Processes Against Tests: Defining Contextual Equivalences

Clément Aubert, Daniele Varacca

Processes Against Tests: Defining Contextual Equivalences

Clément Aubert*

School of Computer and Cyber Sciences, Augusta University, Georgia, USA

Daniele Varacca

LACL, Université Paris-Est Créteil, France

Abstract

In this position paper, we would like to offer and defend a 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 such 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 in three conceptually separated entities, that we call *process terms*, (completed) *processes* and *tests*, and by stressing the importance of formalizing the *completion* of process terms and the *instrumentation* that results from placing a (completed) processes into a test. 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 term 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–in this respect, the theory of monitor is a counter-examples that we discuss and compare to our approach. We argue that this blind spot comes from the under-specification of contexts—environments in which comparisons occur—that play multiple distinct roles but are generally—at least, on the surface of it—given only one definition that fails to capture all of their aspects.

Keywords: Process Algebra, Concurrency, Testing Equivalences, Process Semantics
2020 MSC: 68N19, 68Q85

*Corresponding author
Email addresses: caubert@augusta.edu (Clément Aubert), daniele.varacca@u-pec.fr (Daniele Varacca)

1. Introduction – What *Is* A Context?

In the study of programming languages, contextual equivalences play a central role: to study the behavior of a program, or a process, one needs to observe its interactions with multiple environments, e.g. what outcomes it produces based on different inputs or configurations. 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, e.g. by surrounding implementations of algorithms—snippets—with user-input validation or variable declarations. The user employs them to provide input and obtain an output, e.g. by providing a username and a set of preferences. Finally, the tester or attacker uses them to debug and compare the program or to try to disrupt its intended behavior, e.g. using a fake user environment endowed with a timer for timing attack.

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. Furthermore, we believe that "this purity has a price" [2, p. 1:2], and that unraveling the required subtleties of syntactical definitions will "bring us closer to a realistic programming language or modeling language—[and] that [this] is not always a bad thing." [2, p. 1:2]

Non-technical articles can at times have a tremendous impact [3], 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 also believe that ignoring or downplaying the distinctions we stress have repeatedly caused confusions.

The tone and scope of this essay may seem "timeless", but discussions during the *Combined 27th International Workshop on Expressiveness in Concurrency and 17th Workshop on Structural Operational Semantics*—where our first draft [4] was presented—and the *14th Interaction and Concurrency Experience* (particularly with Ilaria Castellani, Alceste Scalas and Emilio Tuosto) and with colleagues (among which David Baelde and Ross Horne) finished anchoring it into concrete and timely questions.

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

1.1. Changelog

Compared to our post-proceedings publication [5], this version contains an in-depth discussion on monitors (Sect. 6), clarifies a distinction between complete processes and process terms that was absent previously, refrains from using the loaded term "system", and generally improve the wording and intuitions with better and more precise examples and explanations. The general tone of the paper shifted from being focused on contexts to being focused on the interconnection of the components we are discussing, and large sections were re-organized and streamlined to help with readability.

2. The Flow of Testing – A Java Metaphor

We begin by illustrating with a simple Java example the four syntactic notions—*process term*, *(completed) process*, *instrumentation* and *test*—we will be using. Imagine giving a user the code

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

or any other partial implementation of a useful algorithm, as found e.g. in a textbook or on a message board. That user cannot execute or use this "snippet" unless it is *wrapped* into a method, with an adequate header, possibly variable declarations and a `return` statement. Once the programmer has performed this operation, the user can use the obtained program, and the tester can interact with it further, e.g. 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 in Figure 1. We believe this is a fair rendering of "the life of a snippet", that includes other scenarios: typically, the snippet could be shipped already wrapped, and additionally with pre-loaded tests—in this cases, the role of the programmer or the user would be to substitute one wrapping or test with another, but that would not change the different nature of those elements and the division of labor we sketched. Note that we do not distinguish between tests, attacks and interactions with users: they all correspond in our terminology to tests, only their objectives (debugging, compromising, using) differ, but we decided against sub-dividing them further.

In this example, the snippet is what we will call a *process term*, the snippet once wrapped is what we will call a (completed) *process* and the "test" part (i.e., without the completed process in it, but with additional "observations" such as measures on the execution or terminal output) is what we will call a *test*. The result of the insertion of a completed process into its test is called an *instrumentation*, or, sometimes, a *system* and it is the only element that can actually be executed—as neither the process term, the completed process nor the test can be run in isolation. We synthesize this vocabulary and the Java analogy in Figure 2. Our terminology comes from the study of concurrent process algebras, where most of our intuitions and references are located, but we

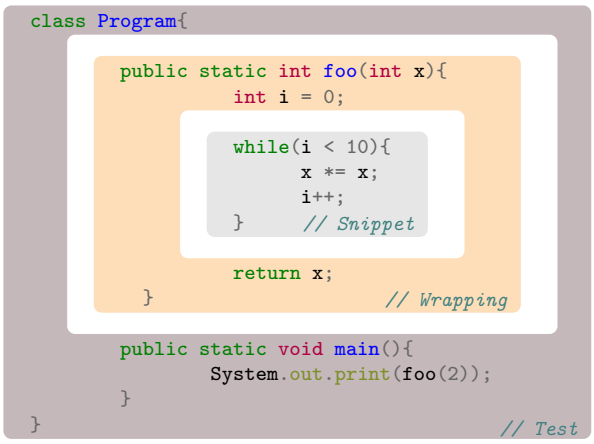

Figure 1: Life of a Snippet

Term	*Built by*	*Obtained by*	*Java Analogy*
Process Term	Programmer	Implementing an algorithm	Snippet
(Completed) Process	Programmer	"Wrapping" the process term	`foo` method
Test	User / Tester	Implementing an usage for the process	Test (`main`)
Instrumentation	Computer	Executing a test calling a process	Compiled `Program`

Figure 2: Roles and Terminology

will make a brief detour to examine how our lens applies to λ-calculus, and was partially inspired by it.

3. A Foreword on λ-Calculus and Its Feedback Loop

Theoretical languages often take λ-calculus as a model or a comparison basis. It is often said that the λ-calculus is to sequential programs what the π-calculus is to concurrent programs [6, 7]. Indeed, pure λ-calculus (i.e. without types or additional features like probabilistic sum [8] or quantum capacities [9, 10]) is a reasonable [11], Turing-complete and elegant language, that requires only a couple of operators, one reduction rule and one equivalence relation to produce a rich and meaningful theory, sometimes seen as an idealized target language for functional programming languages.

Since most terms do not reduce as they are[2], studying their behavior requires to make them interact with an environment, formally represented by a context. Contexts are generally defined as "term[s] with some holes" [13, p. 29, 2.1.18], that we prefer to call *slots* and denote $[\Box]$. Under this apparent simplicity, they should not be manipulated carelessly, as having multiple slots or not being careful

[2]Actually, if application, abstraction and variables all count as one, the ratio between normal term and term with redexes is unknown [12]. We imply here "since *most interesting* terms", i.e. terms that represent programs.

when defining how to place a term in the slot can lead to e.g. lose confluence [14, pp. 40–41, Example 2.2.1], and as those issues persist even in the presence of a typing system [15]. Furthermore, definitions and theorems that use contexts frequently impose some restrictions on the contexts considered, to exclude e.g. $(\lambda x.y)[\Box]$ that simply "throws away" the term placed in the slot in one step of reduction.

In technical terms, contexts generally come in two flavors, depending on the nature of the term for which the context is constructed:

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* [13, 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* [13, p. 246, 10.3.3], which is equivalent [13, p. 246, 10.3.4] to a series of abstractions followed by a series of applications, and sometimes called "head context" [16, p. 25].

Being closed corresponds to being "wrapped"—ready to use—, and feeding arguments to a term corresponds to interacting with it from a `main` method: the Böhm transformation actually encapsulates two operations—closing, then applying the arguments—in one notion. In the λ-calculus, 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 when testing, as only application is allowed: the tested term should not be wrapped in additional layers of abstraction if it is already closed.

Adding features to the λ-calculus 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 [10, p. 1126] and surface context [8, pp. 4, 10] for respectively quantum and probabilistic λ-calculus, to "tame" the power of contexts. In resource sensitive extensions of the λ-calculus, the quest for full abstraction even led to a drastic separation of λ-terms between terms and tests [17], a separation naturally extended to contexts [18, p. 73, Figure 2.4].

This variety happened after the 2000's formal studies of contexts was undertaken [14, 15, 19], 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" [19, p. 29]. It is ironic that λ-calculists took inspiration from a concurrent language to split their syntax in two right at its core [17, 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" of their contexts and their indistinguishability from terms—beside their slot. This re-definition of contexts had impacts on other fields, e.g. on modal type theory [20], but it

seems to us that an opportunity to benefit from this feedback loop was missed in process algebras.

4. Contextual Relations Alter the Definition of Context

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 it is valid in every possible context, an idea formally captured by the notion of (pre-)congruence. 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 $C[\Box]$, $(C[P], C[Q]) \in \mathcal{R}$. Additional requirements occur sometimes, such as requiring that terms in the relation needs to be similar up to uniform substitution [21].

A notable example of congurence is *barbed congruence* [22, Definition 2.1.4][23, Definition 8], which closes by context a reduction-closed relation used to observe "barbs"—the channel(s) on which a process can emit or receive. It is often taken to be *the* "reference behavioural equivalence" [22, p. 4], as it observes the interface of processes, i.e. on which channels they can interact.

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:

In the calculus of communicating systems, notions as central as contextual bisimulation [24, pp. 223-224, Definition 421] and barbed equivalence [24, p. 224, Definition 424] considers only *static* contexts [24, p. 223, Definition 420], which are composed only of parallel composition with an 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" [24, p. 227]. But there is no justification—other than technical, i.e. because they "they persist after a transition" [24, p. 223]—as to *why* one should consider only some contexts in defining contextual equivalences.

In the π-calculus, contexts are sometimes [25, p. 516, Definition 2] drastically restricted to be only closure by substitution. But they are more generally defined liberally [26, p. 19, Definition 1.2.1], however still excluding contexts like e.g. $[\Box] + 0$ without really justifying why. Congruences are then defined using this notion of context [26, p. 19, Definition 1.2.2], and strong barbed congruence is no exception [26, p. 59, Definition 2.1.17]. Other notions, like strong barbed equivalence [26, p. 62, Definition 2.1.20], are shown to be a non-input congruence [26, p. 63, Lemma 2.1.24], which is a notion relying on contexts that forbids the slot to occur under an input prefix [26, 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." [27, p. 245]

In the distributed π-calculus, contexts are restricted right from the beginning to particular operators [27, Definition 2.6]. Then, relations are defined to be contextual if they are preserved by static contexts [27, Definition 2.6], which contains only parallel composition with an arbitrary term and name binding. These contexts also appear as "configuration context" [28, p. 375] or "harness" in the ambient calculus [29, p. 372]. Static operators are deemed "sufficient for our purpose" [27, p. 37] and restrictions to static contexts are implemented "[t]o keep life simple" [27, 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" [21, 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 [21, p. 450] to obtain a notion of *generic reduction* [21, p. 451] that allows a satisfactory definition of sound theories [21, p. 452]. Insensitive terms are essentially the collection of terms that do not interact with contexts [21, p. 451, Proposition 3.15], an analogue to λ-calculuus' genericity Lemma [13, 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"—inspired by λ-calculus [24, p. 85], and matching the "wrapping" of a snippet—can be found in e.g. typed versions of the π-calculus [26, p. 479], in mobile ambient [7, p. 134], in the applied π-calculus [2, p. 7], and in the fusion calculus [30, p. 6]. Also known as "completing context" [31, p. 466], those contexts are parametric in a term, the idea being that such a context will "close" the term under study, making it amenable to tests and comparisons.

Let us try to extract some general principles from this short survey. It seems that contexts are

1. *in appearance* given access to the same operators than terms, with the addition of the slot $\Box$ operator,
2. sometimes deemed "un-reasonable", without always a clear justification—other than technical,
3. shrunken by need, to bypass some of the difficulties they raise, or to preserve some notions[3],
4. sometimes picked by the term itself—typically because the same "wrapping" cannot be applied to all processes.

[3] We expand on this comment, taking as an example the "context lemma", in Appendix A.

Additionally, in all those cases, contexts are given access to a subset of operators, or restricted to contexts with particular behavior, *but never extended*—aside from $\Box$. If we consider that contexts are the main tool to test the equivalence of processes, then why should testers—or attackers—always have access to *fewer* tools than the programmer, and observe only what programmers can observe—i.e. channel names or termination? This perspective, among other flaws, prevents the study and formalization of e.g. side-channel attack, which relies precisely on exploiting different operators and observations to corrupt programs. 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, justifies and acknowledges the restrictions we discussed instead of adding them *passing-by*, and seems closer to common usage and applications.

5. Processes and Tests – Principles in Principled Developments

5.1. Acknowledging Pre-Existing Distinctions

As in the λ-calculus, most concurrent calculi make a distinction between open and closed terms. For instance, the distributed π-calculus [27] implements a distinction between closed terms (called processes [27, 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 (CCS), the theory starts by considering only programs—"closed behaviour expression[s], i.e. ones with no free variable" [32, 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, and can be subject to undesirable or uncontrolled interferences—exactly like a Java snippet cannot be compiled on its own, even if it has been wrapped in a method.

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 [32, p. 68], and by "refusing to admit channels as entities distinct from agents" [33, p. 16] and defining two different notions of scopes [33, 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 [34, 35], there is no notion of closed terms, as every (input) binding on a channel introduces a new and free occurrence of a variable. However, the language they build upon—ECCS [36]—made this distinction clear, by separating channel constants and variables.

Once again in an attempt to mimic the "economy" [37, p. 86] of λ-calculus, but also taking inspiration from the claimed "monotheism" of the actor model [38], 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."

[34, p. 20]. However, it seems that a distinction between those notions always needs to be carefully re-introduced when discussing technically the language [24, p. 258, Remark 493], extensions to it [2, p. 4] or possible implementations [39, p. 13][40]. Finally, let us note that extensions of π-calculus can sometimes have different binders, e.g. making output binders binding in the private π-calculus [41, 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 [26, p. 416] or recursion [33, p. 166]. However, these operators are sometimes not excluded from the start, even if they can never be parts of completed terms. The usual strategy [24, Remark 414][33] is to retain them as long as possible, and to exclude them at the last moment, when their power cannot be tamed any more to fit the framework or prove the desired result, such as the preservation of weak bisimulation by all contexts.

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. Process algebras generally study terms by

1. completing them if needed,
2. inserting them in an environment that contains a test,
3. executing them,
4. observing them thanks to predicates on the execution ("terminates", "emitted the barb a", etc.),

hence constructing equivalences, preorders or metrics [42] on them. Often, the environment is essentially made of another process composed in parallel with the one studied, and tweaked to improve the likeliness of observing a particular behavior: hence, we would like to think of them as tests that the observed completed process has to pass, justifying the terminology we will be using.

5.2. Our Terminology – Using Different Names For Different Objects

Process Terms are "partial" programs, still under development; sometimes called "open terms", they correspond to *incomplete terms*. They would be called code fragments, or snippets, in standard programming. They come with a definition of completion, that transform them into "(closed) terms", or *completed processes*.

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

Instrumentations are obtained by joining together compatible completed processes and tests. How they are joined is part of their definition. Often a notion of empty test is provided to obtain a "dummy instrumentation", or *system*. Instrumentations are ready to be executed and observed, and would correspond to a compiled binary ready to be executed in standard programming.

In the literature of process algebra, the term "process" is commonly used to denote these objects—process terms, completed processes and tests—and the same operator (parallel composition) is used to construct process terms, tests, and to instrument them together, possibly generating confusion. We believe this usage comes from a strong desire to keep all layers uniform, using the same name, operators and rules, but this principle is actually constantly dented (as discussed in Sect. 4), for reasons we expose below. Before doing so, let us note that our terminology is close to the one used e.g. in aDpi [27, Chapter 5] or mobile ambients [43], **in bold below**:

In aDpi [27, Figure 5.2], a **process term** that is closed is **a process** [27, p. 131], and a **located process** is a **system** that can execute. Hence, this calculus actually offers two orthogonal levels of completion (being closed and being located), and accounts for instrumentations later on with the addition of type system and **configurations** [27, Definition 6.1] that let the author defines **actions-in-context**. To perfectly fit our description, aDpi would need to define **systems** (completed processes) reduction using a trivial **configuration** (test) and **action-in-context** (instrumentation) instead of defining its reduction semantics "without test", but it otherwise closely matches our description.

In mobile ambients [43, Table 1], a **process** needs to be placed under an **ambient construct** to become a **system** that can execute. Tests are formulated in terms of **system contexts**, a restriction on **static contexts** that preserve the closedness of the system, and that comes with the trivial instrumentation (i.e., "replace the slot with a system").

5.3. The Design Phases – Separating Tools and Layers

To conculde the proposal, we identify three phases during the design of a process formalism. We believe those three phases should be carried out after having decided what the purpose of the formalism would be—since, as surprising as this may seem, the answer to that question in existing formalisms fluctuated. For instance, CCS was originally supposed to be a programming *and* specification language—an original perspective that was reminded to us by Ilaria Castellani, who we wish to thank. 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" [44, p. 1]. This original research project slightly shifted—from specifying programs to specifying behaviors—, and it is sometimes not clear what the *ultimate* goal of the process algebra is: we believe that the purpose of the formalism should precedes its definition.

1. The first step is to select a set of operators called *construction operators*, used by the programmer to write process terms. Those operators should be expressive, easy to combine, with constraints as light as possible, and selected with in mind the situation that is being modeled—and not thinking whether they fare well with not-yet-defined relations, as it is often done when one chooses to have guarded sum over the internal choice. To ease their usage, a "meta-syntax" can be used, something that is generally represented by a structural equivalence. Another interesting approach, proposed in "the π-calculus, at a distance" [45, p. 45], bypasses the need for a structural equivalence without losing the flexibility it usually provides.

2. The second step requires to define

   (a) a set of *testing operators*,

   (b) a notion of environment—avoiding to use the loaded word "context"—constructed from those operators, along with instructions on which types of completed processes can be placed in it, and how to place them in it,

   (c) a set of observables, i.e. a function from completed processes in environments to a subset of a set of atomic proposition (like "emits barb a", "terminates", "contains recursion operator", etc.).

3. The last step requires to define

   (a) a deployment criteria, explaining what makes a process complete,

   (b) an instrumentation criteria, explaining how a complete process can be placed inside a test,

   (c) and an operational semantics that establishes how the whole can be executed.

   The first two criteria should be defined as a series of conditions on the binding of variables, the presence or absence of some construction operators at top-level, and even the addition of *deployment operators*, marking the process as ready to be deployed in an external environment[4]. Having a set of deployment operators that restricts, expands or intersects with the set of construction operators is perfectly acceptable, and it should enable the transformation of processes and tests into executable instrumentations.

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

Some of those definitions could be mutually referenced: typically, how to complete a process is needed to define correct tests, but may not come before instrumentations are defined, which itself requires a definition of tests.

The important message to take home is that each step uses its own set of operators and generates its own notion of context—to construct, to test or to deploy—and that different actors will manipulate those tools—we discussed for instance *who* should be in charge of constructing the instrumentation in Appendix B.

6. A(n other) Theory of Monitors

As a major example, we will relate Francalanza's monitors [46] to our terminology. Monitors fit our framework for the most part, but there are also some divergences we want to discuss.

Generally speaking, monitors are pieces of code that run alongside another program and observes it to enforces e.g. security policy rules or test some of its properties. Sometimes named "secretaries" [47, p.6], "meta-code" [48] or "edit automata" [49], this range of tools all share the same key component: they are tests in our terminology, and are often thought about in those terms. While they can be interfering or non-interfering with the process they monitor (e.g., change its output or observational behavior or not), all monitors are always thought as being composed in parallel using sometimes a different operator than the one used by programs[5], and as being able to access different functionalities than programs. In Francalanza's work, monitors are described from an operator algebra perspective as running alongside π-calculus processes. We present this formalism, following closely the most recent presentation [46], to which we refer if more details or examples are needed.

6.1. Brief Presentation of Existing Theory

We remind of the standard synchronous early π-calculus with name matching [46, Figure 1]:

Syntax We let $a, b, c, \ldots$ range over channel names, $x, y, z, \ldots$ range over channel name variables, $X, Y, \ldots$ range over process variables, and u, v range over channel names and name variables.

$$
\begin{array}{rlrl}
P, Q \coloneqq & u!v.P & \text{(output)} & \mid u?x.P \quad \text{(input)} \\
\mid & \text{nil} \quad \text{(nil)} & \mid & \text{if } u = v \text{ then } P \text{ else } Q \quad \text{(conditional)} \\
\mid & \text{rec } X.P \quad \text{(recursion)} & \mid & X \quad \text{(process variable)} \\
\mid & P \| Q \quad \text{(parallel)} & \mid & \text{new } c.P \quad \text{(scoping)}
\end{array}
$$

[5] "Inlined reference monitors" [50] are to our knowledge the only type of monitors that are embedded "inside" the code they observe, but they conserve nevertheless a different status than the code that they observe.

$$\frac{}{I \rhd c!d.P \xrightarrow{c!d} P}\ \textsc{pOut} \qquad \frac{}{I \rhd c?x.P \xrightarrow{c?d} P[d/x]}\ \textsc{pIn}$$

$$\frac{}{I \rhd \text{if } c = c \text{ then } P \text{ else } Q \xrightarrow{\tau} P}\ \textsc{pThn} \qquad \frac{}{I \rhd \text{if } c = d \text{ then } P \text{ else } Q \xrightarrow{\tau} Q}\ \textsc{pEls}$$

$$\frac{}{I \rhd \text{rec } X.P \xrightarrow{\tau} P[\text{rec } X.P/X]}\ \textsc{pRec} \qquad \frac{I \rhd P \xrightarrow{\mu} P'}{I \rhd P \| Q \xrightarrow{\mu} P' \| Q}\ \textsc{pPar}$$

$$\frac{I \cup \{d\} \rhd P \xrightarrow{c!d} P' \qquad I \cup \{d\} \rhd Q \xrightarrow{c?d} Q'}{I \rhd P \| Q \xrightarrow{\tau} P' \| Q'}\ \textsc{pCom}$$

$$\frac{I \cup \{d\} \rhd P \xrightarrow{\mu} P' \qquad d \# \mu}{I \rhd \text{new } d.P \xrightarrow{\mu} \text{new } d.P'}\ \textsc{pRes} \qquad \frac{I \cup \{d\} \rhd P \xrightarrow{c!d} P'}{I \rhd \text{new } d.P \xrightarrow{c!d} P'}\ \textsc{pOpn}$$

$$\frac{I \rhd P \xrightarrow{c!d} P' \qquad I \rhd Q \xrightarrow{c?d} Q' \qquad d \# I}{I \rhd P \| Q \xrightarrow{\tau} \text{new } d.(P' \| Q')}\ \textsc{pCls}$$

Figure 3: Semantics of π-calculus

Semantics The labeled transition system (LTS), defined in Figure 3, produces judgments of the form $I \rhd P \xrightarrow{\mu} P'$, where

1. the set of channel names I is the *interface* of names shared by the process and an implicit observer with which the interactions occur, and it must be s.t. the set of names in P is included in I,
2. P is a closed term, i.e. with no free channel name variable, knowing that $c?x.P$ binds x in P,
3. μ ranges over input actions $c?d$, output actions $c!d$ and a distinguished silent action τ.

We write $o \# o'$ if the two syntactical objects o and o' share no free names, and $P[Q'/Q]$ for the result of the substitution of Q by Q' in P. We also omitted the symmetric rules for pPar, pCom and pCls.

A monitor may reach either of two *verdicts*, detection ($\checkmark$) or termination (end), denoting respectively success and an inconclusive verdict.

Syntax We let α range over external actions—all actions but τ—o, r range over names, variables and variable binders, p, q range over *patterns*—input and output actions—, and define $\text{match}(p, \alpha)$—which is either undefined, or a substitution σ—in an usual way [46, Sect. 3].

$$\frac{}{w \xrightarrow{\alpha} w}\ \textsc{mVer} \qquad \frac{\mathrm{match}(p,\alpha) = \sigma}{p.M \xrightarrow{\alpha} M\sigma}\ \textsc{mPat}$$

$$\frac{M \xrightarrow{\mu} M'}{M + N \xrightarrow{\mu} M'}\ \textsc{mChL} \qquad \frac{}{\mathrm{rec}\, X.M \xrightarrow{\tau} M[\mathrm{rec}\, X.M/X]}\ \textsc{mRec}$$

$$\frac{}{\mathrm{if}\, c = c\, \mathrm{then}\, M\, \mathrm{else}\, N \xrightarrow{\tau} M}\ \textsc{mThn} \qquad \frac{c \# d}{\mathrm{if}\, c = d\, \mathrm{then}\, M\, \mathrm{else}\, N \xrightarrow{\tau} N}\ \textsc{mEls}$$

Figure 4: Semantics of monitors

$$\frac{I \rhd P \xrightarrow{\alpha} P' \qquad M \xrightarrow{\alpha} M'}{I \rhd P \lhd M \xrightarrow{\alpha} P' \lhd M'}\ \textsc{iMon} \qquad \frac{I \rhd P \xrightarrow{\tau} P'}{I \rhd P \lhd M \xrightarrow{\tau} P' \lhd M}\ \textsc{iAsyP}$$

$$\frac{I \rhd P \xrightarrow{\alpha} P' \qquad M \overset{\alpha}{\not\rightarrow} \qquad M \overset{\tau}{\not\rightarrow}}{I \rhd P \lhd M \xrightarrow{\alpha} P' \lhd \mathrm{end}}\ \textsc{iTer} \qquad \frac{M \xrightarrow{\tau} M'}{I \rhd P \lhd M \xrightarrow{\tau} P \lhd M'}\ \textsc{iAsyM}$$

Figure 5: Instrumentation semantics of configurations

$$\begin{array}{rll} p, q \coloneqq o?r & \text{(input pattern)} & \mid o!r \quad \text{(output pattern)} \\ w \coloneqq \checkmark & \text{(detection verdict)} & \mid \mathrm{end} \quad \text{(termination verdict)} \\ M, N \coloneqq w & \text{(verdict)} & \mid p.M \quad \text{(pattern match)} \\ \mid M + N & \text{(choice)} & \mid \mathrm{if}\, u = v\, \mathrm{then}\, M\, \mathrm{else}\, N \quad \text{(branch)} \\ \mid \mathrm{rec}\, X.M & \text{(recursion)} & \mid X \quad \text{(monitor variable)} \end{array}$$

Semantics The labeled transition system (LTS) for monitors, defined in Figure 4, produces judgments of the form $M \xrightarrow{\mu} M'$. We omitted the rule mChR, that can easily being inferred.

A *monitored system* $P \lhd M$ consists of a process P instrumented with a monitor M analyzing its external behavior. The *instrumentation semantics* for *configurations*, $I \rhd P \lhd M$, is given in Figure 5, where $M \overset{\alpha}{\not\rightarrow}$ and $M \overset{\tau}{\not\rightarrow}$ are used to denote that M cannot reduce.

The study generally looks for a couple of key properties:

- That verdicts are irrevocable [46, Proposition 1], i.e. that a monitor will not "change its mind" on the result of its analysis.
- That verdict do not interfere [46, Proposition 2], i.e. that a process will execute the same way regardless of it being "wrapped" and tested in a configuration or executing on its own.
- The definition and characterization of pre-orders on potential (resp. deterministic) detections [46, Definition 3], that quantifies over some (resp. all)

traces of configurations.

6.2. How Does the Theory of Monitors Fits Our Frame?

The first (crucial) point to note is that *monitors, and not processes, are tested*, as it is *on monitors* that pre-order are defined, using *processes to test them*. Once this "flipping" of terminology is acknowledged, one can observe that the theory fits pretty closely our frame: process terms ("pre-monitors") have a completion mechanism to become processes (typically, a pattern cannot have free variables [46, Section 3] in a monitor) and are syntactically different from tests (π-calculus terms). Aside from the conditional and recursion, tests and processes do not share any constructor, as monitors are shaped like patterns, trees: they do not have access to parallelism, terminate in a state of success or failure, and have access to non-deterministic sum and a match operator. The semantics is also different: while the main execution mechanism of tests represents the passing of messages (pOut and pIn), monitors mostly proceeds by pattern-matching (mPat).

The instrumentation mechanism is also formally specified and far from trivial, as it involves an explicit interface I containing the names shared by the tests and the observer that acts like a "deployment operator": it marks that a test is ready to be used and contains its instructions, explaining how to use it. Not every test is suited for every monitor—they must agree on the interface—and their interactions is complex: the test drives the execution of the configuration, and hence of the monitor, on external actions (iMon and iTer), but they can proceed independently on internal actions (iAsyp and iAsym), providing a rich and delicate mechanism to execute configurations.

Last but not least, the completion mechanisms are formally stipulated for all syntactical objects (tests, monitors and configurations), also thanks to the clear distinction between channel names and channel name variables. That monitors can not alter the behavior of their tests is an interesting feature, that is harder to obtain because of inessential syntactical complications [46, Example 5]. It is also interesting to note that the monitor preorders and their characterizations do not impose any additional constraints on the syntactical objects, but re-inforces the idea that "a proper definition of monitor correctness needs to take into consideration system instrumentation" [46, p. 25]: a theory of tests or processes without a solid, technical, definition of their interactions (discussing e.g. whenever monitors can learn new channel names [46, Section 9.1]) should always be taken with a pinch of salt. Also, instead of being contrived and simpler, tests "are usually far more complex than the monitors that analyse them." [46, p. 25].

This theory could still be technically refined on a couple of aspects to get closer to our guiding principles. Typically, we believe that the current τ-transitions of monitors (mRec, mThn and mEls) should be replaced by a structural congruence: those transitions do not correspond to an interesting action, simply to some syntactical simplification, and should be treated as such. It could also be more elegant to close the processes and monitors, and to observe only silent transitions (that would need to carry a mention of the channel name on which they took place): a proper reduction semantics for configurations would then

allow to execute only independent, self-contained, configurations, still letting processes and monitors access each other internal transitions. As a consequence, the preorders would not need to quantify over traces [46, Definition 4], since *the configuration* could evolve based on the channel names used in the transition, this quantification could be removed from the predicates `pp` and `pd` [46, Definition 3].

7. Further Applications

We propose here some more applications of our framework.

7.1. Re-Framing Existing Issues

Co-defining observations and contexts Originally, the barb was a predicate [23, p. 690], whose definition was purely syntactic. Probably inspired by the notion of observer for testing equivalences [51, p. 91], an alternative definition was made in terms of parallel composition with a tester process [22, p. 10, Definition 2.1.3]. This illustrates perfectly how the set of observables and the operators allowed in testing contexts are inter-dependent, and that tests should always come with a definition of observable. We believe our proposal could help in clarifying this interplay, and in opening up the possibility of obtaining 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 transition was dubbed "unobservable internal activity" [27, p. 6] and sometimes opposed to "externally observable actions" [52, p. 230]. While we agree that "[t]his abstraction from internal differences is essential for any tractable theory of processes" [33, 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.

The question "to what extent should one identify processes differing only in their internal or silent actions?" [53, p. 6] is sometimes asked, and discussed as if it was a property of the process algebra and not something that can be *internally* tuned when 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 sense to let a programmer observe them when testing to help in deciding which process to prefer based on information not available to users.

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" [52, p. 160]. Sometimes, a similar question is phrased in terms of choosing the right level of abstraction to obtain meaningful language comparisons [54, Section 3]. We firmly believe that the best answer to both questions is to acknowledge that different relations and comparisons tools match different needs, and that there is no "one size fits all" answer for the needs of all the variety of testers. Of course, comparing multiple relations is an interesting and needed task [55, 56], 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 tests 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 [43]. The authors start with a two-level syntax that distinguishes between processes and systems [43, p. 966]. Processes have access to strictly more constructors than systems [43, p. 967, Table 1], that are supposed to hide the threads of computation [43, 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 [43, p. 969, Definiton 2.2].

The authors even extend further the syntax for processes with a special $\circ$ operator [43, 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 two different reduction barbed congruences [43, 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 [43, 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 to assert that, sometimes, systems can and *should* do more than processes—for instance, interacting with users!—, 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 [57, p. 280] require it, and multiple languages [58, p. 24] use it to provide finer-grained equivalences. For technical reasons [26, 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 extensions of processes algebras as different completion strategies for the same construction operators. For instance, reversible [59] or timed [60] extensions of CCS could be seen as two completion strategies—different conditions for a process term to become a process—for the same class of process term, inspired from the usual CCS syntax [24, Chapter 28.1]. Those completion strategies would be suited for different needs, as one could e.g. complete a CCS process term as a RCCS [61] process to test for relations such as hereditary history-preserving bisimulation [62], and then complete it with time markers to obtain a safety-critical process, with possibly a different way of constructing instrumentation and a different reduction semantics. This would correspond to having multiple compilation, or deployment, strategies, based on the need, similar to "debug" and "real-time", versions of the same piece of software. We think also of Debian's DebugPackage, enabling 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.

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 [26, Part III], but diverse difficulties remain. Among them, the co-existence of multiple calculi for e.g. session types [63], the difficulty to tie them precisely to other type systems, such as Linear Logic [64], and the doubts about the adaptation of the "proof-as-program" paradigm in a concurrent setting [65], make this problem 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 process terms 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 [66, pp. 253–258], that

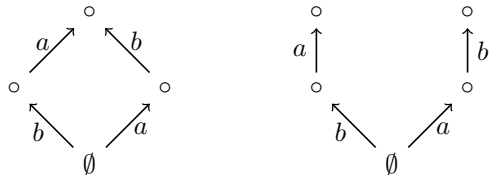

Figure 6: Labeled Configuration Structures

checks the typing discipline at compilation time, but not at run-time.

We hope this series of examples illustrates how our proposal could clarify pre-existing distinctions. In the next sections we show how additional progresses could be made using it, e.g. in CCS, π-calculus, and security.

7.2. Applications to CCS

Testing for auto-concurrency Auto-concurrency (a.k.a. auto-parallelism) is when a system has two different transitions—leading to different states—labeled with the same action [67, p. 391, Definition 5]. Systems with auto-concurrency are sometimes excluded as non-valid terms [68, p. 155] or simply not considered in particular models [69, p. 531], as the definition of bisimulation is problematic for them.

Consider e.g. the labeled configuration structures (a.k.a. stable family [70, Section 3.1]) in Figure 6, where the label of the event executed is on the edge and configurations are represented with $\circ$. Non-interleaving models of concurrency [71] distinguish between them, as "true concurrency models" would.

Some forms of "back-and-forth-bisimulations" cannot discriminate between them if $a = b$ [72]. 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 processes—that are, after all, unjustifiably excluded—in the realm of comparable processes.

Representing man-in-the-middle One could add to the testing operators an operator $\nabla a.P$, which would forbid P to act silently on channel a. This novel operator would add the possibility for the environment to "spy" on a determined channel, as if the environment was controlling (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 to forward, prevent, or alter 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.

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

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

While the first rule enforces that, once secured, channel names are α-equivalent (the process can decide to migrate to a different channel without being spied on), the second illustrates that, once a channel is tapped, a process cannot retrieve confidentiality on it.

Improving reversible calculi Reversible CCS (RCCS) [61] and CCS with keys (CCSK) [73] are two extensions to CCS aiming at formalizing reversible concurrent computation, that actually are the two faces of the same coin [59]. However, as a recent survey on the state-of-the-art in reversible computation puts it bluntly, "[u]nderstanding which notions of behavioural equivalences are suitable for reversible process calculi is a non-trivial, and still open, problem" [74, p. 15]. Two recent studies [75, 76] tried to overcome this shortcoming, starting by defining contexts, but they came to exactly opposite conclusions: while the RCCS-inspired system seems to provide definitive evidence that *no relation can be a congruence if the context can change the history of the process* [75, Theorem 2], the CCSK approach came to the conclusion that a particular barbed bisimilarity was a congruence [76, Corollary 5.12]. Does the difference rest only on the definition of context, or is there a more subtle distinction at work? It is our hope that the approach sketched here could help to solve this mystery, by studying the difference between those instrumentations.

7.3. Applications to the π-Calculus

In the π-calculus, tests must be instantiating contexts (in the sense that the process term needs to be either already closed, or closed by the context), and instantiating contexts can use only construction operators, and hence are construction contexts. This situation corresponds to Situation A in Figure 7.

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 would obtain Situation B in Figure 7. 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" [77, p. 57], being added to the construction operators while most authors seem to agree that they would prefer not to add it to the internals

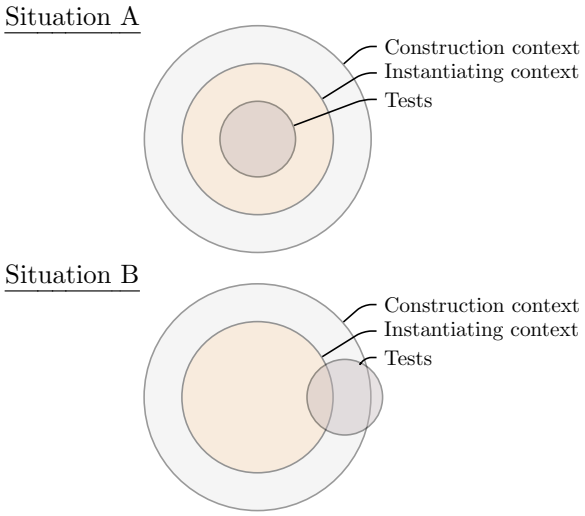

Figure 7: Opening up the testing capacities of π-calculus

of the language[6]. 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 [79] to *quasi-open bissimilarities* [80, 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 completion not requiring closedness 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." [15, 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.

In the distributed π-calculus, one 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 process terms and the other on processes. Such a distinction would allow to observationally distinguish e.g. the execution of a program with two threads on a dual-core computer and the execution of two single-thread programs on two single-core computers.

[6]To be more precise: while "most occurences of matching can be encoded by parallel composition [...,] mismatching cannot be encoded in the original π-calculus" [78, p. 526], which makes it somehow suspicious.

7.4. Applications to the Formal Study of Security

Separating equivalences Semantical approaches to security revealed multiple small gaps that are hard to explain or solve with usual notions of contexts and equivalences. For instance, the study of equivalence properties led to distinguish between semantics where all the communications have to be made via the environment (i.e., the attacker) and semantics where communications inside the process and communications between the process and its environment are treated differently. The (surprising) result [81] is that while both treatments coincide for reachability properties, they are incomparable for indistinguishability properties. The resulting *classical*, *private* and *eavesdropping* semantics each yield their own may-testing and observational equivalences [82, Section 4.1.6], that could be elegantly captured by our frame as different instrumentations.

Studying cryptographic protocols 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 [83]. This perspective was already partially at play in the spi calculus for cryptographic protocols [58, 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 processes, and need to be detached from construction contexts. This could make "the formalization of attackers as contexts [...] continue to play a role in the analysis of security protocols" [2, p. 35].

Recent progresses in the field of verification of cryptographic protocols [82] hinted in this direction as well. By taking "[t]he notion of test [to] be relative to an environment" [82, p. 12], a formal development involving "frames" [82, 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" [82, p. 12], that cannot be used to construct process terms, to construct "experiments".

8. Concluding Remarks

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 [84, Remark 5.2.21]. In a 1981 article, it is assumed that "by varying the ambiant ('weather') conditions, an experimenter" [85, 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 execute the process 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 process—which is something closer to specifying the instrumentation, something formally captured by e.g. "a language for testing concurrent processes" [86, p. 1] that typically included a termination operator and probabilistic features not available to the programmer.

Before daring writing such a lengthy, 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 process terms, processes, tests and instrumentations. 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. 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 is discussed and argued in more details in Appendix A.

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 lens to read and answer some of the un-spoken questions suggested in the margin or the footnotes—but rarely upfront—of the multiple references we consulted. Even without mathematical proofs, we consider this contribution a good way of stirring the community, and to question the traditional wisdom.

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 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 [87]—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 contextwill 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.

Acknowledgment

The authors wish to thank the organizers of EXPRESS/SOS 2020 and ICE 2021 for organizing those welcoming, open venues (especially considering they

had to switch to virtual events), as well as the reviewers who kindly shared their comments, suggestions and insights with us. This paper benefited a lot from them, as well as from the discussions mentioned at the end of Sect. 1.

Appendix A. About Context Lemmas – How Contexts Are Sometimes Shrunken by Need

What is generally refereed to as *the* context lemma[7] 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 [26, p. 95, Definition 2.4.5] as the closure of barbed bisimilarity under all contexts, or only under contexts of the form $[\Box]\sigma \mid P$ for all substitution σ and term P. In its first version [89, 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*".

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, and often done. For instance, contexts like $[\Box] + 0$ are routinely removed—as we mentioned in Sect. 4—to define the barbed congruence of π-calculus, or contexts are restricted to what is called harnesses in the mobile ambients calculus [29] before proving such results. As strong bisimulation [25, p. 514, Definition 1] is not preserved by input prefix [25, 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 [26, p. 62, Definition 2.1.22] do, and then to establish a context lemma for this limited notion of context. Another way of convincing oneself that context lemmas use specific features of languages, in a narrow sense, and that they may not be the cornerstone they sometimes seem to be, is to remark that no context lemma can exist in the "Situation B" of Figure B.8 [26, p. 117], but that this did not prevent from studying those type of relations.

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 having ahead of time excluded all the construction operators that would not fit, hence producing a result 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 testing context belongs to the algebra: on the opposite, one would want contexts to *increase* the discriminating power—as for the mismatch operator, mentioned in Sect. 7.1—and not to "conform" to

[7] At least, in process algebra, as the same name is used with a different meaning in e.g. λ-calculus [88, p. 6].

what some of the construction operators (typically, substitution and parallel composition) have already decided.

Context lemmas seem to embrace an uncanny perspective: instead of being used to prove properties about tests more easily, they should be considered from the perspective of the ease of use of testing systems. Stated differently, we believe that the set of testing operators should come first, and then *then*, if the language designer wishes to add operators to ease the testers' life, they can do so providing they obtain a context lemma proving that those operators do not alter the original testing capacities. Once again, varying the testing suite is perfectly acceptable, but once 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.*

Appendix B. When Should Contexts Come into Play?

The interesting question of *when* to use contexts when testing terms [26, 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 [26, p. 116]), or being defined in two steps, i.e. as the contextual closure of a pre-existing relation (e.g. strong barbed congruence [26, p. 61, Definition 2.1.17], which is the contextual closure of strong barbed bisimilarity [26, p. 57, Definition 2.1.7])?

Indeed, bisimulations can be presented as an "interaction game" [90] generally played as

1. Pick an environment for both terms (i.e., complete them, then embed them the same way 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 observational congruence" [91], provides a better software modularity and reusability, as it allows to study the similarity of terms that can be reconfigured 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 [21, p. 455, Proposition 3.24], and that corresponds to open bisimilarities [92, p. 77, Proposition 3.12]

Those two approaches have been extensively compared and studied—still are [2, p. 24]—but to our knowledge they rarely co-exist, as if one had to take a side at the early stage of the language design, instead of letting the tester decide later on which approach is best suited for what they wish to observe. We argue that both approaches are equally valid, *provided we acknowledge they play different roles.*

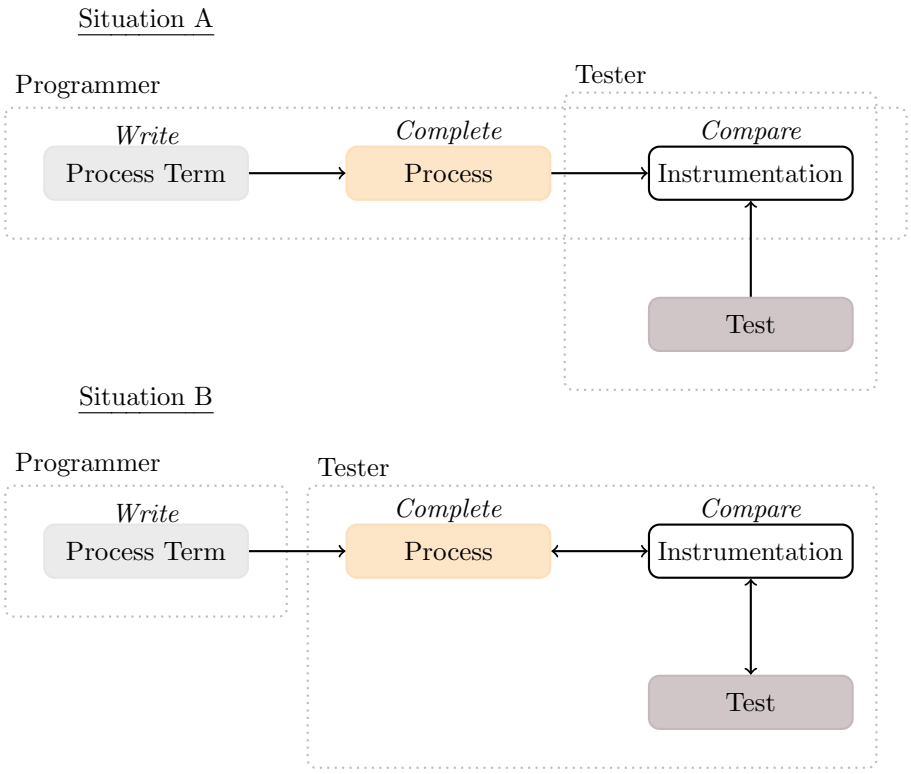

Figure B.8: Distinguishing between divisions of labor

This question of *when are the process 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 e.g. 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—how the instrumentation will be obtained is agreed upon and cannot be changed, the tester can only change the nature of the test—to a *dynamic* one—where the tester can change the completion, how the instrumentation is obtained and the test repeatedly—would corresponds to going from Situation A to Situation B in Figure B.8.

This illustrates two aspects worth highlighting:

1. Playing on the variation "*should I complete the process terms before or during their comparison?*" is not simply a technical question, but reflects a choice between two different situations equally interesting.

2. This choice can appeal to different notions of process terms, completions, instrumentation and tests: for instance, while completing a process term 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)—including how its instrumentation is obtained.

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, some comparing binaries—to get a better, fuller, picture of the program.

References

[1] P. Taylor, Comment to "substitution is pullback", On-line comment, last accessed: 2022-01-03 (09 2012).
URL `http://math.andrej.com/2012/09/28/substitution-is-pullback/`

[2] M. Abadi, B. Blanchet, C. Fournet, The applied pi calculus: Mobile values, new names, and secure communication, Journal of the ACM 65 (1) (2018) 1:1–1:41. `doi:10.1145/3127586`.

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

[4] C. Aubert, D. Varacca, Process, Systems and Tests: Three Layers in Concurrent Computation (Short Paper), working paper (Jul. 2020).
URL `https://hal.archives-ouvertes.fr/hal-02899123`

[5] C. Aubert, D. Varacca, Processes, systems & tests: Defining contextual equivalences, in: J. Lange, A. Mavridou, L. Safina, A. Scalas (Eds.), Proceedings 14th Interaction and Concurrency Experience, Online, 18th June 2021, Vol. 347 of Electronic Proceedings in Theoretical Computer Science, Open Publishing Association, 2021, pp. 1–21. `doi:10.4204/EPTCS.347.1`.

[6] D. Sangiorgi, Pi-calculus, in: D. A. Padua (Ed.), Encyclopedia of Parallel Computing, Springer, 2011, pp. 1554–1562. `doi:10.1007/978-0-387-09766-4_202`.

[7] C. A. Varela, Programming Distributed Computing Systems: A Foundational Approach, The MIT Press, 2013.

[8] C. Faggian, S. Ronchi Della Rocca, Lambda calculus and probabilistic computation, in: LICS, IEEE, 2019, pp. 1–13. doi:10.1109/LICS.2019.8785699.

[9] P. Selinger, B. Valiron, Quantum lambda calculus, in: S. Gay, I. Mackie (Eds.), Semantic Techniques in Quantum Computation, Cambridge University Press, 2009, p. 135–172. doi:10.1017/CBO9781139193313.005.

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

[11] B. Accattoli, U. Dal Lago, Beta reduction is invariant, indeed, in: T. A. Henzinger, D. Miller (Eds.), CSL, Association for Computing Machinery, 2014, p. 8. doi:10.1145/2603088.2603105.

[12] O. Bodini, Personal communication (01 2021).

[13] H. P. Barendregt, The Lambda Calculus – Its Syntax and Semantics, Vol. 103 of Studies in Logic and the Foundations of Mathematics, North-Holland, 1984. doi:10.1016/B978-0-444-87508-2.50006-X.

[14] M. Bognar, Contexts in lambda calculus, Ph.D. thesis, Vrije Universiteit Amsterdam (2002).
URL https://www.cs.vu.nl/en/Images/bognar_thesis_tcm210-92584.pdf

[15] M. Hashimoto, A. Ohori, A typed context calculus, Theoretical Computer Science 266 (1-2) (2001) 249–272. doi:10.1016/S0304-3975(00)00174-2.

[16] B. Accattoli, U. D. Lago, On the invariance of the unitary cost model for head reduction, in: A. Tiwari (Ed.), 23rd International Conference on Rewriting Techniques and Applications (RTA'12), RTA 2012, May 28 - June 2, 2012, Nagoya, Japan, Vol. 15 of Leibniz International Proceedings in Informatics, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2012, pp. 22–37. doi:10.4230/LIPIcs.RTA.2012.22.

[17] A. Bucciarelli, A. Carraro, T. Ehrhard, G. Manzonetto, Full Abstraction for Resource Calculus with Tests, in: M. Bezem (Ed.), CSL, Vol. 12 of Leibniz International Proceedings in Informatics, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2011, pp. 97–111. doi:10.4230/LIPIcs.CSL.2011.97.

[18] F. Breuvart, Dissecting denotational semantics, Ph.D. thesis, Université Paris Diderot — Paris VII (2015).
URL https://www.lipn.univ-paris13.fr/~breuvart/These_breuvart.pdf

[19] M. Bognar, R. C. de Vrijer, A calculus of lambda calculus contexts, Journal of Automated Reasoning 27 (1) (2001) 29–59. doi:10.1023/A:1010654904735.

[20] A. Nanevski, F. Pfenning, B. Pientka, Contextual modal type theory, ACM Transactions on Computational Logic 9 (3) (Jun. 2008). doi:10.1145/1352582.1352591.

[21] K. Honda, N. Yoshida, On reduction-based process semantics, Theoretical Computer Science 151 (2) (1995) 437–486. doi:10.1016/0304-3975(95)00074-7.

[22] J.-M. Madiot, Higher-order languages: dualities and bisimulation enhancements, Ph.D. thesis, École Normale Supérieure de Lyon, Università di Bologna (2015).
URL https://hal.archives-ouvertes.fr/tel-01141067

[23] R. Milner, D. Sangiorgi, Barbed bisimulation, in: W. Kuich (Ed.), Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings, Vol. 623 of Lecture Notes in Computer Science, Springer, 1992, pp. 685–695. doi:10.1007/3-540-55719-9_114.

[24] R. M. Amadio, Operational methods in semantics, Lecture notes, Université Denis Diderot Paris 7 (Dec. 2016).
URL https://hal.archives-ouvertes.fr/cel-01422101

[25] J. Parrow, An introduction to the π-calculus, in: J. A. Bergstra, A. Ponse, S. A. Smolka (Eds.), Handbook of Process Algebra, North-Holland / Elsevier, 2001, pp. 479–543. doi:10.1016/b978-044482830-9/50026-6.

[26] D. Sangiorgi, D. Walker, The Pi-calculus, Cambridge University Press, 2001.

[27] M. Hennessy, A distributed Pi-calculus, Cambridge University Press, 2007. doi:10.1017/CBO9780511611063.

[28] I. Lanese, M. Lienhardt, C. A. Mezzina, A. Schmitt, J. Stefani, Concurrent flexible reversibility, in: M. Felleisen, P. Gardner (Eds.), Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, Vol. 7792 of Lecture Notes in Computer Science, Springer, 2013, pp. 370–390. doi:10.1007/978-3-642-37036-6_21.

[29] A. D. Gordon, L. Cardelli, Equational properties of mobile ambients, Mathematical Structures in Computer Science 13 (3) (2003) 371–408. doi:10.1017/S0960129502003742.

[30] M. Merro, On the expressiveness of chi, update, and fusion calculi, in: I. Castellani, C. Palamidessi (Eds.), Fifth International Workshop on Expressiveness in Concurrency, EXPRESS 1998, Satellite Workshop of CONCUR 1998, Nice, France, September 7, 1998, Vol. 16 of Electronic Notes in Theoretical Computer Science, Elsevier, 1998, pp. 133–144. doi:10.1016/S1571-0661(04)00122-7.

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

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

[33] R. Milner, Communication and Concurrency, PHI Series in computer science, Prentice-Hall, 1989.

[34] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I, Information and Computation 100 (1) (1992) 1–40. doi:10.1016/0890-5401(92)90008-4.

[35] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, II, Information and Computation 100 (1) (1992) 41–77. doi:10.1016/0890-5401(92)90009-5.

[36] U. Engberg, M. Nielsen, A calculus of communicating systems with label passing - ten years after, in: G. D. Plotkin, C. Stirling, M. Tofte (Eds.), Proof, Language, and Interaction, Essays in Honour of Robin Milner, The MIT Press, 2000, pp. 599–622.

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

[38] C. Hewitt, P. B. Bishop, I. Greif, B. C. Smith, T. Matson, R. Steiger, Actor induction and meta-evaluation, in: P. C. Fischer, J. D. Ullman (Eds.), Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973, ACM Press, 1973, pp. 153–168. doi:10.1145/512927.512942.

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

[40] S. Fowler, S. Lindley, P. Wadler, Mixing metaphors: Actors as channels and channels as actors, in: P. Müller (Ed.), ECOOP 2017, Vol. 74 of Leibniz International Proceedings in Informatics, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017, pp. 11:1–11:28. doi:10.4230/LIPIcs.ECOOP.2017.11.

[41] C. Palamidessi, F. D. Valencia, Recursion vs replication in process calculi: Expressiveness, Bulletin of the EATCS 87 (2005) 105–125.
URL http://eatcs.org/images/bulletin/beatcs87.pdf

[42] E. Horita, K. Mano, A metric semantics for the π-calculus extended with external events, Kôkyûroku 996 (1997) 67–81.
URL http://hdl.handle.net/2433/61239

[43] M. Merro, F. Zappa Nardelli, Behavioral theory for mobile ambients, Journal of the ACM 52 (6) (2005) 961–1023. doi:10.1145/1101821.1101825.

[44] R. 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/

[45] B. Accattoli, Evaluating functions as processes, in: R. Echahed, D. Plump (Eds.), Proceedings 7th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2013, Rome, Italy, 23th March 2013, Vol. 110 of EPTCS, 2013, pp. 41–55. doi:10.4204/EPTCS.110.6.

[46] A. Francalanza, A theory of monitors, Information and Computation 281 (2021) 104704. doi:10.1016/j.ic.2021.104704.

[47] P. B. Hansen, Monitors and concurrent pascal: A personal history, in: J. A. N. Lee, J. E. Sammet (Eds.), History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20-23, 1993, ACM, 1993, pp. 1–35. doi:10.1145/154766.155361.

[48] F. B. Schneider, Enforceable security policies, ACM Transactions on Privacy and Security 3 (1) (2000) 30–50. doi:10.1145/353323.353382.

[49] J. Ligatti, L. Bauer, D. Walker, Edit automata: enforcement mechanisms for run-time security policies, International Journal of Information Security 4 (1-2) (2005) 2–16. doi:10.1007/s10207-004-0046-8.

[50] U. Erlingsson, The inlined reference monitor approach to security policy enforcement, Ph.D. thesis, Cornell University (2004).
URL https://hdl.handle.net/1813/5628

[51] R. De Nicola, M. Hennessy, Testing equivalences for processes, Theoretical Computer Science 34 (1984) 83–133. doi:10.1016/0304-3975(84)90113-0.

[52] D. Sangiorgi, J. Rutten (Eds.), Advanced Topics in Bisimulation and Coinduction, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2011. doi:10.1017/CBO9780511792588.

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

[54] I. Lanese, C. Vaz, C. Ferreira, On the expressive power of primitives for compensation handling, in: A. D. Gordon (Ed.), Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings, Vol. 6012 of Lecture Notes in Computer Science, Springer, 2010, pp. 366–386. doi:10.1007/978-3-642-11957-6_20.

[55] C. Fournet, G. Gonthier, A hierarchy of equivalences for asynchronous calculi, Journal of Logical and Algebraic Methods in Programming 63 (1) (2005) 131–173. doi:10.1016/j.jlap.2004.01.006.

[56] R. J. van Glabbeek, The linear time - branching time spectrum II, in: E. Best (Ed.), CONCUR '93, Vol. 715 of Lecture Notes in Computer Science, Springer, 1993, pp. 66–81. doi:10.1007/3-540-57208-2_6.

[57] M. Boreale, R. D. Nicola, Testing equivalence for mobile processes, Information and Computation 120 (2) (1995) 279–303. doi:10.1006/inco.1995.1114.

[58] M. Abadi, A. D. Gordon, A calculus for cryptographic protocols: The spi calculus, Information and Computation 148 (1) (1999) 1–70. doi:10.1006/inco.1998.2740.

[59] I. Lanese, D. Medić, C. A. Mezzina, Static versus dynamic reversibility in CCS, Acta Informatica (Nov. 2019). doi:10.1007/s00236-019-00346-6.

[60] W. Yi, CCS + time = an interleaving model for real time systems, in: J. L. Albert, B. Monien, M. Rodríguez-Artalejo (Eds.), Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings, Vol. 510 of Lecture Notes in Computer Science, Springer, 1991, pp. 217–228. doi:10.1007/3-540-54233-7_136.

[61] V. Danos, J. Krivine, Reversible communicating systems, in: P. Gardner, N. Yoshida (Eds.), CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings, Vol. 3170 of Lecture Notes in Computer Science, Springer, 2004, pp. 292–307. doi:10.1007/978-3-540-28644-8_19.

[62] C. Aubert, I. Cristescu, How reversibility can solve traditional questions: The example of hereditary history-preserving bisimulation, in: I. Konnov, L. Kovács (Eds.), 31st International Conference on Concurrency Theory, CONCUR 2020, September 1–4, 2020, Vienna, Austria, Vol. 2017 of Leibniz International Proceedings in Informatics, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 13:1–13:24. doi:10.4230/LIPIcs.CONCUR.2020.13.

[63] B. van den Heuvel, J. A. Pérez, Session type systems based on linear logic: Classical versus intuitionistic, in: S. Balzer, L. Padovani (Eds.), PLACES@ETAPS 2020, Vol. 314 of EPTCS, 2020, pp. 1–11. doi:10.4204/EPTCS.314.1.

[64] L. Caires, F. Pfenning, B. Toninho, Linear logic propositions as session types, Mathematical Structures in Computer Science 26 (3) (2016) 367–423. doi:10.1017/S0960129514000218.

[65] E. Beffara, V. Mogbil, Proofs as executions, in: J. C. M. Baeten, T. Ball, F. S. de Boer (Eds.), IFIP TCS, Vol. 7604 of Lecture Notes in Computer Science, Springer, 2012, pp. 280–294. doi:10.1007/978-3-642-33475-7_20.

[66] P. Niemeyer, D. Leuck, Learning Java, 4th Edition, O'Reilly Media, Incorporated, 2013.

[67] M. Nielsen, C. Clausen, Bisimulation for models in concurrency, in: B. Jonsson, J. Parrow (Eds.), CONCUR '94, Vol. 836 of Lecture Notes in Computer Science, Springer, 1994, pp. 385–400. doi:10.1007/BFb0015021.

[68] R. De Nicola, U. Montanari, F. W. Vaandrager, Back and forth bisimulations, in: J. C. M. Baeten, J. W. Klop (Eds.), CONCUR '90, Vol. 458 of Lecture Notes in Computer Science, Springer, 1990, pp. 152–165. doi:10.1007/BFb0039058.

[69] M. Nielsen, U. Engberg, K. S. Larsen, Fully abstract models for a process language with refinement, in: J. W. de Bakker, W. P. de Roever, G. Rozenberg (Eds.), Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings, Vol. 354 of Lecture Notes in Computer Science, Springer, 1989, pp. 523–548. doi:10.1007/BFb0013034.

[70] G. 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

[71] V. Sassone, M. Nielsen, G. Winskel, Models for concurrency: Towards a classification, Theoretical Computer Science 170 (1-2) (1996) 297–348. doi:10.1016/S0304-3975(96)80710-9.

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

[73] I. Phillips, I. Ulidowski, Reversing algebraic process calculi, in: L. Aceto, A. Ingólfsdóttir (Eds.), Foundations of Software Science and Computation Structures, 9th International Conference, FOSSACS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-31, 2006, Proceedings, Vol. 3921 of Lecture Notes in Computer Science, Springer, 2006, pp. 246–260. doi:10.1007/11690634_17.

[74] B. Aman, G. Ciobanu, R. Glück, R. Kaarsgaard, J. Kari, M. Kutrib, I. Lanese, C. A. Mezzina, L. Mikulski, R. Nagarajan, I. C. C. Phillips, G. M. Pinna, L. Prigioniero, I. Ulidowski, G. Vidal, Foundations of reversible computation, in: I. Ulidowski, I. Lanese, U. P. Schultz, C. Ferreira (Eds.), Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405, Vol. 12070 of Lecture Notes in Computer Science, Springer, 2020, pp. 1–40. doi:10.1007/978-3-030-47361-7_1.

[75] C. Aubert, D. Medić, Explicit identifiers and contexts in reversible concurrent calculus, in: S. Yamashita, T. Yokoyama (Eds.), Reversible Computation - 13th International Conference, RC 2021, Virtual Event, July 7-8, 2021, Proceedings, Vol. 12805 of Lecture Notes in Computer Science, Springer, 2021, pp. 144–162. doi:10.1007/978-3-030-79837-6_9.

[76] I. Lanese, I. Phillips, Forward-reverse observational equivalences in CCSK, in: S. Yamashita, T. Yokoyama (Eds.), Reversible Computation - 13th International Conference, RC 2021, Virtual Event, July 7-8, 2021, Proceedings, Vol. 12805 of Lecture Notes in Computer Science, Springer, 2021, pp. 126–143. doi:10.1007/978-3-030-79837-6_8.

[77] Y. Fu, Z. Yang, Tau laws for pi calculus, Theoretical Computer Science 308 (1-3) (2003) 55–130. doi:10.1016/S0304-3975(03)00202-0.

[78] J. Parrow, D. Sangiorgi, Algebraic theories for name-passing calculi, in: J. W. de Bakker, W. P. de Roever, G. Rozenberg (Eds.), A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium, Noordwijkerhout, The Netherlands, June 1-4, 1993, Proceedings, Vol. 803 of Lecture Notes in Computer Science, Springer, 1993, pp. 509–529. doi:10.1007/3-540-58043-3_27.

[79] R. Horne, K. Y. Ahn, S. Lin, A. Tiu, Quasi-open bisimilarity with mismatch is intuitionistic, in: A. Dawar, E. Grädel (Eds.), LICS, Association for Computing Machinery, 2018, pp. 26–35. doi:10.1145/3209108.3209125.

[80] D. Sangiorgi, D. Walker, On barbed equivalences in pi-calculus, in: K. G. Larsen, M. Nielsen (Eds.), CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings, Vol. 2154 of Lecture Notes in Computer Science, Springer, 2001, pp. 292–304. doi:10.1007/3-540-44685-0_20.

[81] K. Babel, V. Cheval, S. Kremer, On the semantics of communications when verifying equivalence properties, Journal of Computer Security 28 (2020) 71–127, 1. doi:10.3233/JCS-191366.

[82] D. Baelde, Contributions à la vérification des protocoles cryptographiques, Habilitation à diriger des recherches, Université Paris-Saclay (Feb. 2021). URL http://www.lsv.fr/~baelde/hdr/habilitation_baelde.pdf

[83] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, J. 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, IEEE, 2019, pp. 256–271. doi:10.1109/CSF.2019.00025.

[84] D. Sangiorgi, Introduction to Bisimulation and Coinduction, Cambridge University Press, 2011. doi:10.1017/CBO9780511777110.

[85] R. Milner, A modal characterisation of observable machine-behaviour, in: E. Astesiano, C. Böhm (Eds.), CAAP '81, Trees in Algebra and Programming, 6th Colloquium, Genoa, Italy, March 5-7, 1981, Proceedings, Vol. 112 of Lecture Notes in Computer Science, Springer, 1981, pp. 25–34. doi:10.1007/3-540-10828-9_52.

[86] K. G. Larsen, A. Skou, Bisimulation through probabilistic testing, Information and Computation 94 (1) (1991) 1–28. doi:10.1016/0890-5401(91)90030-6.

[87] B. Accattoli, A fresh look at the lambda-calculus (invited talk), in: H. Geuvers (Ed.), CSL, Vol. 131 of Leibniz International Proceedings in Informatics, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019, pp. 1:1–1:20. doi:10.4230/LIPIcs.FSCD.2019.1.

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

[89] B. C. Pierce, D. Sangiorgi, Typing and subtyping for mobile processes, Mathematical Structures in Computer Science 6 (5) (1996) 409–453. doi:10.1017/S096012950007002X.

[90] C. Stirling, Modal and temporal logics for processes, in: F. Moller, G. M. Birtwistle (Eds.), Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings), Vol. 1043 of Lecture Notes in Computer Science, Springer, 1995, pp. 149–237. doi:10.1007/3-540-60915-6_5.

[91] U. Montanari, V. Sassone, Dynamic congruence vs. progressing bisimulation for CCS, Fundamenta Informaticae 16 (1) (1992) 171–199. doi:10.3233/FI-1992-16206.

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