruences under study should being *defined* as congruences (e.g. like reduction-closed barbed congruence [22, p. 116]), or being defined in two steps, i.e. as the contextual closure of a pre-existing relation (as e.g. strong barbed congruence [22, p. 61, Definition 2.1.17], which is the contextual closure of strong barbed bisimilarity [22, p. 57, Definition 2.1.7])?

Indeed, bisimulations can be presented as an ‘interaction game’ [59] generally played as follows: **1.** Pick an environment for both terms (i.e., complete them, embed them in the same context), **2.** Have them ‘play’ (i.e. have them try to match each other’s step). But a more dynamic version of the game let picking an environment *be part of the game*, so that each process can not only pick the next step, *but also in which environment it needs to be performed.* This version of the game, called ‘dynamic observational congruence’ [48] provides a better software modularity and reusability, as it allows to study the similarity of terms that can be re-configured ‘on the fly’. Embedding the contexts in the definitions of the relations is a strategy that was also used to obtain behavioral characterization of theories [34, p. 455, Proposition 3.24], and that corresponds to open bisimilarities [17, p. 77, Proposition 3.12].[13]

Those two approaches have been extensively compared and analyzed, but to our knowledge they rarely co-exist: it is as if an author—or the process algebra under study—had to ‘take a side’ and dogmatically use and defend one of the two solution. It seems to us that both approaches are equally valid, *provided we acknowledge they play different roles.* This question *when are the terms completed?* can be rephrased as *who is completing them?* if one acknowledges a potential separation between the programmer and the environment. In this frame, moving from the *static* definition of congruence to *dynamic* one would corresponds to going from situation A to situation B in Fig. 3.

This illustrates two aspects that we are worth highlighting:

1. Playing on the variation ‘*should I complete the terms before or during their comparison?*’ is not simply a technical question, but reflects a choice between two different situations equally interesting.
2. This choice can appeal to different notions of environment, completions and tests: for instance, while completing a term before testing it (Situation A) may indeed be needed when the environment is perceived as an external deployment plat-form, it makes less sense if we think of the environment as part of the development workflow, in charge of providing feedback to the programmer (Situation B).

If completion is seen as compilation, this opens up the possibility of studying how the bindings performed *by the user*, on *their* particular set-up, during a *remote* compilation, can alter a program. One can then compare different relations—some obtained by observing the release of the source code, some obtained by observing the release of binaries—to get a better, fuller, picture of the program.

[13] And it should be noted, once again, that *quasi-open bissimilarities* [21, p. 300, Definition 6] opens up the possibility of having multiple notions of completion competing.

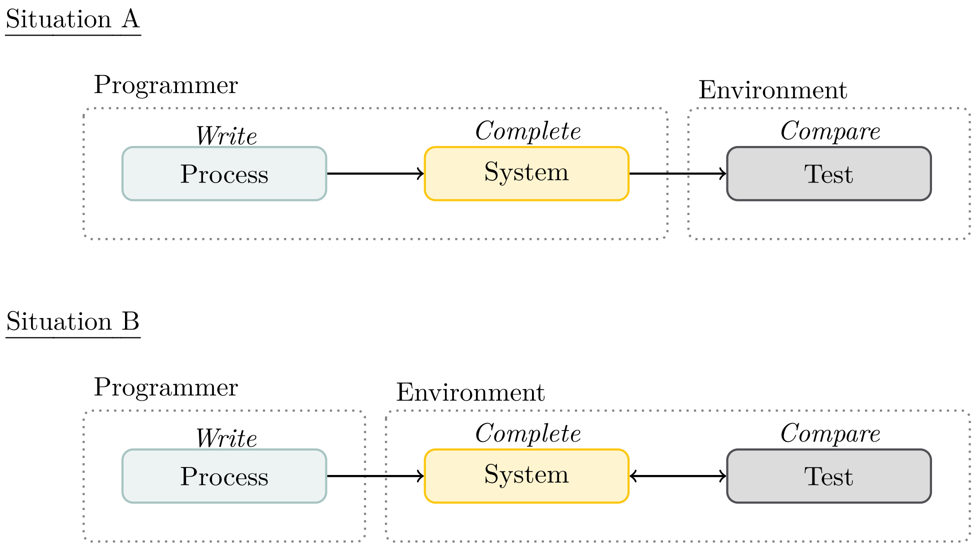

Figure 3 Distinguishing between completing strategies

9.2 Context Lemmas

What is generally refereed to as *the* context lemma[14] is actually a series of results stating that considering all the operators when constructing the context for a congruence may not be needed. For instance, it is equivalent to define the barbed congruence [22, p. 95, Definition 2.4.5] as the closure of barbed bisimilarity under all context, or only under contexts of the form $[\square]\sigma \mid P$ for all substitution σ and term P. In its first version [55, p. 432, Lemma 5.2.2], this lemma had additional requirements e.g. on sorting contexts, but the core idea is always the same: '*there is no need to consider all contexts to determine if a relation is a congruence, you can consider only contexts of a particular form*'. A study of what a 'generic' context lemma [57, p. 1534, Theorem 5.12] may look like was undertaken for higher-order abstract syntax, but is unfortunately of little use for process algebras.

The 'flip side' of the context lemma is what we would like to call the 'anti-context pragmatism': whenever a particular type of operator or context prevents a relation from being a congruence, it is tempting to simply exclude it. For instance, contexts like $[\square] + 0$ were omitted—like discussed in Sect. 5—to define the barbed congruence of π-calculus, or contexts were restricted to what is called harnesses in the mobile ambients calculus [30] before proving such results. As strong bisimulation [53, p. 514, Definition 1] is not preserved by input prefix [53, p. 515, Proposition 4] but is by all the other operators, it is sometimes tempting to simply remove input prefix from the set of constructors allowed at top-level in contexts, which is what is done with non-input contexts [22, p. 62, Definition 2.1.22], and then to establish a context lemma for this limited notion of context.

Taken together, those two remarks produce a strange impression: while it is mathematically elegant and interesting to prove that weaker conditions are enough to satisfy an interesting property, it seems to us that this result is sometimes 'forced' into the process algebra by beforehand excluding the operators that would not fit, hence producing a result that is not only weaker, but also somehow artificial, or even tautological. Furthermore the

[14] At least, in process algebra, as the meaning of this lemma in e.g. λ-calculus is different [41, p. 6].

criteria of 'not adding any discriminating power' should not be a positive criterion when deciding if a context should be added to the algebra: on the opposite, one would want contexts to *increase* the discriminating power—as for the mismatch operator—, and not to 'conform' to what substitution and parallel composition have already decided.

9.3 Embracing the Diversity

Before daring to submit a non-technical paper, we tried to conceive a technical construction that could convey our ideas. In particular we tried to build a syntactic (even categorical) meta-theory of processes, systems and tests. We wanted to define congruences in this meta-theory, and to answer the following question: what could be the minimal requirements on contexts and operators to prove a generic form of context lemma for concurrent languages?

However, as the technical work unfolded, we realized that the definitions of contexts, observations, and operators, were so deeply interwoven that it was nearly impossible to extract any general or useful principle. Context lemmas use specific features of languages, in a narrow sense,[15] and we could not yet find a unifying framework. This also suggests that context lemmas are often *fit* for particular process algebras *by chance*, and dependent to the extreme of the language considered, for no deep reasons.

This was also liberating, as all the nuances of languages we had been fighting against started to form a regular pattern: every single language we considered exhibited (at least parts of) the structure we sketched in the present proposal. Furthermore, our framework was a good lense to read and answer some of the un-spoken questions suggested in the margin or the footnotes—but rarely upfront—of the many research papers, lecture notes and books we consulted. So, even without mathematical proofs, we consider this contribution a good way of stirring the community, and to question the traditional wisdom.

It seems indeed to us that there is nothing but benefits in altering the notion of context, as it is actually routine to do so, and that clearly stating the variation(s) used will only improve the expressiveness of the testing capacities and the clarity of the exposition. It is a common trope to observe the immense variety of process calculi, and to sometimes wish there could be a common formalism to capture them all—to this end, *the* π-calculus is often considered an excellent candidate. Acknowledging this diversity is already being one step ahead of the λ-calculus—that keeps forgetting that there is more than one λ-calculus, depending on the evaluation strategy and on features such as sharing [3]—and this proposal encourages to push the decomposition into smaller languages even further, as well as it encourages to see whole theories as simple 'completion' of standard languages. As we defended, breaking the monolithic status of context will actually make the theory and presentation follow more closely the technical developments, and liberate from the goal of having to find *the* process algebra with *its unique* observation technique that would capture all possible needs.

References

1 Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *Journal of the ACM*, 65(1):1:1–1:41, 2018. doi:10.1145/3127586.

2 Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999. doi:10.1006/inco.1998.2740.

[15] For instance, no context lemma can exist in the 'Situation B' we discussed earlier [22, p. 117].

3 Beniamino Accattoli. A fresh look at the lambda-calculus (invited talk). In Herman Geuvers, editor, *CSL*, volume 131 of *Leibniz International Proceedings in Informatics*, pages 1:1–1:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.1.

4 Beniamino Accattoli and Ugo Dal Lago. Beta reduction is invariant, indeed. In Thomas A. Henzinger and Dale Miller, editors, *CSL*, page 8. Association for Computing Machinery, 2014. doi:10.1145/2603088.2603105.

5 Roberto M. Amadio. Operational methods in semantics. draft of lecture notes, Université Paris Denis Diderot, 2016. URL: https://hal.archives-ouvertes.fr/cel-01422101.

6 Clément Aubert and Ioana Cristescu. How reversibility can solve traditional questions: The example of hereditary history-preserving bisimulation. In Igor Konnov and Laura Kovács, editors, *CONCUR*, volume 2017 of *LIPIcs*, pages 13:1–13:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CONCUR.2020.13.

7 Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

8 Emmanuel Beffara and Virgile Mogbil. Proofs as executions. In Jos C. M. Baeten, Thomas Ball, and Frank S. de Boer, editors, *IFIP TCS*, volume 7604 of *Lecture Notes in Computer Science*, pages 280–294. Springer, 2012. doi:10.1007/978-3-642-33475-7_20.

9 J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science, Amsterdam, 2001. doi:10.1016/B978-044482830-9/50017-5.

10 Mirna Bognar. *Contexts in Lambda Calculus*. PhD thesis, Vrije Universiteit Amsterdam, 2002. URL: https://www.cs.vu.nl/en/Images/bognar_thesis_tcm210-92584.pdf.

11 Mirna Bognar and Roel C. de Vrijer. A calculus of lambda calculus contexts. *Journal of Automated Reasoning*, 27(1):29–59, 2001. doi:10.1023/A:1010654904735.

12 Michele Boreale and Rocco De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120(2):279–303, 1995. doi:10.1006/inco.1995.1114.

13 Flavien Breuvart. *Dissecting denotational semantics*. PhD thesis, Université Paris Diderot — Paris VII, 2015. URL: https://www.lipn.univ-paris13.fr/~breuvart/These_breuvart.pdf.

14 Antonio Bucciarelli, Alberto Carraro, Thomas Ehrhard, and Giulio Manzonetto. Full Abstraction for Resource Calculus with Tests. In Marc Bezem, editor, *CSL*, volume 12 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 97–111, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CSL.2011.97.

15 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. doi:10.1017/S0960129514000218.

16 Vincent Danos and Jean Krivine. Reversible communicating systems. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004. doi:10.1007/978-3-540-28644-8_19.

17 Sangiorgi Davide. A theory of bisimulation for the pi-calculus. *Acta Informatica*, 33(1):69–97, 1996. doi:10.1007/s002360050036.

18 Sangiorgi Davide. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1-2):457–493, 1999. doi:10.1016/S0304-3975(99)00040-7.

19 Sangiorgi Davide. Pi-calculus. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1554–1562. Springer, 2011. doi:10.1007/978-0-387-09766-4_202.

20 Sangiorgi Davide and Jan Rutten, editors. *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2011. doi:10.1017/CBO9780511792588.

21 Sangiorgi Davide and David Walker. On barbed equivalences in pi-calculus. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR*, volume 2154 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2001. doi:10.1007/3-540-44685-0_20.

22 Sangiorgi Davide and David Walker. *The Pi-calculus*. Cambridge University Press, 2001.

23 Matthew De Nicola, Roccoand Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984. doi:10.1016/0304-3975(84)90113-0.

24 Rocco De Nicola, Ugo Montanari, and Frits W. Vaandrager. Back and forth bisimulations. In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR '90*, volume 458 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 1990. doi:10.1007/BFb0039058.

25 Uffe Engberg and Mogens Nielsen. A calculus of communicating systems with label passing - ten years after. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 599–622. The MIT Press, 2000.

26 Claudia Faggian and Simona Ronchi Della Rocca. Lambda calculus and probabilistic computation. In *LICS*, pages 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785699.

27 Cédric Fournet and Georges Gonthier. A hierarchy of equivalences for asynchronous calculi. *Journal of Logical and Algebraic Methods in Programming*, 63(1):131–173, 2005. doi:10.1016/j.jlap.2004.01.006.

28 Simon Fowler, Sam Lindley, and Philip Wadler. Mixing metaphors: Actors as channels and channels as actors. In Peter Müller, editor, *ECOOP 2017*, volume 74 of *LIPIcs*, pages 11:1–11:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. URL: http://www.dagstuhl.de/dagpub/978-3-95977-035-4, doi:10.4230/LIPIcs.ECOOP.2017.11.

29 Yuxi Fu and Zhenrong Yang. Tau laws for pi calculus. *Theoretical Computer Science*, 308(1-3):55–130, 2003. doi:10.1016/S0304-3975(03)00202-0.

30 Andrew D. Gordon and Luca Cardelli. Equational properties of mobile ambients. *Mathematical Structures in Computer Science*, 13(3):371–408, 2003. doi:10.1017/S0960129502003742.

31 Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. *Theoretical Computer Science*, 266(1-2):249–272, 2001. doi:10.1016/S0304-3975(00)00174-2.

32 Matthew Hennessy. *A distributed Pi-calculus*. Cambridge University Press, 2007. doi:10.1017/CBO9780511611063.

33 Carl Hewitt, Peter Boehler Bishop, Irene Greif, Brian Cantwell Smith, Todd Matson, and Richard Steiger. Actor induction and meta-evaluation. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *POPL*, pages 153–168. ACM Press, 1973. URL: http://dl.acm.org/citation.cfm?id=512927, doi:10.1145/512927.512942.

34 Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151(2):437–486, 1995. doi:10.1016/0304-3975(95)00074-7.

35 Ross Horne, Ki Yung Ahn, Shang-Wei Lin, and Alwen Tiu. Quasi-open bisimilarity with mismatch is intuitionistic. In Anuj Dawar and Erich Grädel, editors, *LICS*, pages 26–35. Association for Computing Machinery, 2018. doi:10.1145/3209108.3209125.

36 Ivan Lanese, Michael Lienhardt, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Concurrent flexible reversibility. In Matthias Felleisen and Philippa Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 370–390. Springer, 2013. doi:10.1007/978-3-642-37036-6_21.

37 Ivan Lanese, Doriana Medić, and Claudio Antares Mezzina. Static versus dynamic reversibility in CCS. *Acta Informatica*, November 2019. doi:10.1007/s00236-019-00346-6.

38 Jean-Marie Madiot. *Higher-order languages: dualities and bisimulation enhancements*. PhD thesis, École Normale Supérieure de Lyon, Università di Bologna, 2015. URL: https://hal.archives-ouvertes.fr/tel-01141067.

39 Massimo Merro. On the expressiveness of chi, update, and fusion calculi. In Ilaria Castellani and Catuscia Palamidessi, editors, *EXPRESS*, volume 16 of *Electronic Notes in Theoretical Computer Science*, pages 133–144. Elsevier, 1998. URL: https://www.sciencedirect.com/journal/electronic-notes-in-theoretical-computer-science/vol/16/issue/2, doi:10.1016/S1571-0661(04)00122-7.

40 Massimo Merro and Francesco Zappa Nardelli. Behavioral theory for mobile ambients. *Journal of the ACM*, 52(6):961–1023, 2005. doi:10.1145/1101821.1101825.

41 Robin Milner. Fully abstract models of typed *lambda*-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977. doi:10.1016/0304-3975(77)90053-6.

42 Robin Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science. Springer-Verlag, 1980. doi:10.1007/3-540-10235-3.

43 Robin Milner. *Communication and Concurrency.* PHI Series in computer science. Prentice-Hall, 1989.

44 Robin Milner. Elements of interaction: Turing award lecture. *Communications of the ACM*, 36(1):78–89, January 1993. doi:10.1145/151233.151240.

45 Robin Milner and Sangiorgi Davide. Barbed bisimulation. In Werner Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer, 1992. doi:10.1007/3-540-55719-9_114.

46 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.

47 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992. doi:10.1016/0890-5401(92)90009-5.

48 Ugo Montanari and Vladimiro Sassone. Dynamic congruence vs. progressing bisimulation for CCS. *Fundamenta Informaticae*, 16(1):171–199, 1992. URL: http://eprints.soton.ac.uk/id/eprint/261817.

49 Mogens Nielsen and Christian Clausen. Bisimulation for models in concurrency. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR '94*, volume 836 of *Lecture Notes in Computer Science*, pages 385–400. Springer, 1994. doi:10.1007/BFb0015021.

50 Mogens Nielsen, Uffe Engberg, and Kim S. Larsen. Fully abstract models for a process language with refinement. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, volume 354 of *Lecture Notes in Computer Science*, pages 523–548. Springer, 1989. doi:10.1007/BFb0013034.

51 Patrick Niemeyer and Daniel Leuck. *Learning Java.* O'Reilly Media, Incorporated, 4th edition edition, 2013.

52 Catuscia Palamidessi and Frank D. Valencia. Recursion vs replication in process calculi: Expressiveness. *Bulletin of the European Association for Theoretical Computer Science*, 87:105–125, 2005. URL: http://eatcs.org/images/bulletin/beatcs87.pdf.

53 Joachim Parrow. An introduction to the π-calculus. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 479–543. North-Holland / Elsevier, 2001. doi:10.1016/b978-044482830-9/50026-6.

54 Joachim Parrow and Sangiorgi Davide. Algebraic theories for name-passing calculi. volume 803 of *Lecture Notes in Computer Science*, pages 509–529. Springer, 1993. doi:10.1007/3-540-58043-3_27.

55 Benjamin C. Pierce and Sangiorgi Davide. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996. doi:10.1017/S096012950007002X.

56 Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. Models for concurrency: Towards a classification. *Theoretical Computer Science*, 170(1-2):297–348, 1996. doi:10.1016/S0304-3975(96)80710-9.

57 Manfred Schmidt-Schauß and David Sabel. On generic context lemmas for higher-order calculi with sharing. *Theoretical Computer Science*, 411(11-13):1521–1541, 2010. doi:10.1016/j.tcs.2009.12.001.

58 Peter Selinger and Benoît Valiron. Quantum lambda calculus. In Simon Gay and Ian Mackie, editors, *Semantic Techniques in Quantum Computation*, page 135–172. Cambridge University Press, 2009. doi:10.1017/CBO9781139193313.005.

59 Colin Stirling. Modal and temporal logics for processes. In Faron Moller and Graham M. Birtwistle, editors, *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings)*, volume 1043 of *Lecture Notes in Computer Science*, pages 149–237. Springer, 1995. doi:10.1007/3-540-60915-6_5.

60 Bas van den Heuvel and Jorge A. Pérez. Session type systems based on linear logic: Classical versus intuitionistic. In Stephanie Balzer and Luca Padovani, editors, *PLACES@ETAPS 2020*, volume 314 of *EPTCS*, pages 1–11, 2020. doi:10.4204/EPTCS.314.1.

61 Robert J. van Glabbeek. The linear time - branching time spectrum II. In Eike Best, editor, *CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993. doi:10.1007/3-540-57208-2_6.

62 André van Tondervan. A lambda calculus for quantum computation. *SIAM Journal on Computing*, 33(5):1109–1135, 2004. doi:10.1137/S0097539703432165.

63 Carlos A. Varela. *Programming Distributed Computing Systems: A Foundational Approach*. The MIT Press, 2013.

64 Glynn Winskel. Event structures, stable families and concurrent games. Lecture notes, University of Cambridge, 2017. URL: https://www.cl.cam.ac.uk/~gw104/ecsym-notes.pdf.

65 Wang Yi. CCS + time = an interleaving model for real time systems. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez-Artalejo, editors, *ICALP*, volume 510 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 1991. doi:10.1007/3-540-54233-7_136.