Process, Systems and Tests: Three Layers in Concurrent Computation (Short Paper)

Clément Aubert, Daniele Varacca

Process, Systems and Tests: Three Layers in Concurrent Computation (Short Paper)

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

Daniele Varacca
LACL, Universit Paris-Est Crteil, France
daniele.varacca@u-pec.fr

In this short 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 ambiguities 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.

1 Introduction: An Apparently Tightwad Godfather

Theoretical languages for concurrent computation often take λ-calculus as a model or a comparison basis[1]: one wish concurrent computation could have a language as mature and as stable as this functional language,[2] and "achieve the same economy" [30, p. 86]. As β-normal terms cannot reduce, to study their behaviour, one needs first to make them interact with an "environment", represented by the notion of *context*. However, being careless when defining the notion of context can lead to e.g. losing confluence [6, pp. 40–41, Example 2.2.1], even in the presence of a typing system [20]. When λ-calculus is enriched with e.g. quantum or probabilistic features, then contexts are narrowed down to term [37, p. 1126] or surface [18, pp. 4, 10] contexts respectively. In resource sensitive extensions of the λ-calculus was implemented a more drastic separation, as λ-terms were split between terms and tests [9], a separation that was later on naturally extended to contexts [8, p. 73, Figure 2.4].

It seems ironic that λ-calculists took inspiration from a concurrent language to split their syntax in two right at its core [9, p. 97], or to study formally the *communication* between a context and its expression, while concurrent languages apparently tried to maintain the "purity" and indistinguishability of their contexts—i.e. "a context *is* a term, period"—as we will discuss below.

[1] That the "λ-calculus is to sequential programs what the π-calculus is to concurrent programs" is a common trope [12, 38].
[2] This common belief actually needs some revision [2], but that's not our point here.

Submitted to:
EXPRESS/SOS 2020

2 Contexts

In this section, we would like to discuss how, under their apparent monolithic status, contexts in concurent calculi are actually multifaceted due to their different purposes.

2.1 Open and Closed Terms

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" [28, 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. For contexts, this means that we are actually interested only in *closing contexts*, a.k.a. "completing context" [11, p. 466], contexts that guarantee that the term under study is closed.

However, in concurrent calculi, the central notions of binders and of variables have been changing, and still seem sometimes "up in the air". For instance, in the original CCS, restriction was not a binder [28, p. 68], and by "refusing to admit channels as entities distinct from agents" [29, p. 16] and defining two different notions of scopes [29, 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 [32, 33], 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 [17]—made this distinction clear.

Adding to the confusion, and taking inspiration from the claimed "monotheism" of the actor model [22], notions such as values, variables, or channels have been united under the common terminology of "names", making it even harder to decide what "close" would refer to. Finally, let us note that extensions of π-calculus sometimes include different binders, as e.g. output binders in the private π-calculus [36, p. 113].

2.2 Not All Contexts Are Equal

In addition to this original, common, restriction, contexts are routinely modified and altered, to ease the study of particular relations or to preserve interesting properties. We would like to briefly list some examples. In the Calculus of Communicating Systems, notions as central as contextual bisimulation [3, pp. 223-224, Definition 421] and barbed equivalence [3, p. 224, Definition 424] considers only *static* contexts [3, p. 223, Definition 420], which are composed only of parallel composition with arbitrary term and restriction. There is no justification—other than technical, i.e. because they "they persist after a transition" [3, p. 223]—as to *why* should only some contexts being considered in contextual equivalences. In the π-calculus, despite their liberal definition [14, p. 19, Definition 1.2.1], contexts like e.g. $[\Box] + 0$ are completely excluded. The notion of congruence [14, p. 19, Definition 1.2.1] is refined with non-input congruence [14, p. 62, Definition 2.1.23] thanks to a modified notion of context [14, p. 62, Definition 2.1.22]. In the distributed π-calculus, contexts are restricted to particular operators [21, Definition 2.6], and then restricted to static contexts [21, Definition 2.6], which contains only parallel composition with arbitrary terms and name binding,[3] deemed "sufficient for our purpose" [21, p. 37].

[3] Such contexts have been called "configuration context" [24, p. 375] or "harness" in the ambient calculus [19, p. 372].

2.3 Purposes of Congruences

As we just saw, contexts are often "tuned" when a particular relation is under study, and shrunken by need, to bypass some of the difficulties they raise, or to preserve some notions. This variety of refinements on the notion of context is justified by the central roles of congruences in concurrent calculii, which captures the idea that a comparison is deemed of interest only if its results are valid in every possible context. However, we argue that there are two different perspectives in the use of congruence, depending on whether one would like to know

1. if an incomplete portion of a process can be substituted for another,
2. if a tested 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?*").

That different congruences capture different dimensions is witnessed e.g. by the treatment made to "silent" transitions, seen as an "unobservable internal activity" [21, p. 6]. It is indeed routine to consider *strong* and a *weak* versions of a congruence, the later focusing on the "externally observable actions" [13, p. 230]. While we agree that "[t]his abstraction from internal differences is essential for any tractable theory of processes" [29, p. 3], we would like to stress that *both can and should be accommodated*, and that *"internal" transition should be treated as invisible* to the user*, but accessible* to the programmer.

Sometimes is asked the question "to what extent should one identify processes differing only in their internal or silent actions?" [5, p. 6], but the question is treated as a property of the process algebra,[4] and not as something that can *internally* be tuned as needed, and in that particular example, this distinction is later on *simply discarded* [5, p. 6]! We argue that the answer to that question is "*it depends who is asking*", and that different types of tests should be available to highlight or hide those features, based on the needs.

3 Our Proposal

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 [14, p. 416] or recursion [29, 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, 3. sometimes not required to test a term[5], we would like to use the slightly more generic adjectives *incomplete* and *complete*.

With this in mind, we argue that concurrent languages would benefit from being articulated as follows:

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.

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

[5] In distributed programming, when "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." [20, p. 250].

II. Define deployment criteria The programmer should define how can a *process* become a *system* thanks to 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—complete. Having a set of operators for systems that restrict,[6] expand or intersect with the set of construction operators is perfectly acceptable.

III. Define tests The last step defines 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", "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 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. As in most calculus, one cannot decide which set of observations is more basic [23, p. 444], this is again conforming to the existing yet unspoken "tradition".

4 Benefits and Perspectives

4.1 A Pre-Existing Distinction

We believe the distinction we offer is already used, and maybe in the mind of most of the experts is our proposal obvious. However, if this is the case, we believe that it is "folklore" and has never been written, we are to remain in darkness.

Let us highlight two places where our distinction is already used. The mismatch operator has been introduced to provide "reasonable" testing equivalences [7, p. 280], and is considered across languages [1, p. 24] to provide finer-grained equivalences. But for technical reasons [14, p. 13], this operator is generally not part of the "core" of π-calculus, and is added *by need* to obtain better equivalences. A similar reasoning guided the introduction of a special $\circ$ operator [27, p. 971, Definition 3.1] in mobile ambients, but symmetrically, as the equivalences studied *did not* consider this additional constructor. 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–added to improve the testing capacities.

There is also an on-going debate between which to enrich—observations or contexts? For instance, at its origin, the barb was a predicate [31, p. 690], whose definition was purely syntactic. Probably inspired by the notion of observer for testing equivalences [15, p. 91], an alternative definition was made in terms of parallel composition with a tester process [26, 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.

4.2 A Fruitful Lens

In the literature, processes and systems often have the same structure as tests and are—at least on the surface—indistinguishable from what they are supposed to test. We believe that exploiting a distinction between them could lead to fruitfully revisit some results and explore new possibilities.

[6] 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.

For instance, one could treat conservative extensions of processes algebras as completion strategies for the same construction operators. Indeed, reversible [25] or timed [39] 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 [3, Chapter 28.1]. Those completion strategies would be suited for different needs, as one could e.g. complete a CSS process as a RCCS [10] system to test for relations such as hereditary history-preserving bisimulation [4], 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"[7] versions of the same piece of software.

Auto-concurrency (a.k.a. auto-parallelism) is when a system have two different transitions labeled with the same action [34, p. 391, Definition 5]. Systems with auto-concurrency do not fare well with back and forth bisimulation, and are sometimes excluded as non-valid terms [16, p. 155] or simply not considered in particular models [35, p. 531]. 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 $!a.P$, and would re-integrate auto-concurrent systems in the realm of comparable systems.

5 Concluding Remarks on the Diversity of Concurrent Calculi

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, to answer the question: "*what could be the minimal requirements on contexts and operators to prove a generic form of context lemma*[8] *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, 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.

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 [2]—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.

[7] Similar to Debian's DebugPackage which enables 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.

[8] In a nutshell, this lemma is actually a series of results stating that considering all the operators when constructing the context for a congruence may not be needed.

References

[1] Martín Abadi & Andrew D. Gordon (1999): *A Calculus for Cryptographic Protocols: The spi Calculus*. *Information and Computation* 148(1), pp. 1–70, doi:10.1006/inco.1998.2740.

[2] Beniamino Accattoli (2019): *A Fresh Look at the lambda-Calculus (Invited Talk)*. In Herman Geuvers, editor: *CSL*, *Leibniz International Proceedings in Informatics* 131, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 1:1–1:20, doi:10.4230/LIPIcs.FSCD.2019.1.

[3] Roberto M. Amadio (2016): *Operational methods in semantics*. draft of lecture notes, Universit Paris Denis Diderot. Available at `https://hal.archives-ouvertes.fr/cel-01422101`.

[4] Clment Aubert & Ioana Cristescu (2020): *How Reversibility Can Solve Traditional Questions: The Example of Hereditary History-Preserving Bisimulation*. In Igor Konnov & Laura Kovács, editors: *CONCUR*, *LIPIcs* 2017, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 13:1–13:23, doi:10.4230/LIPIcs.CONCUR.2020.13.

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

[6] Mirna Bognar (2002): *Contexts in Lambda Calculus*. Ph.D. thesis, Vrije Universiteit Amsterdam. Available at `https://www.cs.vu.nl/en/Images/bognar_thesis_tcm210-92584.pdf`.

[7] Michele Boreale & Rocco De Nicola (1995): *Testing Equivalence for Mobile Processes*. *Information and Computation* 120(2), pp. 279–303, doi:10.1006/inco.1995.1114.

[8] Flavien Breuvart (2015): *Dissecting denotational semantics*. Ph.D. thesis, Universit Paris Diderot Paris VII. Available at `https://www.lipn.univ-paris13.fr/~breuvart/These_breuvart.pdf`.

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

[10] Vincent Danos & Jean Krivine (2004): *Reversible Communicating Systems*. In Philippa Gardner & Nobuko Yoshida, editors: *CONCUR*, *Lecture Notes in Computer Science* 3170, Springer, pp. 292–307, doi:10.1007/978-3-540-28644-8_19.

[11] Sangiorgi Davide (1999): *The Name Discipline of Uniform Receptiveness*. *Theoretical Computer Science* 221(1-2), pp. 457–493, doi:10.1016/S0304-3975(99)00040-7.

[12] Sangiorgi Davide (2011): *Pi-Calculus*. In David A. Padua, editor: *Encyclopedia of Parallel Computing*, Springer, pp. 1554–1562, doi:10.1007/978-0-387-09766-4_202.

[13] Sangiorgi Davide & Jan Rutten, editors (2011): *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, doi:10.1017/CBO9780511792588.

[14] Sangiorgi Davide & David Walker (2001): *The Pi-calculus*. Cambridge University Press.

[15] Matthew De Nicola, Roccoand Hennessy (1984): *Testing Equivalences for Processes*. *Theoretical Computer Science* 34, pp. 83–133, doi:10.1016/0304-3975(84)90113-0.

[16] Rocco De Nicola, Ugo Montanari & Frits W. Vaandrager (1990): *Back and Forth Bisimulations*. In Jos C. M. Baeten & Jan Willem Klop, editors: *CONCUR '90*, *Lecture Notes in Computer Science* 458, Springer, pp. 152–165, doi:10.1007/BFb0039058.

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

[18] Claudia Faggian & Simona Ronchi Della Rocca (2019): *Lambda Calculus and Probabilistic Computation*. In: *LICS*, IEEE, pp. 1–13, doi:10.1109/LICS.2019.8785699.

[19] Andrew D. Gordon & Luca Cardelli (2003): *Equational Properties Of Mobile Ambients*. *Mathematical Structures in Computer Science* 13(3), pp. 371–408, doi:10.1017/S0960129502003742.

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

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

[22] Carl Hewitt, Peter Boehler Bishop, Irene Greif, Brian Cantwell Smith, Todd Matson & Richard Steiger (1973): *Actor Induction and Meta-Evaluation*. In Patrick C. Fischer & Jeffrey D. Ullman, editors: *POPL*, ACM Press, pp. 153–168, doi:10.1145/512927.512942. Available at `http://dl.acm.org/citation.cfm?id=512927`.

[23] Kohei Honda & Nobuko Yoshida (1995): *On Reduction-Based Process Semantics*. *Theoretical Computer Science* 151(2), pp. 437–486, doi:10.1016/0304-3975(95)00074-7.

[24] Ivan Lanese, Michael Lienhardt, Claudio Antares Mezzina, Alan Schmitt & Jean-Bernard Stefani (2013): *Concurrent Flexible Reversibility*. In Matthias Felleisen & Philippa Gardner, editors: *ESOP*, *Lecture Notes in Computer Science* 7792, Springer, pp. 370–390, doi:10.1007/978-3-642-37036-6_21.

[25] Ivan Lanese, Doriana Medić & Claudio Antares Mezzina (2019): *Static versus dynamic reversibility in CCS*. *Acta Informatica*, doi:10.1007/s00236-019-00346-6.

[26] Jean-Marie Madiot (2015): *Higher-order languages: dualities and bisimulation enhancements*. Ph.D. thesis, cole Normale Suprieure de Lyon, Universit di Bologna. Available at `https://hal.archives-ouvertes.fr/tel-01141067`.

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

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

[29] Robin Milner (1989): *Communication and Concurrency*. PHI Series in computer science, Prentice-Hall.

[30] Robin Milner (1993): *Elements of Interaction: Turing Award Lecture*. *Communications of the ACM* 36(1), p. 7889, doi:10.1145/151233.151240.

[31] Robin Milner & Sangiorgi Davide (1992): *Barbed Bisimulation*. In Werner Kuich, editor: *ICALP*, *Lecture Notes in Computer Science* 623, Springer, pp. 685–695, doi:10.1007/3-540-55719-9_114.

[32] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, I*. *Information and Computation* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[33] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, II*. *Information and Computation* 100(1), pp. 41–77, doi:10.1016/0890-5401(92)90009-5.

[34] Mogens Nielsen & Christian Clausen (1994): *Bisimulation for Models in Concurrency*. In Bengt Jonsson & Joachim Parrow, editors: *CONCUR '94*, *Lecture Notes in Computer Science* 836, Springer, pp. 385–400, doi:10.1007/BFb0015021.

[35] Mogens Nielsen, Uffe Engberg & Kim S. Larsen (1989): *Fully abstract models for a process language with refinement*. In J. W. de Bakker, Willem P. de Roever & 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*, *Lecture Notes in Computer Science* 354, Springer, pp. 523–548, doi:10.1007/BFb0013034.

[36] Catuscia Palamidessi & Frank D. Valencia (2005): *Recursion vs Replication in Process Calculi: Expressiveness*. *Bulletin of the European Association for Theoretical Computer Science* 87, pp. 105–125. Available at `http://eatcs.org/images/bulletin/beatcs87.pdf`.

[37] Andr van Tondervan (2004): *A Lambda Calculus for Quantum Computation*. *SIAM Journal on Computing* 33(5), pp. 1109–1135, doi:10.1137/S0097539703432165.

[38] Carlos A. Varela (2013): *Programming Distributed Computing Systems: A Foundational Approach*. The MIT Press.

[39] Wang Yi (1991): *CCS + Time = An Interleaving Model for Real Time Systems*. In Javier Leach Albert, Burkhard Monien & Mario Rodríguez-Artalejo, editors: *ICALP*, *Lecture Notes in Computer Science* 510, Springer, pp. 217–228, doi:10.1007/3-540-54233-7_136.