Processes, Systems & Tests: Defining Contextual Equivalences

Clément Aubert, Daniele Varacca

Processes, Systems & Tests: Defining Contextual Equivalences

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

March 26, 2021

In this position paper, we would like to offer and defend a new template to study equivalences between programs—in the particular framework of process algebras for concurrent computation. We believe that our layered model of development will clarify the distinction that is too often left implicit between the tasks and duties of the programmer and of the tester. It will also enlighten pre-existing issues that have been running across process algebras as diverse as the calculus of communicating systems, 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, that we call *Processes*, *Systems* and *Tests*. While the role of what can be observed and the subtleties in the definitions of congruences have been intensively studied, the fact that *not every process can be tested*, and that *the tester should have access to a different set of tools than the programmer* is curiously left out, or at least not often formally discussed. We argue that this blind spot comes from the under-specification of contexts—environments in which comparisons takes place—that play multiple distinct roles but supposedly always "stay the same".

We illustrate our statement with a simple Java example, the "usual" concurrent languages, but also back it up with λ-calculus and existing implementations of concurrent languages as well.

1 Introduction

In the study of programming languages, contextual equivalences play a central role. The core idea is that to study the behavior of a program, or a process, one needs to make it interact with different environments, and observe how it reacts, e.g. what outcomes it produces. If the program is represented by a term in a given syntax, environments are often represented as contexts surrounding the terms.

But contexts play multiple roles that serve different actors with different purposes. The programmer uses them to construct larger programs, the user crafts them to provide input and observe the output, and the tester or attacker uses them to test the program or to try to disrupt its behavior.

We believe that representing those different purposes with the same "monolithic" syntactical notion of context forced numerous authors to repeatedly "adjust" their definition of context without always acknowledging it. We also argue that collapsing multiple notions of contexts into one prevented further progress. In this article, we propose a way of clarifying how to define contextual equivalences, and show that having co-existing notions of equivalences legitimates and explains recurring choices, and supports a rigorous guideline to separate the development of a program from its usage and testing.

Maybe in the mind of most of the experts in the formal study of programming language is our proposal too obvious to discuss. However, if this is the case, we believe that this "folklore" remains unwritten, and that since we were not at *that* "seminar at Columbia in 1976",[1] we are to remain in darkness.

We believe the Annual Symposium on Logic in Computer Science is the ideal place to reach the principal actors in the field, and to either be proven wrong or—hopefully—impact some researchers. Non-technical articles can at times have a tremendous impact [32], and even if we do not claim to have Dijkstra's influence or talent, we believe that our precise, documented proposition can shed a new light on past results, and frame current reflections and future developments. We believe in any case that ignoring or downplaying the distinctions we would like to stress have repeatedly caused confusions in the past, and risks to continue doing so if not addressed.

2 The Flow of Testing

We begin by illustrating with a simple Java example the three syntactic notions–*process*, *system* and *test*–that we will be using. Imagine a user is given the following lines of code

```
while(i < 10){
x *= x;
i++;
}
```

A user cannot execute or use this "snippet" unless it is properly *wrapped* into a method, with adequate header, and possibly variable declaration(s) and `return` statement. Once

[1] To re-use in our setting Paul Taylor's witty comment [73].

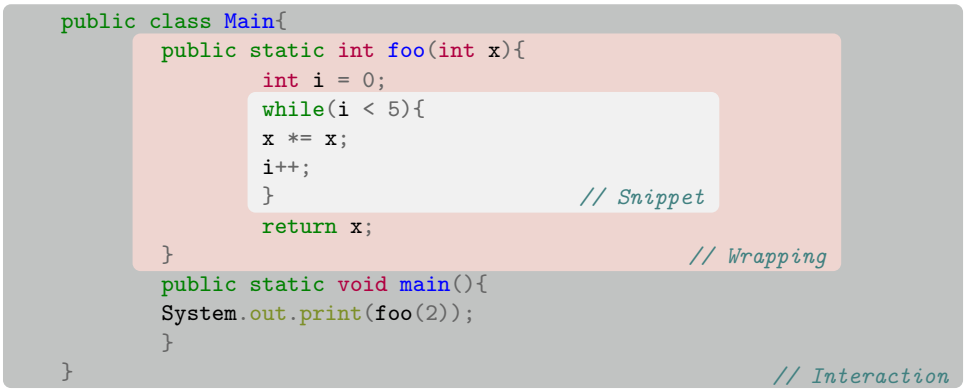

Figure 1: The three layers of a Java program

the programmer performed this operation, the user can *use* the obtained program, and the tester can *interact* with it further, for instance by calling it from a `main` method.

All in all, a programmer would build on the snippet, then the tester would build an environment to interact with the resulting program, and we could obtain the code displayed in Figure 1. Other situations could arise–for instance, if the snippet was already wrapped–, but we believe this description to be a fair rendering of the "life of a snippet".

In this example, the snippet is what we will call a *process*, the snippet once wrapped is what we will call a *system* and the "Interaction" part without the system in it, but with additional "observations"—i.e. measures on the execution—, is what we will call a *test*. Our terminology comes from the study of concurrent process algebras, where most of our intuitions and references will come from. But we will first make a detour to briefly examine how our lens applies to λ-calculus.

3 A Foreword on λ-Calculus

Theoretical languages often take λ-calculus as a model or a comparison basis.[2] And indeed, pure λ-calculus (i.e. without types or additional features like probabilistic sum [34] or quantum capacities [76, 71]) is a reasonnable [6], 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[3] do not reduce as they are, to study their behavior, one needs first to make them interact with an environment, represented by the notion of context. Already

[2] For instance it is often said that the "λ-calculus is to sequential programs what the π-calculus is to concurrent programs" [26, 77], and other process algebras share similar lineages.

[3] Actually, if application, abstraction and variables all count as one, the ratio between normal term and term with redexes is unknown [15]. We imply here "since *most interesting* terms", in the sense of terms that represent programs.

in such a simple set-up, the notion of context is subtle. Contexts are generally defined as "term[s] with some holes" [11, p. 29, 2.1.18], that we prefer to call *slots* and that we denote $[\Box]$. Under this apparent simplicity, they should not be manipulated carelessly, as having multiple slots or not being careful when defining what it means to "fill a slot" can lead to e.g. losing confluence [16, pp. 40–41, Example 2.2.1], and as those issues persist even in the presence of a typing system [39]. Furthermore, definitions and theorems that use contexts frequently impose some restrictions on the contexts considered, to exclude e.g. contexts like $(\lambda x.y)[\Box]$ that simply "throw away" the term put in the slot in one step of β-reduction. Following the above observations, we conclude that contexts often come in two flavors, depending on the nature of the term under consideration:

For closed terms (i.e. without free variables), a context is essentially a series of arguments to feed the term. This observation allows to define e.g. *solvable terms* [11, p. 171, 8.3.1 and p. 416, 16.2.1].

For open terms (i.e. with free variables), a context is a *Böhm transformation* [11, p. 246, 10.3.3], which is equivalent [11, p. 246, 10.3.4] to a series of abstractions followed by a series of applications, and sometimes called "head context" [7, p. 25].

One can see that being closed corresponds to being "wrapped"–ready to use–, and that feeding arguments to a term corresponds to interacting with it from a `main` method: the Böhm transformation actually encapsulates two operations at once. In this case, the interaction can observe different aspects: whether the term terminates, whether it grows in size, etc., but it is generally agreed upon that no additional operator or reduction rule should be used. Actually, the syntax is restricted for the testing suite, as only application is allowed for testing: the tested term should not be wrapped in additional layers of abstraction if it is already closed.

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 [76, p. 1126] and surface context [34, 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 a more drastic separation, as λ-terms are split between terms and tests [20], a separation that was later on naturally extended to contexts [19, p. 73, Figure 2.4].

All this variety happened after the 2000's formal studies of contexts was undertaken [16, 17, 39], 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" [17, p. 29]. It seems ironic that λ-calculists took inspiration from a concurrent language to split their syntax in two right at its core [20, p. 97], or to study formally the *communication* between a context and the term in its slot, while concurrent languages sometimes tried to keep the "purity" and indistinguishability

of their contexts.[4]

In the case of concurrent calculi like the calculus of communicating systems (CCS) or the π-calculus, one also has to represent interactions with environments using a notion of context. But the status of contexts in concurrent calculi is even more unsettling when one note that, while "wrapping" 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. Indeed, as the notion of "feeding arguments to a concurrent process" concurs with the idea of wrapping it into a larger process, it seems that the distinction we just made between two kinds of contexts in λ-calculus cannot be ported to concurrent calculi. Our contribution starts by questioning whenever, indeed, process calculi have treated contexts as a uniform notion independently from the nature of the term or what it was used for.

4 Contextual Relations

Comparing terms is at the core of the study of programming languages, and process algebra is no exception. 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, an idea formally captured by the notions of pre-congruences and congruences. An equivalence relation $\mathcal{R}$ is usually said to be 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 $C[\Box]$, $(C[P], C[Q]) \in \mathcal{R}$ holds.[5]

A notable example of congurence is *barbed congruence* [57, Definition 8][48, Definition 2.1.4], which closes by context a reduction-closed relation used to observe "barbs"–the channel(s) on which a process can emit or receive. This congruence is often taken to be *the* "reference behavioural equivalence" [48, p. 4], as it observes the interface of processes, i.e. on which channels they can interact over the time and in parallel.

But behind this apparent uniformity in the definition of congruences, the definition of contextual relations itself have often been tweaked by altering the definition of context, with no clear explanation nor justification for this modification. Let us back up this assertion by considering five different treatments of contexts before trying to make general statements.

In the calculus of communicating systems, notions as central as contextual bisimulation [8, pp. 223-224, Definition 421] and barbed equivalence [8, p. 224, Definition 424] considers only *static* contexts [8, 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" [8, p. 227]. But there

[4] 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. 4.

[5] In some cases, the additional requirement that terms in the relation needs to be similar up to uniform substitution is added [42], and sometimes [65, p. 516, Definition 2], only the closure by substitution—seen as a particular kind of context—is required.

is no justification—other than technical, i.e. because they "they persist after a transition" [8, p. 223]—as to *why* one should consider only some contexts in defining contextual equivalences.

In the π-calculus, contexts are defined liberally [29, p. 19, Definition 1.2.1], but still exclude contexts like e.g. $[\Box] + 0$ right from the beginning. Congruences are then defined using this notion of context [29, p. 19, Definition 1.2.2], and strong barbed congruence is no exception [29, p. 59, Definition 2.1.17]. Other notions, like strong barbed equivalence [29, p. 62, Definition 2.1.20], are shown to be a non-input congruence [29, p. 63, Lemma 2.1.24], which is a notion relying on contexts that forbids the slot to occur under an input prefix [29, 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." [40, p. 245]

In the distributed π-calculus, contexts are restricted right from the beginning to particular operators [40, Definition 2.6]. Then, relations are defined to be contextual if they are preserved by static contexts [40, Definition 2.6], which contains only parallel composition with arbitrary terms and name binding.[6] Static operators are deemed "sufficient for our purpose" [40, p. 37] and static contexts only are considered "[t]o keep life simple" [40, p. 38], but no further justification is given.

In the semantic theory for processes, at least in the foundational theory we would like to discuss below, one difficulty is that the class of formal theories restricted to "reduction contexts" [42, 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 [42, p. 450] to obtain a notion of *generic reduction* [42, p. 451] that allows a satisfactory definition of sound theories [42, p. 452]. Insensitive terms are essentially the collection of terms that do not interact with contexts [42, p. 451, Proposition 3.15], an analogue to λ-calculuus' genericity Lemma [11, p. 374, Proposition 14.3.24]. 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.

Across calculi, a notion of "closing context"—that emerged from λ-calculus [8, p. 85], and matches the notion of "wrapping" a snippet—can be found in typed versions of the π-calculus [29, p. 479], in mobile ambient [77, p. 134], in the applied π-calculus [1, p. 7], and in the fusion calculus [49, p. 6], among others. Also known as "completing context" [24, p. 466], those contexts are parametric in a term, the idea being that such a context will "close" the term under study and make it amenable to tests and comparisons.

[6] Such contexts have varying name, e.g. "configuration context" [45, p. 375]. They have been studied under the name *harness* in the ambient calculus [38, p. 372].

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—typically because the same "wrapping" cannot be applied to any process. Additionally, in all those cases, contexts are given access to a subset of operators, or restricted to contexts with particular behavior, but *never extended*.[7] If we consider that contexts are the main tool to test the equivalence of processes, then why should the testers 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 justify and acknowledge the restrictions we just discussed instead of adding them *passing-by*.[8]

5 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 [40] implements a distinction between closed terms (called processes [40, 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" [52, 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– this would corresponds, in Java, to being wrapped, i.e. inserted into a proper header and ready to be used or tested.

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 [52, p. 68], and by "refusing to admit channels as entities distinct from agents" [55, p. 16] and defining two different notions of scopes [55, 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 [58, 59], there is no notion of closed terms, as every (input) binding on a channel introduces a new and free

[7] We discuss some other approaches that explicitly considered more expressive testing languages and hence contexts in Sect. 9.1.

[8] One could also note that a negative side-effect to the practices we just discussed is to have changing notions: as the notion of context often evolves between the beginning of the article and its end, the intuition that "programmers simply wrap snippets in larger contexts" would be correct at one point, and then become incorrect, as the "construction contexts" they use have more flexibility than what is actually allowed by contexts.

occurrence of a variable.[9] However, the language they build upon—ECCS [33]—made this distinction clear.

Once again in an attempt to mimic the "economy" [56, p. 86] of λ-calculus, but also taking inspiration from the claimed "monotheism" of the actor model [41], different notions such as values, variables, or channels have been united under the common terminology of "names". This is at times identified as a strength, to obtain a "richer calculus in which values of many kinds may be communicated, and in which value computations may be freely mixed with communications. " [58, p. 20] However, it seems that a distinction between those notions always needs to be carefully re-introduced when discussing technically the language [8, p. 258, Remark 493], extensions to it [1, p. 4] or possible implementations [36], [14, p. 13]. 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 [64, 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—no matter how it is defined—is often not enough, as it is routine to exclude e.g. terms with un-guarded operators like sum [29, p. 416] or recursion [55, p. 166].[10]

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 *complete* and *incomplete*–wrapped or not, in our Java terminology. 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 behavior of terms, by completing them if needed, and then by executing them and observing the outcome. 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 process 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 behavior of the term. Because the environment is generally tweaked to improve the likeliness of observing a particular behavior, 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 will use for the rest of the paper:

Processes are "partial" programs, still under development; they are sometimes called "open terms", and correspond to *incomplete terms*. They would be called code fragments, or snippets, in standard programming.

[9] And, still, in standard π-calculus [33, Table 2], [78], the term $a(x).P$ has at least the same number of free variables than P: every occurrence of x in P is captured by the binder, but a is now free.

[10] However, note that those operators–un-guarded sum and recursion– are often not excluded from the start, even if they can never be parts of tested terms. The usual strategy [55], [8, Remark 414] is often to keep them "as long as possible", and to exclude them only when their power cannot be tamed no more to fit the framework or prove the desired result, such as the preservation of weak bisimulation by all contexts.

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, and ready to be executed.

Tests are defined using contexts and observations, and aims at executing and testing systems. They would be `main` methods calling a library or an API in standard programming, along with a set of observables.

Our terminology is close to the one used e.g. in ADPI [40, Chapter 5] or mobile ambients [50, 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 (as we discussed in Sect. 4), for reasons we will expose below.

6 Designing Layered Concurrent Languages

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

Define processes The first step is to select a set of operators called *construction operators*, used by the programmer to write processes. Those operators should be expressive, easy to combine, with constraints as light as possible, and selected with the situation that is being modeled in mind—and not depending on whenever they fare well with not-yet-defined relations, as it is often done to privilege the guarded sum over the internal choice. To ease their usage, a "meta-syntax" can be used, something that is generally represented by the structural equivalence.[11]

Define deployment criteria Then how a process can become a system ready to be executed and tested should be defined as a series of conditions that can deal with 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.[12] Having a set of

[11] 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", on any sub-term. This was technically done with the introduction of "π-calculus, at a distance" [4, p. 45], that bypasses the need for a structural equivalence without loosing the flexibility it usually provides.

[12] Exactly like a Java method header can use keywords—`extends`, `implements`, etc.—that cannot be used in a method body.

deployment operators for systems that restricts,[13] expands or intersects with the set of construction operators is perfectly acceptable, and it should enable the transformation of processes into systems and their composition.

Define tests The last step requires to define 1. a set of *testing operators*, 2. a notion of environment constructed from those observators, along with instructions on how to place a system in it, 3. a system of reduction rules regimenting how a system can execute in an environment, 4. a set of observables, i.e. a function from systems in environments to a subset of a set of atomic proposition (like "emits barb a", "terminates", "contains recursion operator", etc.),

Observe that each step uses its own set of operators and generate its own notion of context—to construct, to deploy, or to test. Tests would be key in defining notions of congruence, that would likely be "reduction-closed", "observational" "contextually-closed" relations. Whenever how a process is wrapped into a system is part of the test or not is discussed in Sect. 9.2, we will see that it actually resonates with a long-standing debate in process algebra. Note that compared to how concurrent languages are generally designed, our approach is refined along two axis: 1. every step previously exposed allows the introduction of novel operators, 2. multiple notions of systems or tests can and should co-exist in the same process algebra, depending on the situation.

7 Addressing Existing Issues

In the literature, processes and systems often have the same structure as tests and all use the same operators and contexts—at least on the surface of it. However, we believe that the distinction we offer is constantly used "under the hood" without always a clear discussion, as it disturbs the supposedly required simplicity of process algebras. We would like to stress below how we believe our frame captures and clarifies some of the choices, debates, improvements and explanations that have been proposed in process algebras over the time.

Co-defining observations and contexts Originally, the barb was a predicate [57, p. 690], whose definition was purely syntactic. Probably inspired by the notion of observer for testing equivalences [30, p. 91], an alternative definition was made in terms of parallel composition with a tester process [48, 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: we believe our proposal could help in clarifying the interplay between observations and contexts.[14]

[13] 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.

[14] One could even imagine having a series of "contexts and observations lemmas" illustrating how certain observations can be simulated by some operators, or reciprocally.

Justifying the "silent" transition's treatment 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" [40, p. 6] and sometimes opposed to "externally observable actions" [27, p. 230]. While we agree that "[t]his abstraction from internal differences is essential for any tractable theory of processes" [55, 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* when they are running their own tests.

Sometimes is asked the question "to what extent should one identify processes differing only in their internal or silent actions?" [13, p. 6], but the question is treated as a property of the process algebra,[15] and not as something that can *internally* be tuned as needed. We argue that the answer to that question is "*it depends who is asking!*": from a user perspective, internal actions should *not* be observed, but it makes perfect sense to let a programmer observe them when testing and decide which process to prefer based on this information.

Letting multiple comparisions co-exist 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" [27, 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 needs of all the variety of testers. Of course, comparing multiple relations is an interesting and needed task [35, 75], but one should also state 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 required purity and uniformity of the syntax. A good example is probably given by mobile ambients [50]. The authors start with a two-level syntax that distinguishes between processes and systems [50, p. 966]. Processes have access to strictly more constructors than systems [50, p. 967, Table 1], that are supposed to hide the threads of computation [50, 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 [50, p. 969, Definiton 2.2].

The authors even extend further the syntax for processes with a special $\circ$ operator [50, 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, where operators are added and removed based on different needs, and where the language needs not to be monolithic. The authors furthermore introduce

[15] More precisely, in the considered article, as a property of concurrency semantics.

two different reduction barbed congruences [50, 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 [50, 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 systems do not do more than processes—was a clear goal right from the start. We believe it would have been fruitful to embrace this distinction 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 to argue that constructors can and should be added to calculi to access better discriminatory power, but without necessarily changing the "original" language. The mismatch operator, for instance, has a similar feeling: "reasonable" testing equivalences [18, p. 280] require it, and multiple languages [2, p. 24] use it to provide finer-grained equivalences. For technical reasons [29, p. 13], this operator is generally not part of the "core" of π-calculus, but resurfaces *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 improve the testing capacities.

Treating extensions as different completions It would benefit their study and usage to consider different conservative extensions of processes algebras as different completion strategies for the same construction operators. For instance, reversible [46] or timed [80] extensions of CCS could be seen as two completion strategies—different conditions for a process to become a system—for the same class of processes, inspired from the usual CCS syntax [8, Chapter 28.1]. Those completion strategies would be suited for different needs, as one could e.g. complete a CSS process as a RCCS [22] system to test for relations such as hereditary history-preserving bisimulation [9], 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"[16] 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 a theoretical and abstract perspective, one always wants to have some distance with implementation meanders, 2. it is sometimes tempting to tweak reality to have it better fit the model.

[16] 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.

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.[17]

Obtaining fine-grained typing systems The development of typing systems for concurrent programming languages is a notoriously difficult topic. Some results in π-calculus have been solidified [29, Part III], but diverse difficulties remain. Among them, the co-existence of multiple systems for e.g. session types [74], the difficulty to tie them precisely to other type systems as Linear Logic [21], and the doubts about how to adopt the "proof-as-program" paradigm in a concurrent setting [12], make 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 programmers—to track bugs and produce meaningful error messages—and one aimed at users—to track security leaks or perform user-input validation. Once again, having a system developed along the layers we recommend would allow to have e.g. a type system for processes only, and to erase the information when completing the process, so that the typing discipline would be enforced only when the program is being developed, but not executed. This is similar to arrays of parameterized types in Java [63, pp. 253–258], that checks the typing discipline at compilation time, but not at run-time.

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

[17] Typically, testers involved in the quality assurance and developers in test [44] *should* have access to different tools than software developers.

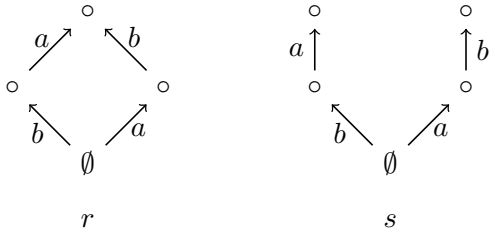

Figure 2: The trouble with auto-concurrency

with the same action [61, p. 391, Definition 5]. Systems with auto-concurrency are sometimes excluded as non-valid terms [31, p. 155] or simply not considered in particular models [62, p. 531], as the definition of bisimulation is problematic for them. Consider for instance the labeled configuration structures (a.k.a. stable family [79, Section 3.1]) r and s of Figure 2, where the label of the event executed is indicated on the edge and configurations are represented with $\circ$. Non-interleaving models of concurrency [69] distinguishes between r and s, as a "true concurrency model" would. Some forms of "back-and-forth-bisimulations" cannot discriminate between r and s if $a = b$ [67].

While not being able to distinguish between those two terms may make sense from an external—user's—point of view, we argue that a programmer should have access to an internal mechanism that could answer the question "*Can this process perform two barbs with the same label at the same time?*". Such an observation—possibly coupled with a testing operator—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, useful—in the realm of comparable systems.

Representing man-in-the-middle One could add to the testing operators an operator $\nabla a.P$ for each channel name a, which would forbid P 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" in a context $\nabla a[\Box]$ if the channel is still secure:

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

But in the case where $a = b$, the environment could intercept the communication and then decide if it forwards it, prevents it, or alters it. Adding this operator to the set of testing operators would for instance open up the possibility of interpreting $\nu a(P)$ as an operation securing the channel a in P,

enabling the study of relations $\sim$ that could include e.g.

$$\nabla a(\nu a(P|Q) \sim \nabla a(\nu b(P[a/b]|Q[a/b])) \qquad \text{(For } b \text{ not in the free names of } P \text{ nor } Q\text{)}$$

$$\nu a(\nabla a(P|Q)) \sim \nabla a(P|Q) \qquad \text{(Uselessness of securising a hacked channel)}$$

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, and hence are "construction contexts". This situation corresponds to Situation A in Figure 3.

We believe the picture could be much more general, with tests having 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 Figure 3. 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" [37, p. 57], standing "inside" the "Construction operators" circle while most authors seem to agree that they would prefer not to add it to the internals of the language.[18] 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 [43] to *quasi-open bissimilarities* [28, 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." [39, 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 and the execution of two programs on two single-core computers.

[18] To be more precise: while "most occurences of matching can be encoded by parallel composition [...,] mismatching cannot be encoded in the original π-calculus" [66, p. 526], which makes it somehow suspicious.

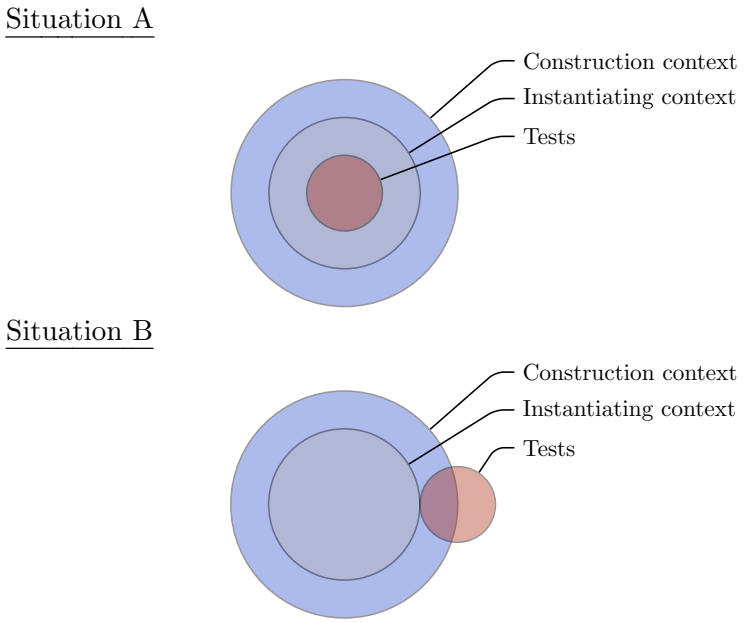

Figure 3: Opening up the testing capacities of π-calculus

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].

Recent progresses in the field of verification of cryptographic protocols [10] hinted in this direction as well. By taking "[t]he notion of test [to] be relative to an environment" [10, p. 12], a formal development involving "frames" [10, Definition 2.3] can emerge and give flesh to some ideas expressed in our proposal. It should be noted that this work also "enrich[...] processes with a success construct" [10, p. 12] that cannot be used to construct processes, to construct "experiments".

9 Concluding Remarks

We conclude by discussing related approaches, by offering a new light on a technical issue in the definition of barbed congruences, by offering a new interpretation of the so-called context lemmas, and by coming back to our motivations.

9.1 An Approved and Promising Perspective

We would like to stress that our proposal resonates with previous comments, and should not be treated as an isolated historical perspective that will have no impact on the future.

In the study of process algebras, in addition to the numerous hints toward our formalism that we already discussed, there are at least two instances when the power of the "testing suite" was explicitly discussed [25, Remark 5.2.21]. In a 1981 article, it is assumed that "by varying the ambiant ('weather') conditions, an experimenter" [53, p. 32] can observe and discriminate better than a simple user could. Originally, this idea seemed to encapsulate two orthogonal dimensions: the first was that the tester could run the tested system any number of times, something that would now be represented by the addition of the replication operator ! to the set of testing operators. The second was that the tester could enumerate all possible non-deterministic transitions of the tested system. This second dimension gave birth to "a language for testing concurrent processes" [47, p. 1] that is more powerful than the language used to write the programs being tested. In this particular example, the tester have access to a termination operator and probabilistic features that are not available to the programmer: as a result, the authors "may distinguish non-bisimilar processes through testing" [47, p. 19].

Looking forward, the vibrant field of secure compilation made a clear-cut distinction between "target language contexts" representing adversarial code and programmers' "source context" to explore property preservation of programs [3]. This perspective was already partially at play in the spi calculus for cryptographic protocols [2, p. 1], where the attacker is represented as the "environment of a protocol". We believe that both approaches—coming from the secure compilation, from the concurrency community, but also from other fields—concur to the same observation that the environment—formally captured by a particular notion of context—deserves an explicit and technical study to model different interactions with systems, and need to be detached from "construction" contexts.

9.2 When Should Contexts Come into Play?

The interesting question of *when* to use contexts when testing terms [29, pp. 116–117, Section 2.4.4] raises a technical question that is put under a different perspective by our analysis. Essentially, the question is whether the congruences under study should being *defined* as congruences (e.g. reduction-closed barbed congruence [29, p. 116]), or being defined in two steps, i.e. as the contextual closure of a pre-existing relation (e.g. strong barbed congruence [29, p. 61, Definition 2.1.17], which is the contextual closure of strong barbed bisimilarity [29, p. 57, Definition 2.1.7])?

Indeed, bisimulations can be presented as an "interaction game" [72] generally played as follows: 1. Pick an environment for both terms (i.e., complete them, then embed them in the same testing environment), 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

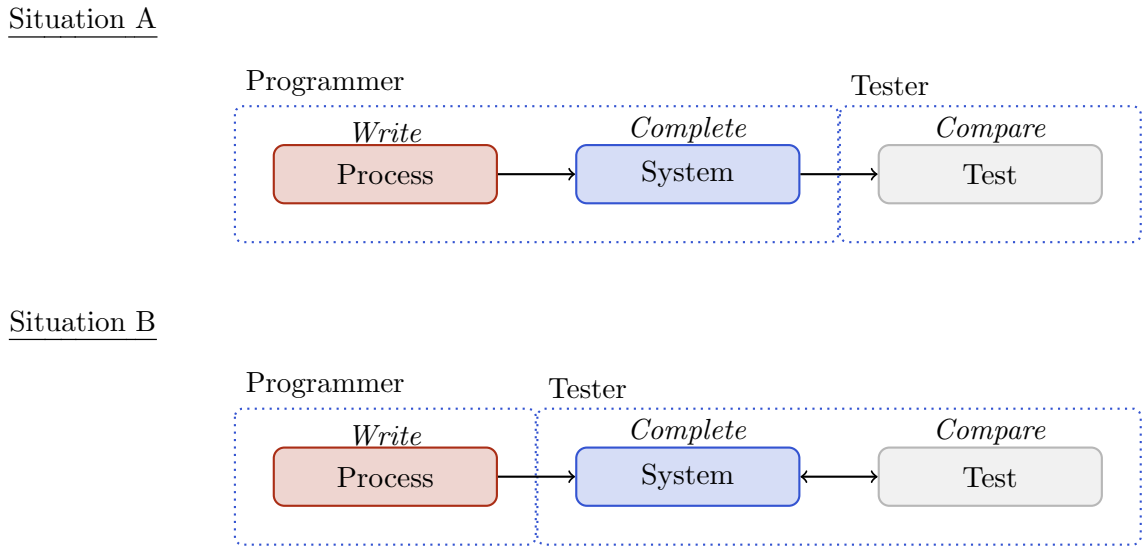

Figure 4: Distinguishing between completing strategies

observational congruence" [60] 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 [42, p. 455, Proposition 3.24], and that corresponds to open bisimilarities [23, p. 77, Proposition 3.12].[19]

Those two approaches have been extensively compared and analyzed–and still are [1, p. 24]—but to our knowledge they rarely co-exist: it is as if a side had to be taken at the early stage of the language design, instead of letting in a second moment the tester decide which approach is best suited for the aspects they wish to observe. It seems that both approaches are equally valid, *provided we acknowledge they play different roles.*

This question of *when are the terms completed?* can be rephrased as *what is it that you are trying to observe?*, or even *who is completing them?*: is the completion provided by the programmer, once and for all, or is the tester allowed to explore different completions and to change them as the tests unfold? Looking back at our Java example from Sect. 2, this corresponds to letting the tester repeatedly tweak the parameter or return type of the wrapping from `int` to `long`, allowing them to have finer comparisons between snippets. In this frame, moving from the *static* definition of congruence to *dynamic* one would corresponds to going from situation A to situation B in Figure 4. 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

[19] And it should be noted, once again, that *quasi-open bissimilarities* [28, p. 300, Definition 6] opens up the possibility of having multiple notions of completion readily available.

different situations equally interesting.

2. This choice can appeal to different notions of systems, completions and tests: for instance, while completing a process before testing it (Situation A) may indeed be needed when the environment represents an external deployment platform, it makes less sense if we think of the environment as part of the development workflow, in charge of providing feedback to the programmer or as a powerful attacker than can manipulate the conditions in which the process is executed (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 comparing source code's releases, some comparing binaries' releases—to get a better, fuller, picture of the program.

9.3 Penetrating Context Lemmas' Meanings

What is generally refereed to as *the* context lemma[20] 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 [29, p. 95, Definition 2.4.5] as the closure of barbed bisimilarity under all context, or only under contexts of the form $[\Box]\sigma \mid P$ for all substitution σ and term P. In its first version [68, 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 [70, 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 $[\Box] + 0$ are routinely removed—as we discussed in Sect. 4—to define the barbed congruence of π-calculus, or contexts were restricted to what is called harnesses in the mobile ambients calculus [38] before proving such results. As strong bisimulation [65, p. 514, Definition 1] is not preserved by input prefix [65, 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 non-input contexts [29, p. 62, Definition 2.1.22] do, 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

[20] At least, in process algebra, as the meaning of this lemma in e.g. λ-calculus is different [51, p. 6].

that is not only weaker, but also somehow artificial, or even tautological. Furthermore the criteria of "not adding any discriminating power" should not be a positive criterion when deciding if a context belongs 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.

From this perspective, it seems to us that context lemmas embrace an uncanny perspective: instead of being used to prove properties about tests more easily, they should be considered from the perpective of the ease of use of testing systems. Stated differently, we believe that the set of testing operators should come first, and than *then*, if the language designer wish to add operators to ease the testers' life, they can do so providing they prove a context lemma proving that those novel operators will not alter the original testing capacities. Once again, varying the testing suite is perfectly acceptable, but once it has been fixed, *the context lemma is simply present to show that adding some testing operators is innocent, that it will simply make testing certain properties easier.*

9.4 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,[21] 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 intrinsically 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 multiple 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 stating the variations 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

[21] For instance, no context lemma can exist in the "Situation B" of Figure 4 [29, p. 117].

is often considered the best 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 [5]—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[22] 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.

[3] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 256–271. IEEE, 2019. doi:10.1109/CSF.2019.00025.

[4] Beniamino Accattoli. Evaluating functions as processes. In Rachid Echahed and Detlef Plump, editors, *TERMGRAPH 2013*, volume 110 of *EPTCS*, pages 41–55, 2013. doi:10.4204/EPTCS.110.6.

[5] 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.

[6] 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.

[22] This may be a good place, before we terminate this article, to mention that this monolithicity probably comes in part from the original will of making e.g. CCS at the same time a programming and a specification language. The specification was supposed to be the program itself, that would be easy to check for correctness: the goal was to make it "possible to describe existing systems, to specify and program new systems, and to argue mathematically about them, all without leaving the notational framework of the calculus" [54, p. 1]. This original research project slightly shifted—from specifying programs to specifying behaviors—but that original perspective remained.

[7] Beniamino Accattoli and Ugo Dal Lago. On the invariance of the unitary cost model for head reduction. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *Leibniz International Proceedings in Informatics*, pages 22–37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012. doi:10.4230/LIPIcs.RTA.2012.22.

[8] Roberto M. Amadio. Operational methods in semantics. Lecture notes, Université Denis Diderot Paris 7, December 2016. URL: https://hal.archives-ouvertes.fr/cel-01422101.

[9] 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 13:1–13:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CONCUR.2020.13.

[10] David Baelde. *Contributions à la Vérification des Protocoles Cryptographiques*. Habilitation à diriger des recherches, Université Paris-Saclay, February 2021. URL: http://www.lsv.fr/~baelde/hdr/habilitation_baelde.pdf.

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

[12] 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.

[13] 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.

[14] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, 2016. doi:10.1561/3300000004.

[15] Olivier Bodini. personal communication.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] Sangiorgi Davide. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[26] 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.

[27] 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.

[28] 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.

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

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

[31] 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.

[32] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968. doi:10.1145/362929.362947.

[33] 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.

[34] 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.

[35] 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.

[36] 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. doi:10.4230/LIPIcs.ECOOP.2017.11.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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. doi:10.1145/512927.512942.

[42] 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.

[43] 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.

[44] Andy Knight. A software engineer in test must have the heart of a developer, 2018. URL: https://blog.testproject.io/2018/11/06/the-software-engineer-in-test/.

[45] 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.

[46] 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.

[47] Kim Guldstrand Larsen and Arne Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991. doi:10.1016/0890-5401(91)90030-6.

[48] 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.

[49] 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. doi:10.1016/S1571-0661(04)00122-7.

[50] 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.

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

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

[53] Robin Milner. A modal characterisation of observable machine-behaviour. In Egidio Astesiano and Corrado Böhm, editors, *CAAP '81, Trees in Algebra and Programming, 6th Colloquium, Genoa, Italy, March 5-7, 1981, Proceedings*, volume 112 of *Lecture Notes in Computer Science*, pages 25–34. Springer, 1981. doi:10.1007/3-540-10828-9_52.

[54] Robin Milner. A calculus of communicating systems. LFCS Report Series ECS-LFCS-86-7, The University of Edinburgh, 08 1986. URL: http://www.lfcs.inf.ed.ac.uk/reports/86/ECS-LFCS-86-7/index.html.

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

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

[57] 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.

[58] 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.

[59] 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.

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

[61] 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.

[62] 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.

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

[64] Catuscia Palamidessi and Frank D. Valencia. Recursion vs replication in process calculi: Expressiveness. *Bulletin of the EATCS*, 87:105–125, 2005. URL: http://eatcs.org/images/bulletin/beatcs87.pdf.

[65] 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.

[66] Joachim Parrow and Sangiorgi Davide. Algebraic theories for name-passing calculi. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium,*

Noordwijkerhout, The Netherlands, June 1-4, 1993, Proceedings, volume 803 of *Lecture Notes in Computer Science*, pages 509–529. Springer, 1993. doi:10.1007/3-540-58043-3_27.

[67] Iain Phillips and Irek Ulidowski. Reversibility and models for concurrency. *Electronic Notes in Theoretical Computer Science*, 192(1):93–108, 2007. doi:10.1016/j.entcs.2007.08.018.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] Paul Taylor. Comment to "substitution is pullback". URL: http://math.andrej.com/2012/09/28/substitution-is-pullback/.

[74] 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.

[75] 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.

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

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

[78] Wikipedia contributors. π-calculus — syntax, 2020. [Online; accessed 5-January-2021]. URL: https://en.wikipedia.org/wiki/%CE%A0-calculus#Syntax.

[79] 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.

[80] 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.