PROCESS ALGEBRAS INSIDE LUDICS: AN INTERPRETATION OF THE CALCULUS OF COMMUNICATING SYSTEMS

Stefano Del Vecchio

Contents

1. INTRODUCTION

This thesis is the result of a research inside the fields of theoretical computer science and logic. It studies the connection between *process calculi*, that model parallel and concurrent computation, and *ludics*, that is a reformulation of logic generated from multiplicative-additive linear logic and based on the concept of *interaction*. The relation between these two systems falls inside the broader topic of the Curry-Howard correspondence, that originally ties together *intuitionistic logic* and λ-calculus. We will mostly work with the paradigm of process algebras, Milner's *Calculus of Communicating Systems*, and the original formulation of ludics by Girard's, to find a Curry-Howard-like correspondence between them. Our focus will be, rather than on the objects of the two systems, on their *dynamic*, i.e. the way these systems are *used* and thus the way terms are transformed and obtain their meaning. The core notion of *interaction* found in ludics will fit our approach particularly well.

A correspondence between proofs and programs

The *Curry-Howard correspondence* strongly ties *formal proofs* and *programs*, forming an extremely prolific connection between proof theory and theory of computation, thus between (intuitionistic) logic and programming languages. This remarkable realization has made possible great advances in both fields, and given the possibility to apply the tools of logic on programs, giving birth, for instance, to *type systems* and *denotational semantics*.

Proof theory is the study of proofs as formal objects, and is mainly based on two formal syntaxes for proofs: *natural deduction*, and *sequent calculus*, introduced by Gentzen in 1935 [31, 32]. The main theorem of Gentzen's works, the **cut-elimination theorem**, is at the core of proof-theory, defining the *dynamic* of proofs: any sequent calculus proof with a *cut*-rule, that is the formalization of classical deductive reasoning as the *syllogism*, can be re-written as a proof of the very same statement, but *without cuts*. Theory

of computation arises in the same years with the birth of various *models of computation*, as *Turing's machines*, by Turing, and the λ-*calculus*, by Church, that can be considered the first programming language, and is still widely used as a theoretical tool with direct applications to functional programming.

A correspondence between proofs and programs, in the sense of λ-calculus terms or Turing's machines, was at first proposed by Curry, and then later refined by Howard, forming a clear connection between simply-typed λ-calculus and intuitionistic natural deduction. The correspondence interests multiple levels of depth:

Intuitionistic Logic Formulas correspond to **Types**.

Atomic formulas to *Base types*.

Logical connectives to *Type constructors*.

Proofs correspond to **λ-terms**.

the $\Rightarrow$ *introduction rule* to λ-*abstraction*.

the $\Rightarrow$ *elimination rule* to *term-application*.

Cut-elimination corresponds to **β-reduction**.

cuts to β-*redexes*.

The most interesting level of the correspondence, and the one which is the focus of this thesis, is the last one, that states a correspondence in the *dynamic* of proofs and programs: it tells us that the procedure of **cut-elimination** is actually a form of **computation**, and vice-versa.

Our position inside the setting of the Curry-Howard correspondence is within the attempts to extended this correspondence to models of different kind of computations (as *parallel and concurrent computation*), by using different logical systems (as linear logic or *ludics*).

Linear logic and Ludics

Resource consumption and duplication of connectives

Linear logic, introduced by Girard's [33], is a *resource-sensitive* logic. This attention to resources becomes clear once put in contrast with intuitionistic

logic: the intuitionistic implication $A \Rightarrow B$ in linear logic is refined into a particular case of the *linear implication*:

$$!A \multimap B,$$

where ! is a special modality that explicitly allows *multiple uses* of the hypothesis A, that otherwise would be consumed for each "creation" of B; that is because $\multimap$ is a *linear* function that consumes one occurrence of A (the input) to obtain one occurrence of B (the output).

The consequence on the syntax is a restriction of the *structural rules* (weakening and contraction rules) to the *exponential* connectives ! and ?, that are the only ones allowing *replication* or *absorption* of formulas, and the *duplication* of the standard connectives:

- $\wedge$ into $\&$ and $\otimes$;
- $\vee$ into $\parr$ and $\oplus$,

depending on their nature as respectively *additive* ($\&$ and $\parr$), or *multiplicative* ($\otimes$ and $\oplus$) connectives. However, with this duplication, another distinction arises between the connectives, that can be grouped depending on their *polarity* – **positive** or **negative**– a property linked with the *reversibility* of rules in a proof, i.e. the fact that we can prove the premises of a rule if we proved its conclusion, and vice-versa. We have that

- *reversible connectives* ($\&$ and $\parr$) are said **negatives**;
- *irreversible connectives* ($\otimes$ and $\oplus$) are said **positives**.

This distinction of connectives leads to the *focalization theorem*, by Andreoli [5], that uses the reversibility of rules to *optimize proof-search*, by considering only *focused* proofs: i.e. linear logic *cut-free* proofs in sequent calculus with a strict alternation of *positive* and *negative* clusters of rules, where the *negative*, or reversible, ones are always decomposed first. Since any proof admits a focused form, the theorem holds for all linear logic proofs. These clusters of rules of a single polarity are called *synthetic connectives*, and focused proofs are a fundamental step in the construction of ludics.

The abstraction of focused proofs: designs.

Ludics, introduced by J.-Y. Girard in [38] is a *purely interactive* approach to logic, that has the objective of giving a new framework for logic, built from scratch with the concept of **interaction** at its core. It comes from the abstraction of focused linear logic proofs, but can be defined in many

forms, especially inside *game semantics*, as a particular form of *strategies* (as *Hyland-Ong innocent strategies* [44]). The notion of *interaction* is primitive in the theory, in the sense that the objects of ludics, called **designs**, overcome the difference between *syntax* and *semantics*: the structure of a design *makes explicit and defines* its possible *interactions* with the other designs, and, vice-versa, a design *is defined by all its possible interactions*, as expressed by the *separation theorem*, stating that two designs that interact in the same way with the other are the very same object.

Designs can be seen as a *syntactical support* for interaction, born from an abstraction of *focused Multiplicative-Additive Linear Logic* ($MALL$) proofs in sequent calculus, where the logical content, i.e. the *formulas* or *types*, is forgotten in favor of *addresses* (sequences of natural numbers) which point at particular *places* in the proof. What becomes important is the *locative structure* of a proof, not the specific *type* of it, i.e. the formula of its *conclusion*. Leaving behind the syntax-semantics distinction, *designs* are thus purely *interactive* proof-like tree structures, whose syntactical structure *is* their semantics, defining the logical content of a design instead of its conclusion.

Anticipating the background section, here is an example of how this works. We start from a simple $MALL$ focused proof

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash A_1, \Delta_1} \qquad \dfrac{\vdots}{\vdash A_2, \Delta_2}}{\vdash A_1 \otimes A_2, \Delta} \qquad \vdash B, C, \Delta}{\vdash ((A_1 \otimes A_2) \,\&\, B) \parr C, \Delta}$$

Being focused, A_1, A_2 are *negative formulas*; B, C are *positive formulas*, and the *negative* connectives $\&$ and $\parr$ are introduced at the same time, forming a *cluster* of two rules of the same polarity. Going further, and considering only positive formulas by *De Morgan duality*, it becomes

$$\dfrac{\dfrac{\dfrac{\vdots}{A_1^\perp \vdash \Delta_1} \qquad \dfrac{\vdots}{A_2^\perp \vdash \Delta_2}}{\vdash A_1 \otimes A_2, \Delta} \qquad \vdash B, C, \Delta}{((A_1 \otimes A_2)^\perp \oplus B^\perp) \otimes C^\perp \vdash \Delta}$$

and in ludics, forgetting about the formulas and keeping only the information on their respective locations as *addresses* (sequences of natural numbers denoted $\xi 1, \xi 2, \ldots$), we have

$$\dfrac{\dfrac{\dfrac{\vdots}{\xi 11 \vdash \Delta_1} \qquad \dfrac{\vdots}{\xi 12 \vdash \Delta_2}}{\vdash \xi 1, \Delta} \qquad \vdash \xi 2, \xi 3, \Delta}{\xi \vdash \Delta}$$

where the conclusion ξ is any arbitrary address: indeed, the *type* does not matter anymore! The *sub-formulas* become *sub-addresses*, tied to the address generating them by *suffixes*; therefore $\xi 1$, $\xi 2$, $\xi 3$ are generated by ξ, and $\xi 11$, $\xi 12$ by $\xi 1$, and so on. At each step the polarity of an address is inverted; in our example the conclusion, being at the *left* of the deduction symbol $\vdash$ is *negative* – indeed, it comes from the abstraction of a *negative* formula – making the design itself negative; its immediate sub-addresses are, instead, *positive*.

Since what matters is the *structure* of the proof, and we can keep track of it through the sub-design relation, another representation for designs is a sequence of rules, or *actions*, each corresponding to a logical rule with the essential information to re-create it. For instance, the design above can be written as:

$$(-, \xi, \{\{1\}, \{2, 3\}\})(+, \xi 1, \{1, 2\})$$

a simple alternate pair of a negative and a positive action. More specifically, we have

- $+, -$ denote the *polarity* of a rule;
- ξ and $\xi 1$ are the *focus* of the rules, that is the address introduced corresponding the the *active formula* of the rule;
- $\{\{1\}, \{2, 3\}\}$ and $\{1, 2\}$ are the *ramification* of their respective rules, i.e. the sub-addresses generated by the rule, that stand for the sub-formulas of the conclusion, that become now the active ones.

The different forms of the ramification, for the negative rule a *set of sets* of natural numbers, and a simple *set* for the positive, is due to the different polarity of the rule, that allows multiple sub-formulas in its premises in the first case, and only one sub-formula for each premise in the latter.

Interaction, the dynamic of ludics

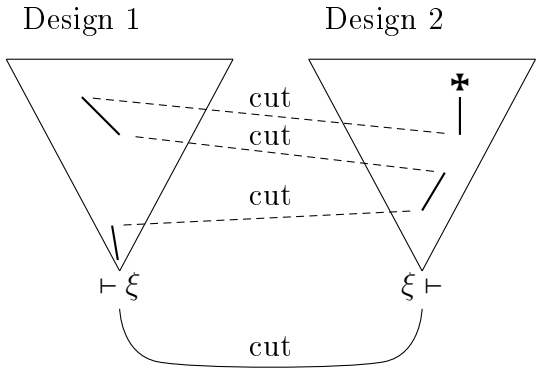
Ludics, considering the structure of designs and how they interact, is actually closer to game semantics than to proof theory. While interaction is strongly tied to cut-elimination, it does not require formulas, as the correctness of

its objects that is not tied to the specific statement found in the conclusion. Designs can be formulated as Hyland-Ong innocent linear strategies [44], where the notion of *play* expresses the dynamic of the theory, and stands for interaction. Even the correspective of types in ludics, called *behaviours*, are defined with respect to interaction, contrary of *games* in game semantics: a behaviour is a set of designs closed with respect to their *orthogonals*, which are the counter-designs of the set, the ones with which interaction ends successfully, and does not diverge. Ludics is also a realization and an expression of the *Geometry of Interaction* [35], a program started by Girard with the objective to develop logic and proofs from a *geometric* viewpoint, and the possibility to give a central role to the *dynamic* of formal objects, instead of their content; as *proof nets* already do for linear logic. This presentation of logic proposes a change in the standard paradigms, and gives the possibility to strengthen the connection between logic, game semantics, and theory of computability, as many works, in particular by C. Faggian (with M. Basaldella [6] and M. Piccolo [27]) and K. Terui [66], already have developed.

Interaction, as cut-elimination, starts from *cuts*. However, these *cuts*, in standard ludics, are not *internal* to a design, as is the cut-rule in sequent calculus proofs, but can only occur between two different designs. A cut is formed when two designs share a common address, but in *opposite position*, thus of opposite *polarity*, as for instance $\vdash \xi$ and $\xi \vdash$. The *closed case* of interaction corresponds to the elimination of a cut between two designs, as if it were a *cut-rule* linking the two proofs; but, this kind of cut-elimination is actually extremely counter-intutive. Indeed, we are trying to eliminate a cut between two proofs with an *opposite* conclusion, that in linear logic would correspond to considering *at the same time* a proof of A and $A^{\perp}$, which is obviously impossible, since one of the two proofs must be incorrect. However, in ludics there are no formulas, as there are no *axioms* or *atomic-formulas*: this entails that the meaning of "correct proofs" cannot be tied nor to the specific conclusion of the proof, nor to the choice of its premises. Again, what matters in ludics is the *locative structure* of the proof, and the strict alternation of rules of opposite polarity; in this sense it is more a *strategy* than a *proof*, and therefore closer to game semantics than sequent calculus. To overcome the difficulties that may arise having "incorrect" proofs in a Logical system, in place of the axioms we find a special rule, called **daimon**, and denoted ✠: the daimon let us *stop proof-search* at any time, by allowing us to prove anything at will. It can interpreted also as a symbol standing for an *error*, or, in game-semantical terms, a move meaning that the player performing it is *giving up*, and ending the game.

To give an intuition, interaction between two designs forms a *path* between

them, that *visits* pairs of dual actions by switching design at each step, and finding new pairs by passing through the premises of the action part of a *cut* that has been visited; it *stops if it reaches* ✠, meaning that interaction has ended *successfully* and the designs involved are *orthogonals*; otherwise interaction diverges. An **interaction path** between two *orthogonals designs* looks like this:



Closed interaction let us define the types of ludics: *behaviours*. Behaviours are *sets of designs closed by bi-orthogonality*, i.e. a set $\mathbb{B}$ of designs such that $\mathbb{B} = \mathbb{B}^{\perp\perp}$. They are one of the most interesting and studied aspects of ludics, and have multiple tools and properties at their disposal, developed in various works (as for instance [28, 29, 30]) including the original article by Girard, were linear logic is interpreted into ludics by interpreting formulas into behaviours, defining logical operations on them corresponding to the connectives $\otimes, \oplus, \parr, \&$, and proving the *full and internal completeness* theorems for polarized *MALL*.

Process calculi

The other field we are going to study and hopefully advance, by forming a connection with ludics, is the one of **process calculi** (or *algebras*). Process calculi are a model of *reactive* systems, i.e. systems where objects, called *processes*, react to stimuli from an external *environment*, which can determine their transformations; in particular, they model *parallel and concurrent computation*, and consequently *concurrent programming*. The name "process algebra" was first used by D. Milner, who introduced it due to the realization that processes have an algebraic structure: processes can be combined with operations to form new processes – for instance by sequential or parallel composition – and the resulting term will behave differently, depending upon the generating processes, and the kind of operation used. In this sense the

syntax of processes is algebraic, since it is formed by basic elements and operations to build new and more complex terms. For instance we can combine the processes $a.b.1$ and $\overline{a}.c.1$ obtaining:

- $a.b.1 \mid \overline{a}.c.1$ by *parallel composition*, or
- $a.b.1 + \overline{a}.c.1$ by *non-deterministic choice*.

were $a, b, 1$ are *channel names*, notion tied to the key concept of process algebras, that is the one of **communication**.

Indeed, the heart of these systems is to specify how parallel processes interact with each other, interaction that takes the form of *communication between processes* running at the same time. Communication can happen in various way, the most common being between *channels*. It often takes the form of an exchange of information: two dual channels can *synchronize* during computation, called **execution**, with one sending a data, and the other receiving it, to then continue depending on the specifications of each process. To give an intuition, in the process $\overline{a}.b.S \mid a.\overline{b}.R$ the pairs of dual channels $(\overline{a}, a)$ and $(b, \overline{b})$ (were $\overline{a}$ is the dual of a, the same for b) can synchronize and communicate; an execution sequence performing both synchronizations looks like this:

$$\overline{a}.b.S \mid a.\overline{b}.R \to b.S \mid \overline{b}.R \to S \mid R$$

Senders and receivers are the very same kind of objects, and processes consist of channels with one or the other role, combined using the aforementioned operations. Processes can therefore communicate via data-passing, interpreted as synchronized communication between agents, giving shape to their dynamic. These ideas are at the core of the paradigm of process algebras, which will be the focus of our work alongside ludics: Milner's *Calculus of Communicating Systems* (*CCS*) [55].

The Calculus of Communicating Systems

Milner's original process calculus has seen many extensions during the years, in particular the widely used π-calculus (introduced in [57, 58] by R. Milner, J. Parrow and D. Walker) that extends standard *CCS* with *name-passing*, allowing channels to send/receive other channel-names as data during synchronization/communication. This feature gives the calculus a much greater expressive power, making it closer to actual concurrent programming, and an ideal model to work with, especially with the solution of practical problems in mind, as for instance with *session types* [47, 64, 50], dealing with security of communication protocols. However the nature of the problem we are going to study requires a purely theoretical approach, letting us disregard practical

implementation problems; therefore, we will mostly stick to standard CCS, that being the paradigmatic process algebras has the advantage of retaining only the essential aspects of process calculi, thus giving us a simple and elegant setting to work with, that will make a correspondence with ludics actually *more general* in comparison with the more complex π-calculus.

The grammar of CCS is actually extremely simple, and consists of

- an empty process, usually denoted 0, from which all processes are built, and that cannot communicate by itself with any other process;
- a denumerable set of channel names, denoted $a, b, c, \ldots$, who have *duals* $\overline{a}, \overline{b}, \overline{c}, \ldots$, that are the means by which processes communicate and are built;
- the *action-prefix* operation, that let us build processes by putting a new channel name as a *prefix* of an already existing process, as $a.\overline{b}.0$;
- the *parallel composition* operation, denoted $P \mid Q$, that is what *enables* communication between processes, and thus makes execution a kind of *parallel computation*;
- the *non deterministic choice*, or sum, denoted $P+Q$, that is an *exclusive choice*, forcing us to decide with which process member of the sum we want to communicate, discarding the other;
- the *private names* or *hidden names* operator νa, that makes all channels named a inside its scope *hidden* from the external environment or context;
- and the *recursive operator* $A(\vec{a})$, on the parameters $\vec{a} \in \mathcal{A}$, that allows recursive definitions of processes; sometimes appearing as $!a.P$, an exponential or *replicable* channel.

Communication between channels happens during execution, the *reduction semantics* for CCS, that makes two *dual channel names*, belonging to processes in *parallel composition*, synchronize – intuitively meaning that they are *exchanging information* – to then be *consumed* and let the process continue depending on its specifications. As we showed in the example above, if a process $P = \overline{a}.b.S \mid a.\overline{b}.R$, then the pair of dual channels $(\overline{a}, a)$ can synchronize, generating the following reduction of the process

$$\overline{a}.b.S \mid a.\overline{b}.R \to b.S \mid \overline{b}.R$$

The reduced form $b.S \mid \overline{b}.R$ now can synchronize the pair $(\overline{b}, b)$, that have no prefixes blocking communication with them, and thus perform the next step

$$b.S \mid \overline{b}.R \to S \mid R$$

In the case of a non deterministic choice, execution will only keep the processes on which we are synchronizing the two channels:

$$a.P + Q \mid \overline{a}.P' + Q' \to P \mid P'$$

Non-determinism and non-confluence in execution

The most important fact that we have to note is that such a simple grammar and reduction rule, without particular restrictions, allows execution on CCS processes to be *non-deterministic* and *non-confluent*. Non-determinism comes directly from the *non-deterministic choice*, that, as the name suggests, forces us to choose between two processes during execution. Instead, *non-confluence* comes from the simple fact that channel names are *not unique*, and therefore we can find processes as

$$P = \overline{a}.b.S \mid a.c.Q \mid a.\overline{b}^{i}.R$$

where there are multiple synchronizations available at the same time, due to the two pairs $(\overline{a}, a)$.

In such cases there are *forks* in execution paths that bring the process to different reduced, and ultimately *normal*, forms. In our example, we have two possibilities:

- $P \to b.S \mid c.Q \mid a.\overline{b}.R$, or
- $P \to b.S \mid a.c.Q \mid \overline{b}.R$

and only in the second case we are allowed to continue execution on the pair $(b, \overline{b})$, obtaining

$$b.S \mid a.c.Q \mid \overline{b}.R \to S \mid a.c.Q \mid R$$

while in the first case we have reached a *normal form*.

These are the most relevant properties of execution regarding the objectives of our research, and what makes a Curry-Howard correspondence with a logical system an extremely delicate matter. Indeed, more often than not, when trying to find a logical model or interpretation of a process calculus into a logical system, *restrictions* are imposed to the syntax of the calculus, in order to *determinize* execution, also making it *confluent*; that is a necessary

step to make execution compatible with *cut-elimination*, whose confluent nature cannot be changed without depriving proofs of their very meaning.

Execution will be connected to interaction of designs in ludics, through an interpretation of the calculus into the logical system: CCS terms will be interpreted into *sets* of designs, and *execution sequences* will correspond to *interaction paths* on *behaviours*, with the scope in mind to gain insights of both theories, and give another logical foundation to process algebras.

A Curry-Howard correspondence between process algebras and ludics

The standard way to build a correspondence between a logical system and a model of computation (in general a *calculus*, as CCS or the λ-calculus) is through a syntactical **translation** of one system into the other. Each basic element of the grammar of one system is translated into a basic element of the other, and then *term constructors* of one system – as the introduction rules for connectives in logic or as *parallel composition*, or *non deterministic choice*, in process algebras – are interpreted into term constructors of the other, thus forming a correspondence. This often takes the form of a typing of the calculus: each elementary term of the grammar is *typed* by an elementary term of the logical system (often a formula), then for each term-construction or term-reduction operation, rules on the types are given, to form the *type of the constructed or reduced term.*

In the case of a *Curry-Howard* correspondence the aim of the translation is also to obtain a correspondence in *meaning*, as a correspondence in *how terms are used and are reduced.* In the classical correspondence between λ-calculus and intuitionistic logic, *cut-elimination* on proofs corresponds to β-reduction of λ-terms. In this way, there is a match in *how terms are used* in the two system, forming a strong parallelism and giving the possibility to shift from one system to the other, and thus transfer properties of one system into the other. Taking into account process algebras, however, there are some intrinsic problems that hinder a clean correspondence with a logical system, and give *strong limitations* when trying to *match their meaning* in the above sense.

A correspondence in dynamic, rather than language

While we are trying to obtain a Curry-Howard correspondence between process algebras, specifically CCS, and a logical system – ludics in our case –

our focus is on the latter part of the correspondence, that is also the *core* of a Curry-Howard correspondence: an interpretation achieving a correspondence in meaning, in the sense of *how the objects of the two systems are used.* In more general terms, this means a correspondence in the dynamic of the two systems; only such a correspondence can give us the ability to transfer not only properties, but also meaning between them, and give process calculi another logical foundation.

To this end, our interpretation does not take the usual form of a translation, and presents a clear shift in complexity between terms of CCS – processes – and objects of ludics – designs. Moreover, the actual translation will only be a small part of the interpretation, that is going to be far more complex than a simple correspondence of syntax and term-construction rules. Indeed, CCS *processes* will not correspond to single designs, but to *sets* of designs; meaning that to a single term of CCS are associated multiple objects in ludics, but with a *well defined common structure.* As we will see, this is a necessary step to achieve the sought correspondence in dynamic, free of limitations or restriction in either the syntax or execution.

Previous works with linear logic

The topic of the extension of the Curry-Howard correspondence outside the intuitionistic and functional world is not a new trend in research, and also our specific case regarding process algebras have already been studied in various works. The goal of the research in this direction is to find a proper Curry-Howard counterpart for models of concurrent and parallel computation, in particular CCS and the π-calculus; while the π-calculus has more expressive power, as we mentioned, for its essential and paradigmatic nature, CCS has enough reasons to be considered as well.

Linear logic has been the favorite target for such a correspondence, being considered a particularly refined interactive system, giving a solid ground to form connections with process algebras, whose main feature is the interaction between a process and the environment. Indeed, what matters in process algebras is not much the *result of computation*, but what happens to get there, all the intermediate steps, and thus which channels synchronize, and what data is passed or received during communication. A good example is the work of Kobayashi, Saito and Sumii on session types [50], where there is a strong focus on *ensuring communication on specific channels*, and thus that certain data will actually be passed and received from a process to another; in this case, the final result of the reduction process does not matter at all, if it is unable to pass through some specific steps first. To this end, of the main tools at the disposal of linear logic to properly represent an interactive

system, *proof nets* are the most effective ones.

Proof nets [40] are a purely *geometrical* syntax for proofs of *Multiplicative Exponential Linear Logic* ($MELL$), with some possible, though not definitive, extensions to *additive connectives*, that are surprisingly hard to deal with in a geometrical representation. The aim of proof nets is to give a more general syntax for linear logic proofs, able to overcome the irrelevant differences found in sequent calculus: provable formulas admit multiple equivalent proofs, where the only difference lies in the *order* in which rules are applied. With proof nets we can abstract from sequentiality and obtain a single form for all equivalent proofs in sequent calculus, with a simple, *local*, cut-elimination procedure.

Many approaches have related models of concurrency to linear logic focusing on the interpretation of processes as proof nets, as presented first by Abramsky [1] and Bellin with Scott [13], then refined by Beffara [7]. The systems born from these works are based on the correspondence between *cut elimination* and the equivalent of reduction/normalization on the side of process algebras, i.e. execution; indeed proof nets have similar dynamics to process calculi, being cut elimination a local (and asynchronous) procedure, which suggests a close relation with parallel computation.

The intrinsic difficulties of cut-elimination

A few later works in this direction, by E. Beffara and V. Mogbil [12] for CCS, then refined in [9], and by T. Ehrhard and O. Laurent [22] for the π-calculus, stress the difficulties with the usual attempts with linear logic and proof nets: the main issue involved in finding a typing for such calculi is a proper representation of their natural *non confluence* lost in the correspondence because of the confluent nature of normalization of proofs, which effectively restrains a process to functional behaviour. In order to form a correspondence between execution and cut-elimination, the non-confluent traits of the first must be repressed in some way, often by limiting the syntax of processes, or by determinizing execution.

Differential proof nets. In the work by Ehrhard and Laurent *differential proof nets* are used as the target into which translate a version of the π-calculus, continuing the ideas and results of Honda and Laurent found in [42], where the π-calculus is translated into *polarized linear logic* – linear logic under focusing discipline. Differential linear logic is non deterministic, as it admits *sums of proofs*, and the core idea of their translation is to interpret the *parallel composition* as a cut between a contraction and a cocontraction (a novelty of differential linear logic) link, which allows to have

several premises connected (then contracted), as outputs or inputs, depending on their polarity, respectively negative and positive.

As a downside, the translation accepts only *replicable receiving channels*, and *it is not modular*, i.e. the parallel composition of processes, which allows interaction between them, cannot be represented by a combination of their proof nets interpretation; moreover, since a process corresponds to a differential proof net, while it admits a cut-elimination path for each non deterministic choice via the sum of proofs, the work does not consider the *non-confluence* of execution, that cannot be represented by the still confluent cut-elimination. Nevertheless, though the π-calculus already have various translations into several kind of nets (as in [11] by Beffara and Maurel), differential nets provide a solid logical ground to the calculus, and are equipped with a denotational semantics, also model of the λ-calculus.

Scheduling of execution. Another interesting approach, and starting point of this thesis, are the ones of Mogbil and Beffara [12] and of Beffara alone [9]. Limiting the context to the *multiplicative fragment* of *CCS* (*MCCS*), that is without non-deterministic choice and *recursive definitions*, the idea of their work consists in a *shift in the correspondence* from "proofs as processes" to *"proofs as executions"*. The reason of this shift is *a difference in meaning* between proofs and processes; where the meaning of proofs is its normal form reached by cut elimination, that does not affect the *conclusion* of the proof, while the meaning of a process is not one of its multiple irreducible forms, generated by non-confluent execution sequences, but the communications that happen, in the form of synchronizations and exchange of information between channels, at each step of the execution. Indeed, every step counts, since depending on the context at each intermediate reduced form of a process a permanent fork in the sequence might be generated. Therefore the *dynamic* of a process calculus, intended as how processes are used, interacts with each other, and transform, lies in this complex kind of behaviour, that a confluent procedure as cut elimination, even if possibly non deterministic, cannot completely represent. Cut elimination does not affect the formula in the conclusion of a proof (its *type*), and no matter the path taken – that is which cut is eliminated first – the final result is the same; instead, regarding processes, each execution step *should not preserve the meaning of the original process*, and thus its type, since at any single execution step the possible normal forms reachable by the process might change, by permanently excluding some of the ones of the previous form.

Being cut elimination confluent even in proof nets, the approach of [12] attempts to find a solution via the *scheduling* of execution, i.e. following an execution *plan*. To each process is associated a proof net, and by choosing in

advance which execution will take place in the process, a deterministic path is chosen, therefore in the normalization of its proof-net counterpart nothing is lost: the result is that for every *deadlock free* execution on a process P a corresponding proof net can be build whose cut elimination corresponds to the chosen execution. The downside of this approach is that a process *does not* have a determined type, which depends mostly on the particular execution, and composition with the environment. In [9] some improvements are made, where the typing of a process is fixed by some typing rules in such a way that to each $MCCS$ process P is associated a MLL proof net $[P]$, by a translation of the $MCCS$ syntax and construction rules. Then, the *scheduling* of the execution is clearly separated from the type of the process, since at each execution, say $P \to P'$, is associated the *linear implication* $[P] \multimap [P']$, which, by construction, is still a correct proof net.

However the same drawbacks apply to the refined results, since execution is made confluent by determining in advance which step is going to be performed, and then translated into linear logic. While each process does correspond to a single proof net, the core result of the translation, that is the correspondence between cut-elimination and execution, is fragmented into *multiple types*: to each execution sequence corresponds a different proof-net, that to be built require to *know in advance the form of the process after the execution sequence*.

Relations between process calculi and ludics

On of the goals of this work is to try to obtain results similar to the ones found in the above mentioned works, while overcoming the limitations of linear logic and proof nets, in particular with respect to non-confluence, by using a logical system with interaction at its core as ludics. Indeed, while non-determinism can be modeled in many ways (by sums in differential linear logic for instance), the non-confluence of processes cannot be properly represented by normalization on proofs for its very (confluent) nature: this gap in meaning cannot be closed inside the framework of linear logic.

Recent works relating ludics with process calculi are in particular by C. Faggian: first we have *ludics net*, along with F. Maurel [17], then, with M. Piccolo, *ludics as a model for the finitary and linear π-calculus* [26] and an interpretation of linear strategies in a particular class of *event structures* [27], a model of concurrent calculi by Winskel (introduced in [69]). These studies underline a strong connection between process algebras and ludics, presented as game semantics. What standard ludics is lacking is *replication* and thus possibly *non determinism* in interaction (each address, being an unique place in a proof, is itself unique), and one thing that C. Faggian and M. Piccolo

aim at gaining from this connection is a way to represent replication, and non determinism, inside ludics.

Event structures also have a close relation with our thesis, as Winskel showed they are a model for CCS-like calculi in [68]; moreover our interpretation actually represent in ludics the relations giving the structure to the set of *events* that defines Winskel's model.

LUDICS AS A π-CALCULUS MODEL. In [26] ludics is used to model the π-calculus, where processes are interpreted as linear strategies, but the calculus is restricted to its finitary and *linear* fragment, along with other constraints (about internal mobility on names, asynchrony, and locality of input-outputs), which heavily restrict its non-determinism; moreover the ludics model *does not* represent execution, i.e. *interaction* in ludics does not correspond to *execution* in the calculus, since it holds a confluent subject reduction property: from the ludics interpretation $[\![P]\!]$ of a process P, it holds that if $P \to Q$ (there is an execution from P to Q), $[\![P]\!] = [\![Q]\!]$. As we already stressed, if we are faithful to the meaning of processes, an execution step should not preserve the interpretation of a process.

LUDICS OUTSIDE LINEARITY. In *Ludics with repetitions* [6] is built a version of ludics formulated in game semantics-terms allowing repetitions of addresses, where *silent actions* are added to make interaction non-deterministic but consistent with the *internal* and *full* completeness theorems of ludics (found in [60, 41, 38]). The repetition of addresses makes ludics non-linear and requires *duplication of designs* during interaction. The drawback of letting addresses and rules on them be repeated is that the *separation theorem* does not hold anymore.

In *Event structures and linear strategies* [27], focused on representing linear strategies as *confusion free* event structures, non determinism is added to ludics, since these latter have controlled non-deterministic traits. Based on the πI-calculus typing by Varacca and Yoshida presented in [67], it starts from the intuition, found in [43], that (typed) processes can be seen as Hyland-Ong innocent strategies, and from other works describing game semantical objects as event structures ([45], [2]): non determinism is represented as non-deterministic sums, and composition of strategies is represented as parallel composition of event structures, thus allowing parallelism and modularity.

Still, the property of *confusion freedom* is a not an irrelevant restriction with respect to process behaviour (event structure can be seen as processes, being a model of CCS and the πI-calculus [63]), since it is a generalization of *confluence* to non-deterministic systems. It restricts processes to those

where each non deterministic choice is *local*, i.e. internal to one process, and independent by the environment; what we want to achieve, instead, is an interpretation of *unrestricted* process behaviour into ludics.

COMPUTATIONAL LUDICS. Another work in this direction is Terui's *computational ludics* [66]. It consists in a reformulation of ludics in terms more suited to deal with and describe *computational power* and complexity of functions. It is presented in a π-calculus-like syntax, with both a β-reduction and Krivine's abstract machine-style normalization. In this setting ludics is naturally non-linear, and admits *cuts* inside designs as a construction rule contrary to standard designs, that being focused *cut-free* $MALL$ proofs, can generate a cut only in combination with an orthogonal design. Computational ludics let us define by recursion infinite design using *finite design generators* and perform interaction on them as a finite procedure. Computational designs are then related to various complexity classes, depending on the features we consider (for example cut-free or with internal cuts).

Problems of exponentials in ludics

On our side, we start our work with the *standard* syntax of ludics. This syntax is linear, does not allow replication of addresses and does not use exponentials, therefore it is a more solid ground to build our interpretation. Both computational ludics and ludics with repetitions have drawbacks that do not allow us to achieve the complete correspondence in dynamics that we are seeking; we will see later why the syntax of *computational ludics* hinders our interpretation of CCS, and why the modified interaction of *ludics with repetitions* does not fit our needs, not counting the fact that in the latter the *separation theorem*[1] *does not hold anymore.*

In linear logic to recover the expressive power of the lost structural rules – i.e. *weakening* and *contraction* – two unary connectives dual of each other, called *exponentials*, are added: ! (*bang*, or *of course*) and ? (*why not?*). When used before a formula they allow the use of structural rules on that formula; for $!A$ when in the premises (left of $\vdash$), for $?A$ when in the conclusion part (right of $\vdash$). In this way formulas with exponentials are replicable and non-linear resources. Since ludics is based on $MALL$, it does not admit any kind of exponentials; furthermore, it would be against ludics theoretical foundation, that is to only keep the geometrical structure of the proof, by relations between *locations* in the proof itself: if an address is a particular place in a proof, being able to replicate it would *not make any sense*. However, without

[1] We mentioned it above: it states that two designs with the very same interactions are the very same object.

extending ludics to non-linearity, representing processes defined by *recursion* is simply *impossible*, as $MALL$ has a rather weak expressive power by itself. If a design, or a part of a design, corresponds to a recursive part of a process, i.e. a program that calls it self arbitrarily many times, this design must be able to replicate itself or some of its rules, and thus addresses, at least during interaction.

We will stumble in this theoretical contradiction when trying to deal with recursion and replication, and attempt to find different solutions than to just switch to one of the non-linear syntax available for ludics, with the goal in mind to keep the *standard ludics interaction* for its intrinsic linear and resource-sensitive nature.

Main contributions of the thesis

Translating processes into ludics. Our interpretation goes somewhat in the opposite direction of [27], by representing the *causal* and *conflict* relation of *event structures* into ludics, using carefully defined *sets of designs*, through a ludics technique inspired by Christophe Fouqueré and Myriam Quatrini, called *pruning*. Limiting our setting to *multiplicative* CCS at first (as in [9]), and then to *replication-free* CCS, the pruning lets us easily represent inside ludics the structure of replication-free processes (or, equivalently, event structures): the *partial order between channels* generated by the *prefix operation*; the *parallel composition*; and any *exclusive relation* between channels or processes, as the *non deterministic choice*, or the conflict between synchronizable pairs that *share the same channel name occurrence*, as we saw in the example about *non confluence*. Adding an *assignment function* associating elements of a process P (channel names, synchronizations, etc.) to *addresses* of the designs we built, leads us to associate to a replication-free CCS process P a set of design, that then is *closed by bi-orthogonality*, obtaining a *behaviour*, that together with the assignment function becomes the *interpretation in ludics of* P, denoted $[\![P]\!]$.

A correspondence between execution and interaction. The relations giving structure to a process are indirectly coded into designs along with the *channel name occurrences* used to built it; then, these designs are put together in a *set*, in such a way that the full relations will appear and be respected *during interaction with the orthogonal set*. Then this set is closed by bi-orthogonality, obtaining a behaviour: in this way the *full dynamic* of processes can be represented by *interaction* on a type of ludics. To each interaction path in its ludical interpretation $[\![P]\!]$ (between $[\![P]\!]$ and its orthogonal) is associated an execution sequence on a replication-free CCS process P, in

such a way that to each interaction is associated *exactly one actually possible execution* on P, and each execution is associated to *at least one interaction*. This correspondence takes into account each singular step of execution, thus, from an interaction we can recover the exact sequence and order of the synchronized pairs of channels, as well as the channel names themselves, therefore we are able to read-back step-by-step the execution of the process from any of the interactions to which it is associated. The correspondence we build forms a connection between the dynamics of the two systems, and to each process is associated a single interpretation, that is a well structured object, and moreover a *type* of ludics. This interpretation is able to represent through interaction the whole range of *executions* on the process, embracing both the non-determinism and non-confluence of the dynamic of CCS. It is also a connection in *meaning* between interaction and execution: as the meaning of execution are its intermediate steps, the meaning of interaction is the *path* it takes to get to a successful termination; indeed, the result of interaction would be otherwise completely un-informative, as it only tell us "convergence" with ✠ or "divergence".

Partial modularity of the interpretation. By modifying and adapting to our interpretation the operation $\otimes$ on behaviours, that corresponds to the linear logic tensor $\otimes$, we try to give modularity to the interpretation, in such a way that the interpretation $[\![P \mid Q]\!]$ of two processes in *parallel composition* is obtainable by an operation on their respective interpretations, i.e. that

$$[\![P \mid Q]\!] = [\![P]\!] \mathbin{⦶} [\![Q]\!]$$

This is partially achieved, since in the trivial case where there is *no communication* between P and Q, $⦶$ is exactly $\otimes$. However, in a general case where P and Q can synchronize through some channels, the operation is much more complex and artificial, going beyond a simple ludical operation on behaviours. We must act on the *set of designs* generating the behaviours, however it is possible to still keep the basic operation $\odot$ on pairs of designs, called *merging*, that is used to define $\otimes$.

A reduction on the interpretation. We deepen the connection in dynamic between processes and behaviours by defining a *reduction* on $[\![P]\!]$. Note that a standard *subject reduction* property stating that *the interpretation of a process is unaffected by execution* must not hold, if we want to be faithful to the meaning of a process. Therefore we define an operation based on ludics' *projection* on behaviours (already defined by Girard in [38]) that *reduces* $[\![P]\!]$ to $([\![P]\!])_u$, matching execution on a given synchronization

$u = (a, \overline{a})$. We thus obtain the interpretation of the process P *after* the execution step on the chosen synchronization u, in such a way that

$$\begin{array}{ccc} P & \to_u & P' \\ {\scriptstyle [\![-]\!]}\downarrow & & \downarrow{\scriptstyle [\![-]\!]} \\ [\![P]\!] & \rightsquigarrow_u & ([\![P]\!])_u = [\![P']\!] \end{array}$$

By extending this result to execution sequences leading to the *empty process*, that we denote 1 instead of 0, we obtain that if $P \to_* 1$ for an execution sequence $*$, then the *reduction* on $[\![P]\!]$ by $*$ results in the design

$$\frac{}{\vdash \xi}\ (+, \xi, \varnothing)$$

which is the design called $\mathbb{O}ne$, whose generated behaviour *corresponds to the linear logic multiplicative unit* 1, giving us good reasons to denote the empty process 1 instead of 0.

Deadlock freedom. We can give a nice sufficient condition for *deadlock-freedom* of a process by checking a property of ludics called *incarnation* of $[\![P]\!]$, a property of behaviours that tells us which are the *parts of its designs actually visited by interaction* (with the orthogonal behaviour). Designs that are fully visited are *material*, i.e. they are equal to their *incarnation*: if this is the case in $[\![P]\!]$, then we can conclude that P is *deadlock-free*. This property holds for the correspondence between execution and interaction: if the whole behaviour $[\![P]\!]$ is accessible during interaction, then so is the process P during execution.

Attempted extension to recursive definitions. At last we point at some possible directions that could lead to an extension of ludics able to both represent *replication* in CCS and respect our correspondence in dynamic. At first we hint at a possible solution by taking the *controlled replication* in the typing of Sangiorgi's πI-calculus by Varacca and Yoshida [67]: the restrictions there imposed let us have a still expressive replication, but much easier to represent – by using infinite sequences of actions instead of replication of addresses.

Another possible route is to just reformulate the whole interpretation into *computational ludics*, having thus available *finite design generators*, that can naturally represent recursive definition of processes, or programs in general, being introduced by Terui with that goal in mind. However, the syntax of computational ludics does not fit our interpretation at all; many problems arise in the translation of processes into sets of designs, and a clear simple

correspondence between elements of the process and addresses is lost. Therefore, what we attempt to do is to rather reformulate finite design generators into the standard syntax of ludics with the smallest extension possible, by introducing special *exponential addresses* denoted $\xi*, \zeta*, \ldots$, that are *replicated in the context* after a rule is performed on them. However this direction of research is still in a very early stage, and needs further study to be evaluated and bring solid results.

Outline

- In **chapter 2** and **chapter 3** we present the necessary background notions, and present in detail linear logic, ludics, and process algebras, with particular attention to CCS. We also give a formal definition of *event structure*, that we will mention later in the thesis.

- With **chapter 4** we start our original work. It is the central chapter of our interpretation of CCS into ludics: in the first part it shows, inside the limited setting of $MCCS$, how to carry out the translation of processes first into a *graph* resembling event structures, to aid some proofs and intuitions with a geometrical representation of a process, then into sets of designs. In the second part we define a correspondence between execution and interaction, and prove the main theorem of the chapter stating that the interpretation in ludics of a process *characterizes* all its executions.

- In the first part of **chapter 5** we build the machinery necessary to define the operation $\oplus$, that will interpret *parallel composition* and make the translation modular; then we prove a theorem stating that it achieves its objective. The second part is dedicated to defining the *reduction* on the interpretation, and properties about deadlocks and deadlock freedom.

- In **chapter 6** we show how to extend the interpretation to *non deterministic choices* and the *hidden name operator* ν, while preserving the previous results. Then we explore the extensions of ludics to non-linearity, as *computational ludics* and *ludics with repetitions*, and how we could interpret *recursive definitions* of CCS processes to make them compatible with our interpretation and its goals, by suggesting some research directions and future works about *exponentials* in standard ludics.

- In **chapter 7** we show some relations between our interpretation and other works. We start with a short presentation of [9], one of the starting points of this thesis, to see what we have gained with our interpretation; then we try to apply our results also to *session types*, in particular to a work by Kobayashi, Saito, and Sumii [50], despite being in the more complex setting of the π-calculus. At last we point out the obvious relations with *event structures*, and hint at some connections with the work of Faggian and Piccolo about *confusion-free event structures as linear strategies* [27], and how our interpretation could be useful there.

2. Linear Logic and Ludics

We start with a short introduction to linear logic [33], the system which gave rise to ludics, as an abstraction of its multiplicative-additive sequent calculus proofs. We will define the standard sequent calculus for linear logic, the role of the exponentials, as well as the *polarization of connectives*, fundamental for the *focusing* (or *focalization*) of proofs, necessary to link linear logic proofs to *game semantics*, in which ludics can be formulated. We will also spend a few words on the geometrical representation of linear logic given by proof-nets, which are widely used in various works related to ours, mostly about translations of process calculi into linear logic.

2.1 Linear Logic

2.1.1 The role of structural rules

Linear logic[1] focuses on the elimination of the *structural rules* of sequent calculus, eventually followed by their *controlled recovery*. This control makes linear logic *resources-sensitive*, while keeping the same distinctions of intuitionistic logic and the *involutive negation* of classical logic, that makes a *double negation* equal to no negation, i.e. $\neg\neg A \equiv A$.

The structural rules in question are contraction and weakening:

Contraction $\dfrac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}$ C-L $\qquad \dfrac{\Gamma \vdash A, A, \Delta}{\Gamma, A \vdash A, \Delta}$ C-R

Weakening $\dfrac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}$ W-L $\qquad \dfrac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}$ W-R

[1] We assume some knowledge of classical (LK) and intuitionistic (LJ) sequent calculus, as well as the notion of *sequent* of formulas.

where Γ and Δ are multisets of formulas, and $\Gamma \vdash \Delta$ a sequent.

The elimination of the structural rules has as a consequence the *duplication* of the connectives $\wedge$ and $\vee$ in, respectively & (with)/$\otimes$ (tensor); and $\parr$ (par)/$\oplus$ (plus), depending on the choice of presentation: *additive* or *multiplicative*. The two presentations of the rules are the following

Additive

$\wedge$ **rules:** $$\frac{\Gamma, A_i \vdash \Delta}{\Gamma, A_0 \wedge A_1 \vdash \Delta}\ i \in \{0,1\} \qquad \frac{\Gamma \vdash A_0, \Delta \qquad \Gamma \vdash A_1, \Delta}{\Gamma \vdash A_0 \wedge A_1, \Delta}$$

$\vee$ **rules:** $$\frac{\Gamma \vdash A_i, \Delta}{\Gamma \vdash A_0 \vee A_1, \Delta}\ i \in \{0,1\} \qquad \frac{\Gamma, A_0 \vdash \Delta \qquad \Gamma, A_1 \vdash \Delta}{\Gamma \vdash A_0 \vee A_1, \Delta}$$

Multiplicative

$\wedge$ **rules:** $$\frac{\Gamma, A_0, A_1 \vdash \Delta}{\Gamma, A_0 \wedge A_1 \vdash \Delta}\ \wedge\text{-L} \qquad \frac{\Gamma \vdash A_0, \Delta \qquad \Gamma' \vdash A_1, \Delta'}{\Gamma, \Gamma' \vdash A_0 \wedge A_1, \Delta, \Delta'}\ \wedge\text{-R}$$

$\vee$ **rules:** $$\frac{\Gamma \vdash A_0, A_1, \Delta}{\Gamma \vdash A_0 \vee A_1, \Delta}\ \vee\text{-R} \qquad \frac{\Gamma, A_0 \vdash \Delta \qquad \Gamma', A_1 \vdash \Delta'}{\Gamma, \Gamma' \vdash A_0 \vee A_1, \Delta, \Delta'}\ \vee\text{-L}$$

The additive connectives depend on the contexts, since the context of the conclusion needs to be same to the context of the premises. The multiplicative connectives, instead, the context of both premises is kept in the conclusion. Additive and multiplicative connectives are equivalent in classical logic, but only *modulo* the *structural rules*. For instance:

$$\dfrac{\dfrac{\Gamma \vdash A_0, \Delta \qquad \Gamma \vdash A_1, \Delta}{\Gamma, \Gamma \vdash A_0 \wedge A_1, \Delta, \Delta}}{\Gamma \vdash A_0 \wedge A_1, \Delta}$$

where there are multiple contraction rules at the end, on Γ and Δ. Without contraction and weakening this equivalence is lost; $\wedge$ is split in multiplicative $\otimes$ and additive &, and $\vee$ in multiplicative $\parr$ and additive $\oplus$. By De-Morgan duality, we only suffice one of the right/left rules to obtain the

other; this let us formulate the sequent calculus with everything *on the right*, in the following way:

Additives

$$\frac{\vdash A_i, \Gamma}{\vdash A_0 \oplus A_1, \Gamma}\ i \in \{0,1\} \qquad \frac{\vdash A_0, \Gamma \quad \vdash A_1, \Gamma}{\vdash A_0 \,\&\, A_1, \Gamma}$$

Multiplicatives

$$\frac{\vdash A_0, \Gamma_1 \quad \vdash A_1, \Gamma_2}{\vdash A_0 \otimes A_1, \Gamma_1, \Gamma_2} \qquad \frac{\vdash A_0, A_1, \Gamma}{\vdash A_0 \parr A_1, \Gamma}$$

Along the connectives, even their respective *units* (the neutral elements) $\top, \bot$ are split, in $\top/1$ and $\bot/0$, where each unit is paired to the respective connective:

- $\bot$ to $\parr$.
- 1 to $\otimes$.
- 0 to $\oplus$.
- $\top$ to $\&$.

Being the neutral element, we have $A \otimes 1 \equiv A$, $A \parr \bot \equiv A$, etc.
The rules for units are:

Units $\dfrac{}{\vdash 1} \qquad \dfrac{\vdash \Gamma}{\vdash \bot, \Gamma} \qquad \dfrac{}{\vdash \top, \Gamma}$

There is *no* rule for 0.

In the setting we have now introduced *negation*, noted $A^\perp$ for a formula A is defined as he *De Morgan normal form* of A, defined through the following set of equivalences, which match multiplicative with multiplicative, and additive with additive:

$$(A \otimes B)^\perp \dashv\vdash (A^\perp) \parr (B^\perp);\ (A \oplus B)^\perp \dashv\vdash (A^\perp) \,\&\, (B^\perp);$$
$$1^\perp \dashv\vdash \bot;\ \top^\perp \dashv\vdash 0.$$

We call $A^\perp$ the *dual* of A. Negation is *involutive*, i.e. $A^{\perp\perp} \equiv A$, and we have that $A \vdash B \equiv \vdash A^\perp, B$, contrary to intuitionism, where there is a difference between sides (inputs and outputs). Nevertheless, linear logic is still able to

make the more refined distinctions typical of intuitionistic logic, as $\vdash A \parr A^\perp$ is provable, but $\vdash A \oplus A^\perp$ is not, maintaining the *disjunction* property at the same time as the involutive negation. Aware of this difference, we can now define the *linear implication*, $A \multimap B$, as $A^\perp \parr B$, opposed to $A^\perp \oplus B$.

The last rules we need to add to our sequent calculus are the Axiom and Cut rule.

$$\textbf{Axiom} \quad \frac{}{\vdash A, A^\perp} \qquad \textbf{Cut} \quad \frac{\vdash A, \Gamma_1 \qquad \vdash A^\perp, \Gamma_2}{\vdash \Gamma_1, \Gamma_2}$$

Note that the cut rule is essentially the same as a multiplicative $\otimes$ rule, and the axiom rule has no hidden weakening, contrary to the calculus LK. Using the cut rule, it is easy to see how linear logic treats formulas as resources.

For example:

$$\frac{\Gamma_1 \vdash A^\perp \parr B \qquad \dfrac{\Gamma_2 \vdash A \qquad \dfrac{}{\vdash B^\perp, B}}{\Gamma_2 \vdash A \otimes B^\perp, B}}{\Gamma_1, \Gamma_2 \vdash B}$$

To obtain B one instance of A is used, in $A \vdash B$ (which becomes $\vdash A^\perp \parr B$). If we had, instead, $A, A \vdash B$, contraction would let us obtain B with only one instance of A.

The lack of structural rules heavily hinders the expressiveness of linear logic; to recover this lost power they are added back to the calculus, but in a particular controlled setting through the *exponentials* ! and ?.

2.1.2 The exponentials

The exponentials are two modalities added to the calculus (in the form of *unary connectives*): ! (*of course*, or *bang*) and its dual ? (*why not*); so $(?A)^\perp \equiv !A^\perp$. Their purprose is to note that a formula is considered as if it were derived in a *classical* context, or environment; i.e a context with unlimited resources. This allows the use of weakening and contraction on such formulas, since they can potentially be obtained an arbitrary amount of times. Therefore, the structural rules hold for the exponentials:

$$\frac{\vdash ?A, ?A, \Gamma}{\vdash ?A, \Gamma} \qquad \frac{\vdash \Gamma}{\vdash ?A, \Gamma}$$

Formulas marked with a modality can be used as many times as one likes, and introduced as context of a sequent. The rules for their introduction are:

$$\textbf{Dereliction}\ \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A} \qquad \textbf{Promotion}\ \frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}$$

Note that a ! can only be introduced in a interrogated context. Indeed, another way to present the exponentials is via an *underlining* of formulas, as in $\underline{A}$, which means that the formula is treated as a formula of classical logic. In this setting the *dereliction* is allowed only on underlined formulas, and the meaning of the promotion is immediately clear: the rule, which let us use a formula as many times as we want, is possible only in an underlined (i.e. resources-free) context, because if $\Gamma \vdash A$, and Γ can be used repeatedly, then, and only then, we can obtain A as many times as we want.

Remark 2.1.1. *Adding the exponentials increases the complexity of linear logic proof-search, making it* undecidable.

2.1.3 Polarized connectives

Proof-search motivated reasons (especially the *focalization* theorem [5]) gave rise to the identification of other kinds of properties of linear logic connectives, beside the multiplicative-additive characterization. This let us group the connectives regarding their **polarity**: *positive* or *negative*.

Negatives $\{⅋, \&, \forall\}$ are the connectives which leave no choice to the rule to apply, in a proof-search strategy; when they are the principal connective of a formula, we are forced to apply the next rule on them. Each occurrence of a negative rule can be read bottom-up, and does not influence the other rules: a negative rule can always be the *last* move of the proof. Negative connectives are thus: *invertible*; *deterministic*; *passive*; or, in game semantical terms, *an opponent move*, made from an *internal choice*.

For the same reasons, the quantifier $\forall$ can also be considered negative, even if we have not introduced quantifiers in the calculus. Dually,

Positives $\{\oplus, \otimes, \exists\}$ are the connectives which do not determine univocally the first move to apply in a proof-search, when they are the principal connective of the formula. It can be due to many choices of repartition of context (in the $\otimes$ case), or the rule itself (for example $A \oplus B$ can be obtained from either A or B, but we can't know which on we need to carry out a proof). Positive connectives are: *irreversible*; *non-deterministic*; *active*; or a

move up to the *player*, which requires an *external choice*.

Let us further explore the reasons of this grouping. At first, we can note that the dual of a negative is positive and vice-versa: the linear negation $\perp$ can invert the polarity of a formula. Moreover, connectives of the same polarity *commute*, leading also to distributivity as a particular case, (for instance$(B \oplus C) \otimes A \equiv (A \otimes B) \oplus (A \otimes C)$) making possible to consider **clusters of rules** of the same polarity as a **single connective** (or **synthetic rule**), since the order of application of rules of the same polarity does not matter.

Indeed, the following sequents are easily provable:

$$\vdash (A \otimes (B \oplus C))^{\perp}, (A \otimes B) \oplus (A \otimes C)$$

and

$$\vdash ((A \otimes B) \oplus (A \otimes C))^{\perp}, A \otimes (B \oplus C).$$

Since negatives are reversible – we can derive $\vdash \Gamma, A, B$ from $\vdash \Gamma, A \parr B$, and $\vdash A, \Gamma$ and $\vdash B, \Gamma$ from $\vdash A \,\&\, B, \Gamma$ – we lose nothing in a proof-search strategy by applying the rule as soon as the connective in question is encountered (we have only one way to apply the rule). The sub-formulas A, B can then be sent in different premises when decomposing a context-sensitive rule as the $\otimes$. This property is at the center of the *focused discipline* in proof-search, as seen in Andreoli's *focalization* theorem [5] (of which we will give some hints), which gives priority to the decomposition of reversible rules. Ludics will arise from this focused discipline and use of synthetic rules, by forgetting the *type* of a proof (the formulas in its conclusion and premises).

A last remark must be made on ! and ?. There is not a general consensus on the polarity of the exponentials; the problem being nor the ? nor the ! can be considered *negative*. At first, the ! might seem invertible, since:

$$\dfrac{\vdash ?\Gamma, !A \qquad \dfrac{\vdash A^{\perp}, A}{\vdash ?A^{\perp} = (!A)^{\perp}, A}}{\vdash ?\Gamma, A}\ \text{cut}$$

However, ! is **not** deterministic and cannot be assumed to be the *last* rule of a proof, during proof-search, making the ! **not** negative. Indeed if we have

$$\dfrac{\dfrac{\dfrac{\vdash A^{\perp}, A}{\vdash ?A^{\perp}, A}}{\vdash ?A^{\perp}, !A} \qquad \vdash 1}{\vdash ?A^{\perp} \otimes 1, !A}\ \otimes$$

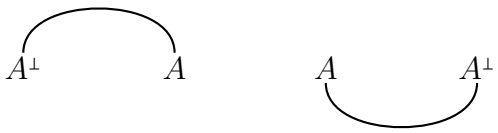
clearly $!A$ *cannot be decomposed first*, going bottom-up, since the context is not under ?, making ! not negative; while ? is clearly not-invertible as well.

2.1.4 Proof nets

Proof nets were introduced to have a more *geometrical* presentation of proof, without the constraint of a sequential representation, where we are often forced to choose irrelevant orders of application of rules, and therefore differentiate proofs which are essentially the same. Their aim is to abstract from sequentiality, make cut-elimination a local procedure, and to explicit the relations between formulas in a proof (axiom/premise and conclusion) by drawing links and giving importance to their *place* in the proof. Proof nets work well in *multiplicative* linear logic (MLL) while they struggle trying to deal with additive connectives in a natural way. We will give a short presentation, since variants of them have been used to translate either the π-calculus (in [22]) or CCS (in [9]), and are widely used in linear logic.

Proof nets are s particular case of *proof structure*, which are directed graphs where the nodes are labeled with formulas (we assume a calculus formulated *on the right*), and links between them (the edges) are introduced by the following rules:

Axiom link **Cut**



The axiom has no premise and two conclusions (and mimic the $\vdash A, A^\perp$ of the calculus), while the cut has two premises and no conclusion. Then we have the rules for $\otimes$ and $\parr$, which are simple binary-links

$\otimes$ link **$\parr$ link**

Definition 2.1.2. *A* proof structure *is a finite set of closed formulas of MLL and axioms, cut, $\otimes$ or $\parr$ links between them such that:*

- *Every formula is conclusion of* exactly *one link.*
- *Every formula is premise of* at most *one link.*

Formulas which are not *premise of any link are called* conclusions.

We have defined how to build a proof structure, the next step is to describe normalization on them: since it is a *local* reduction procedure, even if the proof structure *is not* a proof net, it still holds. Proof nets are indeed proof structures satisfying specific correctness criteria, which assure us that the structure in question is indeed a correct proof in MLL, and *sequentializable*, i.e. can be translated to one or more correct sequent calculus proofs, since we are abstracting from irrelevant orders of application of rules.

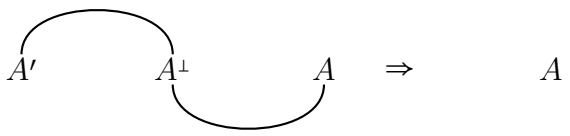
Normalization

Firstly, we can note that the translation from sequent calculus to proof structure is very straightforward. If we have a sequent calculus proof of $\vdash \Gamma$, then we can build a proof structure step-by-step, using an inductive definition:

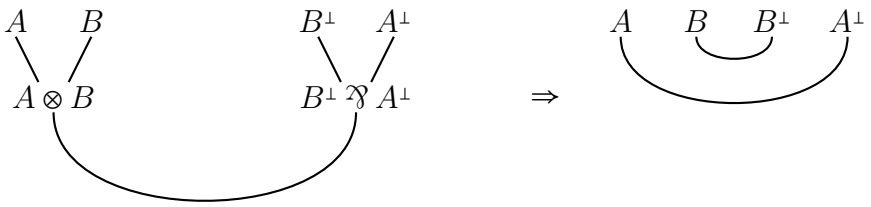
- **Axiom** $\vdash A, A^\perp$, is just a proof structure with an axiom link between A and $A^\perp$.
- **Cut** and $\otimes$ – remember that the cut is a particular case of $\otimes$ – have as premises $\vdash A, \Gamma, \vdash B, \Delta$ ($A^\perp, \Delta$ for the cut), and conclusion $\vdash A \otimes B, \Gamma, \Delta$ ($\vdash \Gamma, \Delta$). For inductive hypothesis we have two proof structure of respective conclusions A, Γ and B, Δ ($A^\perp, \Delta$), and using the respective link ($\otimes$ or cut) we obtain a proof structure with the same conclusion.
- $\parr$ goes from $\vdash A, B, \Gamma$ to $\vdash A \parr B, \Gamma$. For inductive hypothesis we have a proof structure of conclusion A and B, thus using a $\parr$ link between them we obtain the desired proof structure.

The difference between $\otimes$ and $\parr$ is that one is done between the conclusions of two different proof nets, the other between two conclusions of the same proof net. This distinction will make sense once the correctness criteria for proof structure are introduced. Indeed they are both a binary link where the only difference lies in the label; however they are treated differently when checking the correctness of a proof structure. The normalization procedure is extremely simple, and consists of only two cases:

1. Cut with an axiom link:



2. Cut between $\otimes$ and $\parr$.



The normalization procedure for MLL proof structures is *confluent* and verify the Church-Rosser property[2].

Proof-nets are then defined as *proof structures* satisfying certain *correctness criteria* (the mostly applied is the *Danos-Regnier* criterion, [21]), that are needed to make sure that a proof net is *sequentializable*. Indeed, proof nets are supposed to be proof structures corresponding to sequent calculus proofs in MLL. So, we stumble upon the problem of *sequentialization*: a proof structure can have many conclusions, but it does not tell us which is the *last* rule of the proof. Being sequentializable means that we are able to built a proof structure following a sequential order on the rules, order which must correspond to the rules of the sequent calculus.

To check the correctness of proof structures we need the notion of *switching*:

Definition 2.1.3. *Let $\mathcal{P}$ be a proof-structure, and L a $\parr$ link. A* switching *on L is a choice* **left** *or* **right** *on the link, which keeps an edge of the link while erasing the other. The resulting links are the following:*

Left switch Right switch

Then, we have that

Definition 2.1.4 (Proof net)**.** *A proof structure $\mathcal{P}$ is* correct *if for every choice of switching S on $\parr$ links of $\mathcal{P}$, the resulting graph is* connected and acyclic*. A correct proof structure is a* proof net*.*

[2]It is very easy to check the property. For reference, see [40].

To end this section we state Girard's sequentialization theorem [40], skipping its proof.

Theorem 2.1.5. *A proof net is* sequentializable*, i.e. it correspond to the translation of at least one sequent calculus proof in MLL.*

When introducing ludics, we will get back to linear logic to present the *focused discipline* for proofs, a proof-search algorithm based on polarization and alternation of positive and negative rules. Focused proofs form a clear connection with *game semantics* and its objects, called *strategies*, that are an alternation of a *Player* (a positive agent) and an *Opponent* (a negative agent), in which the objects of ludics, designs, can be defined.

2.2 Ludics

Ludics can be described as a *purely interactive approach to logic*. It is a research program started by Girard[3] to give a new foundation to logic based on the core notion of *interaction*. Indeed, the theory itself was developed inside the setting of *Geometry of Interaction* (GOI) whose objective is to give an account of the dynamics of proofs (in logic) – i.e. the normalization process – to be able to consider proofs through a notion of *duality*, and *interaction*. The GOI finds so a realization in ludics, as an approach to logic where interaction is its core notion.

Interaction is the equivalent of cut-elimination for the objects of ludics, called *designs*; the dynamic which let them combine and transform. However, unlike cut elimination for proofs in sequent calculus, it is the very founding notion of ludics, which means that designs are only a syntactical support for this notion: their meaning lies in all the possible interactions with the other designs; therefore they keep as information only what is essential for interaction to be carried out. Only once we put together the syntactical definition of designs together with the notion of interaction we can recover the usual objects of logic. This new and different approach makes formulas and proofs only particular instances of interaction, and let us formulate new concepts and properties to apply to the standard logical framework.

A design is an *abstraction* of a formal sequent calculus proof of multiplicative additive linear logic ($MALL$) where the *logical content* of the proof, the *formulas* in it, is *forgotten*. These proof-like objects are the syntactic support on which interaction can be performed: they are what is left once we isolate and keep only the elements which are necessary to be able to *use* a proof-like object in cut-elimination. In other words, only what tells us how this object *behaves* is kept, its *invariants* when we vary the counter-proofs with which cut elimination becomes possible. In this sense a designs is defined by its interactions, and, ideally, it is completely determined by the set of its counter-designs. This property is called *separation*: two different designs can always be *separated* by a third design that is orthogonal (i.e. can interact successfully) to only one of the two. If two designs have the very same interactions, then they are the very same object. However, the *separation theorem does not hold anymore* if we extend ludics beyond its natural linearity, as done in the works of C. Faggian and M. Basaldella [6], and K. Terui [66]. The procedure of cut elimination travels from the conclusions up to the axioms of a proof, it is indeed a proof-search procedure, and as we will see the formulation of designs is closely related to the *focusing discipline* of

[3] In [38].

proofs, used in the focalization theorem by Andréoli [5].

What we need to keep of a proof, to know how it interacts with its counter proofs, is its *locative* structure, or tree-structure, and the relation of subordination between its elements – the sub-formula relation – seen as places in the proof. Formulas $A, B, \ldots$ becomes *addresses* $\xi, \zeta, \ldots$, which are sequences of natural numbers; *sub-formulas* becomes sub-addresses, i.e. suffixes of these sequences as $\xi.1, \zeta.3.2, \ldots$. For instance, if we have ξ at the place of $(A \parr B) \oplus A$, then $\xi 1$ (or, equivalently, with a separator $\xi.1$) is $A \parr B$, $\xi 2$ is A, while $\xi 12$ is B and $\xi 11$ is A again, though explicitly the sub-formula of that particular occurrence of $A \parr B$. It becomes clear how ludics is intrinsically linear and locative: two instances of the same formula become two different addresses, since they stay in different places, and so a rule like contraction would not make any sense: ludics has a strict control on resources *by default*[4].

The formulation of designs as locative and interactive structures motivate the introduction of the **Daimon**, noted $\maltese$, a tool whose function is to *end* proof search, fulfilling the role of axioms, letting us have access to *enough* counter-proofs for interaction to be complete – even *incorrect ones*. Indeed $\maltese$ can be interpreted as a symbol standing for an *error*, ending a design whose equivalent in linear logic would be an *incorrect proof*; however this is not an issue in ludics. The reason that explains why we need enough *counter-proofs* is very simple: if every design must have an orthogonal one, then obviously one of the two must stand for an "incorrect proof", since if the conclusion of a proof is $\vdash A$, obviously we can't prove $\vdash \neg A$ at the same time, otherwise the system would be *inconsistent*. However when *types* are forgotten, and there are no axioms anymore, proving a "false" statement becomes possible, but irrelevant: what remains, and all that matters, is the geometrical structure of a proof.

So, designs can be defined as abstractions of focused $MALL$ proofs, that means with a strict alternation of *maximal cluster of rules of the same polarity*. As we anticipated in the polarization of the connectives, every proof have a *focused form* that brings ludics very close to *game semantics*, where a *play* is a strict alternation of a player move and an opponent counter-move. Designs can be formulated as Hyland-Ong *innocent linear strategies* on an universal arena, as found in [6] and [27]. In a focused formulation of the sequent calculus *on the right*, we will have an alternation of positive ($\oplus$ and $\otimes$)

[4] Curien, in [16], links the relation between formulas and addresses to the one between typed and untyped λ-calculus, noting that ludics can provide what categories lacks, regarding the semantics of *subtyping*. He gives also an interpretation of the *daimon* as an *error* elements, in the sense of a recoverable error, and stress the importance of errors in denotational semantics.

and negative ($\parr$ and &) rules; but ludics goes a bit further, actually considering only *positive formulas* by means of De Morgan duality: positive rules introduce addresses *on the right of* $\vdash$, and their sub-addresses, being negative, are put *on the left*, which by duality have their polarity inverted. Thus if $\vdash \xi$, then ξ is introduced by a positive rule, therefore its sub addresses ξi will be *negative* (since the rule is a maximal synthetic connective), and shifted on the left ($\xi i \vdash$) – each in a different premise – introduced by a *negative rule*, which still corresponds, in linear logic, to the introduction of a positive connective. While any number of addresses can be on the right, at most one can be on the left, since the rule introducing it must be the next to be performed, following the focusing discipline. Two considerations naturally arise from this link with linear logic proofs, the first being that proof search stops when reaching an axiom, which is of the form $\vdash A, A^{\perp}$. However, when proofs are *untyped* by substituting formulas with addresses, recognizing an axiom becomes impossible, since it would become just $\vdash \xi.i, \xi.j$, a sequent made of two different addresses without any particular connection between them except having the same prefix. Moreover, a sequent of the form $\xi \vdash \xi$ is not allowed, since in linear logic negation is not a rule, but only defined trough De Morgan duality; moreover, having the same address name two different places in a proof would not make any sense. Indeed, axioms and atomic formulas are completely *arbitrary*; in a sequent of the form $\vdash A, A^{\perp}$, A can be regarded as a *variable*, and could be substituted by any formula. However addresses do not have an atomic format, and designs are built bottom-up, where each rule *extends* the length of an address, by adding a suffix: there is no sub-formulas property, and there is nothing telling us when we are done, and when we can stop the proof-search. The Daimon ($\maltese$) is thus needed to play the role of axioms, telling us when to stop. It can be seen as the player "giving up" a game, or the acknowledgment that we do not know how to continue. Reading the proof top-down, instead the daimon resembles closely a *weakened* axiom, of the form $\dfrac{}{\vdash \xi, \Gamma}\ \maltese$.

For the counter-proofs argument that we explained above, the Daimon can be considered a sort of *error*: it always allow a counter-proof to exists (infinitely many, indeed), giving us an arbitrary means to stop proof-search. We can therefore always build a *counter-design* by simply performing the dual rule (or, with a game-semantical term, *action*), and then terminate proof search with a $\maltese$. Having counter-proofs is essential to ludics where the meaning of a design are its interactions against its *orthogonals* (the counter-designs), and not the arbitrary address in its conclusion (or *base*); just as in the contextual observational semantics of a program P, where a program meaning is given by how it behaves and composes with any evaluation context

$C[\ ]$ – i.e. a program with a *placeholder* that can be substituted by any other program P – via the possible reduction sequences of the composition $C[P]$. Therefore, interaction stops once it finds a ✠, which notes that it ended *successfully*, while it will *diverge* if a ✠ can't be reached: designs can indeed be *infinite* objects, and *interaction* never-ending.

2.2.1 Focalization

In this section we will briefly explain the *focusing discipline* and focalization theorem, to clarify and give a foundation to ludics' designs. Andréoli explored further the division in *positive* and *negative* connectives, trying to optimize proof-search in linear logic: the starting point of the focusing discipline is to consider *maximal cluster of connectives of the same polarity*, called *synthetic connectives*. A formula is thus decomposed up to its sub formulas of the opposite polarity; for example $((A_1 \otimes A_2)\,\&\,B) \parr C$ is a negative connective, with as (positive) sub formulas $A_1 \otimes A_2$, B and C.

We consider sequents of only *positive formulas*, as noted before, and with at most one formula on the left of $\vdash$ (the negative one). This follows from the use of synthetic connectives: we will have a left rule and a set of right rules for each connective; the right rules, being positive, are the irreversible ones; dually, the left is reversible, and can always be performed without consequences in proof search. Using synthetic connectives, the last rules of a focused proof of the previous formula are:

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash A_1, \Delta_1} \qquad \dfrac{\vdots}{\vdash A_2, \Delta_2}}{\vdash A_1 \otimes A_2, \Delta} \qquad \vdash B, C, \Delta}{\vdash ((A_1 \otimes A_2)\,\&\,B) \parr C, \Delta}$$

Being focused, A_1, A_2 are *negative formulas*; B, C are *positive formulas*, and the *negative* connectives & and $\parr$ are introduced at the same time, forming a *cluster* of two rules of the same polarity. Going further, and considering only positive formulas, it becomes

$$\dfrac{\dfrac{\dfrac{\vdots}{A_1^\perp \vdash \Delta_1} \qquad \dfrac{\vdots}{A_2^\perp \vdash \Delta_2}}{\vdash A_1 \otimes A_2, \Delta} \qquad \vdash B, C, \Delta}{((A_1 \otimes A_2)^\perp \oplus B^\perp) \otimes C^\perp \vdash \Delta}$$

and in ludics, untyping the proof, we have

$$\dfrac{\dfrac{\dfrac{\vdots}{\xi 11 \vdash \Delta_1} \qquad \dfrac{\vdots}{\xi 12 \vdash \Delta_2}}{\vdash \xi 1, \Delta} \qquad \vdash \xi 2, \xi 3, \Delta}{\xi \vdash \Delta}$$

(the two instances of $\xi 3$ are allowed only in the case of a negative rule, meaning that a & is taking place between some other addresses).

The *focusing discipline* is defined thus:

1. Once the proof-search starts the decomposition of a formula, it keeps decomposing its sub-formulas of the same polarity, until the principal connective is of the opposite polarity.

2. If there is a negative formula, it is decomposed first.

The result of the focusing discipline is that we have, at any time, at most one negative formula, which then get decomposed up to its positive sub-formulas. Then we need to chose which positive formula to decompose, and repeat the cycle.

The consequence of the focusing discipline is that if the active formula is positive, then all the formulas in the sequents are positive as well; after the decomposition we will have a single negative formula in each premise, which will be the active one, then generating again an only-positive premise, and so on, obtaining an alternation of *positive* and *negative* rules, each decomposing its conclusion up to the premises of the opposite polarity. By this way we consider positive sequents with at most a negative formula, and proofs made of synthetic connectives. As connectives are clusters of rules, we may consider them as just *partitions on the set of subformulas* of opposite polarity of the introduced formula. The positive rules will introduce formulas *on the right*, while the negatives *on the left*. Furthermore positive rules can be expressed as a single subset of subformulas, one element constituting a different premise (the choice lies in how to partition the context), while the negative ones as a subset of the *power set* (i.e. a set of subsets) of the subformulas, with each subset as a different premise (that can include more than a single subformula). For example, consider the following rules:

$$\frac{\vdash A, B, \Delta \qquad \vdash C, B, \Delta}{(A^\perp \oplus C^\perp) \otimes B^\perp \vdash \Delta}\ \{\{A, B\}, \{C, B\}\}$$

and $\dfrac{A \vdash \Gamma \qquad B \vdash \Delta}{\vdash (A^\perp \oplus C^\perp) \otimes B^\perp, \Gamma, \Delta}\ \{A, B\}$

The first is a negative rule, the second one is positive, introducing dual formulas. In ludics we assign sub addresses to each sub-formula, with as result

$$\frac{\vdash \xi 1, \xi 3, \Delta \qquad \vdash \xi 2, \xi 3, \Delta}{\xi \vdash \Delta}\ (-, \xi, \{\{1,3\},\{2,3\}\}) \text{ for the negative rule, and}$$

$$\frac{\xi 1 \vdash \Gamma \qquad \xi 3 \vdash \Delta}{\vdash \xi, \Gamma, \Delta}\ (+, \xi, \{1,3\})$$

for the positive one. We have that $\xi 1 = A$; $\xi 2 = C$ and $\xi 2 = B$, and $+/-$ makes explicit the polarity of the rule.

Note that any *isomorphic* connective has the same rule, once synthesized. This leads to the formulation of a *single* general rule for each polarity. Via distributivity and associativity we can consider a single positive rule and a single negative rule, from which we can derive all the particular instances. We have:

$$\textbf{positive rule:}\quad \frac{A_{i1} \vdash \Delta_1 \qquad \dots \qquad A_{in_i} \vdash \Delta_{n_i}}{\vdash (A_{11}^{\perp} \otimes \dots \otimes A_{1n_1}^{\perp}) \oplus \dots \oplus (A_{p1}^{\perp} \otimes \dots \otimes A_{pn_p}^{\perp}), \Delta}$$

with $(\Delta_1 \cup \ldots \cup \Delta_{n_i}) \subseteq \Delta$ and $\Delta_k \cap \Delta_l = \varnothing$ for $k, l \in \{1, \ldots, n_i\}$, i.e. the contexts of the premises are pairwise disjoint.

$$\textbf{negative rule:}\quad \frac{\vdash A_{11}, \dots, A_{1n_1}, \Delta \qquad \dots \qquad \vdash A_{p1}, \dots, A_{pn_p}, \Delta}{(A_{11}^{\perp} \otimes \dots \otimes A_{1n_1}^{\perp}) \oplus \dots \oplus (A_{p1}^{\perp} \otimes \dots \otimes A_{pn_p}^{\perp}) \vdash \Delta}$$

The *hypersequentialized* presentation of proofs let us keep only the part of the proof relevant for proof search, and thus for cut elimination. For example, the isomorphic formulas $A \otimes (B \oplus C)$ and $(A \otimes B) \oplus (A \otimes C)$ have the same presentation as synthetic rules. In both cases, regarding proof-search, we need to find a proof of A and a proof of B, or a proof for A and a proof of C; while for cut elimination we can proceed either with a cut on A and B or with a cut on A and C; there is no relevant distinction, except that the search-space for proofs is significantly smaller.

It is possible to introduce a *focusing sequent calculus*[5], by which we can build only proofs respecting the focusing discipline, using two operators for the *change of polarity*, $\downarrow$ and $\uparrow$, called *shift* operators. Atoms can be considered either positive (as is usually done) or negative without any loss, since for each atom we have its dual, of opposite polarity. The question about the polarity of atoms is not completely irrelevant, though. Indeed atoms are such for an arbitrary choice, as is the *Daimon*: we decide that we do not want, or need, further decomposition of a formula; otherwise atoms should

[5] A formal presentation can be found in [5] and [15].

be a special kind of formulas, with neutral polarity (or better, introduced by a neutral rule), since they have no connectives, subformulas or premises in the axiom rule.

Skipping the rules of the calculus, the theorem lies in proving that in this smaller search-space nothing is lost with respect to provability (we refer to [5] for the proof, or [15] for a short sketch). At last we have the focalization theorem:

Theorem 2.2.1 (Focalization)**.** *Focused proofs are* complete *with respect to provability. If* $\vdash A$ *is provable in the LL sequent calculus, then it is provable by a cut-free focused proof in the* focused sequent calculus*.*

2.2.2 Designs

Designs are an abstraction of focused proofs, built from synthetic connectives, from which the logical content – the formulas – has been substituted by addresses indicating their *place* in the proof, in order to keep only the essential structure needed for interaction. The structure of a design *is* the way it interacts (with its counter-designs), and thus its *semantics*.

Designs are based on hypersequentialized sequences of rules of *alternate* polarity, where these rules are called **actions**, of the form $(+/-, \xi, I/N)$, where $I \subset \mathbb{N}$ and $N \subset \mathcal{P}_{fin}(\mathbb{N})$. Such alternated sequences of actions are called **chronicles** or **branches**. A design is thus a set of chronicles verifying some constrains and properties, that can be represented as a set of sequences of actions, or in sequent calculus style as a proof-tree structure.

Designs as *dessin* or *desseins*

The basic notion defining designs are those of *action* and *justification*, which are defined on *addresses* as *focus* of an action.

Definition 2.2.2 (Abstract sequent)**.** *Addresses are sequences of* natural numbers*, noted* $\xi, \zeta, \ldots$*. An* abstract sequent*, or* base *is of the form* $\xi \vdash \Delta$ *if* negative*, or* $\vdash \Delta$ *if* positive*, with* Δ *a set of* addresses*; moreover we assume that all addresses in a base are* pairwise disjoint*. With* $\xi * J$ *we note any sub address of* ξ*.*

Remark 2.2.3. *Note that the disjunction requirement is not essential to make designs work. It is the practical expression of the locative principle, that an address should denote a particular* place *or* position *inside a proof, and thus must be unique. However, once the syntactic rules are given, we could drop this linearity constrain without occurring in particular inconsistencies:*

we would be just disregarding the meta-theory behind ludics. Still, allowing multiple copies of an address in a base does have some consequences on the properties of designs, as explained in [6].

At first, we can define designs as *dessein* (the original term of [38]), i.e. a set of sequences of actions of alternate polarity (in short *alternate sequences*). We need the following definitions[6]:

Definition 2.2.4 (Action)**.** *A* proper action *is a triple* $(\epsilon, \xi, I/\mathcal{N} = \{I_1, \ldots, I_n\})$*, noted* κ *such that:*

- $\epsilon \in \{+, -\}$ *is the* **polarity** *of* κ*;*
- ξ *is the* **focus** *of* κ*;*
- I *or* $\mathcal{N} = I_1, \ldots, I_n$ *is the* **ramification** *of* κ*; sometimes we improperly say* ramifications*, and call each element of* I *or* $\mathcal{N}$ *a single* ramification*. If* $\epsilon = +$ *then* $I \subset \mathbb{N}$*, i.e. a finite set of natural numbers; otherwise* $\mathcal{N} \subset \mathcal{P}_{fin}(\mathbb{N})$*, i.e. a finite element of the power set of natural numbers. We improperly denote both cases with* $I, J, \ldots$*, for any polarity of the action.*

The improper *action is the* **daimon***, a positive action noted with* ✠*. The dual of* ϵ *is noted* $\overline{\epsilon}$*.*

Next, we must define *alternated sequences* of actions as *chronicles*, based on the notion of *justification*:

Definition 2.2.5. *An action* $\kappa = (\epsilon, \xi i, J)$ *is* justified *by the action* $\tau = (\overline{\epsilon}, \xi, I)$ *if its* focus ξi *is produced by the ramification* I *of* τ*, i.e. is a sub address* ξi *of the focus* ξ *of* τ *such that* $i \in I$*.*

Definition 2.2.6 (Chronicle)**.** *A* chronicle*, noted* C*, is a non empty* alternated sequence *of actions such that:*

> **Justification:** *A* proper positive action *is either* justified *by a previous action, or* initial *if belongs to a* base*.*
>
> *A negative action is initial, if is the only negative action in a base, or justified by the* previous *positive action.*
>
> **Linearity:** *Actions of the sequence have not the same focus.*
>
> **Daimon:** *The daimon can be only the* last *action of the sequence.*

[6] For this part, we follow [60]

We say that a chronicle C is of base $\boldsymbol{base}$ $\Gamma \vdash \Delta$, with Γ a single address or $\varnothing$, if all the foci of the *initial positive actions* of C are in Δ, and the focus of a possible *initial negative action* is Γ.

We also need the following relation on chronicles

Definition 2.2.7. *Two chronicles* C_1 *and* C_2 *are* coherent, *in symbols* $\mathrm{C}_1 \sim \mathrm{C}_2$, *if:*

- *Either one extends the other or they differ on a* negative action.
- *If two chronicles have a different negative action, then there are no two actions in the chronicles with the same* focus.

Remark 2.2.8. *The second condition gives an implicit restriction on the division of the* context *on negative rules: the context must not be necessarily disjoint in the premises of a negative rule, but if the same address is kept twice, only one chronicle may have a rule with it as focus, to be coherent with the others.*

Now we have all the elements for the definition of design as *dessein*.

Definition 2.2.9 (Dessein)**.** *A* **dessein**, *noted* $\mathcal{D}$, *of base* $\Gamma \vdash \Delta$ *is a* set of chronicles *of the same base such that:*

(i) $\mathcal{D}$ *is* closed by prefix, *i.e. if* $\xi i \in \mathcal{D}$ *then* $\xi \in \mathcal{D}$.

(ii) $\forall \mathrm{C}_1, \mathrm{C}_2 \in \mathcal{D}, \mathrm{C}_1 \sim \mathrm{C}_2$, *therefore* $\mathcal{D}$ *is a* clique *of chronicles, i.e. they are pairwise coherent.*

(iii) A chronicle of $\mathcal{D}$ *without extension ends with a* positive *action.*

(iv) If the base of $\mathcal{D}$ *is* positive *then* $\mathcal{D} \neq \varnothing$.

We will occasionally use the *dessein* presentation. However, for the more familiar syntax and intuitive structure of sequent calculus proofs, we will rely mostly on the *dessin* definition of designs, and say *design* in general, letting the context specify the presentation used.

Definition 2.2.10 (Dessin)**.** *A* dessin $\mathcal{D}$ *is a tree of* abstract sequents *built and labeled by actions* κ, *such that the leaves of the tree are* negative sequents, *or positive sequents introduced by a* **daimon** *action* $\maltese$ *(thus, each branch must end with a positive action). Actions are noted in the following way:*

- Daimon *(*Dai^{+}_{Δ} *:)* $\dfrac{}{\vdash \Delta}\ \maltese$
  where Δ *is a finite set of addresses. The Daimon is a* positive *action.*

- Positive action: $\dfrac{\dots \quad \xi.i \vdash \Delta_i \quad \dots}{\vdash \Delta, \xi}\ (+, \xi, I)$
  where $i \in I$, *with* $I \subset \mathbb{N}$. I *is the* ramification *of the rule,* ξ *is the* focus, Δ_i*s are* disjoint *and* included *in* Δ.

- Negative action: $\dfrac{\dots \quad \vdash \xi.I, \Delta_I \quad \dots}{\xi \vdash \Delta}\ (-, \xi, \mathcal{N})$
  where $\mathcal{N}$ *is the ramification of the rule, and* ξ *its focus.* $\mathcal{N} \subset \mathscr{P}_{fin}(\mathbb{N})$, $\xi.I = \xi.1, \dots, \xi.n$, *with* $1, \dots, n \in I \in \mathcal{N}$, *and the* Δ_I *are* included *in* Δ.

Positive and negative actions are alternated *and Dai*$^+$ *can only be the* last *action of a dessin. As we said we sometimes improperly call* ramifications *of a rule the whole chronicles (or branches) generated by that rule.*

Remark 2.2.11. *We must note that, despite being an abstraction of* linear logic *proofs, the actions hide an implicit* weakening, *as noted in [15]. While the daimon may be seen as a* weakened axiom *rule, the requisite on the* context Δ *is such that the contexts* Δ_i *of the premises must only be* included, *thus there might be some unique addresses in* Δ, *which then disappear from the bases of the* ramifications. *Indeed, if no action is performed on these unique addresses, they would not appear at all in the* dessein *representation, as set of chronicles.*

Moreover, the case of the empty ramification $(+, \xi, \varnothing)$ *is admissible by definition. The linear logic correspondent of this action can be either* $\dfrac{}{\vdash \top, \Gamma}$, *or* $\dfrac{}{\vdash 1}$, *depending on the presence or not of the context (on which, however, no rule is performed). Still, this is not an actual* weakening, *since addresses disappearing from the context are irrelevant with respect to the correctness of the sequence of actions, i.e. they are still chronicles, and the* dessein *representation, as any game semantical one, is thus unaffected.*

The examples previously given show how dessins comes directly from the abstraction of sequent calculus proofs. Therefore we can interpret dessins as **cut free** focused proofs, built from the unique negative and positive rules defined above. As a proof have many *sub proofs*, we can define the notion of *sub design*, which is a sub tree whose root is the base of one of the sub chronicle of the designs, i.e. is one of the premises generated by the ramification of an action.

Example 2.2.12. *Let $\mathcal{D}$ be*

$$\dfrac{\dfrac{\dfrac{\xi011 \vdash}{\vdash \xi01}}{\xi0 \vdash} \qquad \dfrac{\dfrac{\xi100 \vdash}{\vdash \xi10}}{\xi1 \vdash} \qquad \dfrac{\dfrac{}{\vdash \xi30}}{\xi3 \vdash}\ \maltese}{\vdash \xi}$$

Then $\dfrac{\dfrac{\xi011 \vdash}{\vdash \xi01}}{\xi0 \vdash}$ *and* $\dfrac{\dfrac{\xi100 \vdash}{\vdash \xi10}}{\xi1 \vdash}$ *are sub designs of $\mathcal{D}$, while*

$$\dfrac{\dfrac{\xi100 \vdash}{\vdash \xi10} \qquad \dfrac{\dfrac{}{\vdash \xi30}}{\xi3 \vdash}\ \maltese}{\vdash \xi}$$

is not.

The two presentations of *dessin* and *dessein* are equivalent, and we can always shift from one to the other. Going from dessin to dessein is extremely simple: the correspondent *dessein* of a *dessin* is the set of chronicles obtained by following, **bottom-up**, the sequence of actions of each branch of the tree, where the *base* of the *dessein* is the same one of the starting *dessin*.

Therefore at each *dessin* we can associate an unique *dessein*. The converse, however, is not immediately possible, because of the choices we have on the *context*. The context can be partitioned in many ways, but any addresses not focus of an action is not represented in the *dessein* counterpart of a *dessin*, therefore we need a stronger notion of *dessin* in order to have a one-to-one correspondence.

Informally[7], we are going to naively *delete* from the (bases of the) ramifications all addresses in the *base* $\vdash \Gamma, \Delta$ of a dessein which *are not* focus of an action, and build a dessin from this sequence of actions, which in this way is going to be *unique*. The resulting dessin is called the *associated sober dessin*.

Example 2.2.13. *Let $\mathcal{D}$ be the dessein*

$$\{\kappa_1 = (+, \xi, \{1,2,3\}),$$
$$\kappa_2 = (-, \xi1, \{\{0\}\})\ \kappa_3 = (+, \xi10, \{0\});$$
$$\kappa_4 = (-, \xi2, \{\{0\}\})\ \kappa_5 = (+, \xi20, \varnothing);$$

[7] The formal definitions can be found in [60].

$\kappa_6 = (-, \xi 3, \{\{0\}\})(+, \xi 30, \maltese)\}$

of base $\vdash \xi, \sigma, \tau$. *Then its* associated sober dessin *is*

$$\dfrac{\dfrac{\dfrac{\xi 100 \vdash}{\vdash \xi 10}\,\kappa_3}{\xi 1 \vdash}\,\kappa_2 \qquad \dfrac{\dfrac{}{\vdash \xi 20}\,\kappa_5}{\xi 2 \vdash}\,\kappa_4 \qquad \dfrac{\dfrac{}{\vdash \xi 30}\,\maltese}{\xi 3 \vdash}\,\kappa_6}{\vdash \xi, \sigma, \tau}\,\kappa_1$$

where σ *and* τ *are erased from the ramifications of the base.*

2.2.3 Interaction

We presented interaction as the core notion of ludics, which gives meaning to its objects, that at the same time serve as its necessary supporting structure. The most familiar way to describe interaction is via a cut-elimination like algorithm, defined on *dessins* and easily transferred to desseins, via the associated sober dessin.

We must note, however, that interaction, as well as designs, can be defined in *game semantics* terms: designs as *Hyland-Ong* games, and interaction as an *abstract machine* (as done in [6], [66], both modification of the abstract machine of [20]). We give a definition close to the original one given in [38], and presented using the notion of *interaction paths*.

Interaction is performed in *cut-nets*, set of designs which enjoy certain properties.

Definition 2.2.14. *A* cut-net, *denoted* $\mathscr{R}$, *is a set of designs such that:*

- *Each address appears at most in the base of two different designs, in dual position. Each such pair is called a cut, as for instance* $(\xi \vdash, \vdash \xi)$.
- *The graph with as nodes the bases of the designs, and as edges the cuts must be* acyclic and connected.

The base *of a cut-net is the sequent* $\Gamma \vdash \Delta$, *where* Γ *is either empty or the only un-cut negative address of a base (the connectedness and acyclicity of the graph force it to be at most one), and* Δ *are all the positive un-cut addresses (of the bases of the designs).*

The principal design *of the cut-net is the design whose base has the same negative part of the base of the cut-net.*

A cut-net is said closed *if its base is empty, otherwise is* open.

The closed case interaction is the most interesting to us, since it let us define the notion of *orthogonal design*, which is the correspective of logical negation and duality. An interaction between two designs ends *successfully* if the result is ✠, and is defined thus:

Definition 2.2.15 (Closed interaction)**.** *Let $\mathscr{R}$ be a closed cut-net. Let $\mathcal{D}$ be the principal design of the net. Its base must be positive, since all negative bases are part of a cut. The result of interaction, noted $[\![\mathscr{R}]\!]$, is defined by the following algorithm. Let k be the first rule of $\mathcal{D}$, then*

1. *If k is ✠ then $[\![\mathscr{R}]\!] = ✠$.*
2. *Otherwise $k = (+, \xi, I)$, and ξ is part of a cut with a rule $(-, \xi, \mathcal{N})$ of a distinct design $\mathcal{C}$, with $\mathcal{N} = \{J_1, \ldots, J_n\}$. Then:*
   - *If $I \notin \mathcal{N}$ then interaction diverges, and $[\![\mathscr{R}]\!] = \vdash$, an empty set of chronicles, which is not a design.*
   - *If $I \in \mathcal{N}$, then interaction continues on the sub designs of $\mathcal{D}$ generated by the ramification I, and the sub-design of C generated by the sub-ramification $I \in \mathcal{N}$ (and the other designs of $\mathscr{R}$). So, interaction continues on*

     $$[\![\mathscr{R}]\!] := [\![\mathscr{R}[{}^{\mathcal{D}_j (j \in I)}/_{\mathcal{D}}][{}^{\mathcal{C}_\mathcal{I}}/_{\mathcal{C}}]]\!]$$

     the same cuts net where $\mathcal{D}$ and $\mathcal{C}$ are substituted by, respectively, all its sub-designs for the first, and the sub-design with the matching ramification for the second. Note that here a change in focus can occur, as the new principal design will be $\mathcal{C}_I$, and its first (positive) rule might be on what was context in the step before (not one of the $\xi * I$). The new principal design is, then, $\mathcal{C}_I$.

The open *case is very similar to the closed one, except for a few points:*

- *The* principal *design $\mathcal{D}$ might be* negative*: in this case, the cut-net $\mathscr{R}$ is* split *in one cut-net for each premise of $\mathcal{D}$; each one will substitute the principal design in its respective cut-net.*
- *The focus of the action k of the positive principal design might not be cut: in this case, we consider for each sub-designs $\mathcal{D}_i$ of $\mathcal{D}$ a cut-net $\mathscr{R}_i$, containing $\mathcal{D}_i$ and the designs of $\mathscr{R}$ whose base has a cut with any address of the context of $\mathcal{D}_i$.*

Each action k (each first rule of the current $\mathcal{D}$), and its dual $k^\perp$, are said **visited** *during interaction.*

Example 2.2.16.

$$\mathcal{D} = \dfrac{\dfrac{\vdots}{\xi.1 \vdash} \quad \dfrac{\vdots}{\xi.2 \vdash}}{\vdash \xi} (+, \xi, \{1,2\})$$

$$\mathcal{C} = \dfrac{\ldots \quad \dfrac{\vdots}{\vdash \xi.1, \xi.2, \Gamma_I} \quad \ldots}{\xi \vdash \Gamma} (-, \xi, \{\ldots, \{1,2\}, \ldots\})$$

The two ξ form a cut. Interaction checks the premises of the positive action, and continues if its sub-addresses (the ramification*) match with one set of the negative counterpart, otherwise diverges. It stops if it reaches a $\maltese$.*

We can now give the definition of orthogonal, and duality in general.

Definition 2.2.17 (Orthogonal)**.**

- *Two designs $\mathcal{D}, \mathcal{C}$ of respective base $\vdash \xi$ and $\xi \vdash$ are orthogonals, or duals, if $[\![\mathcal{D}, \mathcal{C}]\!] = \maltese$. In symbols we write $\mathcal{D} \perp \mathcal{C}$.*
- *A set of designs $R = \{\mathcal{C}_0, \ldots, \mathcal{C}_n\}$ of respective bases $\vdash \xi, \sigma_1 \vdash, \ldots, \sigma_n \vdash$, is orthogonal to a design $\mathcal{D}$ of base $\xi \vdash \sigma_1, \ldots, \sigma_n\}$ if $[\![\mathcal{D}, R]\!] = \maltese$; in symbols we write $\mathcal{D} \perp R$.*
- *The set of all orthogonals of $\mathcal{D}$ is noted $\mathcal{D}^\perp$; the orthogonal of a set of designs R is*

$$R^\perp = \bigcap_{\mathcal{D} \in R} \mathcal{D}^\perp.$$

Interaction can be defined through a very intuitive token-machine between two designs forming a cut on their bases. We define the algorithm for the *closed case*, thus starting with a positive design.

Definition 2.2.18 (Token-machine interaction)**.** *Let t be a token and t_n the token position at the n-th step. Let $\mathcal{D}$ be a design of positive base $\vdash \xi, \Delta$, introduced by $(+, \xi, I)$, and $\mathcal{C}$ a design of negative base $\xi \vdash \Delta'$, introduced by $(-, \xi, \mathcal{N})$.*

$t_0 = (+, \xi, I)$.

Let t_n be a positive rule:

1. *If $t_n = (+, \Delta, \maltese)$, then interaction stops (successfully). The result of interaction is $\maltese_\Delta$.*

2. *If* $t_n = (+, \zeta, I)$ *in* $\mathcal{D}$ *(without loss of generality), then we have two cases:*

   - *If* $\exists (-, \zeta, \mathcal{N}) \in \mathcal{C}$, *and* $I \in \mathcal{N}$, *then* $t_{n+1} = (-, \zeta, \mathcal{N})$.
   - *Otherwise* $t_{n+1} = \perp$ *(is undefined), and interaction diverges.*

3. *If* t_n *is a negative rule of the form* $(-, \zeta, \mathcal{N})$, *and* $t_{n-1} = (+, \zeta, I)$, *with* $I \in \mathcal{N}$, *then* $t_{n+1} = (+, \zeta.i, I')$, *the rule introducing the* I*-th premise of* $(-, \zeta, \mathcal{N})$, *where* $i \in I$.

An example of interaction

Example 2.2.19.

$$\textit{Let } \mathcal{D} = \dfrac{\dfrac{\dfrac{\overline{\xi.1.1.1 \vdash}}{\vdash \xi.1.1}\,(+, \xi.1.1, \{1\})^4}{\xi.1 \vdash}\,(-, \xi.1, \{\{1\}\})^3 \quad \dfrac{\dfrac{\overline{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}\,(+, \xi.2.1, \{1\})^8}{\xi.2 \vdash}\,(-, \xi.2, \{\{1\}\})^7}{\vdash \xi}\,(+, \xi, \{1, 2\})^0$$

$$\textit{and } \mathcal{C} = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\quad}}{\vdash \xi.2.1.1, \xi.1.1.1}\,\maltese^{10}}{\xi.2.1 \vdash \xi.1.1.1}\,(-, \xi.2.1, \{\{1\}\})^9}{\vdash \xi.1.1.1, \xi.2}\,(+, \xi.2, \{1\})^6}{\xi.1.1 \vdash \xi.2}\,(-, \xi.1.1, \{\{1\}\})^5}{\vdash \xi.1, \xi.2}\,(+, \xi.1, \{1\})^2}{\xi \vdash}\,(-, \xi, \{\{1, 2\}\})^1$$

The numbers denote the n*-th interaction step. Interaction starts from the cut on the bases* $(\vdash \xi, \xi \vdash)$, *and checks the premises of the* $+$ *rule: if* $\{1, 2\}$ *finds a match in the corresponding negative rule, interaction continues, and ends* successfully *if it reaches a* $\maltese$. *In the following we omit to note the ramifications for brevity; the interaction steps are:*

(1) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\xi.1.1.1 \vdash}}{\vdash \xi.1.1}\,(+,\xi.1.1)^4}{\xi.1 \vdash}\,(-,\xi.1)^3 \qquad \dfrac{\dfrac{\dfrac{}{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}\,(+,\xi.2.1)^8}{\xi.2 \vdash}\,(-,\xi.2)^7}{\vdash \xi}\,(+,\xi)^0 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \xi.2.1.1, \xi.1.1.1}\,\maltese^{10}}{\xi.2.1 \vdash \xi.1.1.1}\,(-,\xi.2.1)^9}{\vdash \xi.1.1.1, \xi.2}\,(+,\xi.2)^6}{\xi.1.1 \vdash \xi.2}\,(-,\xi.1.1)^5}{\vdash \xi.1, \xi.2}\,(+,\xi.1)^2}{\xi \vdash}\,(-,\xi)^1$$

Visited actions: $(+,\xi)^0(-,\xi)^1$

(2) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\xi.1.1.1 \vdash}}{\vdash \xi.1.1}\,(+,\xi.1.1)^4}{\xi.1 \vdash}\,(-,\xi.1)^3 \qquad \dfrac{\dfrac{\dfrac{}{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}\,(+,\xi.2.1)^8}{\xi.2 \vdash}\,(-,\xi.2)^7 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \xi.2.1.1, \xi.1.1.1}\,\maltese^{10}}{\xi.2.1 \vdash \xi.1.1.1}\,(-,\xi.2.1)^9}{\vdash \xi.1.1.1, \xi.2}\,(+,\xi.2)^6}{\xi.1.1 \vdash \xi.2}\,(-,\xi.1.1)^5}{\vdash \xi.1, \xi.2}\,(+,\xi.1)^2$$

Visited actions: $(+,\xi)^0(-,\xi)^1 \to (+,\xi.1)^2(-,\xi.1)^3$

(3) :

$$\dfrac{\dfrac{\dfrac{}{\xi.1.1.1 \vdash}}{\vdash \xi.1.1}\,(+,\xi.1.1)^4 \qquad \dfrac{\dfrac{\dfrac{}{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}\,(+,\xi.2.1)^8}{\xi.2 \vdash}\,(-,\xi.2)^7 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \xi.2.1.1, \xi.1.1.1}\,\maltese^{10}}{\xi.2.1 \vdash \xi.1.1.1}\,(-,\xi.2.1)^9}{\vdash \xi.1.1.1, \xi.2}\,(+,\xi.2)^6}{\xi.1.1 \vdash \xi.2}\,(-,\xi.1.1)^5$$

Visited actions: $(+,\xi)^0(-,\xi)^1 \to (+,\xi.1)^2(-,\xi.1)^3 \to (+,\xi.1.1)^4(-,\xi.1.1)^5$

(4) :

$$\dfrac{\dfrac{\dfrac{}{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}\,(+,\xi.2.1)^8}{\xi.2 \vdash}\,(-,\xi.2)^7 \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \xi.2.1.1, \xi.1.1.1}\,\maltese^{10}}{\xi.2.1 \vdash \xi.1.1.1}\,(-,\xi.2.1)^9}{\vdash \xi.1.1.1, \xi.2}\,(+,\xi.2)^6$$

Visited actions: $(+,\xi)^0(-,\xi)^1 \to (+,\xi.1)^2(-,\xi.1)^3 \to (+,\xi.1.1)^4(-,\xi.1.1)^5 \to (+,\xi.2)^6(-,\xi.2)^7$

(5) :

$$\dfrac{\dfrac{}{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}\ (+,\xi.2.1)^{8} \qquad \dfrac{\dfrac{}{\vdash \xi.2.1.1, \xi.1.1.1}\ \maltese^{10}}{\xi.2.1 \vdash \xi.1.1.1}\ (-,\xi.2.1)^{9}$$

Visited actions: $(+,\xi)^0(-,\xi)^1 \to (+,\xi.1)^2(-,\xi.1)^3 \to (+,\xi.1.1)^4(-,\xi.1.1)^5 \to (+,\xi.2)^6(-,\xi.2)^7 \to (+,\xi.2.1)^8(-,\xi.2.1)^9$

(6) :

$$\dfrac{}{\vdash \xi.2.1.1, \xi.1.1.1}\ \maltese^{10}$$

Visited actions: $(+,\xi)^0(-,\xi)^1 \to (+,\xi.1)^2(-,\xi.1)^3 \to (+,\xi.1.1)^4(-,\xi.1.1)^5 \to (+,\xi.2)^6(-,\xi.2)^7 \to (+,\xi.2.1)^8(-,\xi.2.1)^9 \to \maltese^{10}$.

Note that the least number *at each step denotes the* first action k *of the current* principal design. *We can see how the cut-net* $\mathscr{R} = [\![\mathcal{D}, \mathcal{C}]\!]$ *after each step is restricted to sub-designs of* $\mathcal{D}$ *and* $\mathcal{C}$. *For instance,* $\mathscr{R}^3$, *noting the cut-net at the third step, is composed by:*

$\mathcal{D}_1$: $\dfrac{\dfrac{}{\xi.1.1.1 \vdash}}{\vdash \xi.1.1}\ (+,\xi.1.1,\{1\})^4$; $\mathcal{D}_2$: $\dfrac{\dfrac{\dfrac{}{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}}{\xi.2 \vdash}\ (+,\xi.2.1,\{1\})^8\ (-,\xi.2,\{\{1\}\})^7$;

and $\mathcal{C}_1$: $\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \xi.2.1.1, \xi.1.1.1}\ \maltese^{10}}{\xi.2.1 \vdash \xi.1.1.1}\ (-,\xi.2.1,\{\{1\}\})^9}{\vdash \xi.1.1.1, \xi.2}\ (+,\xi.2,\{1\})^6}{\xi.1.1 \vdash \xi.2}\ (-,\xi.1.1,\{\{1\}\})^5$

The core theorems

Interaction and designs yield the following results, called *analytical theorems*, at the core of the theory[8].

1. The *associativity theorem*, a diamond (confluence) property for interaction.

2. The *monotonicity theorem*, and

3. the *stability theorem*.

[8] All the theorems have a correspondent in the λ-calculus, as noted in [16].

The **separation theorem** is what keeps them together, stating that a design is completely identified by its interactions: two designs are the same if and only if they have the same interactions (the set of their orthogonals is the same), i.e. there is no third design orthogonal to only one of the two.

The theorem is tied to the intrinsic *linearity* of ludics. For instance in the already cited *ludics with repetitions* we find a formulation in game-semantics terms where repeated actions with the same focus are admitted; however, once the linearity constrain is dropped – allowing repetition of addresses – the separation theorem does not hold anymore[9].

As a consequence of interaction and the separation theorem, we can define a notion of *inclusion* between designs, using the orthogonals as method of comparison. Note that this is not the *sub design* relation, which is not an inclusion with respect to interaction, since a sub-design has as base a sub-address of the main one. In the same way, shortening branches of a design $\mathcal{D}$ using $\maltese$ (an idea to which we will resort later), while apparently could be seen as an inclusion, has the opposite effect on the orthogonal, since $\mathcal{D}^{\perp} \subseteq (\mathcal{D}^{\maltese})^{\perp}$.

The relation that we need is an inclusion of the following kind:

Definition 2.2.20. *Let $\mathcal{D}$ and $\mathcal{C}$ be designs with the same base. We say that $\mathcal{D}$ is* more defined *than $\mathcal{C}$, in symbols $\mathcal{D} \preceq \mathcal{C}$, if $\mathcal{D}^{\perp} \subseteq \mathcal{C}^{\perp}$.*

$\preceq$ is transitive, reflexive and *antisymmetric* by the *separation theorem*. Indeed it can never be $\mathcal{D}^{\perp} = \mathcal{C}^{\perp}$ if $\mathcal{C} \neq \mathcal{D}$.

The fact that designs are fully described by their interactions is a notable result, which goes in accord with ludics' aim of closing the gap between syntax and semantic. The stability and monotonicity theorems are usually *semantics* property; while confluence (the diamond property) is syntactic, depending of how the calculus is performed. In ludics designs are defined by their dynamics, and thus only *interaction* remain as a meter of evaluation; therefore we can express all these properties via interaction.

Using the $\preceq$ as ordering between designs, we can express the previous properties in the following way:

1. **Associativity:** let $\{\mathscr{R}_0, \ldots, \mathscr{R}_n\}$ be a set of cut-nets, then $[\![\mathscr{R}_0, \ldots, \mathscr{R}_n]\!] = [\![[\![\mathscr{R}_0]\!], \ldots, [\![\mathscr{R}_n]\!],]\!]$

2. **Stability:** let $\mathscr{R}$ be a cut-net, and $\mathscr{R}_i \subset \mathscr{R}$, for all $i \in I \neq \varnothing$. Then $[\![\bigcap_{i\in I} \mathscr{R}_i]\!] = \bigcap_{i\in I} [\![\mathscr{R}_i]\!]$.

3. **Monotonicity:** if $\mathcal{D}_0 \preceq \mathcal{C}_0, \ldots, \mathcal{D}_n \preceq \mathcal{C}_n$ then $[\![\mathcal{D}_0, \ldots, \mathcal{D}_n]\!] \preceq [\![\mathcal{C}_0, \ldots, \mathcal{C}_n]\!]$.

[9] While still allowing *internal* and *full* completeness, [6], section 9 and 10.

Using the notion of orthogonality we can now define **behaviours**, sets of designs of same base *closed by bi-orthogonality* (i.e. of designs which *behave* the same with respect to interaction), which are a fundamental tool of ludics. They are ludics equivalent of *types*, in which we can interpret linear logic formulas, via a BHK-like semantics: a formula A is the set of its proofs, thus a behaviour $\mathscr{A}$ interpreting A contains the designs corresponding to the proofs of A. Operations on behaviours corresponding to linear logic connectives are definable, noted $\otimes, \parr, \with, \oplus$, which let us compose the types, along with properties of **full** and **internal** completeness with respect to linear logic proofs (we state the theorems below).

2.2.4 Behaviours

A behaviour is a set of designs closed by bi orthogonality: a set $\mathscr{B}$ of designs with the same base such that $\mathscr{B} = \mathscr{B}^{\perp\perp}$. A behaviour is *positive* if of base $\vdash \Gamma$, negative if of base $\xi \vdash \Gamma$.

Definition 2.2.21. *A set of designs of same base $\mathscr{B}$ is a* behaviour *if $\mathscr{B} = \mathscr{B}^{\perp,\perp}$. A behaviour has the same polarity of its base.*

If $\mathbb{B} = \{\mathcal{D}_0, \ldots, \mathcal{D}_n\}$ (with each $\mathcal{D}_i$ of same base), then we say that $\mathbb{B}$ is the set of generators *of $\mathbb{B}^{\perp\perp} = \mathscr{B}$.*

A behaviour $\mathscr{B}$ is connected *if all $\mathcal{D} \in \mathscr{B}$ have the* same first action.

Be aware that any set orthogonal of a *net* of designs, i.e. a set $\mathscr{R}$ of designs with the same base, for example defined as $\mathscr{R}^{\perp}$, is a behaviour, since $\mathscr{R}^{\perp\perp\perp} = \mathscr{R}^{\perp}$.

Another notable fact is that, if we take a single design $\{\mathcal{D}\}$, then the smallest behaviour containing it is

$$\{\mathcal{D}\}^{\perp\perp} = \{\mathcal{D}' \mid \mathcal{D} \preceq \mathcal{D}'\}.$$

This follows directly by the definition of $\preceq$. Some immediate properties are:

- Let $\mathscr{B}$ be a behaviour, then if $\mathcal{D} \in \mathscr{B}$ and $\mathcal{D} \preceq \mathcal{C}$, then $\mathcal{C} \in \mathscr{B}$.
- If, $\forall \mathcal{D}_k \in \mathscr{B}$, with $k \in K \neq \varnothing$, $\exists \mathcal{D}$ such that $\forall \mathcal{D}_k, \mathcal{D}_k \preceq \mathcal{D}$, then $\bigcap_{k\in K} \mathcal{D}_k \in \mathscr{B}$, by the stability theorem.
- Let $\mathscr{B}$ and $\mathscr{G}$ be behaviours. Then $\mathscr{B} \cap \mathscr{G}$ is a behaviour. Assuming that $\mathscr{B} = \mathbb{B}^{\perp}$ and $\mathscr{G} = \mathbb{G}^{\perp}$, then we have $\mathbb{B}^{\perp} \cap \mathbb{G}^{\perp} = (\mathbb{B} \cup \mathbb{G})^{\perp}$, by De-Morgan

duality, which is a behaviour, since it is defined as the *orthogonal* of a given set. In short, every intersection of behaviours is a behaviour.

- If $\mathscr{G} \subset \mathscr{B}$ then $\mathscr{B}^\perp \subset \mathscr{G}^\perp$.

Example 2.2.22. *Here are a few examples of notable behaviours.*

- *If* $\mathbb{B} = \{Sk_\xi\}$ *the empty design of negative base* $\xi \vdash$, *then* $\mathbb{B}^\perp = \{\maltese_\xi\}$, *the design* $(+, \xi, \maltese)$ *of base* $\vdash \xi$. *Then,* $\mathbb{B}^{\perp\perp}$ *contains* all designs *of base* $\vdash \xi$. *We note* $\mathbb{B}^{\perp\perp}$ *with* $\top$, *while* $\{\maltese\}$ *with* 0.
- *Let* $\mathbb{B} = \{(+, \xi, \varnothing)\}$, *where* $\dfrac{}{\vdash \xi}\,(+, \xi, \varnothing)$ *is the positive design of empty ramification, denoted with* $\mathbb{O}ne_\xi$. *Then* $\mathbb{B}^\perp$ *contains all designs which include the chronicle* $\{(-, \xi, \varnothing)\maltese\}$, *where* $\maltese$ *introduces* $\vdash$. *We note* $\{\mathbb{O}ne\}^{\perp\perp}$ *with* 1 *(indeed* $\vdash 1$ *is the only linear logic respective of an empty ramification, since there is no weakening without the exponentials).*

Behaviours let us interpret linear logic into ludics, giving us a tool to find the usual notions and operations of logic in this different setting. However, designs do not always correspond to correct linear logic proofs in sequent calculus, since $\maltese$ can be interpreted as a symbol for *error*, and a tool to include *incorrect* proofs: we are arbitrarily stopping a proof-search, in order to always be able to build a counter-proof for any given design. Therefore, we need a way to discriminate between designs representing correct linear logic proofs, and the incorrect ones: a behaviour interpreting a formula can thus be seen as a set of *correct* designs, which stand for the proofs of this formula. The next step is to define operations of behaviours corresponding to linear logic connectives, that can be used as a semantic for linear logic (in fact for $MALL_2$, the propositional second order linear logic, with *no exponentials*[10]).

Behaviours satisfy two remarkable completeness properties:

- **Full completeness**, which says that if $\mathcal{D} \in \mathscr{A}$, and the behaviour $\mathscr{A}$ is the interpretation of a formula A, then $\mathcal{D}$ is the interpretation of a proof of A; and
- **Internal completeness**, which says that we can decompose a behaviour which is the interpretation of a *decomposable* formula, into sub behaviours interpreting its sub formulas. For example if $\mathscr{A} \otimes \mathscr{B}$ is a behaviour interpreting $A \otimes B$ (for an operation $\otimes$ to be defined on behaviours), then $\mathscr{A}$ is the interpretation of A, and $\mathscr{B}$ is the interpretation of B.

[10] As seen in [60].

We saw that a behaviour $\mathscr{B}$ is closed under $\preceq$; hence if a design $\mathcal{D}$ is in a behaviour, all less definite designs are also in the behaviour. The dual notion is the one of *incarnation*, i.e. the *smallest* design included in $\mathcal{D}$ (with respect to the *set of chronicles*), such that it still belongs to $\mathscr{B}$. The definition is the following:

Definition 2.2.23 (incarnation)**.** *Let $\mathscr{B}$ be a behaviour, and let $\mathcal{D} \in \mathscr{B}$. The* incarnation *of $\mathcal{D}$ in $\mathscr{B}$, denoted $|\mathcal{D}|_{\mathscr{B}}$, is the smallest design $\mathcal{D}'$* included *in $\mathcal{D}$ that still belongs to $\mathscr{B}$ (it exists by the closure property already remarked). Moreover:*

1. *A design $\mathcal{D}$ is* material *in $\mathscr{B}$ if $\mathcal{D} = |\mathcal{D}|_{\mathscr{B}}$.*
2. *The incarnation of $\mathscr{B}$, denoted $|\mathscr{B}|$ is the set of the material designs of $\mathscr{B}$.*

Intuitively the notion of incarnation is strongly tied to the one of *visitable path*[11] in interaction. The incarnation of a behaviour are all the (sets of) chronicles of its designs which are actually visited by some interaction with the orthogonal. For example:

- $|\top| = Sk_{\xi}$, while $|0| = \{\maltese\}$ and $|1| = 1$.
- $|\{\mathcal{D}\}^{\perp\perp}| = \mathcal{D}^{\maltese}$, where $\mathcal{D}^{\maltese}$ is the set of designs obtained by replacing, in some chronicles of $\mathcal{D}$, the last positive rule with a $\maltese$.

Another notion we require is that of *directory* of a behaviour $\mathscr{B}$, denoted $Dir_{\mathscr{B}}$ or $\mathcal{R}ep_{\mathscr{B}}$ (from *répertoire*).

Definition 2.2.24 (Directory)**.** *Let $\mathscr{B}$ be a behaviour. The* directory *of $\mathscr{B}$, denoted $Dir_{\mathscr{B}}$ is the set of ramifications of the first actions of the material designs of $\mathscr{B}$.*

This notion will be important when defining operations on *disjoint* behaviours, in the sense of their ramifications (but still of same base). Here are a few examples:

- $Dir_{\top} = Dir_0 = \varnothing$.
- $Dir_1 = \{\varnothing\}$.
- If the first action of $\mathcal{D}$ is $(+, \xi, I)$, then $Dir_{\{\mathcal{D}\}^{\perp\perp}} = I$, if $\mathcal{D}$ is negative of first action $(-, \xi, \mathcal{N})$, then is $\{I_1, \ldots, I_n\}$ for $I_1, \ldots, I_n \in \mathcal{N}$.

Note that the directory of a *connected* behaviour is a *singleton* containing the sole first action of all its designs.

[11] By which incarnation can be defined, as in [28].

Operations on behaviours

We now present the basic operations on sets of designs which, used in combination with orthogonality, have a precise logical meaning and are able to represent the connectives of linear logic.

At first we can note the following facts about the intersection $\cap$[12]:

- Let $\mathscr{B}$ be a (negative) behaviour of base $\xi \vdash$, then $\mathscr{B} \cap \top_{\xi\vdash} = \mathscr{B}$.
- If $\mathcal{D} \trianglelefteq \mathcal{E}$, then $\{\mathcal{D}\}^{\perp\perp} \cap \{\mathcal{E}\}^{\perp\perp} = \{\mathcal{E}\}^{\perp\perp}$, since $\{\mathcal{E}\}^{\perp\perp} \subset \{\mathcal{D}\}^{\perp\perp}$.
- If $\mathscr{B}$ and $\mathscr{C}$ are *positive* behaviours, and $Dir_{\mathscr{B}} \cap Dir_{\mathscr{C}} = \varnothing$, then $\mathscr{B} \cap \mathscr{C} = 0$.
- If $\mathscr{B}$ and $\mathscr{C}$ are *negative* behaviours, and $Dir_{\mathscr{B}} \cap Dir_{\mathscr{C}} = \varnothing$, then $|\mathscr{B} \cap \mathscr{C}| = |\mathscr{B}| \cup |\mathscr{C}|$.

Definition 2.2.25. *Let $\mathscr{B}$ and $\mathscr{C}$ be two behaviours of the same base. Then $\mathscr{B}$ and $\mathscr{C}$ are* disjoint *if $Dir_{\mathscr{B}} \cap Dir_{\mathscr{C}} = \varnothing$.*

Therefore, we have that $\cap$ on behaviours is *commutative*, *associative* and have a *neutral element*, that is the design $\top$ (for negative behaviours). We can then consider the dual operation, $\cup$, which however must be obviously closed by bi-orthogonality, if we want a behaviour as result. In this case it has the same properties, with the design 0 as neutral element – that is $\top^{\perp} = (\{Sk_{\xi}\}^{\perp\perp})^{\perp}$ restricted to $\{\maltese\}$. We need a few further notions in order to build some well-defined operations that correspond to linear logic connectives – operations and notions that will return later in our interpretation, with slight modifications.

Definition 2.2.26 (Alienation)**.** *Let $\mathscr{B}$ and $\mathscr{C}$ be two behaviours of the same base. They are* alienated *if the* reunions *of their respective directories are* disjoint*, i.e. if $\bigcup Dir_{\mathscr{B}} \cap \bigcup Dir_{\mathscr{C}} = \varnothing$.*

The definition of *alienated behaviours* makes sense when the directories are not a singleton, but have multiple elements. In this case we would have *sets of sets*, since the ramifications are finite subsets of natural numbers (in the negative case finite subsets of $\mathcal{P}(\mathbb{N})$); then it makes sense to take their reunion $\bigcup$, to check the single numbers inside them, and see if there is an intersection there, i.e. one or more common numbers, instead of a whole set.

[12] Found in [16], where properties are formulated differently, and are more focused on interaction. In particular the last fact of the list is proved as a Lemma on disjoint behaviours, a property that Girard calls the "mystery of incarnation", [16], p.43.

Other than the notion of *alienated* behaviours, we need a tool which let us *de-synchronize* a behaviour, in other words change its polarity, in such a way that it will be recognizable (i.e. we can recover its previous form).

Definition 2.2.27. *Let* C *be a chronicle of base* $\vdash \xi i$ *(respectively* $\xi i \vdash$*). With* $\downarrow \mathrm{C}$ *we denote the chronicle* $(-, \xi, \{\{i\}\}).\mathrm{C}$ *of base* $\xi \vdash$ *(respectively* $(+, \xi, \{i\}).\mathrm{C}$ *of base* $\vdash \xi$*).*

Let $\mathcal{D}$ *be a design of base* $\vdash \xi i$ *(respectively* $(\xi i \vdash)$ *if negative), then* $\downarrow \mathcal{D} =$

$$\{\downarrow \mathrm{C} \mid \mathrm{C} \in \mathcal{D}\}$$

Then, let $\mathscr{B}$ *be a behaviour of base* $\vdash \xi i$ *(*$\xi i \vdash$ *if negative), then* $\downarrow \mathscr{B} =$

$$\{\downarrow \mathcal{D} \mid \mathcal{D} \in \mathscr{B}\}^{\perp\perp}.$$

The operations on behaviours corresponding to linear logic connectives are the following:

Definition 2.2.28. *Let* $\mathscr{B}_k$ *be a family of* disjoint *positive behaviours of same base, then*

$$\bigoplus \mathscr{B}_k = (\bigcup \mathscr{B}_k)^{\perp\perp}$$

Let $\mathscr{B}_k$ *be a family of* disjoint *negative behaviours of same base, then*

$$\mathop{\&}\nolimits \mathscr{B}_k = \bigcap \mathscr{G}_k$$

Note that the following facts hold:

1. $(\mathop{\&}\nolimits \mathscr{B}_k)^{\perp} = \bigoplus \mathscr{B}_k^{\perp}$, since

$$\bigcap \mathscr{B}_k = \bigcap \mathscr{B}_k^{\perp\perp} = (\bigcup \mathscr{B}_k^{\perp})^{\perp}$$

2. The property called the *mystery of incarnation* consists in the following fact:

$$|\mathscr{B} \,\&\, \mathscr{C}| = |\mathscr{B}| \times |\mathscr{C}|$$

i.e two disjoint negative behaviours do not have common material designs, therefore the & is equivalent to a Cartesian product on the incarnation.

About these operations, we can note right away that they are natural only up to a certain degree. Firstly, they are polarized, i.e. they depend on the polarity of the behaviours, and the operation $\downarrow$ might be needed to extend the

designs with an action of the wanted polarity; secondly, they require *disjoint* behaviours.

Regarding the mistery of incarnation, J.-Y. Girard talks about a *spiritual dilemma*[13]: in order to carry out these operations, we may need to adapt the *directories* of the behaviours accordingly, which is indeed a *rewriting*, also called *de-localization*[14], on the first rules of the designs of the behaviour, in order to have disjoint directories, and so no superposition of addresses (which can be seen as the *sub formulas* of the conclusion). Note that this operation would *not* change the structure of the design, intended as the *cardinality* of the ramification, and the polarity and number of rules of each chronicle; it is only a substitution of numbers in the prefixes of premises of the first rule(s). This de-localization is indeed a dilemma, since it is a step back from the abstraction of proofs that designs are: we need to adapt to their intended logical meaning.

The last operations we need to define are those for the *multiplicative* connectives. Both are based on an operation noted $\odot$, which we call *merging* for future purposes.

Definition 2.2.29. *Let $\mathcal{D}$ and $\mathcal{C}$ be designs of the same base $\vdash \xi$. We define $\mathcal{D} \odot \mathcal{C}$ by cases:*

- *if $\mathcal{D}$ or $\mathcal{C}$ is $\maltese$, then $\mathcal{D} \odot \mathcal{C} = \maltese$.*
- *Otherwise, let $(+, \xi, I)$ and $(+, \xi, J)$ be the first action of, respectively, $\mathcal{D}$ and $\mathcal{C}$. Then*
  - *if $I \cap J = \varnothing$, we have that:*

$$\mathcal{D} \odot \mathcal{C} = \{(+, \xi, I \cup J).\mathrm{C} \mid (+, \xi, I)\mathrm{C} \in \mathcal{D} \text{ or } (+, \xi, J)\mathrm{C} \in \mathcal{C}\}$$

  - *otherwise $\mathcal{D} \odot \mathcal{C} = \maltese$.*

$\odot$ is commutative, associative and its neutral element is the design $\mathbb{O}ne = (+, \xi, \varnothing)$. We will use this operation along with a *de-localization* defined as a renaming on designs, which can be applied to any of the two designs, members of $\odot$, to make them *alienated*. For the multiplicative connectives, we define the operations on alienated designs, instead of simply disjoint.

Definition 2.2.30. *Let $\mathscr{B}$ and $\mathscr{E}$ be* positive alienated *behaviours (thus of the same base). Then*

$$\mathscr{B} \bigotimes \mathscr{E} = \{\mathcal{D} \odot \mathcal{C} \mid \mathcal{D} \in \mathscr{B}, \mathcal{C} \in \mathscr{E}\}^{\perp\perp}$$

[13] As reported in [60] and in the original [38] by Girard.
[14] [60] p.33.

Let now $\mathscr{B}$ and $\mathscr{E}$ be negative alienated *behaviours. Then*

$$\mathscr{B} \parr \mathscr{E} = (\mathscr{E}^{\perp} \otimes \mathscr{B}^{\perp})^{\perp}$$

To end this section we mention the completeness theorems, referring to [38] for the proofs and definitions in standard ludics, and to [6] for the ones in ludics formulated in game-semantics. Their statements are the following:

- **Internal additive completeness**: let $K \neq \varnothing$, then $\bigoplus_{k\in K} \mathscr{G}_k = \bigcup_{k\in J} \mathscr{G}_k$, i.e. there is no need for the bi-orthogonal closure.

- **Internal multiplicative completeness**: let $\mathscr{G}$ and $\mathscr{H}$ be alienated behaviours, then $\mathscr{G} \otimes \mathscr{H} = \{\mathcal{D} \odot \mathcal{C} \mid \mathcal{D} \in \mathscr{G}, \mathcal{C} \in \mathscr{G}\}$; again, the bi-orthogonal closure is superfluous.

- **Full completeness**: let $\Gamma \vdash \Delta$ be a $MALL_2$ sequent Π^1, and let $\mathcal{D}$ be a design of the behaviour interpreting the sequent $\Gamma \vdash \Delta$. Then, it exists a proof π of $\Gamma \vdash \Delta$ such that $\mathcal{D} = \pi^*$, where π^* is a design interpreting the proof π.

2.2.5 Giving interaction a direction

Using an operation called *pruning*, inspired from a work of C. Fouqueré and M. Quatrini [30] – where is presented in different terms and more developed – it is rather easy to force a *direction* on interaction, when considering a *set* of designs or a behaviour.

The pruning we use is essentially the erasing of a final segment of a branch in a design, that gets meaning once this *pruned design* is considered together with other designs with a common base, and with carefully placed ✠ on these latters. This technique is a simple operation on designs with the same base that, when put together in a set, allows interaction to not diverge only if it visit branches in a specific order (thus giving it a *direction*). This is preserved also once we close the set by bi-orthogonality to obtain a *behaviour*, that is then called **directed behaviour**.

The pruning works in a rather simple way; consider the following design:

$$\dfrac{\ldots \quad \dfrac{\dfrac{\vdots}{\xi.1.1 \vdash}\,(-,\xi.1.1,\mathcal{N})}{\vdash \xi.1}\,(+,\xi.1,\{1\}) \quad \ldots}{\xi \vdash}$$

a pruning on the action $(-,\xi.1.1,\mathcal{N})$ will look like this

$$\dfrac{\ldots \quad \dfrac{\dfrac{}{\xi.1.1 \vdash}\, p}{\vdash \xi.1}\,(+,\xi.1,\{1\}) \quad \ldots}{\xi \vdash}$$

Pruning on negative actions is always a possible operation, since the last action of a branch with no extension must be positive.

When putting together a design and one of its pruned versions, the common orthogonals will be the designs with which interaction respects the order forced by the pruning: i.e. the pruned branch will always be visited last. We improperly call pruned designs also **directed designs**.

Let us take back the interaction example, and then *modify with the pruning* the positive design, to see what happens:

Example 2.2.31. $\mathcal{D}$:
$$\dfrac{\dfrac{\dfrac{\dfrac{}{\xi.1.1.1 \vdash}}{\vdash \xi.1.1}\,(+,\xi.1.1,\{1\})}{\xi.1 \vdash}\,(-,\xi.1,\{\{1\}\}) \qquad \dfrac{\dfrac{\dfrac{}{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}\,(+,\xi.2.1,\{1\})}{\xi.2 \vdash}\,(-,\xi.2,\{\{1\}\})}{\vdash \xi}\,(+,\xi,\{1,2\})$$

$\mathcal{C}$:
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \xi.2.1.1, \xi.1.1.1}\,\Lightcross}{\xi.2.1 \vdash \xi.1.1.1}\,(-,\xi.2.1,\{\{1\}\})}{\vdash \xi.1.1.1, \xi.2}\,(+,\xi.2,\{1\})}{\xi.1.1 \vdash \xi.2}\,(-,\xi.1.1,\{\{1\}\})}{\vdash \xi.1, \xi.2}\,(+,\xi.1,\{1\})}{\xi \vdash}\,(-,\xi,\{\{1,2\}\})$$

$\mathcal{E}$:
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \xi.1.1.1, \xi.2.1.1}\,\Lightcross}{\xi.1.1 \vdash \xi.2.1.1}\,(-,\xi.1.1,\{\{1\}\})}{\vdash \xi.2.1.1, \xi.2}\,(+,\xi.1,\{1\})}{\xi.2.1 \vdash \xi.2}\,(-,\xi.2.1,\{\{1\}\})}{\vdash \xi.2, \xi.1}\,(+,\xi.2,\{1\})}{\xi \vdash}\,(-,\xi,\{\{1,2\}\})$$

Both $\mathcal{C}$ and $\mathcal{E}$ are orthogonal to $\mathcal{D}$. $\mathcal{C}$ visits the $\xi.1$ branch first, and the $\xi.2$ one later; $\mathcal{E}$ does the opposite instead. However

$$\mathcal{D}^*:\ \dfrac{\dfrac{}{\xi.1 \vdash}\, p \qquad \dfrac{\dfrac{}{\vdash \xi.2.1}\,\Lightcross}{\xi.2 \vdash}\,(-,\xi.2,\{\{1\}\})}{\vdash \xi}\,(+,\xi,\{1,2\})$$

where p denotes a pruning *on the branch starting with $\xi.1 \vdash$, is orthogonal only to $\mathcal{E}$: interaction cannot continue on $\xi.1$, since it is not introduced by a rule anymore, but can only pass through $\xi.2 \vdash$. In conclusion $\mathcal{E} \in \{\mathcal{D}, \mathcal{D}^*\}^\perp$, since it visits the $\xi.2$ branch first, while $\mathcal{C} \notin \{\mathcal{D}, \mathcal{D}^*\}^\perp$; in this way we have forced interaction to respect the order $\xi.2 < \xi.1$.*

Example 2.2.32. *If we take the behaviour* $\mathbf{1} = \{\mathbb{O}ne_\xi, Dai^+_\xi\}$*, we can consider* $\uparrow\downarrow 1$*. Its designs are:*

$$\dfrac{\dfrac{\dfrac{}{\vdash \xi.1.1}(+,\xi,\varnothing)}{\xi.1 \vdash}}{\vdash \xi} \quad ; \quad \dfrac{\dfrac{\dfrac{}{\vdash \xi.1.1}\maltese}{\xi.1 \vdash}}{\vdash \xi}$$

Another instance of $\uparrow\downarrow 1$*, with a different ramification of the first rule is:*

$$\dfrac{\dfrac{\dfrac{}{\vdash \xi.2.1}(+,\xi,\varnothing)}{\xi.2 \vdash}}{\vdash \xi} \quad ; \quad \dfrac{\dfrac{\dfrac{}{\vdash \xi.2.1}\maltese}{\xi.2 \vdash}}{\vdash \xi}$$

we can denote this different instance with $\uparrow\downarrow 1^2$*.*

Now, if we consider the tensor *of these two behaviours* $\uparrow\downarrow 1 \otimes \uparrow\downarrow 1$ *(definition 2.2.30) we obtain:*

$$\dfrac{\dfrac{\dfrac{}{\vdash \xi.1.1}}{\xi.1 \vdash} \quad \dfrac{\dfrac{}{\vdash \xi.2.1}}{\xi.2 \vdash}}{\vdash \xi} \quad ;$$

$$\dfrac{\dfrac{\dfrac{}{\vdash \xi.1.1}\maltese}{\xi.1 \vdash} \quad \dfrac{\dfrac{}{\vdash \xi.2.1}}{\xi.2 \vdash}}{\vdash \xi} \quad ; \quad \dfrac{\dfrac{\dfrac{}{\vdash \xi.1.1}}{\xi.1 \vdash} \quad \dfrac{\dfrac{}{\vdash \xi.2.1}\maltese}{\xi.2 \vdash}}{\vdash \xi} \quad ;$$

$$\dfrac{\dfrac{\dfrac{}{\vdash \xi.1.1}\maltese}{\xi.1 \vdash} \quad \dfrac{\dfrac{}{\vdash \xi.2.1}\maltese}{\xi.2 \vdash}}{\vdash \xi}$$

The resulting set of design let interaction visit the two branches in no particular order; it just has to visit both to get to $\maltese$ *in all designs of the set. If we wanted to give precedence instead to the designs of* $\uparrow\downarrow 1$ *obtaining the* **directed behaviour** $\uparrow\downarrow 1 <\uparrow\downarrow 1^2$*, where* $<$ *is a* directed tensor*, we can modify the second one in the following way:*

$$\dfrac{\dfrac{\dfrac{}{\vdash \xi.1.1}\maltese}{\xi.1 \vdash} \quad \dfrac{\dfrac{}{\vdash \xi.2.1}}{\xi.2 \vdash}}{\vdash \xi} \quad \rightsquigarrow \quad \dfrac{\dfrac{\dfrac{}{\vdash \xi.1.1}\maltese}{\xi.1 \vdash} \quad \dfrac{}{\xi.2 \vdash}\,p}{\vdash \xi}$$

by pruning *the rule on* $\xi.2$*. This single design in the set is enough to force interaction to visit the branches starting with* $\xi.1$ *first.*

Remark 2.2.33. *Note that the pruning (of designs) makes behaviours* irregular*. Informally, a behaviour is* regular *if* mixing *two interaction paths (i.e. visiting actions by alternating, at each step, the two paths) is still an interaction path on the behaviour – that means there is an orthogonal design determining that path during interaction*[15]*. Irregularity exactly expresses the fact that the visitable actions of a behaviour cannot be visited in any order, but only some specific paths are allowed.*

[15]The formal definition is that a behaviour is **regular** if interaction paths are *closed* under **shuffle**, a well defined version of the *mixing* we mentioned, found in [30].

3. Process calculi

Programs based on concurrent and parallel computation, where many agents interact with each other, are widely used in various settings, thus increasing the need of an abstract logical framework able to model and describe such objects. The core notion is the one of *communication* and information exchange, more than the one of computation step, which brings us out of the purely functional world.

In a communication-based concurrent computation there are many agents, called *processes*, acting at the same time, which need to communicate – in the form of sending and receiving information – to enable computation. To give a semantic of concurrency, and an abstract setting to treat concurrent computation, we will introduce *process calculi* (or *algebras* as in [46]). We can start by defining a *labeled transition system* (LTS), an abstract framework where terms are reduced by performing a set of *actions*; then, by adding communication between actions in the form of *synchronization* between them, we can define the *Calculus of Concurrent Systems*, in short CCS, first introduced by Milner in [55], as the prototypical process algebra. Then we shortly present an extension of CCS, and the more powerful π-calculus, based on *name passing*.

3.1 LTS

Definition 3.1.1. *A labeled transition system (lts) is a ternary relation subset of $S \times Act \times S$, for a set S of states, and Act of actions. For $s, s' \in S$ and $\alpha \in Act$ it is denoted*

$$s \to^{\alpha} s'.$$

We can give *lts* a syntax that will be later extended to define processes. An *lts* is defined thus

$$P, Q := 1 \mid \alpha.P \mid P + Q \quad \alpha \in Act$$

1 is the empty *lts*[1], $\alpha.P$ is the *prefix* operation (we say that P is *prefixed* by α) and $P+Q$ is the non-deterministic choice, or *sum*, an operation where once one of the two terms is chosen, the other is discarded. With this syntax, the set S of states is made by *actions* put together by the prefix operation and non deterministic sum. The *lts* associated to this syntax is made of transitions $\rightarrow^{\alpha}$ depending on the action α which is prefix of all other actions (or member of a sum).

Definition 3.1.2. *The set of* traces *of a term* P *is*

$$tr(P) = \{\alpha_1, \ldots, \alpha_n \mid P \rightarrow^{\alpha_1} \cdots \rightarrow^{\alpha^n}\}.$$

P and Q are traces equivalent *if* $tr(P) = tr(Q)$.

With these basics notions we cannot still properly model concurrent computation, nor communication between terms. The set of traces describes the internal transitions that happen inside a term – its internal behaviour – but cannot take into account an *environment* with which it may interact via communication. To describe processes and communication we need to add a notion of *dual action* and *synchronization* between duals, which happens when putting terms in *parallel composition*.

To prepare for this extended setting, we define a more refined notion than trace equivalence to treat behavioral equivalence, the one of *bisimulation*:

Definition 3.1.3. *Let* $\rightarrow \subseteq S \times Act \times S$ *be a* labelled transition system, *and* $\mathbb{R}$ *a binary relation on the set of states* S.

$\mathbb{R}$ *is a* simulation *if from* $P\mathbb{R}Q$ *and* $P \rightarrow^{\alpha} P'$, *for an action* α, *it follows that* $\exists Q'$ *such that* $Q \rightarrow^{\alpha} Q'$ *and* $P'\mathbb{R}Q'$.

$\mathbb{R}$ *is a* bisimulation *if it holds the same for* Q, *i.e. if* $P\mathbb{R}Q$ *and* $Q \rightarrow^{\alpha} Q'$ *implies that* $\exists P'$ *such that* $P \rightarrow^{\alpha} P'$ *and* $P'\mathbb{R}Q'$.

The following facts hold:

- The *empty* and *identity* relations are bisimulations (straightforward from definition).
- The class of bisimulations is closed under inverse, composition, and arbitrary unions.
- The union of all bisimulations is thus a bisimulation, denoted with $\sim$ (called *bisimilarity*).
- An intersection of bisimulations is, in general, *not* a bisimulation.[2]

[1] Note that the standard notation is 0; we will explain later our reasons to use 1 instead.
[2] For reference, see [4].

As we are more concerned with *communication* rather than computation, we can imagine that when two processes interact, there are some *internal computations* that produce an output, which then is sent to the other process during synchronization of actions (the moment where communication happens); these internal computations, however, are *not observable*.

We can thus add to an *lts* an *internal action* τ, which cannot synchronize with the environment. In this sense the term $a.\tau.b.1$ is equivalent to $a.b.1$; however the τ action is not completely irrelevant with respect to the behaviour of a process. For example, if we have $a + b$ as a non-deterministic choice, and we add a τ action before each member, becoming $\tau.a + \tau.b$, it becomes an *internal choice*, since a transition on τ is independent from the environment.

In [46] the τ is called *silent action*, and it is *not observable* from an external viewpoint. In general, it is removable only if redundant, i.e. if doing a τ transition does not affect the possible evolutions – the intermediate steps and resulting states – of a term. Indeed, in these cases the τ action can be considered invisible from both a syntactic and semantic point of view. However, in the presence of a choice, for example $a.P + \tau.1$ (which is not equivalent to $a.P$) or $\tau.a + \tau.b$, it cannot be erased, since we cannot know which option can be kept.

By adding the internal action τ we get a *weak lts*:

Definition 3.1.4. *A* weak lts *is an lts* $\Rightarrow \subseteq S \times Act \cup \{\tau\} \times S$*, where* $\tau \notin Act$ *is a distinct internal action. A transition is defined thus:*
$\Rightarrow^{\alpha} = (\rightarrow^{\tau})^*$ *if* $\alpha = \tau$*;*
$(\rightarrow^{\tau})^* \rightarrow^{\alpha} (\rightarrow^{\tau})^*$ *otherwise.*
where * *is a finite sequence of transitions.*

This notion of *weak transition system* is naturally extended to bisimulations. $\mathbb{R}$ is a *weak bisimulation* if it is a bisimulation with respect to a *weak transition system* $\Rightarrow$, where two transition steps between bisimilar terms can be matched without counting τ transitions.

3.2 Milner's Calculus of Communicating Systems

By adding *parallelism*, and enabling *synchronization* between actions to an *lts* we have all the tools needed to represent concurrent computation. The process algebra which we obtain as result is the *Calculus of Communicating Systems*, in short *CCS*, first introduced by Milner in [55].

The first step in building CCS is the notion of *dual action*. The set of actions Act is extended in this way:

$$\mathcal{A} = \{a, \overline{a} \mid a \in Act\}.$$

Where $\overline{a}$ is the dual of the action a. The dual is *involutive*, i.e. it holds $\overline{\overline{a}} = a$. To obtain the full calculus we need to add the *hidden or private action* operator, denoted ν, the *parallel composition* between terms, denoted $P \mid Q$, and the *recursive* operator, which has a variant as an *exponential* operator, denoted as the linear logic exponential !. The recursive operator is of the form $A(\vec{a}) = P$, in its more general version, with $a \in \mathcal{A}$, is such that the same construct $A(\vec{b})$, defined on a different sequence of actions, can occur in P – here lies the recursion. The exponential version $!a.P$ instead *duplicates* the process under !.

Definition 3.2.1 (CCS)**.** *Terms of the* Calculus of Communicating Systems *are called* **processes**, *denoted with* $P, Q, R, S, \ldots$, *and are built by the following grammar:*

$$P, Q := 1 \mid a.P \mid \overline{a}.P \mid (P \mid Q) \mid (P + Q) \mid \nu a(P) \mid A(\vec{a})$$

where $a \in \mathcal{A}$, *with* $\mathcal{A} = \{a, b, c, \ldots\}$ *a denumerable set of* actions, *called* channel names. *For each* $a \in \mathcal{A}$, *exists its* **dual** $\overline{a}$, *such that* $\overline{\overline{a}} = a$. *More in details:*

- 1 *is the* **empty process**, *from which any process is built.*
- $a.P$ *and* $\overline{a}.P$ *are, respectively, the* **positive action prefix** *and the* **negative action prefix**.
- $P \mid Q$ *is the* **parallel composition** *of* P *and* Q.
- $P + Q$ *is the* **non deterministic choice** *or* **sum** *of* P *and* Q.
- νa *is the* **private name operator** *or* **hidden action operator**, *that makes all channels named* a *inside its scope* hidden *from the environment.*
- $A(\vec{a})$ *is the* **recursive operator** *on the parameters* $\vec{a} \in \mathcal{A}$, *that allows recursive definitions of processes.*

We assume that any process is built starting with 1, the empty process, who has **no dual**. While the original syntax use 0, we instead use 1 because the empty process is not only the **neutral element** of the sum +, but also of

the parallel composition $P \mid Q$; moreover, our *interpretation in ludics* shows a clear *multiplicative* behaviour and a correspondence with the linear logic unit 1. The hidden action operator ν *binds* a channel name a, making it *private* for future communication purposes: only a dual channel name inside the scope of the same ν may communicate with it. If a channel name is not in the scope of a ν, then its *free*. With $fc(P)$ we denote the set of free channel names of P, and with $bc(P)$ the set of bound channel names.

The core of CCS lies in its communication-based reduction. An *lts* can be defined for CCS, and then can be refined by considering a particular case of reduction, called *execution*, where we can *synchronize* dual channels. The relevant notions are hence *parallelism* and *synchronization*: we can consider the names $a, b, \ldots$ as *channels* on which parallel processes synchronize in order to communicate. A synchronization can happen when two *dual channels* are *the most external prefixes* of two processes in parallel composition – i.e. are minimal channels of each process, w.r.t. the prefix order – as in $a.P \mid \overline{a}.Q$; once the synchronization is performed, the resulting process will be $P \mid Q$, *consuming* the used channels (execution is indeed *resource sensitive*). In the case of CCS, there is no data sent or received, what matters is where and when synchronizations happen, and thus how a process is transformed during communication with the environment, and which intermediate forms it obtains, that determine all the possible transformations of a process, depending on the environment.

The most basic interpretation of $P = a.P'$ is therefore "P is ready to synchronize on the channel a". Communication may happen only on channels ready for synchronization, or equivalently available for execution, and only when there is a dual channel in another term in parallel composition also ready for synchronization. Instead of defining an *lts* for CCS (which can be found in [4], p.62), since we would not make much use of it, we will give directly some examples and then define execution as the proper reduction relation for normalization of processes.

3.2.1 Execution as a reduction semantic

An *lts* for CCS let us erase a channel as a transition, with some restrictions for the ν and +: in the first case the name in question must not be private – i.e. under the scope of ν – and in the second we must discard the member of the sum not concerned with the transition. As a few examples, let $P = a.1 \mid \overline{a}.1$, then

$$P \rightarrow_a 1 \mid \overline{a}.1 \rightarrow_{\overline{a}} 1 \mid 1$$

has the same result than

$$P \to_{\overline{a}} a.1 \mid 1 \to_a 1 \mid 1$$

and both pairs $(a, \overline{a})$ or $(\overline{a}, a)$ can be denoted as an internal transition of P

$$P \to_\tau 1 \mid 1.$$

The transitions we want to allow are only *pairs of dual channel names* ready for synchronization, with no prefix blocking them (we often say *external channels*). However, we do not care if the channels are contiguous or far away, and moreover since the empty process 1 cannot synchronize we want it to be a neutral element with respect to parallel composition. We will therefore consider processes up to the following congruence relation:

Definition 3.2.2 (Structural congruence)**.** *Structural congruence is the smallest congruence relation, denoted* $\equiv$*, such that parallel composition and* $+$ *are commutative and associative,* $+$ *also idempotent, and* 1 *is the neutral element of parallel composition and sum; i.e.* $P + P \mid R + Q \equiv Q + R \mid P$*;* $P \mid (Q \mid R) \equiv (P \mid Q \,) \mid R$*;* $P \mid 1 \equiv P$*;* $P + 1 \equiv P$*; and such that*

$$\nu a(P \mid Q) \equiv \nu a(P) \mid Q \quad \text{if } a \notin fc(Q)$$

$$\nu a(P) \mid \nu b(Q) \equiv \nu a\ \nu b(P \mid Q) \equiv \nu b\ \nu a(P \mid Q) \quad \text{if } a \notin fc(Q), b \notin fc(P)$$

$$A(\vec{b}) \equiv [\vec{b}/\vec{a}]P \quad \text{if } A(\vec{a}) = P.$$

Our choice of using 1 to denote the empty process also comes from our interpretation of parallel composition, that shows to be a *multiplicative* operation, rather than additive: its correspective in ludics will be a modified version of $\otimes$, which we saw being the representation of the linear logic tensor $\otimes$, a *multiplicative connective*.

Before defining execution we need to add some notation to the syntax. Execution is, in general, a *non deterministic* and *non confluent* operation, and that is due to the fact that channel names are *not unique*. However, it is still a *resource-sensitive operation*. We thus want a way to discriminate different *occurrences* of the same channel name; to this end we are adding to the syntax a set $Loc = \{l, m, n, o, \ldots\}$ of **locations**: *unique labels* for each occurrence of a channel name.

With Loc_P we note the set of locations labeling channels of a process P. Therefore a process will be of the following form:

$$P = a^l.Q \mid \overline{a}^m.R \mid \overline{b}^o.S.$$

The *execution* relation is the core of the calculus, and tells us how processes communicate and transform. It happens between a pair of *dual* channels occurrences, thus of same name and opposite polarity. In this case we say that the two channels **synchronize**, and that there is *communication* between them. Two channels are *ready for synchronization* only if they have *no prefix*, i.e. they are the first channels of two processes in parallel composition (the last added by action prefix), and thus *minimal* with respect to the *prefix*-induced partial order: this makes execution a *synchronous* procedure[3].

Definition 3.2.3 (Execution)**.** *Execution is the following relation on CCS processes, up to structural congruence:*

$$\nu a\ (\overline{a}^m.P + P' \mid a^l.Q + Q') \mid R \rightarrow_{\{(m,l)\}} \nu a\ (P \mid Q\ ) \mid R$$

denoted by $\rightarrow$ *along with as label a pair of locations naming dual channel names.* $\rightarrow^*$ *is the transitive reflexive closure of* $\rightarrow$*, and is denoted in the following way:* $P \rightarrow^*_{\emptyset} P$*, and if* $P \rightarrow^*_{\vec{e}} Q \rightarrow^*_{\vec{i}} R$ *then* $P \rightarrow^*_{\vec{e}\cup\vec{i}} R$*, where* $\vec{e}$ *and* $\vec{i}$ *are sets of disjoint pairs of dual channels.*

An execution sequence *is said* **maximal** *if there is no synchronizable pair* $(b^m, \overline{b}^p)$ *that can* extend *the sequence.*

Regarding the newly introduced *locations*, with $subj_P(l)$ we denote the channel name labeled by l, and with $pol_P(l)$ the polarity of said channel. We can mirror the prefix order via a partial order on locations, and then define some relations on synchronizable pairs of dual channels using the partial order on their locations as support. These relations are essentially the causal and conflict relations of *event structures*[4], a model of *CCS*-like calculi [68], that we will define later (definition 3.2.9).

Definition 3.2.4 (Location order)**.** *Let* P *be a CCS process and* $l, m \in Loc_P$*. The location order on* P*, denoted by* $<_P$*, is the partial order on* Loc_P *such that* $l <_P m$ *if* l *labels a prefix of the channel labeled by* m*. So, if* $P = x^l.Q$*,* $l <_P m$ *for every* $m \in Loc_Q$*.*

A location stands for a particular occurrence of a channel name in a process; being a 1-to-1 correspondence, we do not distinguish between a channel occurrence and the location which labels it. Note that the sets Loc_P, $subj_P(l)$, $pol_P(l)$, with $l \in Loc_P$ and the order $\leq_P$ completely characterize a process up to structural congruence (it characterize its congruence class).

Instead of *traces* as in a *lts*, for execution we have *pairings*:

[3] Execution can, in general, be modified in order to allow communication independently by the prefix order. This makes the calculus *asynchronous* instead.

[4] In the original version: [55].

Definition 3.2.5 (Pairing)**.** *A* pairing *of a CCS process P is a subset $c \subseteq C$ of a partial involution C over Loc_P, such that for all l, $subj_P(l) = subj_P(c(l))$, $pol_P(l) = -pol_P(c(l))$ (each channel is paired with a dual one, thus forming an executable pair).*

A pairing is consistent *if $\equiv_c <_P \equiv_c$ is acyclic, where $\equiv_c$ is the smallest equivalence relation containing c, and $dom(c)$ is downward closed with respect to $<_P$.*

A pairing is maximal *if there is no consistent extension of $dom(C)$, and is* complete *if $\forall l \in Loc_P$, $l \in dom(c)$ or $l \in img(c)$.*

Each consistent pairing correspond to the indexing of a possible execution on P (actually many indexing, up to permutation of the pairs that respect $<_P$).

3.2.2 Non determinism and non-confluence

The execution relation is non deterministic at its core, due to the non uniqueness of channel names, and the presence of the sum +, which also induces non-confluence. For instance in

$$\nu a\ (\overline{a}.P \mid \overline{a}.Q \mid a)$$

we have two $\overline{a}$ channels ready for synchronization, but only one a available as a counterpart. The fact that in each state of a process there are different possible executions excluding each other naturally generates forking paths and multiple *normal forms*, i.e. states where no further execution is possible.

This is always the case if there is a sum $Q_1 + Q_2 \mid \ldots$ in a process P such that Q_1 and Q_2 are unique in P, and $Q_1 \neq Q_2$.

Example 3.2.6. *Let*

$$P = \overline{a}^m.b^h.S \mid a^l.c^k.Q \mid a^n.\overline{b}^i.R$$

there are two possible execution steps:

- $P \rightarrow_{(m,l)} b^h.S \mid c^k.Q \mid a^n.\overline{b}^i.R$
- $P \rightarrow_{(m,n)} b^h.S \mid a^l.c^k.Q \mid \overline{b}^i.R$

(we use circular brackets when only one pair is used). Each execution makes different channels ready for synchronization: in the second case $(b^h, \overline{b}^i)$ is

a new possible execution, while in the first $\overline{b}^i$ *is blocked; moreover, the first execution brings to a normal form, while the second admits further reductions. Therefore, we have different* execution sequences *and* normal forms.

With the sum it is clear that if

$$a^l.P + Q \mid \overline{a}^m.P' + Q' \rightarrow_{(l,m)} P \mid P'$$

then we can no longer access to Q, Q'*, and thus once a choice has been made we have a permanent fork in the execution path.*

In order to properly characterize execution we want to be able to directly deal with synchronizable pairs, and define relations on them that can tell us the necessary and sufficient conditions to perform an execution step on any given pair. To this end we introduce the following notion:

Definition 3.2.7 (synchronizations)**.** *A* synchronization *of a CCS process* P *is* any pair of dual channel names $(a^l, \overline{a}^m)$ *such that* $a^l, \overline{a}^m \in P$*. The set of synchronizations of* P *is denoted* $\mathcal{S}_P$*, its elements* $i, j, u, v, \ldots$*.*

- *We say that a channel name* a *or a location* l *belongs to a synchronization* u*, denoted* $a \in u$ *or* $l \in u$*, if* a^l *belongs to the synchronization named* u*.*

- *Synchronizations order is induced by prefix order on channel occurrences: for* $i, j \in \mathcal{S}_P$ *we say* $i \lesssim_{\mathcal{S}_P} j$ *if* $\exists l \in i$ *and* $m \in j$ *such that* $l <_P m$*, with* $l, m \in Loc_P$*.*

- *A synchronization is* **principal** *if* both its locations *are* **minimal** *w.r.t* $<_P$*.*

Note that the synchronization order admits cycles, since it may be that $i \lesssim_{\mathcal{S}_P} j$ and $j \lesssim_{\mathcal{S}_P} i$, if a location of i is smaller than a location of j and vice-versa: indeed, this is the case with *deadlocks* in the process. For instance

$$P = a^1.b^2 \mid \overline{b}^3.\overline{a}^4$$

has two synchronizations $u = (a^1, \overline{a}^4)$ and $v = (b^2, \overline{b}^3)$, and is *deadlocked*. We have that a^1 is blocking b^2 and $\overline{b}^3$ is blocking $\overline{a}^4$; in this case it holds that $u \lesssim_{\mathcal{S}_P} v$ and $v \lesssim_{\mathcal{S}_P} u$, since each synchronization has a channel blocked by one belonging to the other.

Definition 3.2.8 (XOR)**.** *Let P be a CCS process, and $i, j \in \mathcal{S}_P$. Then, $i \cap j = \varnothing$ if $\forall l \in i, \forall m \in j,\ l \neq m$. with $l, m \in Loc_P$. If $i \cap j \neq \varnothing$, then i and j have a channel name occurrence in common. Then,*

$$\mathcal{X}_P = \{(i, j) \mid i, j \in \mathcal{S}_P, i \cap j \neq \varnothing\}$$

we call $\mathcal{X}_P$ the xor, or conflict*, relation of a process P, and (i, j) a xor clause, or* conflict pair*. For every $i \in \mathcal{S}_P$, let $xor(i) = \{j \in \mathcal{S}_P \mid (i, j) \in \mathcal{X}_P\}$.*

The *xor* relation tells us when two synchronizations can't be in the same execution sequence. If two synchronizations have a location, so a channel occurrence, in common then one excludes the other from the same sequence, since channels are *consumed* during execution. The relation can obviously be applied also to channels of two processes in a +.

Event structures

Event structures, introduced first in [59] by Nielsen, Plotkin and Winskel, are a causal model of concurrency, part of the *true concurrency* models. They are closely tied to CCS, as Winskel gave an interpretation of CCS into event structures in [68], showing that event structures *are a model for CCS-like calculi*. Since we will also give some hints of the relation between event structures and our work later, and form a connection with a specific paper on *confusion-free* event structures and linear strategies ([27], in chapter 7), we briefly introduce them here.

Definition 3.2.9 (Event structure)**.** *An* **event structure** *is a triple $\langle \mathscr{E}, \leq, \smile \rangle$ such that:*

- *$\langle \mathscr{E}, \leq \rangle$ is a* partially ordered set*, where elements of $\mathscr{E}$, denoted $e, e', e_1, \ldots$, are called* events*, $\mathscr{E}$ is at most countable, and $\leq$ is called* **causality relation**.
- *The* downward closure *of $S \subseteq \mathscr{E}$ is $\lceil S \rceil = \{e' : e' \leq e, e \in S\}$. If S is a singleton $\{e\}$, it is denoted $\lceil e \rceil$.*
- *$\smile$ is an* irreflexive symmetric relation*, called* **conflict relation***, such that for every $e_1, e_2, e_3 \in \mathscr{E}$ it holds*

  $$(e_1 \leq e_2 \ \wedge \ e_1 \smile e_3) \rightarrow e_2 \smile e_3$$

  In this case we say that the conflict $e_2 \smile e_3$ is inherited *from $e_1 \smile e_3$; if a conflict is not inherited it is* immediate*, denoted $\smile_\mu$.*

- *Causal order and conflict are* mutually exclusive.
- *Two events that are nor in a* causal *nor* conflict *relation are said* **concurrent**.
- *The set* $\lceil e) = \lceil e \rceil \setminus \{e\}$ *is the* enabling set *of* e.

The particular case of event structure we will mention are *confusion-free* event structures. To define them we need the following notion:

Definition 3.2.10 (Cell)**.** *A* cell c *is a* maximal set of events*, with respect to* $\leq$*, that are pairwise in* immediate conflict*, and have the* same predecessors*, i.e. the same* enabling set*. Therefore*

$$e, e' \in c \to e \smile_\mu e' \text{ and } \lceil e) = \lceil e')$$

Definition 3.2.11 (Confusion free)**.** $\langle \mathscr{E}, \leq \rangle$ *is* **confusion free** *if the following hold:*

- Immediate conflict is transitive:

$$\forall e, e', e'' \in \mathscr{E}, (e \smile_\mu e' \wedge e' \smile_\mu e'') \to e \smile_\mu e''$$

- *Any two events in* immediate conflict *have the same predecessors, i.e. any two events in immediate conflict* form a cell.

Therefore confusion-free event structures are event structures where *choices*, in the form of *conflicts* between events, are localized to *cells*.

For its use and meaning, we can see the *xor* relation on CCS processes as the conflict relation of *event structures*, and pairs of dual channel names as events that can *synchronize*. Note that the conflict relation is *hereditary* in event structures, while ours are only *local*. With the same parallelism in mind, we can call **concurrent** two *synchronizations* $u_1, u_2 \in P$ such that are neither in a *xor* relation, nor in an order relation, i.e. such that $(u_1, u_1) \notin \mathcal{X}_P$ and neither $u_1 \preccurlyeq_{\mathcal{S}_P} u_2$ nor $u_2 \preccurlyeq_{\mathcal{S}_P} u_1$.

Value-passing CCS

We give some hints of an extension of CCS where data, as *values*, can be sent and received during synchronization, which brings CCS a step closer to the more complex π−calculus. Values, however, are distinct from *channel names*, and are considered atomic objects as booleans or integers. We have

thus a countable number of values $Val = \{v_0, v_1, \ldots\}$. The set $\mathcal{A}$ of channels is modified in this way:

$$\mathcal{A} = \{a(v), \overline{a}(v) \mid a \in \mathcal{A}, v \in Val\} \cup \{\tau\}$$

The last ingredient are variables $id ::= x \mid y \mid \ldots$ ranging over values, which together with these latters gives us a generic definition of *term*:

$$t ::= v \mid id$$

with $v \in Val$.

We can consider $a(id)$ as an *input* action, which receives a value v on the channel a, and $\overline{a}(t)$ as an *output* action sending a value v on a. We also want to be able to test values for equality, and act accordingly, thus we need to add to the syntax $[v = v']P, Q$, which is the process that runs P if $v = v'$, and Q otherwise.

The execution relation works as in CCS, but now the output channel sends a value, and the receiving one (input) substitutes a variable in its process with the data received:

$$\overline{a}(v).P \mid a(x).Q \rightarrow_{\overline{a},a} P \mid [{}^{v}/_{x}]Q$$

where the value v has substituted x in Q. For the equality clause we have:

$$[v = v]P, Q \rightarrow P \qquad [v = v']P, Q \rightarrow Q \quad \text{if} \quad v \neq v'.$$

While in this way data-passing can be represented, the $\pi-$calculus is a more general and expressive setting since the values passed are channel names themselves, which may enable further communication. A process so can change its *internal* form during communication.

3.3 π-calculus

Despite not being the focus of our work, due to its large use and importance, being a more complex and expressive process algebra[5], we present the π-calculus as an extension of value-passing CCS.

As we anticipated channels in the π-calculus exchange values which are channel names themselves, thus enabling new synchronizations, and modifying the process internally at each step. It is widely used as a system to model

[5] As denoted in [50], extensions of the π-calculus have almost the same expressive power than actual concurrent programming languages.

concurrent and distributed programs (used for web services for example), by using type systems that check for communication safety and observance of communication protocols, as done with *session types* [50, 64].

We can see the π-calculus as an extension of value-passing CCS, where an output action $\overline{x}(v)$ sends a channel name v to an input action $x(y)$; for instance

$$x(y).P \mid \nu z\ (\overline{x}(z).Q) \rightarrow_{x,\overline{x}} \nu z\ ([^{z}/_{y}]P \mid Q).$$

The *polyadic* variant of the π-calculus is the most used, where a vector of values $\vec{v}$ can be sent and received, and usually the communication can happen only on *private channels*.

Let $\{x, y, z, \ldots\}$ be a countable set of variables for channel names of $\mathcal{A}$. Then the syntax of the π-calculus is the following:

$$P, Q ::= 1 \mid x(y).P \mid \overline{x}(y).P \mid (P \mid Q) \mid \nu x(P) \mid [x = y]P \mid !(x(y).P)$$

The semantics is pretty much the same as value-passing CCS, where: 1 is the empty process; $x(y).P$ is the input action prefix, waiting to receive a name z on x, and substitute y with it, becoming $[^{z}/_{y}]P$; $\overline{x}(z).P$ is the output action prefix, sending a name z on the channel x, becoming P; $P \mid Q$ is the parallel composition of P and Q; νx is the hidden (or new) name operator; $[x = y]P$ runs P if $x = y$, otherwise is undefined; and the *exponential* (in linear logic fashion) $!(x(y).P)$, defined on *inputs*, waits to receive a name z on x, and then duplicates in $[^{z}/_{y}]P \mid !(x(y).P)$ (it is a different way to give a recursive definition of a term).

α-renaming can be performed on names bound by ν, making them closer to *constants*; while the names attached to an input, which are going to be substituted, can be seen as *variables*. We have left aside the non deterministic sum +, because it can be encoded in the syntax, up to a certain extent[6].

Execution in the π-calculus works exactly as we showed from the example. Without restriction to communication on private names only, we have the following relation:

$$\overline{x}(z).P \mid x(y).Q \rightarrow_{(x,\overline{x})} P \mid [^{z}/_{y}]Q$$

extended by rules for ! and $[x = y]$

$$\overline{x}(z).P \mid !(x(y).Q) \rightarrow_{(x,\overline{x})} (P \mid [^{z}/_{y}]Q \mid !(x(y).Q);$$

$$[x = x]P \rightarrow P.$$

[6] See [4], p.95, for the details.

sometimes the ν might note that the *sent or substituted* channel is *fresh*. For example in $\nu y(x(y).P)$, a name y unique to P is going to be substituted in P after communication, different from x. Or $\nu y(\overline{x}(y).P)$ denotes that a fresh channel is going to be sent in output, and $\overline{x}$ becomes free after communication.

Variants

Skipping the discussion about bisimulation in the π-calculus (outside the scopes of this work) we already noted that the most widespread version of the π-calculus is the **polyadic** one, where multiple channel names can be sent and received at the same time in a single communication – obviously the input and output action must agree on the arity of names exchanged. We simply allow inputs and outputs of the form $x(\vec{y} = y_1, \ldots, y_n)$ and $\overline{x}(\vec{y})$.

Another variant is the **asynchronous communication**, where output actions *are not ordered*. This means that there is no notion of prefix, and thus there is no partial order between them, with respect to execution.

The π-calculus concludes our background chapters; the following chapter starts the original part of the thesis and our research, by building the fundamental notions and machinery needed to interpret CCS into ludics.

4. Interpretation of Multiplicative CCS

The objective of this chapter is to carry out the *first step* of our correspondence between the dynamic of *CCS processes*, that is *execution* (definition 3.2.3), and the dynamic of ludics *designs*, that is *interaction* (definition 2.2.15).

We say *first step* because the correspondence we define here will only be *limited* to a small, but relevant, part of CCS: to *simplify the setting as much as possible*, this translation will be carried out, at first, only in a fragment of CCS the multiplicative fragment, in short $MCCS$ – as already defined in [9], by V. Mogbil and E. Beffara – that is *without non deterministic choice* and *recursive definitions*. The name is due to its correspondence with *proof nets* of the multiplicative part of linear logic (restricted to its multiplicative connectives). While lacking some fundamental tools used to define interesting processes and functions, there are still good theoretical reasons to start with the simple fragment consisting of only *action prefix* and *parallel composition*, explained in subsubsection 4.1.1.

Later in chapter 6 we will interpret also the non deterministic choice + and private name operator ν, and at last attempt to extend the correspondence to *recursive definitions* in subsection 6.3.3.

This first step is divided into *two parts*: in section 4.1 we translate processes into *sets of designs*. The correspondence, on ludics' side, requires a more complex object: instead of translating a *single process* into a single design, we use a well-structured *set of designs*, that contains as elements modified copies, via *pruning* (definition subsection 2.2.5), of the same design, whose *chronicles* have an *explicit correspondence* with the *basic elements of the translated process*.

The reason why a single design is not enough to get a meaningful translation is explained during the construction of the set.

To start, in subsection 4.1.1, we give the definition of $MCCS$ found in [9], as a *restriction of* CCS grammar; then in subsection 4.1.2, we associate to $MCCS$ *processes* a *graph* (noted $\mathscr{G}_P$), in order to have a more geometrical representation of the structure of processes: in particular the structure given by the *partial order* on locations $<_P$ (definition 3.2.4) of a $MCCS$ process P, i.e. the *prefix* order on channels, whose correct representation inside ludics is one of the core steps of the interpretation.

Note that the graph associated to a process is only a *different presentation* of a $MCCS$ process that *let us put in advance* the features of a process relevant to *execution* and for its *interpretation as interaction*, and shorten a few proofs, but *is not a necessary step* for either the interpretation or the results that will follow.

Then, we exploit the *pruning* technique (defined in subsection 2.2.5) to indirectly represent the *prefix order* of channels of a process P, once a correspondence is defined between *elements of the process*, and *branches* of a design (defined in subsection 2.2.2).

In definition 4.1.14, is presented a *constructive definition* of the *base design associated to a process* P: a naive correspondence between the elements of the process *relevant for execution* (the *channel names labeled by locations*, the *set of synchronizations* $\mathcal{S}_P$, and the *conflict relation* $\mathcal{X}_P$) and *branches of a design* in ludics, *without carrying on their structure*. That means that the *partial orders* $<_P$ and $\leqslant_{\mathcal{S}_P}$ *are not represented in the base design.*

This construction is made by *merging* together (recall definition 2.2.29) smaller *negative* design as premises of a *positive* one. These negative design are made by chronicles of at most 2 rules, a negative one (with 1 or 2 premises) followed by a positive one, and the address introduced by the first rule is *associated to a specific element of the process* that we are interpreting.

In 4.1.15, we define *restriction designs* by using the *pruning* to *modify* copies of the base design with the aim to *represent the order* $<_P$, and the *conflict relation* $\mathcal{X}_P$.

The intended result is that these modified copies, called *directed modifications*, once put in a *set* together with the *base design* (denoted $\mathcal{D}_P$), force *interaction* with the *orthogonal set* to *respect the relations* $<_P$ and $\mathcal{X}_P$. This set is then *closed by bi-orthogonality* to obtain a *connected and directed behaviour*, that is the main element of the *interpretation in ludics of the process* P, noted $[\![P]\!]$.

This last step is actually superfluous, because *it is not necessary at all* to define a correspondence between the dynamics of processes and ludics. We

could actually stop the translation at the set containing the base design along with the restriction designs, and then define a correspondence between the possible execution paths on P, and interaction paths between this set of designs and its orthogonal. However, there are a few reasons behind this choice, mentioned in the introduction and further explained in subsection 4.1.4.

The first *part* of the correspondence between *dynamics* that we are seeking thus takes the form of a *naive translation*, of processes into sets of designs. The *main result* of this section is therefore the *definition* of the *translation* between $MCCS$ processes and *sets* of ludics' designs, composed of a *base design* and *restriction designs*, then closed by bi-orthogonality into *behaviours*. Then, the machinery needed to form a *correspondence between the dynamics* of $MCCS$ on one side, and the one of ludics on the other, is defined on these sets. The final objective of the work is to extend this correspondence to the full calculus; objective that will be only partially achieved.

In section 4.2, the *second part* is carried out, by finally defining a correspondence between *execution* on a process P and *interaction* between the *behaviour* translating P and its orthogonal set.

In this section we develop the tools needed to form the *correspondence* between *execution on MCCS processes* and *interaction between designs*, linking together the two *dynamcis*. The *main results* of this part are the *definition* of *execution associated* to an *interaction path* (definition 4.2.4) and a theorem stating that the *possible interactions on the process interpretation* $[\![P]\!]$ *characterize all the execution paths* of a $MCCS$ process P (theorem 4.2.14).

We start by defining the notion of *admissible execution* on a process P, to note a *valid* execution on the process that follows its reduction rule; this notion is used to define the concept of *execution associated to an interaction*, to extract from an *interaction path* between $\mathscr{B}_P$ and $\mathscr{B}_P^{\perp}$ an *admissible execution*.

Then in subsection 4.2.1 we show that the information coded in the *interpretation* of P, i.e the sets Loc_P, $\mathcal{S}_P$ and $\mathcal{X}_P$, are the *necessary and sufficient* information to describe the *possible execution paths of the process*, that lead to its multiple possible *reduced normal forms*; i.e. they are *exactly* the information we need to describe the dynamic of a process.

Finally, in subsubsection 4.2.1, we prove Theorem 4.2.14, stating that $[\![P]\!]$ *characterize all executions of* P, for a specific meaning of *characterize*. The theorem is split into *two parts*, and shows that the *interpretation* of a process P, $[\![P]\!]$, can completely *describe* the possible reduction sequences on P via *interaction with its orthogonal*, and that *every interaction* on $[\![P]\!]$ describes a possible execution on P.

4.1 From Multiplicative CCS processes to sets of designs

4.1.1 Multiplicative CCS

We introduce the *multiplicative* fragment of CCS, in short $MCCS$, as defined in [9]. Terms of $MCCS$, noted by the letters $P, Q, R, \ldots$, are built by the following restricted grammar:

Definition 4.1.1. $P, Q := 1 \mid a^l.P \mid \overline{a}^m.P \mid (P \mid Q)$
With $a, \overline{a} \in \mathscr{A}$, *and* $l, m \in Loc_P$.

We use 1 to denote the empty process, instead of the traditional 0, since it shows a *multiplicative* behaviour and a clear connection with the multiplicative unit 1. $\mathscr{A}$ is the a denumerable set of *channel names*, and Loc_P is a denumerable set of *locations*, to discriminate different occurrences of the same channel. The function associating a *location* to each channel name occurrence is a *bijection*, thus each channel occurrence receive an unique label that cannot appear elsewhere. This means that when processes are put together via *parallel composition*, then a *renaming of locations* may occur, in order to preserve the 1-1 correspondence between channel occurrences and locations. The dual operator is involutive, i.e. it holds that $\overline{\overline{a}} = a$.

The operations which we left aside to simplify the setting are:

- the *non deterministic sum* $P + Q$.
- the *new* operator $\nu a(P)$, which binds and makes a channel a *private*.
- the *recursive operator* $A(\vec{a})$.

The structural congruence, partial order $<_P$, the induced order on cuts $\leqslant_{\mathcal{S}_P}$ and the *xor* relation $\mathcal{X}_P$, as well as *execution*, are the same as we defined when introducing CCS (in section 3.2).

Why $MCCS$ is enough

The interpretation will be carried out, at first, for the simple *multiplicative* fragment of CCS that we introduced. The lack of *non deterministic choice* and *recursive definitions* does take away a lot of expressive power; however

our objective is to find a correspondence between the *dynamics* of the two systems, thus all we care about is the *essential part* of a process algebra, the bare minimum that allows us to perform *execution* on a process. Indeed that is one of the reasons we stick to CCS in the first place: it is the *most basic and primitive* process algebra, and does not have the same *expressive power* of the much more used π-calculus. What we are doing is taking away from CCS all that is not essential with respect to *execution* as we defined it in subsection 3.2.1.

What is needed to perform execution, i.e. to let two processes *communicate*, are *channel names of dual polarity* and *the parallel composition* operation, that *enables execution*. Therefore, we are restricting the syntax to this simple fragment, that we call *multiplicative* due to its *linear logic* interpretation into *multiplicative proof-nets*. We show later that an extension to the *non deterministic choice* + and *private name operator* is actually possible, while *recursive definitions* are much harder to interpret in an *intrinsically linear setting* as ludics.

4.1.2 Graph associated to Multiplicative CCS processes

In this section a *graph* will be associated to processes, with the objective to achieve a more geometrical presentation of the process, in order to put in advance the elements necessary to our *interpretation*: the set $\mathcal{S}_P$ (recall definition 3.2.7) of synchronizable channels, and the conflict relation between them $\mathcal{X}_P$.

The presentation is not necessary to prove properties or results about the interpretation, but is meant to help intuition and simplify some steps. A *reduction* on the *graph*, *matching execution on the process* it is associated to, will also be defined. This reduction will help us understand how the relevant relations mentioned above change with *execution*.

In the following, let P be a $MCCS$ process.

Definition 4.1.2. *The* graph associated to P, *noted* $\mathcal{G}_P$, *is a* directed *graphs composed of:*

- *As set of* **nodes** *the set of locations of the process* P, Loc_P, *considered as a* **multiset**, *where each location can appear multiple times, depending on how many synchronizations it belongs to. Each location is* labeled with either synchronizations of $\mathcal{S}_P$ or with $\perp$.
- *Formally, we note nodes as locations labeled in the following way:* $\{l^* \mid l \in Loc_P$ *and* $* = u \in \mathcal{S}_P$ *such that* l *belongs to* u, *if there is such* u, *otherwise* $* = \perp\}$

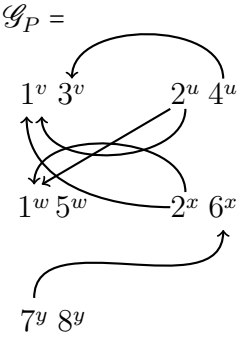
We say l belongs to u *with the meaning of definition 3.2.7.*

All locations appear in the multiset at least once. *A location* l *appears* more than once *only if it belongs to* more than one synchronization. *For instance we would have the nodes* l^u, l^v *if* l *belongs to two synchronizations* u *and* v.

- *As set of* **edges** *the set* $E : \{(l^*, m^*) \mid m <_P l\}$ *with* $<_P$ *the* partial order on locations *of definition 3.2.4. We note an edge with* $l^* \to m^*$.

Example 4.1.3. *Let* $P = a^1.b^2 \mid \overline{a}^3.\overline{b}^4 \mid \overline{a}^5 \mid \overline{b}^6.c^7 \mid \overline{c}^8$.
$\mathcal{S}_P = \{v = (a^1, \overline{a}^3); w = (a^1, \overline{a}^5); u = (b^2, \overline{b}^4); x = (b^2, \overline{b}^6); y = (c^7, \overline{c}^8)\}$.
$\mathcal{X}_P = \{(v, w), (u, x)\}$.

$\mathcal{G}_P =$



The xor relation is easily identifiable, since there is one instance of a location for each synchronization they belong to: if two synchronizations label the same location, then they form an element of $\mathcal{X}_P$.

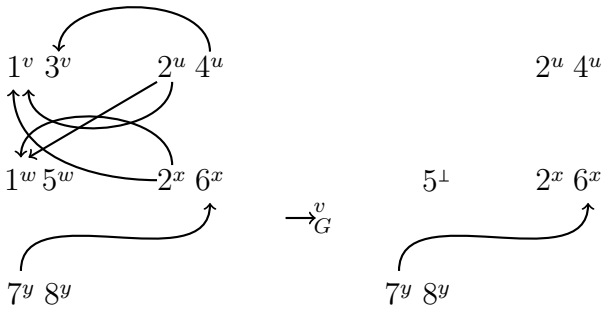
Definition 4.1.4. *One-step reduction on $\mathscr{G}_P$ is a rewriting operation on graphs, defined as follows:*

1. *If exists a pair l^i, m^i in $\mathscr{G}_P$ (i.e. a pair of nodes consisting of two different locations labeled with the same synchronization), such that* no edge *departs from either l or m (i.e. $\neg\exists o$ such that $l \to o$ or $m \to o$)*[1] *then $\mathscr{G}_P$ can be rewritten to $\mathscr{G}' = \mathscr{G}_P \setminus \{l^*, m^*\}$, a subgraph of $\mathscr{G}_P$ determined by the following steps:*
   - *we* erase *from the nodes* all occurrences *of l^* and m^*, and all* edges *between them and the other nodes;*
   - *any location o^* such that $* = u \in \mathcal{S}_P$, and l or m belongs to u (which means that o is, necessarily, the only location labeled with u) is rewritten to $o^\perp$.*
2. *If there is no such a pair, then* no rewriting is possible.

The reduction *on $\mathscr{G}_P$ on a pair (l^i, m^i) is noted by $\mathscr{G}_P \to^i_G \mathscr{G}'$.*

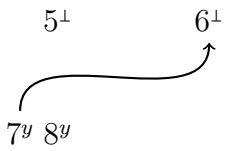
Reduction on $\mathscr{G}_P$ is the transitive closure of $\to_G$.

Example 4.1.5 (Reduction on $\mathscr{G}_P$)**.** *Let P and $\mathscr{G}_P$ be the, respectively, the process and the graph of the previous example. Reduction on $v \in \mathcal{S}_P$ induces the following rewriting:*



Now, further reductions are possible on $(2^u, 4^u)$ or $(2^x, 6^x)$, but not on $(7^y, 8^y)$ (since 7 is not minimal *with respect to $<_P$). The reduction on $(2^u, 4^4)$ would leave $\mathscr{G}_P$ in the following normal form:*

[1] Note that this means that the locations l and m are ***minimal*** with respect to $<_P$



It holds that:

Lemma 4.1.6. *$\mathscr{G}_P \to^{i}_{G} \mathscr{G}'$ if and only if $P \to_i P'$, and $\mathscr{G}' = \mathscr{G}_{P'}$*

Proof. Straightforward by definition of reduction on $\mathscr{G}_P$, and execution on P. Erasing a *synchronizable pair* of channels directly correspond to erasing a synchronization in execution. Since the edges are also erased, and channels (represented by their location in the graph) that are not synchronizable anymore has their label rewritten to $\perp$, the resulting graph $\mathscr{G}'$ is equal to the graph associated to P', by definition of $\mathscr{G}_P$. □

Lemma 4.1.7. *If $\mathscr{G}_P \to^{\vec{i}}_{G} \varnothing$ for some $\vec{i}$, then execution is possible on all channels of P, there are no* cycles *in the partial order on synchronizations $\leqslant_{\mathcal{S}_P}$, and $\mathcal{S} = \{(l^i, m^i) \in \mathscr{G}_P \mid i \in \vec{i}\}$ is a* maximal pairing *(definition 3.2.5), and $P \to^{\vec{i}} 1$.*

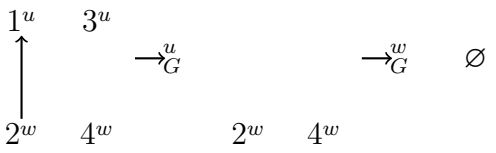
Proof. A cycle in the partial order on $\mathcal{S}_P$ is represented in $\mathscr{G}_P$ by a (l^i, m^i) and (h^j, k^j) such that, without loss of generality, $l \to_G h$ and $k \to_G m$. This prevents either couples to ever be a sink, and so be erased. If $\mathscr{G}_P = \varnothing$, then all locations, and so all channel names, have been erased by reduction; this implies that the corresponding execution on P gives 1 as result. The sequence of synchronizations indexed by $i \in \vec{i}$ is then a maximal pairing. □

Corollary 4.1.8. *If $\mathscr{G}_P \to^{\vec{i}}_{G} \mathscr{G}_{P'}$ and $\mathscr{G}_{P'} \neq \varnothing$ but there are no more pairs of sinks in $\mathscr{G}_{P'}$ with* the same label, *then $\mathcal{S} = \{(l^i, m^i) \in \mathscr{G}_P \mid i \in \vec{i}\}$ is a* maximal pairing.

Proof. Obvious by definition of $\mathscr{G}_P$ and reduction on $\mathscr{G}_P$. If there are no more sinks, there are no synchronizations, and therefore no further reduction is possible. □

Example 4.1.9. *Let $P = a^1.b^2 \mid \overline{a}^3 \mid \overline{b}^4$. Synchronizations are $u = (a^1, \overline{a}^3)$ and $w = (b^2, \overline{b}^4)$. The graph associated to P can easily be reduced to the empty graph, in two reduction-steps.*

$\mathscr{G}_P =$



Remark 4.1.10. *The role of the graph defined here is to put in advance the parts and structure of processes that is needed to form the correspondence in dynamic we seek; in other words, the essential elements needed to describe process behaviour. While being a tool originally made only to help intuition, we found a clear resemblance to* event structures *(definition 3.2.9), in particular when seen as a model for processes: the relations $<_P$ and $\mathcal{X}_P$, which are going to be the core of our* interpretation, *are obviously linked to $\leq$ and $\smile$;* locations *can be seen as* single events, *and* synchronizations *as a* new single event *tied to both its members by the relation $\leq$. We will give further details about the connection between event structures and our work in chapter 7.*

4.1.3 The translation of Multiplicative CCS processes

The aim of the interpretation is to build a *behaviour* capable of characterizing a process by representing *all and only its execution paths*, via interaction with its orthogonal. We need to build designs which will make only the *right interactions* reach ✠. To this end, we need a way to forbid interaction to take certain paths, while allowing others; in particular, we want it to respect the partial order $<_P$ and, by consequence, $\leqslant_{\mathcal{S}_P}$ (we omit the subscript when obvious by the context), as well as the conflict relation $\mathcal{X}_P$, once we have interpreted the relevant elements of the process into designs.

The solution is to exploit the *pruning* (defined in subsection 2.2.5), which lets us build *sets of designs* where interaction with the orthogonal is forced to visit certain branches (the ones generated by the ramification of the first rule) *before* others. Once each branch is associated to an element of the process, we will be able to make interaction *respect the prefix order and conflict relation* that we defined on the process.

We are going to exploit this technique to represent $<_P$, and $\mathcal{X}_P$, by building directed designs representing the two relations *locally*; i.e. one design will represent one element of the relation. Then, once we put them together as elements of a set, interaction with the orthogonal set will automatically respect the *full relations*, by transitivity.

Freedom requisites

What we need next is to associate to each synchronization the local requisites that makes it available for execution, and to each location the synchronizations that let us erase it. To this end, we introduce the following:

Definition 4.1.11 (Freedom requisite)**.** *Let P be a MCCS process. Let $\mathscr{F}()$ be a function going from $\mathcal{S}_P \cup Loc_P$ to* subsets *of $\mathcal{S}_P$ and Loc_P. With* **freedom requisite** *of a synchronization $i \in \mathcal{S}_P$ or* location *$l \in Loc_P$ we mean the* output *of, respectively, $\mathscr{F}(i)$ or $\mathscr{F}(l)$.*

$\mathscr{F}()$ is defined by cases:

1. *Let $i = (a^l, \overline{a}^h) \in \mathcal{S}_P$, then, if i* is not minimal with respect to $<_P$

$$\mathscr{F}(i) = \{m, n \in Loc_P \mid l \to m \vee h \to n \text{ are edges of } \mathscr{G}_P\}.$$

   Otherwise $\mathscr{F}(i) = \varnothing$.

   The definition selects the very precedent location, if there is any, of each location labeling the channels of i.

2. *Let $l \in Loc_P$, then*

$$\mathscr{F}(l) = \{i \in \mathcal{S}_P \mid l \in i\}$$

   Note that if l does not belong to a cut, then $\mathscr{F}(l) = \{\varnothing\}$.

   In this case the definition selects all the synchronizations (possibly none) to which a location l belongs to.

Note that

1) If i is *principal*, $\mathscr{F}(i) = \varnothing$.

2) If i has only *one minimal location*, then $\mathscr{F}(i) = \{p\}$, where p is the *very precedent* location of l or h.

3) If i has *no minimal location*, then $\mathscr{F}(i) = \{m, n\}$ such that, without loss of generality, $m <_P l$, $n <_P h$, and m, n are the respective *predecessors* of l, h.

Remark 4.1.12. *It holds that $\mathcal{X}_P$ and freedom requisites cover any possible case of* not-executable synchronization*. In the simple setting of MCCS, a synchronization is not executable if its blocked by a location l not belonging to a synchronization (such that $\mathscr{F}(l) = \varnothing$), or if its locations are subordinate to two different sides of the same xor clause. So if $u = (a^l, \overline{a}^m)$, $a^l > b^n$, $\overline{a}^m > b^p$, $v = (b^n, \overline{b}^q)$, $w = (b^p, \overline{b}^q)$, and b^n, b^p do not belong to any other synchronization, then $(v, w) \in \mathcal{X}_P$, and u is* not executable.

Building the Base design

In this section we build the **base design** of a process P, denoted $\mathcal{D}_P$, by merging together multiple basic *negative* designs, defined for each location, synchronization, and *xor* condition, via a positive n-ary action. The result is a positive design with the base of each of these negative designs as premise of a positive rule, and the negative designs as sub-designs of this base one.

Definition 4.1.13 (Assignment function)**.** *Let* $[\ ]_P$ *be an injective function associating elements of a process* P *to addresses with the same prefix (as* $\xi 1, \ldots, \xi n$*). The* domain *of* $[\ ]_P$ *is* $Loc_P \cup \mathcal{S}_P \cup \mathcal{X}_P$*. With* $[i]_P$*,* $[l]_P$ *and* $[(u,v)]_P = xor^u \,\&\, xor^v$*, we denote respectively the address assigned to the synchronization* i*, the location* l *or the xor condition* (u,v) *in the base design* $\mathcal{D}_P$*. For instance, we have* $[i]_P = \xi.1$*,* $[l]_P = \xi.3$*, and* $[(u,v)] = \xi.4$*.*

We call $[\ ]_P$ **assignment function**.

The base design will serve as a naive interpretation of a process; then we will build *restriction* designs as *directed designs*, i.e. modifications of the base design using the pruning. There will be one restriction for each *freedom requisite*, which represents the local order and conflict relation between synchronizations and locations. Then, both the restrictions and the base design will be put together in a set that generates a behaviour by bi-orthogonal closure, along with the full order and conflict relation by transitivity.

Definition 4.1.14 (Base design)**.**

- *Let* $x \in \mathcal{S}_P \cup Loc_P$

$$G[x]_P = \dfrac{\dfrac{[x]_P.1.1 \vdash}{\vdash [x]_P.1}}{[x]_P \vdash}$$

- *Let* $(u,v) \in \mathcal{X}_P$*, then*

$$w[u,v] = \dfrac{\dfrac{xor^u.1 \vdash}{\vdash xor^u} \quad \dfrac{xor^v.1 \vdash}{\vdash xor^v}}{xor^u \,\&\, xor^v \vdash}\ \&$$

where xor^u *and* xor^v *are the sub-addresses of* $xor^u \,\&\, xor^v$*. We note the last rule with* $\&$ *since it is a binary negative rule, corresponding to the linear logic connective* $\&$*. They serve as check-addresses, not corresponding to an element of* P*, but linked to the respective synchronizations* u *and* v*. One branch will have precedence on* $[u]_P$*, the other*

on $[v]_P$ through dedicated restriction designs, thus effectively separating the two interaction paths in a non-confluent way.

We have

$$\mathcal{D}_P = (\bigotimes_{x \in \mathcal{S}_P \cup Loc_P, (u,v) \in \mathcal{X}_P} \{G[x], w[u,v]\}).$$

Where $\otimes$ is a sole positive *action with the elements of the set as sub-designs, i.e the premises generated by the ramification of the positive rule.*

The result of the operation is a rather *flat* interpretation, where all the elements needed to characterize execution are naively put together, while losing all the relevant relations between them: nor the partial order nor the conflict relation are coded in this way.

What we need is a way to recover these relations during interaction, and to this end we are going to modify the *base design* accordingly to the *freedom requisites* that we defined, obtaining a set of *restriction designs.*

Restriction designs

Let P be a $MCCS$ process, and $\mathcal{D}_P$ its base design.

Definition 4.1.15 (Restriction Designs)**.** *Let P be an $MCCS$ process. The* **set of restriction designs** *for elements of $\mathcal{S}_P, Loc_P$ and $\mathcal{X}_P$ (also called* non commutative restrictions*), is the set*

$$\mathcal{R}(P) = \mathcal{R}(\mathcal{S}_P) \cup \mathcal{R}(Loc_P) \cup \mathcal{R}(\mathcal{X}_P)$$

where each element of the union is a set of modified copies of $\mathcal{D}_P$*, regarding specific elements of P (we omit the subscript $_P$ when denoting the associated addresses). These copies of $\mathcal{D}_P$ have only a few specific branches changed with prunings or daimons ✠, while all the other chronicles – represented by the various $\cdots$ – are left untouched.*

Restriction designs for synchronizations.

Let $i \in \mathcal{S}_P$. If $\mathcal{F}(i) = \{l\}$, then $\mathcal{R}(i)$ is a copy of $\mathcal{D}_P$ modified in the following way:

$$\mathcal{R}(i) = \dfrac{\cdots \quad \dfrac{\dfrac{}{\vdash [l].1}\ ✠}{[l] \vdash} \quad \cdots \quad \dfrac{}{[i] \vdash}\ p \quad \cdots}{\vdash \xi}$$

If $\mathscr{F}(i) = \{m, n\}$, *then* $\mathcal{R}(i)$ *is the following pair of designs:*

$$\dfrac{\cdots \quad \dfrac{\dfrac{}{\vdash [m].1}\,\maltese}{[m]\vdash} \quad \cdots \quad \dfrac{}{[i]\vdash}\,p}{\vdash \xi} \qquad \dfrac{\cdots \quad \dfrac{\dfrac{}{\vdash [n].1}\,\maltese}{[n]\vdash} \quad \cdots \quad \dfrac{}{[i]\vdash}\,p}{\vdash \xi}$$

If $\mathscr{F}(i) = \varnothing$, *for* $i \in \mathcal{S}_P$, *then* **there is no** $\mathcal{R}(i)$. $\mathcal{R}(\mathcal{S}_P)$ *denotes the set of all* $\mathcal{R}(i)$, *for each* $i \in \mathcal{S}_P$.

Restriction designs for locations.
Let $l \in Loc_P$. *We define* $\mathcal{R}(l)$ *by cases.*

1. *If* l *is a* **minimal location** *(with respect to* $<_P$*), and* $\mathscr{F}(l) = \{i_1, \ldots, i_n\}$, *then* $\mathcal{R}(l)$ *is the following design:*

$$\dfrac{\ldots \quad \dfrac{\dfrac{}{\vdash [i_1].1}\,\maltese}{[i_1]\vdash} \quad \ldots \quad \dfrac{\dfrac{}{\vdash [i_n].1}\,\maltese}{[i_n]\vdash} \quad \ldots \quad \dfrac{}{[l]\vdash}\,p \quad \ldots}{\vdash \xi}$$

2. *If* l is not minimal, *let* m *be the* immediate predecessor of l, *then, we have two cases:*

   (a) *If* $\mathscr{F}(l) = \{i_1, \ldots, i_n\}$, , *then* $\mathcal{R}(l)$ *is the following pair of designs:*

$$\dfrac{\cdots \quad \dfrac{\dfrac{}{\vdash [m].1}\,\maltese}{[m]\vdash} \quad \cdots \quad \dfrac{}{[l]\vdash}\,p}{\vdash \xi}$$

   and

$$\dfrac{\ldots \quad \dfrac{\dfrac{}{\vdash [i_1].1}\,\maltese}{[i_1]\vdash} \quad \ldots \quad \dfrac{\dfrac{}{\vdash [i_n].1}\,\maltese}{[i_n]\vdash} \quad \ldots \quad \dfrac{}{[l]\vdash}\,p \quad \ldots}{\vdash \xi}$$

   with a $\maltese$ *over each* $G[i_1], \ldots, G[i_n]$.

   (b) *If* $\mathscr{F}(l) = \{\varnothing\}$, *then the second design becomes:* $\dfrac{\ldots \quad \dfrac{}{[l]\vdash}\,p \quad \ldots}{\vdash \xi}$

With $\mathcal{R}(Loc_P)$ we denote the set of all $\mathcal{R}(l)$ for each $l \in Loc_P$.

Restriction designs for the conflict relation $\mathcal{X}_P$.
Let $(u,v) \in \mathcal{X}_P$, then $\mathcal{R}(u,v)$ is the following pair of designs:

$$\dfrac{\ldots \quad \dfrac{\dfrac{}{\vdash xor^u} \; \clubsuit \; \dfrac{xor^v.1 \vdash}{\vdash xor^v}}{xor^u \,\&\, xor^v \vdash} \quad \ldots \quad \dfrac{}{[u] \vdash} p \quad \ldots}{\vdash \xi}$$

and equivalently for v:

$$\dfrac{\ldots \quad \dfrac{\dfrac{xor^u.1 \vdash}{\vdash xor^u} \; \dfrac{}{\vdash xor^v} \; \clubsuit}{xor^u \,\&\, xor^v \vdash} \quad \ldots \quad \dfrac{}{[v] \vdash} p \quad \ldots}{\vdash \xi}$$

With $\mathcal{R}(\mathcal{X}_P)$ we denote the set of all $\mathcal{R}(u,v)$ for each $(u,v) \in \mathcal{X}_P$.

With $\mathcal{R}(P)$ we denote the set

$$\mathcal{R}(\mathcal{X}_P) \cup \mathcal{R}(\mathcal{S}_P) \cup \mathcal{R}(Loc_P).$$

Remark 4.1.16. *The non commutative restrictions for locations might seem superfluous at first: indeed, we are making explicit the* local *prefix order regarding each location, i.e. the very preceding location in $<_P$ must have precedence in interaction. This might seem useless to add, if we already have the induced order on* synchronizations, *which are the executable part of the process: the prefix order would be implicitly respected by interaction by just following the freedom requisites on synchronizations. The local precedence order, however, is necessary when* combining *interpretations to represent parallel composition. In this case, when new synchronizations, and execution paths, are generated, then we need the local order on locations to deduce the freedom requisites for these new synchronizations.*

Definition 4.1.17. *Let P be a MCCS process. Then let*

$$\mathbb{B}_P = \{\mathcal{D}_P\} \cup \mathcal{R}(P)$$

be the set of generators *of the behaviour*

$$\mathscr{B}_P = \mathbb{B}_P^{\perp\perp} = (\{\mathcal{D}_P\} \cup \mathcal{R}(P))^{\perp\perp}$$

Since all the restriction designs are modification of $\mathcal{D}_P$, and thus have the same first action, *$\mathscr{B}_P$ is a* connected *and* irregular *behaviour.*

Then, the **interpretation of P** *is*

$$[\![P]\!] = (\mathscr{B}_P, [\ \ ]_P)$$

the pair formed by the behaviour and the assignment function.

The interpretation aims to describe, using a superimposition of terms, the behaviour of a process. In order to get a *modular interpretation*, we will define an operation based on $\otimes$ on behaviours, but on the set of generators, with some further technical and artificial steps, but still based on the *merging* of designs $\odot$.

Although the interpretation is rather *flat*, since the structure of the process is indirectly coded in it, the relations expressing this structure are generated through the effect of the restriction designs on interaction. Having multiple restriction designs let us easily and locally modify each single one, by breaking the relations we are interpreting in smaller local pieces. When building a process by parallel composition, action prefix, or any other operation, we can directly act on single restriction designs to account for each change in the order or conflict relation, without the need to build the interpretation from scratch for every change in the process. In this way achieving a modular interpretation is easier than dealing with a structure coded in a single design. Coding the order and conflict relation information explicitly would be a huge complication of the base design, and for a possible read-back of the process. Moreover many difficulties arise when trying to relate different addresses during interaction, that need to express multiple local relations about the same element of the process, since the only way in which addresses are linked is via *justification sequences*.

4.1.4 Why Behaviours?

When we close the set of the base and restriction designs by bi-orthogonality, we obtain a behaviour; however, a *bi-orthogonal closure is not needed* to achieve our results, and to form the intended correspondence of dynamics between processes and ludics that is the main objective of our interpretation. Indeed, *we never use a property of behaviours* in any formal proof. Still, there are a few reasons to actually take this superfluous step and consider the *bi-orthogonal closure* of the defined set.

Behaviours and types. The first reason to consider is that behaviours are the closest thing to a *type* in ludics: they are the *type* of a set of designs, and working with behaviours is a natural step in ludics. We already

mentioned that, when interpreting linear logic into ludics, formulas are *interpreted as behaviours*, and we showed how to *linear logic connectives* can be represented as *operations on behaviours* (in subsubsection 2.2.4). Using behaviours to interpret processes let us form a connection between the *type* of a certain set of design, and the *type* of a process, in the sense of the way it interacts with the environment.

Behaviours are one of the most interesting and studied aspect of ludics, therefore by forming a correspondence between processes and behaviours, we may be able to exploit the operations and properties found about them to deepen the connection. In particular, checking what happens on one side of the correspondence when something "interesting" is performed on the other. For instance, in the next chapter, we will see how to interpret *parallel composition* of two processes P and Q as an operation on their respective interpretations $[\![P]\!]$ and $[\![Q]\!]$. It turns out that in a *trivial case*, where P and Q *cannot communicate* at all, *parallel composition can be represented by* $\otimes$ (definition 2.2.30). Moreover, a *reduction-like operation* can be defined on $[\![P]\!]$ that *matches execution on* P (in section 5.2.11); by using this operation a few properties can be deduced. This reduction operation on the *interpretation* actually matches an *already existing ludics operation*, defined on *connected behaviours*, called *projection* (found in [38]), that *commutes with the bi-orthogonal closure*.

Does behaviours really add something? While behaviours *potentially* give us a lot of already existing tools to work with, they are not actually *needed*: indeed, while operations on processes do correspond to interesting operations on behaviours (a tensor-like operation for *parallel composition*, and a reduction for *execution*), the converse *is not true*. We have not found particular properties or operations on behaviours that if applied to the interpretation of a process $[\![P]\!]$ result in something interesting on the side of the process P. The interpretation of a process P is easily lost by operations that modify its designs, and moreover all of the above operations matching parallel composition and execution can still be defined on the *set of generators*, without bothering with the bi-orthogonal closure.

However, there is another reason to consider behaviours, despite being *theoretical* rather than practical, since it *does not help* regarding the technical purpose of our interpretation, that is finding a *correspondence in dynamics* between CCS and ludics.

Completeness. By considering the bi-orthogonal of a set of design, we are taking into account *all designs that behave in the same way* as those of the set. This give us a closure of the set *with respect to its orthogonal*.

Indeed, from a theoretical point of view, the *core of the interpretation* is actually the *orthogonal of the set of generators* $\mathbb{B}_P$. That is because a design *is its own semantics*, since *its structure makes explicit its meaning*, that is the way it interacts with the other designs. Since the important feature of our interpretation is the *correspondence in dynamics*, what we care about are actually *the possible interaction paths* on the *base design*, once *limited by the restriction designs*. These interaction paths are actually *already explicited* by the *designs in the orthogonal set* $\mathbb{B}_P^\perp$, since they tell us exactly how the interaction will go: which branches are going to be visited, and in which order. Thus, closing the set of generators *with respect to its orthogonal*, i.e. performing the bi-orthogonal closure, give us a *completeness* regarding the *interaction paths* that form the *correspondence of dynamics*, i.e. all the designs *successfully interacting* with $\mathbb{B}_P^\perp$ (i.e. reach ✠ during interaction), that therefore is the set *giving meaning to the interpretation*.

However this *closure* to achieve completeness is part of a more general theoretical need of logic, that is justified by the *incompleteness theorems*. Since we know that there are sets not only *incomplete* (with respect to something else, as *provability* or *interaction* in our case), but rather *not completable*, it has some importance to wonder if the set we are working with *is completable or not*.

The fact that the *bi-orthogonal closure* does not affect the *correspondence between MCCS processes and sets of designs* is relevant, especially considering the fact that *this might not hold anymore* once the interpretation are extended to full *CCS* with *recursive definitions*. Indeed recursion and replication require *at least* an extension also of *the syntax of ludics*, and thus of *interaction*, meaning that a bi-orthogonal closure *might not preserve the interpretation anymore*.

4.2 Connecting execution to interaction

In order to form a correspondence between *dynamic* of a process P and its interpretation $[\![P]\!]$ we need to be able to extract from interaction some information that can be read as an execution on the process. We need to establish which steps of interaction correspond to execution on which synchronization, in such a way that $[\![P]\!]$ contains *all and only* interaction paths that describe actual executions on the process P, i.e. each interaction must have an associated execution on P, and each execution must be associated to (at least) one interaction.

The notion we need is the one of *visited actions* inside an *interaction path*. Recall that actions considered at each step by interaction are said *visited*

(definition 2.2.15), and an *interaction path* on a design is simply the sequence of visited actions. If we restrict our attention to one design participating in an interaction session (inside a cut-net or with another single design), for instance the base design $\mathcal{D}_P$ inside the set of generators of $\mathscr{B}_P$, we can define the actions visited on this particular design as a simple restriction on the interaction path.

Remark 4.2.1. *Defining an interaction path as the sequence of actions visited during interaction is also found in [30], where with $\langle \mathcal{D} \leftarrow \mathscr{R} \rangle$ is denoted the sequence of actions of $\mathcal{D}$ visited during interaction with $\mathscr{R}$, where $(\mathcal{D}, \mathscr{R})$ is a closed cut-net. The formal definition is by induction on the number of normalization steps.*

Our aim is to give a definition of *associated execution* in order to make any design *orthogonal to $\mathscr{B}_P$ describe an execution*, even if the empty one; at the same time the *directed* restriction designs (definition 4.1.15) will assure us that *if* a design is orthogonal to $\mathscr{B}_P$, *then* its associated execution (on the process P) is *admissible*, with the following meaning:

Definition 4.2.2. *Let P be a MCCS process, and $\vec{i} = i_1, \ldots, i_n$ be an ordered sequence of synchronizations of P. We say that $\rightarrow_{\vec{i}}$ is an* admissible execution *on P if $\{i_1, \ldots, i_n\}$ is a* consistent pairing *(definition 3.2.5).*

Remark 4.2.3. *If $\vec{i}$ describes an* admissible execution *(on P), then it respects the partial order $<_P$ (thus also the induced partial order $\leqslant_{\mathcal{S}_P}$ on synchronizations), and the xor conditions $\mathcal{X}_P$ (i.e. there are not two synchronizations in the sequence that are in a xor relation), thus being a possible execution path on P. This follows immediately from the definition of* consistent pairing.

Using the sequence of *visited actions* with respect to an interaction path we can define the notion of *associated execution* (to an interaction path). Borrowing the notation of [30], we denote the *sequence of visited actions* of a design $\mathcal{D}$ during interaction with another design $\mathcal{C}$ with $\langle \mathcal{D} \leftarrow \mathcal{C} \rangle$. It is the sequence $k_1, \ldots, k_n \restriction_{\mathcal{D}}$, with $k_1, \ldots, k_n$ the sequence of actions visited by the interaction between $\mathcal{C}$ and $\mathcal{D}$; i.e. the actions $k_i, \ldots, k_j$, $1 < i \leq j < n$, such that $k_i, \ldots, k_j \in \mathcal{D}$.

Definition 4.2.4 (Associated execution)**.** *Let P be a MCCS process, and let $\mathcal{C} \in \mathscr{B}_P^{\perp}$. The execution on P* associated *to $\mathcal{C}$ is the execution $\rightarrow_{\vec{i}}$, where the sequence $\vec{i} = i_1, \ldots, i_n$ is a sequence of synchronizations such that*

$$\forall i \in \vec{i},\ (-, [i], \{\{1\}\})(+, [i].1, \{1\}) = G[i] \in \langle \mathcal{D}_{\mathcal{P}} \leftarrow \mathcal{C} \rangle$$

ordered as they are visited by the interaction path. We often say execution associated to $\langle \mathcal{D}_{\mathcal{P}} \leftarrow \mathcal{C} \rangle$ *to make explicit the base design involved in the definition. To note a generic associated execution, not tied to a particular counter-design, we say* execution associated to an interaction path.

Let us give a more precise intuition with an example.

Example 4.2.5. *Let* $\mathcal{D}_P =$

$$\dfrac{\cdots \quad \dfrac{\dfrac{\overline{\xi.1.1.1 \vdash}}{\vdash \xi.1.1}(+,\xi.1.1,\{1\})^4}{\xi.1 \vdash}(-,\xi.1,\{\{1\}\})^3 \quad \dfrac{\dfrac{\overline{\xi.2.1.1 \vdash}}{\vdash \xi.2.1}(+,\xi.2.1,\{1\})^8}{\xi.2 \vdash}(-,\xi.2,\{\{1\}\})^7 \quad \cdots}{\vdash \xi}(+,\xi,I)^0$$

$$\mathcal{C} = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\vdash \xi.2.1.1, \xi.1.1.1, \Delta}}{\vdash \xi.2.1.1, \xi.1.1.1, \Delta}\,\maltese^{10}}{\xi.2.1 \vdash \xi.1.1.1, \Delta}(-,\xi.2.1,\{\{1\}\})^9}{\vdash \xi.1.1.1, \xi.2, \Delta}(+,\xi.2,\{1\})^6}{\xi.1.1 \vdash \xi.2, \Delta}(-,\xi.1.1,\{\{1\}\})^5}{\vdash \xi.1, \xi.2, \Delta}(+,\xi.1,\{1\})^2}{\xi \vdash}(-,\xi,\{I\})^1$$

Assume $\xi.1 = [i]$ *and* $\xi.2 = [j]$. *Then* $\langle \mathcal{D}_{\mathcal{P}} \leftarrow \mathcal{C} \rangle =$

$$(+,\xi,I)^0(-,[i],\{\{1\}\})^3(+,[i].1,\{1\})^4(-,[j],\{\{1\}\})^7(+,[j].1,\{1\})^8.$$

Therefore the execution associated to $\langle \mathcal{D}_{\mathcal{P}} \leftarrow \mathcal{C} \rangle$ *is* $P \rightarrow_i \rightarrow_j$. *Furthermore* i, j *are* principal synchronizations, *since they are visited before any other branch associated to locations or synchronizations in* $\mathcal{D}_P$.

Remark 4.2.6. *It obviously holds that an action* i *is principal – equivalently* $P \rightarrow_i$ *is a valid execution (synchronization on the channels of* i *is possible) – if and only if there is a* $C \in \mathscr{B}_P^{\perp}$ *such that visits* $G[i]$ *before any other ramification of* $\mathcal{D}_P$ *corresponding to a synchronization. Indeed, this is possible only if* $\mathscr{F}(i) = \varnothing$, *meaning that there are no restriction designs for* i *except its xor conditions.*

The proof of this fact follows immediately by applying the definitions of principal synchronization, execution on P *and* associated execution.

Also it must be noted that the opposite direction of the result we are aiming at – i.e. the property stating that *if* the execution associated to a design is admissible, *then* the design itself is orthogonal – does not, and *must*

not hold, otherwise the correspondence between execution and interaction that we seek would be lost. This is because the ✢ in the restriction designs $\mathscr{R}$ is used to impose an order on interaction, but not to *force* the interaction. For example, if we do not want to interact with a certain location l, we don't have to visit the branches corresponding to elements of $\mathscr{F}(l)$ (synchronizations, or $[m]_P$ such that $m <_P l$), which are the *only* ramifications of $\mathcal{R}(l)$ ending with a ✢. Therefore we could have a counter design respecting all restrictions, but not visiting the branch of $[m]_P$ at all, thus not being orthogonal to $\mathcal{R}(l)$, unless it ends with a ✢ itself. Due to how we have build $[\![P]\!]$, since in $\mathcal{D}_P$ there are no daimons, a ✢ must be present in the counter-designs: nevertheless, we could apply the definition of associated execution even to a design not ending with ✢, which will thus not be orthogonal. If we wanted to include all the designs that have an admissible associated execution, we would need to force daimons on the base design, but in this way any interaction would stop immediately, since it would encounter a ✢ on any branch, and so the correspondence itself would not make sense.

Remark 4.2.7. *The fact that only the counter-designs of $\mathscr{B}_P$ end with ✢ implies that, even on* maximal interaction paths*, a ✢ is only found in designs $\mathcal{C} \in \mathscr{B}_P^{\perp}$. This means by definition, that $\mathscr{B}_P$ is a* **pure** *behaviour, a concept tied with* type safety *and the interpretation of* errors *when typing functions and* data types*, but not particularly relevant in our case.*

In this way each execution sequence will be associated to *multiple* interaction paths, and so counter-designs: this is mostly due to the fact that we have ramifications neutral with respect to execution, for example those corresponding to xor conditions, the various $xor_u \,\&\, xor_v \vdash$, which do not correspond to synchronizations or locations, therefore cannot be linked to execution directly, and the order in which they are visited is *not relevant* with respect to execution. This also means that the number of interaction steps that can be linked to a single execution step is not fixed.

Remark 4.2.8. *The requisite to associate an execution to an interaction is rather weak, but straightforward. The only condition we are imposing is that the actions corresponding to $G[i]$ are visited during interaction, without any constrain on the sequence of actions preceding or following $G[i]$, with the order by which they are visited as the order of the associated execution. This simple definition works because of the* restriction designs*: by coding the local prefix order – and xor conditions – they force any interaction with $\mathscr{B}_P$ to respect $<_P$, assuring us that it does correspond to an admissible execution, regardless of the context.*

4.2.1 A correspondence in dynamic

To summarize, we have the following elements:

$P \in MCCS$:	$\mathscr{G}_P$:	$\mathscr{B}_P$:
$Loc_P, \mathcal{S}_P, \mathcal{X}_P$	$\Rightarrow$ vertexes $\in \mathscr{G}_P$	$\Rightarrow$ sub designs $G[x], w[u,v] \subset \mathcal{D}_P$
$<_P$	edges $(u,v) \in \mathcal{A}$	$\varnothing$
$\mathscr{F}(\cdots)$	directed edges and labels on nodes	designs $\mathcal{R}(\cdots)$
$P \to_i$	reduction on $\mathscr{G}_P$	interaction between $\mathscr{B}_P$ and $\mathscr{B}_P^\perp$

In order to prove the correspondence between execution and interaction, we will focus on each step, to justify its choice and prove that it fulfills its tasks.

$Loc_P, \mathcal{S}_P, \mathcal{X}_P$ (defined in subsection 3.2.1). At first we will consolidate the fact that these are all and only the elements and relations on the process relevant for execution: locations, pairs of dual channels (synchronizations), and pairs of synchronizations which share a location, forming a conflict relation. In $\mathscr{G}_P$ the structure of the process is represented via $<_P$ as edges between vertexes (the locations), where locations are duplicated for each decoration, i.e. the synchronizations to which they belong. It is then immediate to identify the full relation $\mathcal{X}_P$.

On the side of ludics we are naively representing all these elements in $\mathcal{D}_P$, forgetting the structure of the process. To each element is associated a *sub design*, a sequence of actions instead of a single one. This is because a design, for the purpose of interaction, should always end with a positive rule. Indeed when reaching a negative rule, the next step is checking the positive directly above, since we are forced to perform a negative rule introducing the negative address, whenever there is any.

Another reason is that the modifications on $\mathcal{D}_P$ needed to shape the restrictions are more natural in this way. If for example $\mathscr{F}(i) = l$, and $G[l]$ were a single (negative) action, we would need to *add* a rule for the daimon in $\mathcal{R}(i)$, above $G[l]$, shifting the correspondence between the base design and the restriction designs. Indeed, the intuition behind $\mathcal{R}(i)$ is that we can

perform execution on i only if l has been already erased; thus execution on l here means reaching the $\maltese$. But, if $G[l]$ were only a single negative action, then interaction would diverge immediately on it.

Lemma 4.2.9. *Loc_P, $<_P$, $\mathcal{S}_P$ and $\mathcal{X}_P$ give necessary and sufficient information about the possible execution paths on P.*

Proof. $\mathcal{S}_P$ is obviously the executable/interactive part of the process. The execution rule tells us that a pair of channels is ready for synchronization if the both channels are *external*, i.e. minimal with respect to $<_P$, which means that they have no prefix. $<_P$ on locations is associated to the prefix order, thus Loc_P is enough to recover this order via the restriction designs.

The other constrain on execution, that generates non-determinism and non-confluence, is the fact that channel names are not unique, and are *consumed* during execution. Therefore a particular synchronization can sometimes prevent others to be executed, generating a permanent fork in the possible execution sequences on the process, i.e. we have non intersecting execution sequences. This is because execution is *resource sensitive*, and channel occurrences can only be used once. Permanent forks in an interaction sequence, which can potentially yield different normal forms, arise only – and *not always* – when a channel is part of one or more synchronizations (or if we have a non deterministic choice +, which we are not considering for now) which thus are in *conflict*: the $\mathcal{X}_P$ relation represents this conflict, and therefore each potential fork during execution.

□

Remark 4.2.10. *The non-deterministic choice + will be represented by an extension of $\mathcal{X}_P$, consistently with its nature as a* conflict *relation – which is used to represent + also in event structures [27].*

$<_P$ (definition 3.2.4) describes the structure of P: the prefix order between channels, and which sub-processes are in parallel composition (lack of order). However, execution does not need the whole partial order, but only a local order relation: a pair of dual locations is ready for synchronization if there is no prefix blocking it. Therefore it suffices to represent it only locally, by checking the *immediate predecessor* of the channel name occurrence in question.

$\mathscr{F}(\cdots)$ (definition 4.1.11) defines a two level alternating dependence relation, still by checking $<_P$ locally: synchronizations are freed by locations, and locations by synchronizations *and* the immediate preceding location. The

main fact is that this is enough to respect the *partial order on synchronizations* during interaction: the full prefix order is then transitively recovered by considering the totality of the possible interactions.

Lemma 4.2.11. *Let P be a MCCS process. The freedom requisites for $\mathcal{S}_P$ and Loc_P correctly respect $<_P$ and the induced order on synchronizations $\preccurlyeq_{\mathcal{S}_P}$, therefore $\mathcal{R}(P)$ forces interaction on $\mathscr{B}_P$ to respect the prefix order as done by execution on processes.*

Proof. Let, without loss of generality, $i = (a^l, \overline{a}^m) \in \mathcal{S}_P$.

- If $\mathscr{F}(i) = \{h, k\}$, with $h, k \in Loc_P$, then $h <_P l$ and $k <_P m$.
- If $h \in j$ and $k \in u$, with $j, u \in \mathcal{S}_P$, we implicitly have $j, u \preccurlyeq_{\mathcal{S}_P} i$. This holds because $j \in \mathscr{F}(h)$ and $u \in \mathscr{F}(k)$.
- The corresponding directed designs $\mathcal{R}(i)$, $\mathcal{R}(l)$ and $\mathcal{R}(m)$ force interaction to respect the (local) order given by freedom conditions: $[j]_P$ must be visited before $[h]_P$ and $[u]_P$ before $[k]_P$, that on their hand must be visited before $[i]_P$, that have precedence on $[l]_P$ and $[m]_P$.
- By transitive closure, $[u]_P$ and $[j]_P$ must be visited before $[i]_P$; $[k]_P$ and $[h]_P$ must be visited before $[l]_P$ and $[m]_P$. Therefore interaction on $\mathscr{B}_P$ respects $<_P$ and $\preccurlyeq_{\mathcal{S}_P}$.

□

We want to obtain a complete *characterization* of execution via interaction between $\mathscr{B}_P$ and $\mathscr{B}_P^\perp$, with the following meaning:

Definition 4.2.12. *Let P be a MCCS process. If each execution $P \to_i$ is* associated *to an interaction path on $[\![P]\!]$, and the* associated execution *of each interaction path on $[\![P]\!]$ is* admissible *(in the sense of definition 4.2.2), we say that $[\![P]\!]$* **characterizes** *execution on P.*

Remark 4.2.13. *What we are describing here is* **not a bijection***: as we already noted, an execution sequence may be associated to multiple interaction paths. This is because of the null interaction steps with respect to execution, i.e. steps that do not visit actions of $G[i]$ for some synchronization i. We could define a general notion of* big step *and an equivalence relation on counter-designs in order to have a bijection, but it would be a variable notion and not needed to form the correspondence we seek.*

What we want to prove is:

Theorem 4.2.14. *Let P be a MCCS process. $[\![P]\!] = (\mathscr{B}_P, [\ \ ]_P)$, characterizes all executions on P.*

We split the proof of the theorem in two main results, the first going *from interaction to execution*, and the second *from execution to interaction*; these results are Corollary 4.2.16 (or its constructive version Lemma 4.2.19), and Lemma 4.2.20.

From interaction to execution

Lemma 4.2.15. *Let P be a MCCS process. It holds that*
1) $C \perp \mathscr{B}_P \Leftrightarrow$ 2) the execution associated to $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$, with $\mathcal{D}_P \perp \mathcal{C}$, is admissible*; i.e. any $G[x]$, with $x \in \mathcal{S}_P$, and any $G[l]$, with $l \in Loc_P$, in $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$ is visited accordingly to $\leqslant_{\mathcal{S}_P}$ and $<_P$, and the interaction never visits both $G[u]$ and $G[v]$ with $(u, v) \in \mathcal{X}_P$.*

Proof. 1) $\Rightarrow$ 2) holds because, as proved in lemma 4.2.11, $\mathcal{R}(P)$ forces interaction to respect $<_P$ and $\leqslant_{\mathcal{S}_P}$, thus the execution *associated* to $\mathcal{C}$ (with respect to $\mathscr{B}_P$), by definition, respects the same order. $\mathcal{R}(\mathcal{X}_P)$ also prevents interaction from visiting both synchronizations of a *xor* clause, since a & forces an exclusive choice in interaction.

2) $\Rightarrow$ 1): We have that $\mathcal{C} \perp \mathcal{D}_P$ (note that this implies that $\mathcal{C}$ ends with a ✠); if furthermore $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$ respects $<_P, <_{Loc_P}$ and $\mathcal{X}_P$, then $C \perp \mathcal{R}(P)$, and thus $\mathcal{C} \in \mathbb{B}_P^\perp = \mathbb{B}_P^{\perp\perp\perp} = (\mathbb{B}_P^{\perp\perp})^\perp = \mathscr{B}_P^\perp$; therefore $\mathcal{C} \perp \mathscr{B}_P$.

□

As an immediate corollary, and as first direction of theorem 4.2.14, we have that (by definition of *admissible execution* 4.2.2):

Corollary 4.2.16. *Let P be a MCCS process. $\forall \mathcal{C} \in \mathscr{B}_P^\perp$, the* execution associated *to $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$ is* admissible *in P.*

and furthermore

Corollary 4.2.17. *Let P be a MCCS process, and $i \in \mathcal{S}_P$. Then i is* principal *(def 3.2.7) if and only if $\exists C \in \mathscr{B}_P^\perp$ such that $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$ visits $G[i]$ before any other $G[j]$ with $j \in \mathcal{S}_P$.*

Proof. Trivially, if i is principal then $\mathscr{F}(i) = \varnothing$, and $\mathcal{R}(i) = \varnothing$, therefore $G[i]$ can be interacted with before any other ramification corresponding to synchronizations. Only the *xor* conditions of i are required to be visited before $G[i]$.

For the other direction, if $\langle \mathcal{D}_{\mathcal{P}} \leftarrow \mathcal{C} \rangle$ visits $G[i]$ before any other $G[x]$, then its associated execution starts with i, and thus $P \rightarrow_i$ is an admissible execution, i.e. i is a principal synchronization. □

The lemma and corollaries assure us that there are no *bad interactions* on $[\![P]\!]$: any orthogonal design always describes an actually possible execution path on P.

Remark 4.2.18. *Note that we indiscriminately talk about interaction and orthogonal design; that is possible due to the lack of distinction of syntax and semantics in ludics. A design explicitly tells us how it interacts: therefore a design* is *an interaction, with respect to its orthogonals.*

Corollary 4.2.16 can be re-phrased as a one-step correspondence between execution and interaction in the following way

Lemma 4.2.19. *Let P be a MCCS process. Then $P \rightarrow_i$ is an admissible execution if and only if $\exists \mathcal{C} \in \mathscr{B}_P^{\perp}$ such that the execution associated to $\langle \mathcal{D}_{\mathcal{P}} \leftarrow \mathcal{C} \rangle$ starts with i.*

Proof. The direction $\Leftarrow$ is the same as in the corollary 4.2.17.

For $\Rightarrow$ we can note that if $P \rightarrow_i$, then i is a principal synchronization, therefore $\mathscr{F}(i) = \varnothing$. Then, we can easily *build* a *minimal* orthogonal design interacting first (and only with) $G[i]$. Let $[i]^{\perp} =$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash [i].1.1, \Delta}\,\maltese}{[i].1 \vdash \Delta}\,(-, i.1, \{\{1\}\})}{\vdash [i], xor^{i_1}.1, \ldots, xor^{i_n}.1, \Delta}\,(+, i, \{1\})}{xor^{i_n} \vdash [i], xor^{i_1}.1, \ldots, xor^{i_{n-1}}\Delta}\,(-, xor^{i_n}, \{\{1\}\})}{\vdots}}{xor^{i_1} \vdash xor^{i_2} \,\&\, xor^{j_2} \ldots, xor^{i_n} \,\&\, xor^{j_n}, [i], \Delta}\,(-, xor^{i_1}, \{\{1\}\})}{\vdash xor^{i_1} \,\&\, xor^{j_1}, \ldots, xor^{i_n} \,\&\, xor^{j_n}, [i], \Delta}\,(+, xor^{i_1} \,\&\, xor^{j_1}, \{xor^{i_1}\})}{\xi \vdash}\,(-, \xi, \{I\})$$

Then $[i]^{\perp} \perp \mathscr{B}_P$ and the execution associated to $\langle \mathcal{D}_P \leftarrow [i]^{\perp} \rangle$ is $\rightarrow_i$.

In Δ must appear the address of each premise of $\mathcal{D}_P$, since the ramifications of $\mathcal{D}_P$ and $[i]^{\perp}$ must match. The sequence $xor^{i_1} \& xor^{j_1}, \ldots, xor^{i_n} \& xor^{j_n}$ includes all the addresses assigned to the clauses $(i, x) \in \mathcal{X}_P$, with x varying on $\mathcal{S}_P$.

□

From execution to interaction

We have thus tied interaction to execution, partially proving theorem 4.2.14. To complete the other direction of the theorem, and complete the correspondence between execution and interaction, we need the following lemma

Lemma 4.2.20. *Let P be a MCCS process and $\vec{i} \in \mathcal{S}_P$, with $\vec{i} = (i_1, \dots, i_n)$. Then $P \to_{\vec{i}}$ is an* admissible *execution if and only if $\exists \mathcal{C} \in \mathscr{B}_P^{\perp}$ such that $\to_{\vec{i}}$ is the* execution associated *to $\langle \mathcal{D}_{\mathcal{P}} \leftarrow \mathcal{C} \rangle$.*

Proof. The direction $\Leftarrow$ holds by corollary 4.2.16.

For $\Rightarrow$ we could be temped to split $P \to_{\vec{i}} P_n$ in $P \to_{i_1} P_1 \to_{i_2} \dots \to_{i_n} P_n$, and apply the one-step correspondence lemma (4.2.19) to each step. However this is not possible, since a standard *subject reduction* theorem[2] *does not hold.* Nothing tells us that if $P \to_i P'$ then $[\![P]\!] = [\![P']\!]$, and indeed it is *not* the case, as we will see in section 5.2. The best approach would be to *build* a **counter design** $\mathcal{C}$ with the intended associated execution. We can do this by recursively building a counter design whose first steps are taken from the one-step lemma.

We proceed by induction on the length of an execution sequence. Let $i = (a^l, \overline{a}^m)$, then:

$$[i]^{\perp} := (+, xor^{i_1} \,\&\, xor^{j_1}, \{xor^{i_1}\})(-, xor^{i_1}, \{\{1\}\}) \dots$$

$$\dots (-, xor^{i_n}, \{\{1\}\})(+, [i], \{1\})(-, [i.1], \{\{1\}\})$$

$$(+, [l], \{1\})(-, [l.1], \{\{1\}\})(+, [m], \{1\})(-, [m.1], \{\{1\}\})$$

where $xor^{i_1} \,\&\, xor^{j_1} \dots xor^{i_n} \,\&\, xor^{j_n}$ are the addresses assigned to the xor clauses for i, and l, m the locations of i. In a more synthetic representation: $\forall x \in \mathcal{S}_P$ such that $(i, x) \in \mathcal{X}_P$, and for $l, m \in i$, we have

$$[i]^{\perp} = w[i, x]^{\perp} * G[i]^{\perp} * G[l]^{\perp} * G[m]^{\perp}$$

where $*$ is the concatenation of rules in ludics.

We can define $\mathcal{E}(\vec{i})$ as the following design (with $(+, \xi, I)$ the first action of $\mathcal{D}_P$):

- $\mathcal{E}(\varnothing) = (-, \xi, \{I\})$
- $\mathcal{E}(i_1, \dots, i_n) = \mathcal{E}(i_1, \dots, i_{n-1}) * [i_n]^{\perp}$

[2] In typing systems is expected that the type of a *program* does not change with reduction on the program, i.e. if $P \to P'$, then $type(P) = type(P')$. This is usually called a *subject reduction* property.

Given an execution $\vec{i}$, let $\mathcal{C}$ be the design $\mathcal{E}(\vec{i}) * \maltese$. Then, by construction, $\mathcal{C} \in \mathscr{B}_P^\perp$ and $\rightarrow_{\vec{i}}$ is the execution associated to $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$.

□

Maximal executions and read-back of the process

With theorem 4.2.14 we have the necessary tool to see what happens with *maximal execution paths* (defined in 3.2.3). A *maximal execution path* on a process P is a reduction sequence that brings P to one of its *normal irreducible forms*, that is a sequence that cannot be extended by further synchronizations. From the correspondence execution-interaction we immediately have:

Corollary 4.2.21. *Let P be a MCCS process, and $\mathcal{C} \in \mathscr{B}_P^\perp$. Then, the execution $\rightarrow_{\vec{i}}$ associated to $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$ is* maximal *if and only if $\neg \exists \mathcal{C}' \in \mathscr{B}_P^\perp$ such that $\rightarrow_{\vec{i}}$ is* properly included *in the execution associated to $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C}' \rangle$; i.e. such that $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C}' \rangle$ extends $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$ by $G[j] \notin \langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$ for some $j \in \mathcal{S}_P$.*

Proof. Let $\mathcal{C} \in \mathscr{B}_P^\perp$ such that its associated execution $\rightarrow_{\vec{i}}$ is *maximal*. Suppose $\exists \mathcal{C}' \mathscr{B}_P^\perp$ that extends $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$ by some $G[j]$. Then by theorem 4.2.14, the execution associated to $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C}' \rangle$ is $\rightarrow_{\vec{i}} \rightarrow_j$; however this is impossible, since $\rightarrow_{\vec{i}}$ is *maximal* by hypothesis.

For the other direction of the implication, by theorem 4.2.14 all executions have an associated $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$, and thus indirectly at least one design $\mathcal{C}$. If there is no $\mathcal{C}' \in \mathscr{B}_P^\perp$ such that its associated execution extends $\rightarrow_{\vec{i}}$, then $\{\vec{i}\}$ is a maximal pairing. □

Another property that we can expect from the interpretation is the ability to deduce, or *read back*, the state of the process, given an interaction path, or just the sequence $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$, for a $\mathcal{C} \in \mathscr{B}_P^\perp$, *without* performing execution on the process. Instead of determining the interpretation after an execution on the process, we want to determine the state of the process after an interaction of the interpretation with an orthogonal design.

We can easily define a *read back* on the interpretation which satisfy this property.

Definition 4.2.22 (read back)**.** *Let P be a MCCS process, $\mathcal{C} \in \mathscr{B}_P^\perp$, and $\rightarrow_{\vec{i}}$ the execution associated to $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$. The* read back *of $\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle$, noted $[\langle \mathcal{D}_\mathcal{P} \leftarrow \mathcal{C} \rangle]$, is*

$$\mathscr{G}_P \setminus \{l \in Loc_P \mid l \in \vec{i}\}$$

where $l \in \vec{i}$ means $l \in j$ for some $j \in \vec{i}$.

The operation we defined is simply a blind erasing on the graph $\mathscr{G}_P$ of the locations labeled with synchronizations part of the execution associated to the interaction we performed (this implies an erasing of all edges to and from the locations in question). We need to prove that the resulting graph is indeed the result of the execution (or reduction in the case of $\mathscr{G}_P$) on $\vec{i}$.

Corollary 4.2.23. *Let P be a MCCS process, and $C \in \mathscr{B}_P^{\perp}$, and $\rightarrow_{\vec{i}}$ the execution associated to $\langle \mathcal{D}_{\mathcal{P}} \leftarrow \mathcal{C} \rangle$. It holds that $\mathscr{G}_P \rightarrow_G^{\vec{i}} [\langle \mathcal{D}_{\mathcal{P}} \leftarrow \mathcal{C} \rangle]$.*

Proof. Reduction on $\mathscr{G}_P$ (definition 4.1.4) is the erasing from the graph of all the nodes corresponding to two chosen locations, who are labeled with the same synchronizationin at least one node (remember definition 4.1.2), and the edges to and from these nodes. Given the sequence $\vec{i} = i_1, \ldots, i_n$ we know that i_1 is a *minimal* synchronization, since $\langle \mathcal{D}_{\mathcal{P}} \leftarrow \mathcal{C} \rangle$ respects $\preccurlyeq_{\mathcal{S}_P}$; therefore we can just erase the nodes corresponding to the locations $l, m \in i_1$ from $\mathscr{G}_P$, and their edges. We can repeat the same reasoning on $i_2, \ldots, i_n$ until we have erased all the nodes labeled with locations of synchronizations in $\vec{i}$ obtaining, at each step, a valid reduction sequence (at each step we are erasing *pairs of minimal locations*). The graph resulting from this reduction is, by definition, the read back $[\langle \mathcal{D}_{\mathcal{P}} \leftarrow \mathcal{C} \rangle]$.

□

Discussion

Theorem 4.2.14 is thus proven, and the correspondence well-defined. This still opens a few interesting questions are about the limit cases of execution: **deadlocks**, and the reduction to the empty process **1**. The latter is closely tied to what will be developed in section 5.2, where the interpretation associated to the empty process **1** will have a precise form in ludics, that is the design $\mathbb{O}ne$ (defined in example 2.2.22). Deadlocks will be studied in section 5.3, by using the notions of *incarnation* and *material design* (definition 2.2.23), that let us isolate in designs of a behaviour only their essential part with respect to interaction, i.e. the branches actually visited by some interaction with the orthogonal behaviour.

5. Properties of the Interpretation

This chapter is divided into four main parts, where each part is dedicated to prove a different property of the interpretation of $MCCS$ processes in ludics. Most of the properties rely on defining *operations* on the interpretation $[\![P]\!]$ of a $MCCS$ process P, to then be able to reflect the *results* of these operations back at the process, to represent fundamental constructors of the syntax of $MCSS$, or simply check interesting properties of P directly on its interpretation in ludics.

The objective of the first part, section 5.1, is to define an *operation on* $[\![P]\!]$ that *interprets* the **parallel composition** of two processes, making the interpretation *modular*. This operation will be defined on the set of generators, and is based on the merging of designs $\odot$ (definition 2.2.29), taht is the core operation of $\otimes$ (definition 2.2.30) on behaviours. The goal is to be able to represent the *parallel composition* of processes via a composition of their *interpretations*, such that this composition is carried out as much as possible via logical operations. As we will see, this is partially achieved.

Since it is often chosen as a fundamental operation of process algebras that *enables communication* between two processes, being able to represent in ludics parallel composition is an important step of the interpretation to deepen the connection between the two systems.

The *main results* of section 5.1 are the *definition* of the operation $\oplus$ *on the interpretation of processes* $[\![P]\!]$, $[\![Q]\!]$ (definition 5.1.16), called **merging**, that interprets the parallel composition $P \mid Q$, and *proof that it commutes with the interpretation operation* (theorem 5.1.18), i.e. that

$$[\![P \mid Q]\!] = [\![P]\!] \oplus [\![Q]\!].$$

In the trivial case, when composing processes that *cannot* communicate, there is a perfect match between parallel composition and the $\otimes$ on behaviours:

$$[\![P \mid Q]\!] = [\![P]\!] \bigotimes [\![Q]\!]$$

however many difficulties arise when two processes that have synchronizable pairs of channels are put together: a tensor-like operation cannot handle the increased complexity of two communicating processes, and some artificial steps are required to make it work while keeping the core of the operation – the *merging* of designs $\odot$ – intact.

We start the section with a discussion on what *cannot work* as a ludics counter-part of parallel composition on processes. Then, a few basic operations are defined on *designs*, that lead to the notion of **compatibility** with respect to the *merging of interpretations*: the notion express the fact that two designs can be merged without falling into inconsistencies.

These basic operations are then extended in order to consider the *new elements* that are generated in the *parallel composition* of two processes P, Q, and is applied to the interpretations $[\![P]\!], [\![Q]\!]$, in the definition of $\oplus$. Eventually, the theorem on $\oplus$ mentioned above is proved, by considering a few intermediate lemmas.

In section 5.2 we try to answer the question of *how the interpretation of an MCCS process change* when *execution* is performed on the process. In particular, *what are the differences* between the interpretation of a *MCCS* process P, and one of its reduced forms P' after one execution step, and if it is possible to pass from $[\![P]\!]$ to $[\![P']\!]$ via an *operation on the interpretation*, without the need to get back at the starting process P.

The *main result* of this section is the definition of a *reduction on the interpretation* (definition 5.2.11) $[\![P]\!]$ of a *MCCS* process P that *matches execution* on processes. This operation is a slight extension of an *already existing ludical operation* called **projection** (found already in Girard's seminal article [38]), that *commutes with the bi-orthogonal closure*, therefore can be performed either on *the set of generators* or the associated *behaviour*. We call this operation *reduction* since it actually *erases specific chronicles* from the designs of the set or behaviour, which thus have a reduced form. The projection operation, once applied to our interpretation, requires as a slight extension that the same erasing it performs on the behaviour must be applied also to the *assignment function* of $[\![P]\!]$.

The reduction operation gives us a nice *commutative property* (theorem 5.2.12) between *execution*, *reduction* on $[\![P]\!]$, and the *interpretation operation* from P to $[\![P]\!]$: given a *MCCS* process P and a *principal* synchronization u (whose channels are *minimal* with respect to the *prefix order*), then we can pass to the interpretation $[\![P]\!]$, and perform *reduction on* $[\![P]\!]$ *based on* u, to *obtain as result the interpretation* $[\![P']\!]$, where $P \rightarrow_u P'$ is a *valid execution*.

This means that, given an *MCCS* process P, we can deduce from its interpretation $[\![P]\!]$ *all of its reduced and normal forms* via reduction on $[\![P]\!]$,

without performing execution on P, all we need to know is the *partial order* $<_P$ to identify *principal synchronizations*, that we have coded in $[\![P]\!]$ anyway.

By matching with the reduction on $[\![P]\!]$ *execution paths* that lead P to the *empty process* 1 (if there is one such execution path), we obtain a remarkable connection with *linear logic*, that also explains why we call the empty process 1 instead of the standard 0. Indeed, in subsection 5.2.1 we show that performing successive reductions on $[\![P]\!]$, matching an *execution path* $\rightarrow_{u_1} \cdots \rightarrow_{u_n}$ such that $P \rightarrow_{u_1} \cdots \rightarrow_{u_n} 1$, has as result the *behaviour* generated by the design $\mathbb{O}ne$ (defined in example 2.2.22), that is denoted 1, being the interpretation of the *linear logic multiplicative unit* 1.

In section 5.3 we study the topic of **deadlocked processes**, and try to find a few properties on the interpretation $[\![P]\!]$ that can tell us something relevant on possible *deadlocks* in a ($MCCS$) process P.

We start with the definition of **normal form** (definition 5.3.1) of a process P, then give the definition of *deadlocked process* (definition 5.3.5).

In subsection 5.3.1 is shown what a deadlock in a process P entails regarding interaction and the *restriction designs* $\mathcal{R}(P)$: *cycles* in the partial order $<_P$ and the induced order on synchronizations $\preccurlyeq_{\mathcal{S}_P}$ means that we can find in the restriction designs requirements *impossible to satisfy*. Then we define a simple order relation on restriction designs that follows $\preccurlyeq_{\mathcal{S}_P}$ to identify deadlocks, which is then *dismissed as not relevant* since it is an extra-ludical and extra-logical property, that is no different than looking directly at the order relations on the process itself.

Therefore, in subsection 5.3.2 we start analyzing the *interpretation* $[\![P]\!]$ in search of a property that *can tell us when a process* P *is deadlock-free*. The answer is found in the *material design* (definition 2.2.23) of the *base design* $\mathcal{D}_P$, that can give us a sufficient *but not necessary* condition to decide if the associated $MCCS$ process P is *deadlock-free* (lemma 5.3.10); however, it may be the case that P *is deadlock-free*, but the sufficient condition *is not satisfied*.

In subsection 5.3.3, a particular case of deadlocks is considered, that is when they are *superficial*, meaning that the *channels involved in the cycle* are *not blocked by a prefix*, and are *directly accessible by the environment*, as for instance a process in *parallel composition*. In this limited case, we attempt to find a way to *solve* them on the interpretation: that means be able to *identify* them and *add a minimal context*, via *merging of interpretations*, that will let the process *unlock* and continue execution. A partial solution is eventually given in 5.3.19, despite being practically un-viable due to its its complexity.

In section 5.4, we extend the interpretation by representing the *action prefix* constructor of the syntax of CCS (the basic process constructor, that let us extend P to $a.P$) as an *operation* on the ludical interpretation of processes. The *result* is *definition* 5.4.2, that exploits the same ideas of the *merging of interpretations* with slight modifications, to achieve the same goal.

At last, in section 5.5, we examine the case of *independent synchronizations* – synchronizations that are *not a pair* belonging to $\preccurlyeq_{\mathcal{S}_P}$, so they can be performed in *any order* – and reductions (execution paths or single steps that *commute*): what causes *forks* in execution paths of $MCCS$ processes and CCS processes (also considering the *non deterministic choice* $+$), and when these forks are *not permanent*, meaning that there are further *execution steps* that actually bring back two execution paths of a process P at the *same reduced form* (thus possibly at the same *normal form*).

In the case of *non-permanent* forks caused by independend synchronizations of a $MCCS$ process P, we give a *diamond property* in lemma 5.5.2, proved by using *different successive reduction steps on the interpretation* $[\![P]\!]$: the conclusion is that it doesn't matter in which order independent synchronizations are executed, the resulting form of P is the same, as its *interpretation* in ludics.

5.1 Representing parallel composition

Let P, Q be $MCCS$ processes, and $[\![P]\!]$, $[\![Q]\!]$ their respective interpretations in ludics. The aim is to build the interpretation of $P \mid Q$, $[\![P \mid Q]\!]$, from $\mathscr{B}_P$, $\mathscr{B}_Q$ and the assignments $[\ ]_P$, $[\ ]_Q$, in the most natural way possible; therefore to define some operation $[\![P]\!] \oplus [\![Q]\!]$ on interpretations that reflects $P \mid Q$, such that $[\![P]\!] \oplus [\![Q]\!] = [\![P \mid Q]\!]$.

The ideal result would be to be able to do this *without touching the processes at all*; so, given the interpretation for P and Q, be able to build the interpretation of $P \mid Q$ without any kind of additional information on P and Q (only what their interpretations tells us). To deduce the possible communications between P and Q, we need to extract the information about *channel names* of P and Q, and *how many times they occur*: the only way to deduce the information without looking at the processes is from the assignment functions $[\ ]_P$ and $[\ ]_Q$, whose domains include $\mathcal{S}_{P|Q}$ and $Loc_{P|Q}$.

Since new synchronizations, and thus execution paths, may arise in the parallel composition (indeed it is what makes execution possible in the first place, and characterize concurrent behaviour) we need to read-back from the assignments the information about the process: how many occurrences of each channel name there are in each process. By checking the domain of the

assignments we can recover the information we need, since channel names labeled by locations are included.

When we put two processes P and Q in parallel composition, however, a *renaming* of their locations may be required in case of superimposition, since they must be unique. This does not change the channel they label, and neither the addresses assigned to each occurrence by the assignment. A renaming could also be required on the addresses of the base designs $\mathcal{D}_P$ (or $\mathcal{D}_Q$), to induce the same substitution in the co-domain of $[\ \ ]_P$ (or $[\ \ ]_Q$) – that is, the *ramifications* of the base designs – and avoid superimposition between $[\ \ ]_P$ and $[\ \ ]_Q$. Substituting an address induce an hereditary substitution also on its sub-addresses and the chronicles generated from there.

We may start by defining *base unification*, i.e. a renaming of the addresses in the bases of two designs, to make them *compatible* for the *merging* $\odot$ (definition 2.2.29). Recall that the operation is defined for designs with *disjoint* ramifications, but same base: an unification of the addresses of the bases, and then a renaming of the ramification (of the first rule) of one of the two designs will make the merging always possible.

Definition 5.1.1 (Unification)**.** *Let $\mathcal{C}$ be a design of base $\xi_0 \vdash \xi_1, \ldots, \xi_n$ and $\mathcal{D}$ a design of base $\zeta_0 \vdash \zeta_1, \ldots, \zeta_n$ of the same arity. The* unification *(of the bases) of $\mathcal{C}$ and $\mathcal{D}$ is either the design $\Theta\mathcal{C}$ or $\Theta\mathcal{D}$, result of a* renaming function on addresses *(called* unification *function) Θ such that:*

$$\Theta(\xi_i) = \zeta_i \ (\textit{or}\ \Theta(\zeta_i) = \xi_i),\ 0 \le i \le n :$$

and for each sub-address ξ_i* of ξ (ζ_i* of ζ):

$$\Theta(\xi_i*) = \Theta(\xi_i)* \quad (\Theta(\zeta_i*) = \Theta(\zeta_i)*).$$

The unification function is no more than a substitution of one or more addresses in the whole design, which preserves the polarity and ramifications of its actions. It obviously suffices to perform it on one of the two designs, renaming addresses of only one of the two bases.

5.1.1 Assignment on the interpretation

When we build the interpretation of a process P we assign to each location $l \in Loc_P$ an address $[l]_P$. These addresses can be renamed at any time in the construction of $\mathcal{D}_P$, but each location corresponds to a different address, as well as a channel name occurrence, thus we can explicit the information, and consider the assignment to be, for example, $[a^l]_P = \xi.1$.

The assignments of two processes P and Q are enough to derive the conflict relation $\mathcal{X}_{P \mid Q}$, and, of course, the set of synchronizations $\mathcal{S}_{P \mid Q}$: we do not need the structure the two processes, which is the *partial order* on locations, i.e. the *prefix order* on channels, but only the respective sets of channel names occurrences. By combining these two sets, we can easily deduce the *new synchronizations* generated in the parallel composition. Freedom requisites (definition 4.1.11), and hence the restriction designs (definition 4.1.15), regarding these new synchronizations, come naturally from the partial orders $<_P$ and $<_Q$, which are preserved in the composition; indeed $<_{P \mid Q} = <_P \cup <_Q$, since there are no new channels. Then, the freedom requisites of a new synchronization will be the locations (possibly only one) which are already a requisite for the locations labeling the channels, one of P and one of Q, that form that new synchronization (the very preceeding locations). The requisites for locations are explicited in $\mathcal{R}(P)$ and $\mathcal{R}(Q)$ (the set of restriction designs of, respectively, $[\![P]\!]$ and $[\![Q]\!]$), thus on their respective interpretations, letting us recover the information we need.

Note that this is the *tricky step* of the result we want to prove: we *need the prefix order* $<_P$ of the processes in order to find the *freedom requisites* of the *new synchronizations* arising in the parallel composition of P and Q. This partial order can be *indirectly deduced* by looking at the *restriction designs* $\mathcal{R}(P)$ and $\mathcal{R}(Q)$, or, equivalently, by looking at the *interaction paths* between $\mathscr{B}_P, \mathscr{B}_Q$ and their respective orthogonals: if a certain *branch* is consistently visited after another, i.e. there is no interaction where it is visited before the other, then we can conclude that the first branch has precedence on the second, and find an order on the *corresponding elements of the process P (or Q), that is* assigned to that branch. Obviously checking *all the interaction paths* is a very long process, however we *do not need* the processes P and Q to recover their structure, since it is indirectly coded *inside interaction*, via the *restriction designs*.

While it might seem an equivalent operation, which means that we can't get the information needed to represent the parallel composition $P \mid Q$ without looking at P and Q directly, once the interpretations $[\![P]\!]$ and $[\![Q]\!]$ are built we can actually *forget* about P and Q, and just look at their interpretations. The only downside of this operation is that we *need the extra-ludical information* of the *assignment functions* $[\ \ ]_P$ and $[\ \ ]_Q$ attached to the behaviours $\mathscr{B}_P$ and $\mathscr{B}_Q$.

5.1.2 Renaming and rewriting

Base unification is meant to be just a tool to simplify the **merging of designs** (definition 2.2.29); we want to always be able to merge two *base de-*

signs, since we can always compose two processes via parallel composition. Intuitively, it is a rather natural operation, even if it does require a *rewriting* of one of the designs involved; still, this rewriting does not affect the part of the designs relevant for the correspondence and interpretation of a process, which is its structure.

Remember that designs are *abstraction* of sequent calculus proofs (subsection 2.2.2), where addresses name *places* in the proof. Thus $\xi 1$ means "the place of the first premise generated by ξ"; the suffix 1 however only affects our logical interpretation of the action (since it can be seen as *the first sub formula* of the conclusion), not the interaction of the design. If, for example, the first (negative) premise of a positive rule on $\vdash \xi$ is $\xi 3 \vdash$, then we can see that as a double $\oplus$ rule with the first $\oplus$ done on $\xi 3$, thus hiding both $\xi 1$ and $\xi 2$. However, what matters is its *place* in the designs with respect to the other addresses, and the design *structure*: the cardinality of each ramification, the number of rules, and occurrences of daimons.

In our case the actual address matters even less, since we base our interpretation of the design and its interactions on the assignment function. What we need thus is the cardinality of the ramification, one premise/sub-design for each element we are coding, and that each sub-design has the structure intended for the element to which it correspond. For instance, if $\xi 3 = [(u, v)]_P$ with $(u, v) \in \mathcal{X}_P$, what matters to us is that $\xi 3$ is the address assigned to the clause (u, v), and that the corresponding chronicle is of the form $(-, xor_u \,\&\, xor_v, \{\{1\}, \{2\}\})(+, xor_u = \xi 31, \{1\})\cdots$.

Regarding interaction, the structure of the orthogonals does not change when performing a renaming, only the displayed addresses change; for example if a design $\mathcal{D}$ is of base $\vdash \xi$, then $\mathcal{D}[^{\sigma}/_{\xi}]$ if of base $\vdash \sigma$. This substitution changes all the prefixes on the chronicles (the branches of a rule, definition 2.2.6) of the ramifications of $\vdash \xi$, but the design preserves its structure and meaning, i.e. its interactions.

However, if this substitution is not performed on an address of the *base*, but on the *suffix numbers* of its sub addresses, then it is indeed a *rewriting* of the design. This is another necessary step to ensure the compatibility of two designs for merging: the ramifications of their respective first actions must be *disjoint*. This means that we must change the elements in the ramification I of the action *justifying* the sub-addresses in the premises, even though not its cardinality. The renaming must still be *coherent* with the focus of the action (it must still be a sub address), therefore we can see it as a renaming of the elements of I. For instance if $\mathcal{D} = \dfrac{\dfrac{\vdots}{\xi 1 \vdash} \quad \dfrac{\vdots}{\xi 2 \vdash} \quad \dfrac{\vdots}{\xi 3 \vdash}}{\vdash \xi}\,(+, \xi, I = \{1, 2, 3\})$

Then $\mathcal{D}[^{\sigma}/_{\xi}]$ is $\dfrac{\begin{array}{ccc}\vdots & \vdots & \vdots\\ \sigma.1\vdash & \sigma.2\vdash & \sigma.3\vdash\end{array}}{\vdash\sigma}\,(+,\sigma,I=\{1,2,3\})$

and $\mathcal{D}[^{J=\{4,5,6\}}/_{I}]$ is $\dfrac{\begin{array}{ccc}\vdots & \vdots & \vdots\\ \xi 4\vdash & \xi 5\vdash & \xi 6\vdash\end{array}}{\vdash\xi}\,(+,\xi,J=\{4,5,6\})$

In the first case we performed a substitution of the address in the base, prefix of the addresses generated by the first rule, in the second the ramification; however the relevant structure of the design has not changed.

With this intuition in mind, we can formally define this operation on designs *and* on the assignment attached to the interpretation of a process. A renaming on the co-domain of an assignment $[\ \ ]_P$ (the suffixes of the ramification of the *first rule* of $\mathcal{D}_P$) affects the corresponding base design as a substitution of the ramification; then the operation of *merging* becomes extremely natural: we can *identify* the bases of two (positive) designs by unification (say $\vdash\xi$), substitute the ramification of one if needed, to make the two *disjoint*, then put all the **negative chronicles** of the two designs together as **premises of a single positive rule introducing the base** ξ. The result is the union of the ramifications, that are no more than (*finite* in our case) sets of natural numbers. In this way the operation $\odot$ will always be well defined on base designs, and thus by extension on the whole behaviours $\mathscr{B}_P$ and $\mathscr{B}_Q$, for two processes P and Q. Before formulating the definition, by putting together the renaming and the merging, we give a short example of what it looks like:

Example 5.1.2. *Let* $\mathcal{C}=\dfrac{\begin{array}{cc}\vdots & \vdots\\ \xi 1\vdash & \xi 3\vdash\end{array}}{\vdash\xi}$, $\mathcal{D}=\dfrac{\begin{array}{ccc}\vdots & \vdots & \vdots\\ \zeta 1\vdash & \zeta 2\vdash & \zeta 3\vdash\end{array}}{\vdash\zeta}$

Let Θ *be an unification of the bases of* $\mathcal{C}$ *and* $\mathcal{D}$ *performed on* $\mathcal{D}$*. We have*

$\Theta\mathcal{D}=\dfrac{\begin{array}{ccc}\vdots & \vdots & \vdots\\ \xi 1\vdash & \xi 2\vdash & \xi 3\vdash\end{array}}{\vdash\xi}$

Then, to make $\mathcal{C}$ *and* $\Theta(\mathcal{D})$ compatible *for merging, we need to rewrite one of the ramifications, to make them disjoint. Let* $I=\{1,2,3\}$ *the ramification of the first action of* $\mathcal{D}$*, then a* rewriting *(of the ramification) of* $\mathcal{D}$ *is*

$\mathcal{D}[^{J=\{4,5,6\}}/_{I}]=\dfrac{\begin{array}{ccc}\vdots & \vdots & \vdots\\ \xi 4\vdash & \xi 5\vdash & \xi 6\vdash\end{array}}{\vdash\xi}$

This let us easily perform the merging *of* $\mathcal{C}$ *and* $\Theta\mathcal{D}$*:*

$$\mathcal{C} \odot (\Theta\mathcal{D}) = \dfrac{\dfrac{\vdots}{\xi 1 \vdash} \quad \dfrac{\vdots}{\xi 3 \vdash} \quad \dfrac{\vdots}{\xi 4 \vdash} \quad \dfrac{\vdots}{\xi 5 \vdash} \quad \dfrac{\vdots}{\xi 6 \vdash}}{\vdash \xi} (+, \xi, \{1, \ldots, 6\})$$

Remark 5.1.3. *Recall that the* $\odot$ *operation is performed on designs of two* disjoint *behaviour of same base (with no superimposition of their* **directories***, definition 2.2.24); the role of the renaming and unification is to create the right conditions for the operation to always succeed, without compromising the correspondence between a process and its interpretation.*

Assignment renaming

Unification of bases and substitution of the ramification of a design is an operation applicable to two generic design to make them compatible for merging. Another route to achieve the same result, but specific to our interpretation, is to work also on the assignment functions $[\ ]_P$ and $[\ ]_Q$. A renaming of the target addresses (the co-domain) of an assignment induces a corresponding renaming on the addresses of the base design; for instance, it suffices a simple forward shift of the numbers of the ramification.

The renaming of an assignment $[\ ]_Q$ with respect to another assignment$[\ ]_P$ is the following:

Definition 5.1.4 (Assignment renaming)**.** *Let* P*,* Q *be MCCS processes,* $\mathcal{D}_P$ *and* $\mathcal{D}_Q$ *have the same base, and let* $(+, \xi, I)$*,* $(+, \xi, J)$ *be their respective first actions. The* renaming *of* $(\mathcal{D}_Q, [\ ]_Q)$ *with respect to* $(\mathcal{D}_P, [\ ]_P)$ *consists of the following steps:*

1. *A renaming of locations in the domain of* $[\ ]_Q$ *such that* $Loc_P \cap Loc_Q = \varnothing$*. This is a necessary step when composing two processes* P *and* Q *in parallel composition, since locations must be unique.*

2. *A substitution of the suffixes of addresses in the* output *of* $[\ ]_Q$*, denoted* $Output([\ ]_Q)$*, such that* $[\ ]_P \cap [\ ]_Q = \varnothing$*, i.e. the addresses assigned to elements of* Q *may be changed in order to avoid conflicts – no address must be in common between the outputs of* $[\ ]_P$ *and* $[\ ]_Q$ *– while keeping the* same prefix ξ*.*

3. *The ramification (of the first action) of* $\mathcal{D}_Q$ *must be rewritten, in order to be coherent with the assignment* $[\ ]_Q$*. So the new ramification* J^* *must be such that* $Output([\ ]_Q) = J^*$*, and* $I \cap J^* = \varnothing$*.*

A renamed *pair* $(\mathcal{D}_Q, [\ ]_Q)$ *is denoted* $(\mathcal{D}^*_Q, [\ ]^*_Q)$*.*

Remark 5.1.5. *The simple reason why renaming is performed on one pair,* with respect to another*, is because it suffices to act on only one assignment function in order to make the two designs compatible; therefore it is equivalent to perform the renaming on one pair or the other.*

The requisites that two (positive) designs $\mathcal{D}$ and $\mathcal{C}$ need to satisfy in order to be *compatible* for merging are:

- $\mathcal{D}$ and $\mathcal{C}$ must have the same base.
- Let $(+, \xi, I)$ be the first rule of $\mathcal{D}$ and $(+, \xi, J)$ be the first rule of $\mathcal{C}$, then $I \cap J = \varnothing$.

  Note that our condition of *compatibility* is a particular case of *alienation*. Recall that two behaviours of the same base are *alienated* (definition 2.2.26) if the *reunion* of the ramification of the first action of their designs are *disjoint*; however in the case of a *connected positive behaviour* (definition 2.2.21), this is no different than having disjoint directories.

When merging the interpretation of two processes P and Q we can distinguish two cases:

1) $\mathcal{A}_P \cap \mathcal{A}_Q = \varnothing$ (the set of *channel names* are disjoint); the two processes have no common channel name, and thus no new synchronization arises from the parallel composition $P \mid Q$.
2) The intersection is not empty, and new synchronizations are generated.

The difficulty here is obviously case 2), which forces us to add new elements not already present in $\mathcal{D}_P$ and $\mathcal{D}_Q$. However, via **merging of designs**, in the first case we can define an operation working exactly as the $\otimes$ on behaviours, and add some more extra-ludical steps in the second case to account for the new elements, while keeping the $\odot$ operation on *base designs* intact.

Let us first look at $P \mid Q$, and what are the consequences of the parallel composition:

- $<_P$ and $<_Q$, the two respective partial orders on locations, are preserved, since *no new location* is added (assuming the *renaming* on locations of P or Q to preserve their uniqueness), so $<_{P \mid Q} = <_P \cup <_Q$ and $Loc_{P \mid Q} = Loc_P \cup Loc_Q$.
- There may be new synchronizations between channels of P and channels of Q, preserving all their previous ones. So $\mathcal{S}_P \cup \mathcal{S}_Q \subseteq \mathcal{S}_{P \mid Q}$ as well as $\mathcal{X}_P \cup \mathcal{X}_Q \subseteq \mathcal{X}_{P \mid Q}$.

- The new synchronizations *extend* the *freedom requisites* of their locations – since there are more ways to delete them through execution. The *freedom requisites* of these new synchronizations, instead, are always already existing locations – we can determine which they are by looking at $<_P$ and $<_Q$ *locally*, thus at the *restriction designs* $\mathcal{R}(P)$ and $\mathcal{R}(Q)$ or, as we already noted, at the *interaction paths* in $\mathscr{B}_P$ and $\mathscr{B}_Q$.

The new objects that need to appear in the merging can be captured by the sets:

$$new\mathcal{S}_{P \mid Q} = \mathcal{S}_{P \mid Q} \setminus \mathcal{S}_P \cup \mathcal{S}_Q$$

$$new\mathcal{X}_{P \mid Q} = \mathcal{X}_{P \mid Q} \setminus \mathcal{X}_P \cup \mathcal{X}_Q.$$

We have that $new\mathcal{S}_{P \mid Q}$ is the set of synchronizations $i, j, \ldots$ that are generated in the parallel composition; so the pairs $(a^l, \overline{a}^m)$ such that, w.l.o.g., $a \in P$ and $\overline{a} \in Q$. $new\mathcal{X}_{P \mid Q}$ by definition is the set containing only the new xor clauses: pairs of synchronizations (u, v) with a common location, such that u or v belongs to $new\mathcal{S}_{P \mid Q}$ (therefore the clause was not in $\mathcal{X}_P$ or $\mathcal{X}_Q$).

While the definitions holds, we want a way to recover these sets *without looking* at (the structure of) $P \mid Q$. We already pointed at the solution, that is to just look at the domain of the assignments $[\ ]_P$ and $[\ ]_Q$ (which includes each channel name occurrence, via their locations, of P and Q): to recover $new\mathcal{S}_{P \mid Q}$, it suffice to take each pair $(a^l, \overline{a}^m)$ such that. w.l.o.g., $a^l \in Domain([\ ]_P)$ and $\overline{a}^m \in Domain([\ ]_Q)$, as showed in the next two lemmas (for brevity we will use $Dom([\ ]_P)$ to denote the domain of an assignment function).

Lemma 5.1.6. *Let P, Q be MCCS processes. It holds that*

$$new\mathcal{S}_{P \mid Q} = \mathcal{S}_{P \mid Q} \setminus \mathcal{S}_P \cup \mathcal{S}_Q = \{(a^l, \overline{a}^m) \mid a^l \in Dom([\ ]_P), a^l \in Dom([\ ]_Q)\}.$$

where $Dom([\ ]_P)$ is the domain *of the assignment function $[\ ]_P$ and $Dom([\ ]_Q)$ is the* domain *of the assignment function $[\ ]_Q$.*

We skip the trivial proof, by *definition* of $new\mathcal{S}_{P \mid Q}$, $[\ ]_P$ and $[\ ]_Q$.

The set $new\mathcal{X}_{P \mid Q}$ instead includes all the pair of synchronizations (u, v) where u and v share a channel name occurrence, and such that either u or v are in $new\mathcal{S}_{P \mid Q}$:

Lemma 5.1.7. *Let P, Q be MCCS processes. It holds that*

$$new\mathcal{X}_{P \mid Q} = \mathcal{X}_{P \mid Q} \setminus \mathcal{X}_P \cup \mathcal{X}_Q =$$

$$= \{(u, v) \mid u \in new\mathcal{S}_{P \mid Q} \vee v \in new\mathcal{S}_{P \mid Q} \wedge u, v \in \mathcal{S}_P \cup \mathcal{S}_Q \cup new\mathcal{S}_{P \mid Q}\}.$$

Again, a trivial proof by definition of $new\mathcal{X}_{P \mid Q}$ and by the previous lemma 5.1.6.

Therefore, if we have $[\ \ ]_P$ and $[\ \ ]_Q$ we can always assume to have $new\mathcal{S}_{P \mid Q}$ and $new\mathcal{X}_{P \mid Q}$, which can be build by a local assignment check (the full structure of the two processes remains unknown, despite being *recoverable*).

We have defined the *renaming* on a specific pair $(\mathcal{D}_P, [\ ]_P)$ with respect to another pair $(\mathcal{D}_Q, [\ ]_Q)$; however, with no need to internalize this operation in the operation that will be the *merging of interpretations*, we can define it for two designs satisfying the requisites of *compatibility*, and then for their possible assignments.

As a last ingredient, we need to define a design containing the elements of $new\mathcal{S}_{P \mid Q}$ and $new\mathcal{X}_{P \mid Q}$, which will be a third member of a merging of designs including the *base designs* of two processes P and Q, in the non-trivial case where these sets are not empty.

Definition 5.1.8. *Given two MCCS process P and Q, let $(+, \xi, I)$ and $(+, \xi, J)$ be the first action of respectively $\mathcal{D}_P$ and $\mathcal{D}_Q$, then $\mathcal{N}_{P \mid Q}$ is the following design:*

$$\frac{G[k_1] \quad \cdots \quad G[k_n] \quad w[x_1, y_1] \quad \cdots \quad w[x_n, y_n]}{\vdash \xi} \ (+, \xi, N)$$

where $\{k_1, \ldots, k_n\} = new\mathcal{S}_{P \mid Q}$, $\{(x_1, y_1), \ldots, (x_n, y_n)\} = new\mathcal{X}_{P \mid Q}$ and $N \cap I = N \cap J = \varnothing$.

We directly explicit the elements assigned to addresses of the ramification of $\mathcal{N}_{P \mid Q}$, since the actual suffixes are not important for its construction. With $[\ \ ]_{\mathcal{N}}$ we note the assignment of $new\mathcal{S}_{P \mid Q}$ and $new\mathcal{X}_{P \mid Q}$ to their corresponding addresses introduced by the ramification N.

Note that $\mathcal{N}_{P \mid Q}$ is **compatible** with $\mathcal{D}_P$ and $\mathcal{D}_Q$ by construction.

5.1.3 Merging of interpretations

As a general case, we may start with the definition of merging of base designs, which is the first step to define the more general *merging of interpretations*. We have:

Definition 5.1.9. *Let P, Q be MCCS processes, and $\mathcal{D}_P, \mathcal{D}_Q$ their respective base designs. Then the* merging *of $\mathcal{D}_P$ and $\mathcal{D}_Q$ is*

$$\mathcal{D}_{\mathcal{P}} \odot \mathcal{D}^*_{\mathcal{Q}} \odot \mathcal{N}_{\mathcal{P} \mid \mathcal{Q}}$$

where $\mathcal{D}^*_Q$ *is the* renaming *of* $(\mathcal{D}_Q, [\ ]_Q)$ *with respect to* $(\mathcal{D}_P, [\ ]_P)$ *(definition 5.1.4).*

The assignment associated to the merging is

$[\ ]_{P\odot Q} = [\ ]_P \cup [\ ]_{Q^*} \cup [\ ]_{\mathcal{N}}$ *(the union is already disjoint by construction).*

We note $\mathcal{D}_{\mathcal{P}} \odot \mathcal{D}^*_{\mathcal{Q}} \odot \mathcal{N}_{\mathcal{P}\ |\ \mathcal{Q}}$ *with* $\mathcal{D}_P \bigodot \mathcal{D}_Q$.

As an immediate lemma we have

Lemma 5.1.10. *Let* P, Q *be MCCS processes. Then* $\mathcal{D}_P \bigodot \mathcal{D}_Q = \mathcal{D}_{P\ |\ Q}$.

Proof. Since we put together the branches of $\mathcal{D}_D$ and $\mathcal{D}_Q$, that appear in $\mathcal{D}_{P\ |\ Q}$, it suffices to show that $\mathcal{N}_{P\ |\ Q}$ has the ramifications corresponding to the elements of $P \mid Q$ missing from either P or Q, i.e. the new synchronizations and their *xor* conditions. Trivially, it holds by construction of $\mathcal{N}_{P\ |\ Q}$, and definition of $new\mathcal{S}_{P\ |\ Q}$ and $new\mathcal{X}_{P\ |\ Q}$.

□

In the limit case where $\mathcal{N}_{P\ |\ Q} = \varnothing$, which means that there are no new synchronizations, or simply no communication, between P and Q, the merging can be defined as just the standard operation $\bigotimes$, defined on alienated behaviours of same base. Recall that the operation is defined thus:

$$\mathscr{B} \bigotimes \mathscr{C} = \{\mathcal{D} \odot \mathcal{C} \mid \mathcal{D} \in \mathscr{B}, \mathcal{C} \in \mathscr{C}\}^{\perp\perp}$$

Remark 5.1.11. *If we look at the case* $\mathcal{R}(P) \odot \mathcal{R}(Q)$ *we may think that the merging of two restriction designs nullifies the role and meaning of the restriction itself, introducing* ✣ *and prunings* ^{p} *where they are not supposed to be. However these merging are actually* irrelevant *if we look at the final result. That is because the merged designs* $\mathcal{R}(P) \odot \mathcal{D}_Q$ *on one hand, and* $\mathcal{D}_P \odot \mathcal{R}(Q)$ *on the other, actually take over* $\mathcal{R}(P) \odot \mathcal{R}(Q)$*: the requisite they force on interaction will still hold in the final behaviour, since the requisite expressed by* $\mathcal{R}(Q)$ *and* $\mathcal{R}(P)$ *are unaffected by the merging with the base design of the respective other process (*$\mathcal{D}_P$ *and* $\mathcal{D}_Q$*). The designs resulting in the merging* $\mathcal{R}(P) \odot \mathcal{R}(Q)$ *do not generate conflict, they are just meaningless restriction designs, which would appear anyway in the* bi-orthogonal closure *of the set of generators* $\mathbb{B}_{P\ |\ Q}$.

Therefore, we can give a simple definition for the case were there is no communication between P and Q, and a more elaborated one for the second one, where there are new synchronizable pairs. The final step is thus to extend the definition of merging to $[\![P]\!]$ and $[\![Q]\!]$, and prove that the merging

of interpretations, which will be denoted $[\![P]\!] \oplus [\![Q]\!]$, **is equal to** $[\![P \mid Q]\!]$.

We need to restrict the operation $\odot$ on the base designs $\mathcal{D}_P$ and $\mathcal{D}_Q$, together with $\mathcal{N}_{P \mid Q}$, *re-build the restriction designs* on the merged base design $\mathcal{D}_P \bigodot \mathcal{D}_Q$, following the *freedom requisites* for the new elements, and only then perform the closure by bi-orthogonality. In this sense, the full merging of interpretations (in the second case) is related to the operation $\otimes$ on behaviours, since it keeps the core operation $\odot$, but goes outside its *logical meaning* by adding a third member on the merging of base designs, and by reaching the results through a few steps that *repeat the construction of the interpretation*. This is indeed close to building the interpretation of $P \mid Q$ from scratch, but our requirement to *avoid directly looking at* P *and* Q has been fulfilled, since we are *only using the information found in* $[\![P]\!]$ *and* $[\![Q]\!]$ – that actually let us *recover the necessary prefix order of* P *and* Q.

To prove the equality (modulo renaming, thus rather an equivalence) of the behaviours $\mathscr{B}_P$ and $\mathscr{B}_Q$ interpreting two processes P and Q, we can prove either that their *set of generators* is the same, or that the bi-orthogonal closure yields the same set; i.e. the interaction paths are the same for both sets. However, since the set of restriction designs is entirely dependent on the base design and *freedom requisites*, from which *restriction designs* are univocally generated, it is much more direct to show the first equivalence.

Note that by construction $\mathscr{B}_{P \mid Q}$ preserves all the interactions of $\mathscr{B}_P$ and $\mathscr{B}_Q$ whose associated executions are an execution sequence of P or Q: it holds that $\mathscr{B}_P^{\perp} \cup \mathscr{B}_Q^{\perp} \subseteq (\mathscr{B}_{P \mid Q})^{\perp}$, since all the *restriction designs* are preserved, even if built on a larger base design. This holds because restrictions are based on *freedom requisites* that are entirely determined by the partial orders $<_P$ and $<_Q$, which are not affected by the parallel composition.

Sets of generators and assignments merging

Let P, Q be $MCCS$ processes, and $\mathcal{D}_P \bigodot \mathcal{D}_Q$ be the merging of their base designs and assignments. We can distinguish two cases as said above.

Let A_P be the *multi*-set of occurrences of channel names in the domain of $[\ \ ]_P$ (thus each occurrence of the same channel name is counted as a separate element), and $A_P \upharpoonright_\epsilon$ the restriction of the set to elements of polarity $\epsilon \in \{+, -\}$. Then, either

1. $A_P \upharpoonright_+ \cap A_Q \upharpoonright_- = \varnothing$ and $A_P \upharpoonright_- \cap A_Q \upharpoonright_+ = \varnothing$, i.e. $[\ \ ]_P$ and $[\ \ ]_Q$ do not have any common occurrence of the same channel name, and of opposite polarity, or

2. $A_P \upharpoonright_+ \cap A_Q \upharpoonright_- = \{a^l, \ldots, a^m, \overline{a}^n, \ldots, \overline{a}^o\}$ or

$A_P \upharpoonright_- \cap A_Q \upharpoonright_+ = \{\overline{a}^l, \ldots, \overline{a}^m, \ldots, a^n, \ldots, a^o\}$.

With these definitions we are able to identify which of the shared channel names (of opposite polarity) are in $[\ \ ]_P$ and which are in $[\ \ ]_Q$ by their polarity.

We have that

Definition 5.1.12. *Let P, Q be MCCS processes and case 1 above hold. Assume $\mathscr{B}_P$ and $\mathscr{B}_Q$ are built as alienated behaviours with the same base, then*

$$[\![P]\!] \oplus [\![Q]\!] = (\mathscr{B}_P \bigotimes \mathscr{B}_Q, [\ \ ]_P \cup [\ \ ]_Q).$$

Lemma 5.1.13. *Let P, Q be MCCS processes and case 1 above hold. Then,*

$$[\![P \mid Q]\!] = [\![P]\!] \oplus [\![Q]\!]$$

Proof. Since *no communications* arises in the *parallel composition* $P \mid Q$, it holds that:

- $new\mathcal{S}_{P \mid Q} = \varnothing = new\mathcal{X}_{P \mid Q}$, $Loc_{P \mid Q} = Loc_P \cup Loc_Q$; therefore

  $$[\ \ ]_{P \mid Q} = [\ \ ]_P \cup [\ \ ]_Q,$$

  modulo *renaming of addresses* to avoid superimposition.

- $\mathcal{R}(P \mid Q) = (\mathcal{R}(P) \odot \mathcal{D}_Q) \cup (\mathcal{R}(Q) \odot \mathcal{D}_P)$, i.e. the union of the *merging* of each design in $\mathcal{R}(P)$ and the base design $\mathcal{D}_Q$, with the merging of each design in $\mathcal{R}(Q)$ with the base design $\mathcal{D}_P$.

  Therefore:

  $$\mathcal{R}(P \mid Q) \subset \{\mathcal{D} \odot \mathcal{C} \mid \mathcal{D} \in \mathscr{B}_P, \mathcal{C} \in \mathscr{B}_Q\}^{\perp\perp} = \mathscr{B}_P \bigotimes \mathscr{B}_Q$$

We can thus conclude that

$$[\![P \mid Q]\!] = [\![P]\!] \oplus [\![Q]\!] = (\mathscr{B}_P \bigotimes \mathscr{B}_Q, [\ \ ]_P \cup [\ \ ]_Q).$$

by definition 5.1.12 of $[\![P]\!] \oplus [\![Q]\!]$ in case 1.

□

In this way we found a simple definition for $\oplus$, that *interprets the parallel composition* **in the trivial case**. However, in case 2 things are much more complicated, since we need to *modify* existing restrictions, and add *new ones*. Our aim is to prove the equivalence between $[\![P]\!] \oplus [\![Q]\!]$ and $[\![P \mid Q]\!]$, where

$\oplus$ will be extended and modified to a more general and complex operation. We need to go through a few more steps for **case** 2, and partially repeat the construction of the interpretation. The definition of $\oplus$ for case 2 will still hold for case 1, but as a drawback entails some direct modifications of the *sets of generators* based on the information we can recover from the assignment functions.

The most direct way to prove the equivalence is therefore via the sets of generators themselves. We already proved the *base designs equality* in lemma 5.1.10; what we are missing is the equality of the restriction designs of $[\![P \mid Q]\!]$ and $[\![P]\!] \oplus [\![Q]\!]$, derived from the set of freedom requisites. What we need are the *new restriction designs*, that is the restriction designs for the new elements $new\mathcal{S}_{P \mid Q}$, $new\mathcal{X}_{P \mid Q}$, and *modify the old restrictions* for locations that appear in new synchronizations $u, v \in new\mathcal{S}_{P \mid Q}$.

Definition 5.1.14 (New restriction designs)**.** *Let P, Q be MCCS processes. We know that $<_{P \mid Q} = <_P \cup <_Q$, therefore $\mathcal{F}(i)$, $\mathcal{F}((u,v))$ (the* freedom conditions*) are unchanged for synchronizations $i \in \mathcal{S}_P \cup \mathcal{S}_Q$, and $(u,v) \in \mathcal{X}_P \cup \mathcal{X}_Q$.*

Instead $\mathcal{F}(l)$ for $l \in Loc_P \cup Loc_Q$ may be extended with any $i \in new\mathcal{S}_{P \mid Q}$ such that $l \in i$. We can then build $\mathcal{R}(P)$ and $\mathcal{R}(Q)$ in the same way as before (definition 4.1.15) – by putting $\maltese$ and prunings accordingly to freedom requisites and xor conditions – on the extended base design $\mathcal{D}_P \odot \mathcal{D}_Q$.

The resulting sets of restriction designs *are denoted $\mathcal{R}(P)_{\mathcal{D}_P \odot \mathcal{D}_Q}$ and $\mathcal{R}(Q)_{\mathcal{D}_P \odot \mathcal{D}_Q}$.*

The restrictions for $new\mathcal{X}_{P \mid Q}$ are immediately recoverable *from the set $new\mathcal{S}_{P \mid Q}$: all we need to know are which new synchronizations are in conflict. Instead restriction designs $\mathcal{R}(j)$ for $j \in new\mathcal{S}_{P \mid Q}$ require us to check the* local partial orders on locations coded into $[\![P]\!]$ and $[\![Q]\!]$. *If $j = (a^l, \overline{a}^m)$, we must recover the* immediate predecessors *of l and m, i.e. the greatest locations h, k such that $h < l$ and $k < m$.*

Since we coded the local partial order on locations in the restriction designs, it is enough to check $\mathcal{R}(l)$ and $\mathcal{R}(m)$: if there is a design with a $\maltese$ as last rule of $G[h]$, for some location h, and a pruning over $[l] \vdash$ (and equivalently for m, for some location k), then we have that $h, k \in \mathcal{F}(j)$ – remember that one of the freedom requisites of a location is its immediate predecessor. Equivalently we can check all the possible interaction paths of $[\![P]\!]$ and $[\![Q]\!]$ *to recover the partial orders. Therefore we can build $\mathcal{R}(j)$ based on $\mathcal{F}(j)$ (we skip the trivial proof of the fact that $\mathcal{F}(j)$ is the same set if build on $(P \mid Q, <_{P \mid Q})$).*

These new sets of restriction designs *are denoted respectively $\mathcal{R}(new\mathcal{S}_{P \mid Q})$ and $\mathcal{R}(new\mathcal{X}_{P \mid Q})$.*

Lemma 5.1.15. *Let P,Q be MCCS processes. It holds that* $\mathcal{R}(P \mid Q) = \mathcal{R}(P)_{\mathcal{D}_P \odot \mathcal{D}_Q} \cup \mathcal{R}(Q)_{\mathcal{D}_P \odot \mathcal{D}_Q} \cup \mathcal{R}(new\mathcal{S}_{P \mid Q}) \cup \mathcal{R}(new\mathcal{X}_{P \mid Q})$.

Proof. We proved the equality of the base designs $\mathcal{D}_{P \mid Q}$ and $\mathcal{D}_P \bigodot \mathcal{D}_Q$ (lemma 5.1.10), thus the *restriction designs* of the interpretation of $P \mid Q$ and the ones on the *merged base design* $\mathcal{D}_P \bigodot \mathcal{D}_Q$ are modified copies of the same design.

Now, these modifications must be the same, since they are based on the same freedom requisites. Indeed, the ones for P and Q remain unchanged, and their respective restriction designs are, by definition, exactly $\mathcal{R}(P)_{\mathcal{D}_P \odot \mathcal{D}_Q}$ and $\mathcal{R}(Q)_{\mathcal{D}_P \odot \mathcal{D}_Q}$.

The other restrictions in $\mathcal{R}(P \mid Q)$ are the ones about $new\mathcal{S}_{P \mid Q}$ and $new\mathcal{X}_{P \mid Q}$, which are exactly the ones included in the union above, i.e. $\mathcal{R}(new\mathcal{S}_{P \mid Q})$ and $\mathcal{R}(new\mathcal{X}_{P \mid Q})$, that are defined in the same way as standard restriction designs for synchronizations and *xor* clauses in the *construction of the interpretation* (definition 4.1.15), but take the needed information from the interpretations $[\![P]\!]$ and $[\![Q]\!]$ instead that from the *processes* P and Q. □

Definition 5.1.16. *Let P,Q be MCCS processes, and* **case** 2 *above hold (there is communication between P and Q). Then* $[\![P]\!] \oplus [\![Q]\!] =$

$$(\mathcal{D}_P \bigodot \mathcal{D}_Q \cup \mathcal{R}(P)_{\mathcal{D}_P \odot \mathcal{D}_Q} \cup \mathcal{R}(Q)_{\mathcal{D}_P \odot \mathcal{D}_Q} \cup \mathcal{R}(new\mathcal{X}_{P \mid Q}) \cup \mathcal{R}(new\mathcal{S}_{P \mid Q}))^{\perp\perp}$$

Let us note its set of generators with $\mathbb{B}_P \oplus \mathbb{B}_Q$, then

Definition 5.1.17. *Let P,Q be MCCS processes, and* case 2 *above hold 5.1.2 (there are possible communications between P and Q). The* **merging of interpretations** *of P and Q, in symbols $[\![P]\!] \oplus [\![Q]\!]$, is the pair*

$$((\mathbb{B}_P \oplus \mathbb{B}_Q)^{\perp\perp}, [\ \ ]_{P \odot Q}).$$

We want to prove the following:

Theorem 5.1.18. *Let P,Q be MCCS processes. Then* $[\![P]\!] \oplus [\![Q]\!] = [\![P \mid Q]\!]$.

Proof. It suffices to prove that $\mathbb{B}_P \oplus \mathbb{B}_Q \equiv \mathbb{B}_{P \mid Q}$. We already proved that:

- the *base designs* of the *merging of interpretations*, and of the *interpretation of $P \mid Q$ are the same*, in lemma 5.1.10;
- also the *restriction designs* are the same, by lemma 5.1.15.

Therefore we only need to prove that an unification of the assignments is possible, i.e. that, via a renaming on the assignments, $[\ ]_{P \odot Q} = [\ ]_{P \mid Q}$. We use the symbol for equivalence, since it holds modulo a renaming of addresses.

The theorem is almost straightforward by construction of $\mathbb{B}_P \oplus \mathbb{B}_Q$ and $\mathbb{B}_{P \mid Q}$. What we need to check is that we can indeed make the assignments equal by a renaming (which implies a rewriting of their corresponding base design). This is possible as long as:

1. The domain of the assignments is the same.
2. The co-domains have the same cardinality and structure (number of rules), which are the ramifications of $\mathcal{D}_P \odot \mathcal{D}_Q$ and $\mathcal{D}_{P \mid Q}$.

These two points hold by construction, since

1. the domain of $[\ ]_{P \mid Q}$ is:
   - $Loc_{P \mid Q} = Loc_P \cup Loc_Q$.
   - $\mathcal{S}_{P \mid Q} = \mathcal{S}_P \cup \mathcal{S}_Q \cup new\mathcal{S}_{P \mid Q}$.
   - $\mathcal{X}_{P \mid Q} = \mathcal{X}_P \cup \mathcal{X}_Q \cup new\mathcal{X}_{P \mid Q}$.

   Which, by construction, is the domain of $[\ ]_{P \odot Q}$.
2. By lemma 5.1.10, $\mathcal{D}_P \odot \mathcal{D}_Q = \mathcal{D}_{P \mid Q}$

Therefore we can assume $[\ ]_{P \mid Q} = [\ ]_{P \odot Q}$ via a simple renaming, i.e. the domain and co-domain of the functions are the same; hence at the *same element* of the common domain can be associated the *same address* in the co-domain (the ramification of the respective base design). We can now conclude that

$$\mathbb{B}_P \oplus \mathbb{B}_Q \equiv \mathbb{B}_{P \mid Q}$$

□

Remark 5.1.19. *The* merging of interpretations*, with the complications of the second case – where there is communication between the two processes – is explicative regarding the dynamic of parallel composition. While in the first case, when there is no interaction between P ad Q, it is nothing more than a $\otimes$ on behaviours, when something new is generated it requires some machinery and modifications, which make it go beyond a purely logical operation.*

It can be seen, though, as a good representation of what a parallel composition actually entails, and what are the minimal information *that we need*

to make it work. That is the local order $<_P$*, all the possible combinations of synchronizable pairs, and the conflict relation between them. All of them add complexity to the process, enough to go beyond the grasp of a simple* $\otimes$*.*

While no syntactic element is added in the parallel composition $P \mid Q$*, in* $[\![P \mid Q]\!]$ *we have something more to account for, with respect to the behaviour of the composed term; the merging of interpretations tries to tell us what that is.*

5.2 A reduction on the interpretation

The question we are going to study in this second part is what happens to the interpretation when we consider a process after an execution step. So, if $P \rightarrow_u P'$, what changes between $[\![P]\!]$ and $[\![P']\!]$? In other words, what happens to the interpretation $[\![P']\!]$ with respect to $[\![P]\!]$?

In general, when dealing with *typing systems*, *types* are *preserved* by reduction/normalization, i.e. if $P \rightarrow P'$, then $type(P) = type(P')$; this property is called *subject reduction*, and, as we pointed out in the discussions about the meaning of processes, it *should not hold* in our case. As Beffara writes in the introduction of [9], the meaning of an execution is not its final result, but what happens to get there, and thus the meaning of a process should not be preserved after execution, since the intermediate steps and forms of the process during execution cannot be recovered anymore. Our aim was to characterize the dynamic of a process, taking into account its non deterministic and non confluent behaviour, hence also any change in the process that influences the possible execution paths, and its potential interactions with the environment, which necessarily change after execution. This means that, if we achieved the *intended result* with our interpretation, then *the interpretation is not supposed to be preserved after execution*; therefore, a standard subject reduction property *should not hold*.

There actually is a sort of strong inclusion of $[\![P']\!]$ in $[\![P]\!]$, that we can momentarily note with $[\![P']\!] \sqsubset [\![P]\!]$. This is not a standard inclusion $\subset$, as could be the sub-design property: sub designs are the sub-sets of branches (or chronicles) whose base is a premise of an action of the main design – for example all the $G[u], G[l], etc.$ are sub designs of $\mathcal{D}_P$ – however, if we take $\mathcal{D}_P \setminus G[u]$ we would not have a sub-design inclusion; indeed this kind of operation would be a *deeper* pruning, up to the ramification of the base of $\mathcal{D}_P$. The pruning consists in erasing all actions in the continuation of a element of the ramification, but not the element itself from the ramification: the base of the sub-design – a premise of the first action of $\mathcal{D}_P$ – is still there; therefore there is no need to rewrite the first action of the design. Instead, by completely removing the whole branch $G[u]$, we would need to rewrite the ramification of the first action $(+, \xi, I)$ of $\mathcal{D}_P$, i.e. the *directory* of the behaviour, since it is *connected* (fact already explained in definition 2.2.24). On connected behaviours, this operation is called **projection**[1]. Note that we have already defined a similar operation in the *merging of designs*, that is the renaming on $[\ \ ]_P$ which induces a rewriting on the ramification I of the first action of the base design, in order to have *disjoint* directories for

[1] Found in the seminal article [38].

the merging. In this case, instead, we would be erasing a particular number from I, that would become $I' = I \setminus \{n\}$ where $[u] = \xi n$.

Theorem 4.2.14 on the correspondence between execution and interaction is the technical reason that explains why the interpretation is not preserved – which, again, was our intended result. $[\![P]\!]$ characterizes all executions on P, thus we know that any execution $P' \rightarrow_u$, for any P' such that $P \rightarrow_u P'$ for some $u \in \mathcal{S}_P$, is already associated to interactions on $[\![P]\!]$. However, it is clear enough that $[\![P']\!]$ cannot and should not include all executions $\rightarrow_{u,\vec{v}}$ from P; in particular, not the execution $P \rightarrow_u P'$. The synchronization u, as well as the synchronizations having a common channel with u – its *xor* conditions – should not appear at all in $[\![P']\!]$. Therefore $[\![P']\!]$ cannot be equal to $[\![P]\!]$, however $[\![P]\!]$ is able to characterize all the executions of $[\![P']\!]$, or at least *include them as sub-executions*, and this is where the inclusion lies. The next step is to formally define an operation that lets us obtain $[\![P']\!]$, for any reduced form P' of P, once applied on $[\![P]\!]$ and a synchronization $u \in \mathcal{S}_P$. Since the operation removes a part of $[\![P]\!]$, is an actual **reduction** on the interpretation.

Remark 5.2.1. *We can immediately note that if $P \rightarrow_u P'$, then $\mathcal{S}_{P'} = \mathcal{S}_P \setminus (\{u\} \cup xor(u))$ (with $xor(u)$ as in definition 3.2.8)*[2]*, and $Loc_{P'} = Loc_P \setminus \{l, m\}$, with $u = (a^l, \overline{a}^m)$. This means that the whole branches $G[u], G[l], G[m]$ and $G[x]$, for all $x \in xor(u)$,* are not in $\mathcal{D}_{P'}$*, as well as $w[u, x]$ and $w[x, y]$ (the xor conditions for u). Still, except for the channels in the synchronization u, the rest of the process is preserved, as well as its possible executions; reduction is thus a strict erasing, as in $\mathcal{G}_P$, always resulting in a smaller structure (of course this is true only for processes* without replication*). Therefore we want to* deep prune *– or* **trim** *to avoid misinterpretation – $\mathcal{D}_P$ and obtain a* strictly smaller base designs*, along with* less *non commutative restrictions.*

We can define the operation on $\mathcal{D}_P$, which then will naturally apply to $\mathcal{R}(P)$.

Definition 5.2.2 (Trimming)**.** *Let P be a MCCS process, $(a^l, \overline{a}^m) = u \in \mathcal{S}_P$ and $(+, \xi, I)$ the first action of $\mathcal{D}_P$. The* **trimming** *of u on $\mathcal{D}_P$, noted $(\mathcal{D}_P)_u$, is a rewriting of $\mathcal{D}_P$ consisting in the removing of the* branches associated to a synchronization *u, which are:*

- *$G[u]$;*
- *$G[l]$ and $G[m]$;*

[2] $xor(u) = \{x \mid (u, x) \in \mathcal{X}_P\}$

- $xor^u \,\&\, xor^{x_1}, \ldots, xor^u \,\&\, xor^{x_j}$ *and* $G[x_1], \ldots, G[x_j]$ *for all clause* $(u, x_1), \ldots, (u, x_j) \in \mathcal{X}_P$*;*
- $xor^x \,\&\, xor^y$ *for each* $x \in xor(u)$ *and* $y \in xor(x)$*;*

and then a rewriting of $(+, \xi, I)$ *accordingly to this removal, i.e.* a restriction to the action $(+, \xi, J)$ *with* $J = I \setminus \{i_1, \ldots, i_n\}$*, where* $[u] = \xi i_1$*,* $[l] = \xi i_2$*,* $[m] = \xi i_3$*,* $G[x_1] = \xi i_4$*, ...,* $[xor^u \,\&\, xor^{x_j}] = \xi i_m$*, ...,* $[xor^x \,\&\, xor^y] = \xi i_n$*.*

This operation on the base design induces a rewriting on all designs of $\mathcal{R}(P)$, and a restriction on the assignment $[\ \ ]_P$, since its co-domain itself get restricted, that is the directory of $\mathscr{B}_P$. For the latter it is a matter of a simple set inclusion, since $[\ \ ]_{P'} = [\ \ ]_P$ except for the elements $[u], [l], [m]$, $[x]$, $[xor^u \,\&\, xor^x]$ and $[xor^x \,\&\, xor^y]$, for each $x \in xor(u)$ and $y \in xor(x)$ which are removed from the domain of $[\ \ ]_P$ (since all $x \in xor(u)$ disappear once u has been erased, even all theirs xor conditions must as well).

To better understand the *trimming*, we give a simple concrete example in the following page.

Example 5.2.3. *The simplest case is when the trimming is done on a synchronization* $u_k = (a^l, \overline{a}^m)$ *with no xor conditions. Thus, let* $\mathcal{D}_P =$

$$\dfrac{\begin{array}{ccccccccccccc} \vdots & & \vdots & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \overline{G[u_1]\vdash} & \overline{G[u_2]\vdash} & \cdots & \overline{G[u_i]\vdash} & \overline{G[u_k]\vdash} & \overline{G[l]\vdash} & \overline{G[m]\vdash} & \cdots & \overline{G[p]\vdash} & \overline{G[o]\vdash} & \overline{w[u_1,u_i]\vdash} & \overline{w[u_i,u_h]\vdash} & \cdots & \overline{w[u_j,u_n]\vdash} \end{array}}{\vdash \xi}\ (+,\xi,I)$$

and $G[u_k] = \xi.x$, $G[l] = \xi.y$, $G[m] = \xi.z$. *Then, the trimming of* u_k *on* $\mathcal{D}_P$, $\mathcal{D}_P^{u_k}$ *is*

$$\dfrac{\begin{array}{ccccccccccc} \vdots & \vdots & & \vdots & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \overline{G[u_1]\vdash} & \overline{G[u_2]\vdash} & \cdots & \overline{G[u_i]\vdash} & \cdots & \overline{G[p]\vdash} & \overline{G[o]\vdash} & \overline{w[u_1,u_i]\vdash} & \overline{w[u_i,u_h]\vdash} & \cdots & \overline{w[u_j,u_n]\vdash} \end{array}}{\vdash \xi}\ (+,\xi,J = I \setminus \{x,y,z\})$$

If we trim $u_1 = (b^p, \overline{b}^o)$ *(with* $G[u_1] = \xi.1$*) instead, with as single xor condition on* u_1, $w[u_1,u_i]$, *with* $G[u_i] = \xi.i$, *of base* xor^{u_1} *&* $xor^{u_i} = \xi.n$, *and as second xor condition on* u_i, $w[u_i,u_h]$, *of base* xor^{u_i} *&* $xor^{u_h} = \xi.n+1$, *then* $\mathcal{D}_P^{u_i} =$

$$\dfrac{\begin{array}{ccccccccc} \vdots & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \overline{G[u_2]\vdash} & \cdots & \overline{G[u_i]\vdash} & \overline{G[u_k]\vdash} & \overline{G[l]\vdash} & \overline{G[m]\vdash} & \cdots & \overline{w[u_j,u_n]\vdash} \end{array}}{\vdash \xi}\ (+,\xi,J = I \setminus \{1,i,n,n+1\})$$

where we erased not only $G[u_1]$ *and the branches corresponding to its locations* $G[p]$ *and* $G[o]$, *but also* $G[u_i]$, *and the xor conditions* $w[u_1,u_i]$ *and* $w[u_i,u_h]$.

As we mentioned, this operation of removal of a sub-ramification from designs of a behaviour in ludics terms is a simple *projection* operation, defined on *connected behaviours*. We could as well define the *trimming* as an operation on $\mathbb{B}_P$, the set of *generators* of the behaviour $\mathscr{B}_P$, since the projection operation *commutes with the bi-orthogonal closure* (shown in [38]). Thus, if $(\mathbb{B}_P)_u$ denotes the *trimmed* set of generators, where for each design, the ramification I of the first action $(+,\xi,I)$ is projected to $J = I \setminus \{i_1,\dots,i_n\}$, then $(\mathbb{B}_P)_u^{\perp\perp} = ((\mathbb{B}_P)^{\perp\perp})_u$.

In the following, let x, y be variables on the set $\mathcal{S}_P$ (i.e. any synchronization satisfying the conditions on x or y).

Definition 5.2.4. *Let P be a MCCS process, and $(a^l,\overline{a}^m) = u \in \mathcal{S}_P$. Then $([\ ]_P)_u$ is $[\ ]_P \upharpoonright_{(\mathcal{D}_P)_u}$, i.e. is $[\ ]_P$ on the restricted domain $(Loc_P \setminus \{l,m\}) \cup (\mathcal{S}_P \setminus (\{u\} \cup \{x \mid \in xor(u)\}) \cup (\mathcal{X}_P \setminus \{(u,x) \cup (x,y)\})$ for each $x \in xor(u)$ and $y \in xor(x)$.*

It is clear that by this definition $([\ ]_P)_u$ matches the ramifications of $(\mathcal{D}_P)_u$, and has the same domain of $[\ ]_{P'}$, with $P \to_u P'$ – for the considerations made in remark 5.2.1. Therefore, via a renaming of the assignments as for the *merging* of types, we can conclude

Lemma 5.2.5. *Let P be a MCCS process. If $P \to_u P'$, then, let $([\ ]_P)_u$ be the resulting assignment on the trimmed base design $(\mathcal{D}_P)_u$; it holds that $([\ ]_P)_u = [\ ]_{P'}$.*

Proof. We already noted that the domains of the two assignments are the same, since $\mathcal{S}_{P'}, \mathcal{X}_{P'}$ and $Loc_{P'}$ matches exactly the defined restrictions on the respective sets of P for $([\ ]_P)_u$. The co-domain, which is the ramification of the first action of $\mathcal{D}_{P'}$ for $[\ ]_{P'}$ and the one of $(\mathcal{D}_P)_u$ for $([\ ]_P)_u$, have the same cardinality, being built on *the same domain* of their respective assignments. Thus we can unify the two ramifications, say I for $\mathcal{D}_{P'}$ and J for $(\mathcal{D}_P)_u$, by a renaming of the latter (we could as well rename I, but in this way we confine the operations on the *trimmed* base design), and obtain the equality

$$([\ ]_P)_u = [\ ]_{P'}.$$

□

Since the directories of $\mathcal{D}_{P'}$ and $(\mathcal{D}_P)_u$ are the same, as well as their structure – all the $G[l]$, $G[u]$ and $w[x,v]$ are the same since are built on the same domain – and their base can be assumed to be the same (it is a single address), as an immediate corollary we have

Corollary 5.2.6. *If $P \rightarrow_u P'$, then $\mathcal{D}_{P'} = (\mathcal{D}_P)_u$.*

About the *restriction designs*, it holds that the *trimming* of a synchronization u on the base design will automatically *remove from the interpretation* the restriction designs regarding the branches *associated* to u. That is because the restrictions in question *do not have anymore their meaningful branches* (the ones ending with ✠ or pruned), thus are *equal* to the base design; therefore they will simply disappear in the union with $\{\mathcal{D}_P\}$ since they become the very same element. We can identify these relevant restrictions in the following way:

Definition 5.2.7. *Let P be a MCCS process, and $(a^l, \overline{a}^m) = u \in \mathcal{S}_P$.*

Let $\mathcal{R}([u])$ be the set of restrictions designs *of $[\![P]\!]$ where $G[u]$ or $G[x]$, for $x \in xor(u)$, are* pruned *along with those where $G[u], G[l]$, $G[m]$ end with ✠, and those where $[xor^u \,\&\, xor^x]$, or $[xor^x \,\&\, xor^y]$, for all $x \in xor(u)$, $y \in xor(x)$ have* a branch *ending with ✠.*

The definition selects all restriction designs where u or its locations are a requisite (ends with ✠) for some other element of P, and the restrictions on all xor conditions regarding u and $x \in xor(u)$ – note that in this way $\mathcal{R}(l)$ and $\mathcal{R}(m)$ are included, since $G[u]$ is a requisite for its own locations, ending with ✠. What happens is that all such synchronizations x disappear along u, because *one of their channels*, the one in common with u, *has been erased.* However the *other channel* is still in the process, thus the restriction design where $G[x]$ ended with ✠ (the restriction on the other channel labeled by a *location*), is still relevant to interaction. Indeed the restriction for the other location of such x will just be missing the branch $G[x]$, but have ✠ on $G[j]$ for any other synchronization j to which it belongs, or just end with a pruning if no such j exists, noting an absence of interaction; thus only the restriction design where $G[x]$ is pruned are going to disappear.

Example 5.2.8. *Let P be a MCCS process, $u, x \in \mathcal{S}_P$ and $(u, x) \in \mathcal{X}_P$. Then, let $u = (b^l, \overline{b}^m)$ $x = (b^l, \overline{b}^o)$, and $y = (b^n, \overline{b}^o)$; $\overline{b}^o$ is a common channel between x and y, therefore $(x, y) \in \mathcal{X}_P$. If we erase from $\mathcal{D}_P$ the* branches associated to *u, the entire chronicle starting with $xor^x \,\&\, xor^y$ is* erased from all designs of $\mathbb{B}_P$, *and the restriction design for $\overline{b}^o$, $\mathcal{R}(b^o)$, from*

$$\dots \quad \frac{\dfrac{}{\vdash [x].1}\,✠}{[x] \vdash} \quad \dots \quad \frac{\dfrac{}{\vdash [y].1}\,✠}{[y] \vdash} \quad \dots \quad \frac{}{[b^o] \vdash}\,p \quad \dots$$
$$\vdash \xi$$

becomes

$$\dfrac{\ldots \quad \dfrac{\dfrac{}{\vdash [y].1}\ \maltese}{[y] \vdash} \quad \ldots \quad \dfrac{}{[b^o] \vdash}\ p \quad \ldots}{\vdash \xi}$$

Note also that we do not require u to be a *principal synchronization*, since we want the definition to be as general as possible, letting us perform the trimming on the interpretation even for non principal synchronizations. Still, in most cases the restrictions where $G[u]$ is pruned *are not relevant*, since our aim is to *match execution on the process*, and thus we are assuming to be in the case where u is ready for execution, i.e. its channels are not blocked by any prefix, and therefore there are no restrictions where $G[u]$ is pruned.

The rewriting of $\mathcal{R}(P)$ (the set of *all restriction designs*) on the trimmed base design, that is $\mathcal{R}(P) \downarrow_{(\mathcal{D}_P)_u} = (\mathcal{R}(P))_u$, does not change the relative positions of $\maltese$ and the *pruning*, except in the case of the erased branches, where obviously they do not appear at all. We can note the behaviour built on the trimmed base design with:

$$(\mathscr{B}_P)_u = (\{(\mathcal{D}_P)_u\} \cup (\mathcal{R}(P))_u)^{\perp\perp}$$

As a last step, we need the following two lemmas:

Lemma 5.2.9. *Let P be a MCCS process, and $u \in \mathcal{S}_P$. All designs of $\mathcal{R}([u])$ disappear in the union $\{(\mathcal{D}_P)_u\} \cup (\mathcal{R}(P))_u$.*

Proof. By rewriting $\mathcal{R}(P)$ on $(\mathcal{D}_P)_u$ all the branches *associated* to u are erased. These are exactly – by definition – the branches ending with $\maltese$ or a pruning of designs of $\mathcal{R}([u])$ which therefore *do not have these relevant branches anymore*, becoming *exactly equal to* $(\mathcal{D}_P)_u$, the only design remaining in the union.

□

Lemma 5.2.10. *Let P be a MCCS process, $u \in \mathcal{S}_P$, and $P \to_u P'$. Then $\mathcal{R}(P') = (\mathcal{R}(P))_u \setminus \mathcal{R}([u])$.*

Proof. All designs of $\mathcal{R}([u])$ are *exactly* the ones about the branches missing from $\mathcal{D}_{P'}$ with respect to $\mathcal{D}_P$, i.e. the branches corresponding the elements *associated to* u in P. The other elements of $(\mathcal{R}(P))_u$ are then exactly the same designs of $\mathcal{R}(P')$, since $<_{P'}$ and $\mathcal{X}_{P'}$, on which restriction designs are based, are exactly $<_P$ and $\mathcal{X}_P$ on a restricted domain, i.e. $<_{P'} \cap <_P = <_{P'}$ and $\mathcal{X}_{P'} \cap \mathcal{X}_P = \mathcal{X}_{P'}$. Therefore $\mathcal{R}(P') = (\mathcal{R}(P))_u \setminus \mathcal{R}([u])$.

□

We have now all the tools needed to define **reduction** on $[\![P]\!]$.

Definition 5.2.11. *Let P be a MCCS process, $[\![P]\!]$ its interpretation, and $u \in \mathcal{S}_P$. The* reduction *of u on $[\![P]\!]$ is $([\![P]\!])_u = ((\mathscr{B}_P)_u, ([\ ]_P)_u)$. We note this operation with $[\![P]\!] \rightsquigarrow_u ([\![P]\!])_u$.*

Note that if u is an internal synchronization the result of the operation is the interpretation of a process that may be *not obtainable* by execution on P. We let the operation apply to a general case, since this could help us characterize deadlocks by performing it on non-executable synchronizations.

The reason we call this operation *reduction* instead of rewriting, other than the fact that we are erasing parts of the base design, is clear if we look at the *interaction paths*. Indeed, once proved that $([\![P]\!])_u = [\![P']\!]$, for $P \rightarrow_u P'$, it is immediate to show that the reduced interpretation contains a *subset* of the interaction paths of $[\![P]\!]$, by the correspondence execution-interaction. On P' we can perform $\rightarrow_{v_i}$ with $v_i = v_j$ or v_0, such that:

1. v_j does not have a common location with u, and is a principal synchronization.

2. v_0 has as only requisite the synchronization u (one or both its locations), i.e. $P \rightarrow_{u,v_0}$ is an admissible execution.

Therefore, by theorem 4.2.14, for all interactions on $[\![P']\!]$ we have that, modulo renaming of the assignments, there is an interaction on $[\![P]\!]$ such that either they have the same associated execution, or the interaction on $[\![P]\!]$ extends – with an initial segment – the one on $[\![P']\!]$.

However we cannot talk about *equality* of the sets of visited actions, because the set of orthogonals $\mathscr{B}_P^\perp$ and $\mathscr{B}_{P'}^\perp$ *cannot* be equal, due to the mismatch of the *directories* of $\mathcal{D}_P$ and $\mathcal{D}_{P'}$. Indeed the first action of $\mathcal{D}_P$ have a smaller cardinality than the one of $\mathcal{D}_{P'}$, since P' lacks all the branches associated to u, and so $[\ ]_{P'}$ have a strictly smaller domain than $[\ ]_P$; however we can still assume that the same addresses are assigned to the same elements of the domains. It remains to prove that $(\mathscr{B}_P)_u$ is actually the interpretation of the process after execution on u.

Theorem 5.2.12. *Let P be a MCCS process, $u = (a^l, \overline{a}^m) \in \mathcal{S}_P$ and $P \rightarrow_u P'$. It holds that $[\![P']\!] = ([\![P]\!])_u$, therefore*

$$\begin{array}{ccc} P & \rightarrow_u & P' \\ {\scriptstyle [\![-]\!]}\downarrow & & \downarrow{\scriptstyle [\![-]\!]} \\ [\![P]\!] & \rightsquigarrow_u & ([\![P]\!])_u \end{array}$$

Proof. By applying lemma 5.2.5, corollary 5.2.6, and lemma 5.2.9 to enable lemma 5.2.10, we have that $\mathcal{D}_{P'} = (\mathcal{D}_P)_u$, $[\ \ ]_{P'} = ([\ \ ]_P)_u$, and that $\mathcal{R}(P') = (\mathcal{R}(P))_u$. Therefore we can conclude:

$$[\![P']\!] = ([\![P]\!])_u.$$

□

For the above discussion about the inclusion of interaction paths, we can give the following result:

Lemma 5.2.13. *Let $P \rightarrow_u P'$. It holds that $\forall \mathcal{C}' \in \mathscr{B}_{P'}, \exists \mathcal{C} \in \mathscr{B}_P$ such that either:*

- $\langle \leftarrow \mathcal{C}' \rangle \cong \langle \leftarrow \mathcal{C} \rangle$ *where* $\cong$ *means that* $(\langle \leftarrow \mathcal{C}' \rangle \setminus (+, \xi, J)) = (\langle \leftarrow \mathcal{C} \rangle \setminus (+, \xi, I))$ *and* $J \subset I$*, with* $(+, \xi, J)$ *the first action of* $\mathcal{D}_{P'}$ *and* $(+, \xi, I)$ *the first action of* $\mathcal{D}_P$*.*

  This is the case where we perform interaction on $[v_i], [\vec{v}]$ *with* v_i *as in case* 1 *above, independent from* u*; or*

- $\langle \leftarrow \mathcal{C}' \rangle \sqsubset \langle \leftarrow \mathcal{C} \rangle$ *where* $\sqsubset$ *means that* $(\langle \leftarrow \mathcal{C}' \rangle \setminus (+, \xi, J)) = (\langle \leftarrow \mathcal{C} \rangle \setminus ((+, \xi, I), [u]^{\perp\perp})$ *and* $J \subset I$*; with* $(+, \xi, J)$ *the first action of* $\mathcal{D}_{P'}$*,* $(+, \xi, I)$ *the first action of* $\mathcal{D}_P$ *and* $[u]^{\perp}$ *the sequence of action defined in the constructive proof of* lemma 4.2.19*, i.e. the sequence of action that visits the addresses of* $xor(u)$*,* $G[u]$*, and* $G[l]$*,* $G[m]$*, for its locations* l, m*. This is a generalization of the notion of final segment – it is not necessarily a final segment since all the interaction steps not corresponding to execution may be done in a very different order.*

  In this case we are performing interaction on $[v_0], [\vec{v}]$*, with* v_0 *as in case* 2 *above, dependent from* u*.*

Proof. By applying theorem 4.2.14 on the correspondence execution-interaction we can conclude that the interaction paths on $\mathscr{B}_{P'}$ are in one of the relations $\cong$ or $\sqsubset$ with the ones of $\mathscr{B}_P$, and thus have either the *same associated executions* or the ones of $\mathscr{B}_{P'}$ are (generalized) *final segments* of those of $\mathscr{B}_P$. □

From lemma 5.2.13, it follows that $[\![P]\!]$ *includes* the interactions of $[\![P']\!]$, in the meaning expressed by the two relations $\cong$ and $\sqsubset$ defined in the lemma.

5.2.1 Reduction to 1

The definition of reduction on the interpretation can naturally be extended to sequences of synchronizations $\vec{u} = u_1, \dots, u_n$ as a simple repetition of the operation: for example $(\mathcal{D}_P)_{u_1,u_2} = ((\mathcal{D}_P)_{u_1})_{u_2}$ and so on, the same for $([\ ]_P)_{\vec{u}}$ and $(\mathscr{B}_P)_{\vec{u}}$, for execution sequences $\to_{\vec{u}}$.

What is left to inquire is what is the form of the interpretation when a process P is reduced to 1. If $P \to_{\vec{u}} 1$, then what happens to the structure of $(\mathscr{B}_P)_{\vec{u}}$? We just need to apply the reduction step by step, and check what is left in $([\![P]\!])_{\vec{u}}$. Following the reduction operation, if $P \to_{\vec{u}} 1$, then *the whole ramification is going to be erased* from $\mathcal{D}_P$: every location, thus every synchronization and xor condition. What is left is just the *base* of $\mathcal{D}_P$ with an *empty* ramification:

$$\frac{}{\vdash \xi}\ (+, \xi, \varnothing)$$

which is the design called $\mathbb{O}ne$ (defined in example 2.2.22), whose generated behaviour *corresponds to the linear logic multiplicative unit* 1, when interpreting linear logic into ludics. This means that the reduction operation on the interpretation has the intended intuitive meaning, and further justify the choice to name the empty process 1, the multiplicative unit, instead of 0. Moreover, $\mathbb{O}ne$ is the **neutral element** of $\odot$, the merging of designs.

5.3 Deadlocks

Before giving a formal definition of *deadlocked process*, we must be able to tell when a process P is in *normal form*, i.e. when there are no synchronizable pairs of channels in P.

Definition 5.3.1. *A CCS process P is in* **normal form** *if there are no synchronizing pairs of channels ready for execution in P; in other words, there is not a minimal synchronization available, w.r.t. $\preccurlyeq_{\mathcal{S}_P}$.*

A CCS process P is **normalizable** *if there is an execution path $P \to_{\vec{u}} P'$ such that P' is in normal form.*

Remark 5.3.2. *In a setting* without replication, *as the multiplicative fragment, obviously any process is normalizable, since there is no infinite execution path. Moreover, if P is in normal form, the only execution associated to the interactions on $[\![P]\!]$ is the* empty one *(this is a direct consequence of theorem 4.2.14 on the correspondence execution-interaction). Furthermore,*

since execution is non-confluent, a process P has, in general, **multiple independent normal forms**.

Example 5.3.3. *For instance, consider the process*

$$P = a^1.\overline{b}^2 \mid b^3.\overline{a}^4 \mid \overline{a}^6.c^7 \mid \overline{b}^8.\overline{c}^9$$

with $u_1 = (a^1, \overline{a}^4)$, $u_2 = (a^1, \overline{a}^6)$, $u_3 = (\overline{b}^2, b^3)$, $u_4 = (b^3, \overline{b}^8)$ *and* $u_5 = (c^7, \overline{c}^9)$. *The execution* $\rightarrow_{u_4,u_1}$ *leads to*

$$P' = \overline{b}^2 \mid \overline{a}^6.c^7 \mid \overline{c}^9$$

which is in normal form, since no further execution is possible.

If we instead performed the execution $\rightarrow u_2, u_4, u_5$ *we would have*

$$P'' = \overline{b}^2 \mid \overline{a}^4,$$

which is another normal form; as is $P''' = \overline{a}^4 \mid c^7 \mid \overline{b}^8.\overline{c}^9$ *after the execution* $\rightarrow_{u_2,u_3}$. *Anticipating again the non deterministic choice +, which denotes a mutual exclusion of two terms during execution, we have that*

$$a.P' + b.Q' \mid \overline{a}.P'' + .\overline{b}.Q''$$

can lead to either $P' \mid P''$ *or* $Q' \mid Q''$, *the first for the execution* $\rightarrow_{(a,\overline{a})}$, *the second for the execution* $\rightarrow_{(b,\overline{b})}$. *If* $P' \neq Q'$ *or* $P'' \neq Q''$, *and there are no possible executions on either of them, then we have two different normal forms.*

Note, however, that two different execution paths may lead to the same normal form. For instance, if we take

$$P = a^1.\overline{b}^2 \mid b^3.\overline{a}^4 \mid \overline{a}^6.c^7 \mid \overline{b}^8.\overline{c}^9 \mid a^{10} \mid b^{11}$$

the process of the previous example, where we added $a^{10} \mid b^{11}$; with $u_6 = (a^{10}.\overline{a}^4)$, $u_7 = (a^{10}, \overline{a}^6)$, $u_8 = (\overline{b}^2, b^{11})$ and $u_9 = (\overline{b}^8, b^{11})$, then both execution paths $\rightarrow_{u_2,u_3,u_6,u_9,u_5}$ and $\rightarrow_{u_2,u_8,u_4,u_5,u_6}$ lead to the empty process 1.

With these facts in mind, we may want to talk about normalization classes, i.e. classes of *execution sequences* which lead to the same *normal form*.

Definition 5.3.4. *A* **normalization class** *of a process P, noted $\mathbb{P}_i$, with $i \in I$ and I a set of indexes, is a set of* execution sequences *of P (each of the form $\vec{u} = (u_1, \ldots, u_n)$), such that they lead to the same* normal form, *i.e. if $\vec{u}, \vec{v} \in \mathbb{P}_i$, then*

$$P \to_{\vec{u}} P' = P'' \ {}_{\vec{v}} \leftarrow P.$$

with P'/P'' in normal form. *With $\mathbb{P}_i$ we note the i–th normalization class of P.*

Now we can give the definition of *deadlocked process*:

Definition 5.3.5. *A process P is* **deadlocked** *if and only if*

- *P is in* **normal form** *and*
- *$\exists \sigma_1, \ldots, \sigma_n$ such that each σ_i is a* non empty *set of synchronizations in a cycle $u_1 \preccurlyeq_{\mathcal{S}_P} u_2 \preccurlyeq_{\mathcal{S}_P} \ldots \preccurlyeq_{\mathcal{S}_P} u_n \preccurlyeq_{\mathcal{S}_P} u_1$. We denote the* set of cycles *of a process P with $\mathscr{C}_P$.*

5.3.1 Cycles in the process

The first consideration we can make about *deadlocks* is that, from the point of view of *interaction*, nothing special is happening. A process is deadlocked when *at least* two synchronizations block each other by prefix order of their channels; for instance if $P = a^1.\overline{b}^2 \mid b^3.\overline{a}^4 \mid c^5.R \mid d^6.S$, there is a deadlock between $i = (a^1, \overline{a}^4)$ and $j = (\overline{b}^2, b^3)$: a^1 is blocking $\overline{b}^2$ and b^3 is blocking $\overline{a}^4$.

Regarding interaction in ludics, a deadlock is no different that any other part of the process where no communication is possible, as $c^5.R \mid d^6.S$; in both cases there are no synchronizations available, and thus no execution can be performed: from the point of view of $[\![P]\!]$ there are simply no $\mathcal{C} \in \mathscr{B}_P^\perp$ interacting with either $G[i]$ or $G[j]$. However, the reason why the interaction is not possible can give us a hint on how to identify deadlocks, or the absence of them.

In the example, $\mathcal{R}(i)$ and $\mathcal{R}(j)$ are *impossible to satisfy together*. The *freedom requisites* (definition 4.1.11) are, respectively, $\mathscr{F}(i) = \{3\}$ and $\mathscr{F}(j) = \{1\}$, but $\mathscr{F}(1) = \{i\}$ and $\mathscr{F}(3) = \{j\}$: there is a cycle $1 < j < 3 < i < 1$. In the *restriction designs* this is translated into designs requiring interaction to visit $G[1]$ before $G[j]$ and $G[2]$; $G[j]$ before $G[3]$ and $G[2]$; $G[3]$ before $G[i]$ and $G[4]$; and $G[i]$ before $G[1]$ and $G[4]$, which is obviously impossible to satisfy. Any design trying to interaction with $[i], [j]$ or the addresses assigned to their locations is bound to not be orthogonal to one of the restrictions, and thus $\notin \mathscr{B}_P^\perp$.

However, the synchronizations in a deadlock *are not bound to be stuck in this cyclic order*: a parallel composition with another process Q could provide the channels and thus synchronizations needed to break the cycle. The deadlock is therefore still a *potential* communication: if either a^1 or b^3 are erased via execution then the other synchronization (respectively j or i) is ready for synchronization; and we can expect this to happen in $P \mid Q$, with $\overline{a}$ or $\overline{b}$ as external channels of Q.

The deadlock is thus a cycle where two or more synchronizations block each other, which can potentially be "unlocked", and it is directly visible in the restriction designs as *two or more incompatible conditions* (possibly by *transitivity*). We can define a simple relationship $\triangleright_P$ (we omit the subscript when clear from the context) on sub designs of $\mathcal{D}_P$, such that $G[i] \triangleright G[l]$ (vice-versa $G[l] \triangleright G[i]$) if

$$\dfrac{\ldots \quad \dfrac{\dfrac{}{\vdash [i].1}\,\maltese}{[i] \vdash} \quad \ldots \quad \dfrac{}{[l] \vdash}\,p \quad \ldots}{\vdash \xi}$$

is in $\mathcal{R}(l)$ (vice versa, $G[l]$ ends with a $\maltese$ in $\mathcal{R}(i)$).

Remark 5.3.6. *Note that this condition* cannot *be checked by looking at all the possible interaction paths of $[\![P]\!]$: if $[l]$ is always visited before $[i]$ in any interaction path, then obviously $[l]$ must have precedence on $[i]$; however if $[l]$ and $[i]$ are in a* deadlock *there will be* no interaction on either of them.

Whenever there is a cycle $G[l] \triangleright G[j] \triangleright \ldots \triangleright G[i] \triangleright G[l]$ we have therefore a deadlock. We can formalize this property as:

Lemma 5.3.7. *A MCCS process P is* deadlock free *if $\triangleright_P$ is* acyclic.

Proof. If $\triangleright_P$ is acyclic, then we can retrace our steps and conclude, by definition of $\triangleright_P$, that there is no restriction design whose pruned ramification is, by transitive closure of $\triangleright_P$, a requisite of its own requisite, i.e. ends with a $\maltese$ in a restriction design for his requisite. By definition of $\mathcal{R}(i)$ and $\mathcal{R}(l)$, with $i \in \mathcal{S}_P, l \in Loc_P$, then there are no sequence of synchronizations $i_1, \ldots, i_n$ in a cycle, implying that $\preccurlyeq_{\mathcal{S}_P}$ is acyclic, and thus that P is deadlock free.

□

This a trivial consequence on the definition of $\triangleright$ and an obvious result once defined the order on locations and synchronizations. However the dual property, i.e. the presence of cycles in $\triangleright_P$, is *not* a property of ludics and it is *not enough* to determine if this cycle in the order implies that there is actually a deadlock, i.e. if execution on the synchronizations in the cycle is actually not possible. Indeed, there could be a cycle in $\triangleright$, but possible synchronizations with channels outside the cycle.

We could of course always check if there are possible executions or not on these channels, however our aim is to be able to *identify deadlock-free processes without performing execution*, by only looking at the interpretation; therefore by using *properties of designs or behaviours*, thus property of ludics, to find from $[\![P]\!]$ if P is deadlock-free.

5.3.2 Deadlock-free processes and material designs

To answer the questions about *deadlock-freedom* we can focus on the *visitable paths* of a behaviour, which applied on the interpretation $[\![P]\!]$ can give us some useful insights on the corresponding process P. It is another way to talk about *incarnation*, which we already defined on behaviours (definition 2.2.23). What we need is the *potentially visited part* of a design (in a behaviour) and, by extension, of the entire behaviour.

In general, we can just consider the incarnation of a behaviour as the incarnation of each design in it. As defined in the background section, the incarnation of a design $\mathcal{D} \in \mathscr{B}$ is *the smallest design $\mathcal{D}'$ included in $\mathcal{D}$ that still belongs to $\mathscr{B}$*; i.e. the set of chronicles of $\mathcal{D}$ visited by an interaction (at least one) with the orthogonal behaviour. We can start by considering the set of *visitable paths* of a behaviour, by which incarnation can be defined, as in [28]. The *set of visitable paths* of a design $\mathcal{D}$ in a behaviour $\mathscr{B}$ is

$$V_{\mathscr{B}}(D) = \bigcup_{\mathcal{C} \in \mathscr{B}^{\perp}} \langle \mathcal{D} \leftarrow \mathcal{C} \rangle$$

The set $V_{\mathscr{B}}(D)$ determines the *incarnation* of $\mathcal{D}$ in the behaviour $\mathscr{B}$, denoted $|\mathcal{D}|_{\mathscr{B}_P}$ (we exclude the subscript if obvious from the context), once we consider the *chronicles* corresponding to the set of visitable paths. In the case of the base design $\mathcal{D}_P$, its incarnation yields some useful information on the process, in particular about *deadlock-freedom*.

Another property of the interpretation that we only mentioned, is its *purity*.

Definition 5.3.8 (Purity)**.** *A behaviour $\mathscr{B}$ is* pure *if all the* maximal *interaction paths with $\mathscr{B}^{\perp}$* do not *end with ✠ in $\mathscr{B}$; i.e. on maximal interaction paths the ✠ is always found in designs of $\mathscr{B}^{\perp}$.*

Trivially, we have that

Lemma 5.3.9. *For any MCCS process P, $\mathscr{B}_P$ is* pure*.*

This holds by construction: all the maximal interaction paths are the ones with the base design $\mathcal{D}_P$ (since the restriction designs are strictly smaller), which does not have branches ending with ✠.

Purity is a property strongly tied with the notion of *type safety*, and so indirectly to behaviours, the *types* of ludics, as they model linear logic formulas, and we can compose them via operators corresponding to the connectives $\otimes, \parr, \&, \oplus$. If we consider the daimon $\dagger$ as a symbol standing for an *error* (one of its many interpretations), which is consistent with the fact that, in ludics, even false (or wrong) proofs are admitted (since each proof has a *dual*, proving the *opposite statement*), then the notion of *purity* tells us that *errors do not occur during interaction*. For instance, if a behaviour $\mathscr{B}$ is the interpretation of a *data type* (as found in [66]), then we know that its type is *safe*, i.e. computation on this type will not encounter errors (the same holds for *functional types*).

In our case, however, *purity* does not tell us much. While it is still a nice property, since we are representing the dynamic of processes, modulo the correspondence execution-interaction, $\dagger$ only means that we are *stopping execution* at some point; then, the purity of $\mathscr{B}_P$ only further underlines the fact that there are no "bad" executions represented by interaction.

Going back to visitable paths, we can consider the incarnation of $\mathcal{D}_P$[3]. Once we determine the *visitable* part of the base design, which correspond the the *executable* part of the process, a few properties naturally follow.

Let P be a $MCCS$ process, $(+, \xi, I)$ be the first action of $\mathcal{D}_P$, and $|\mathcal{D}_P|$ its *incarnation*; then, let $J = \{i_m, \dots, i_n\} \subset I$ be the sub-ramification that is missing in $|\mathcal{D}_P|$, i.e. the suffixes corresponding to the never-visited branches in $\mathcal{D}_P$: by the correspondence execution-interaction, we can conclude that the elements of P corresponding to these branches will *never* be part of an execution sequence.

If we consider the incarnated behaviour $|\mathscr{B}_P|$, the relevant design of this restricted behaviour is of course $|\mathcal{D}_P|$, since the role of the restrictions is just to limit the possible interactions, that also make $[\![P]\!]$ *irregular*. We only mentioned *regularity* as a property defined on interaction paths: in regular behaviours these paths can be *mixed* and still be interaction paths on the behaviour; formally they are *closed under shuffle*, a notion that we do not need – hence have not defined – whose formal definition is found in [30]. Irregularity tells us the opposite: interaction allows only certain *ordered* paths. Indeed, they must respect the partial orders $<_P$ and $\preccurlyeq_{\mathcal{S}_P}$, thus cannot be blindly mixed (that is the very purpose of the restriction designs).

The notion of *associated execution*, and theorem 4.2.14 on the correspondence make sense if we watch at the interaction paths on the base design,

[3] The incarnation can be calculated without checking *all* the interaction paths, as shown in [30].

which are always the *maximal* ones. Indeed, restriction designs are needed to restrain the possible routes interaction can take *on the base design*, so that each possible interaction can, then, represent an admissible execution sequence. Therefore $|\mathcal{D}_P|$ tells us which is the *executable* part of a process, and how its normal forms will look like. $|\mathcal{D}_P|$ is thus enough to deduce a few properties about P, and can give us a *sufficient condition* to know when a process is *deadlock-free*:

Lemma 5.3.10. *Let P be an MCCS process.*

1. *If $|\mathcal{D}_P| = \mathcal{D}_P$, i.e. the base design is* material*, then $|\mathscr{B}_P| = \mathscr{B}_P$, and P is deadlock free.*

2. *If $|\mathcal{D}_P| \neq \mathcal{D}_P$, then let $J = \{i_m, \dots, i_n\} \subset I$ be the sub-ramification that is missing in $|\mathcal{D}_P|$, i.e. the* non visitable *part of $\mathcal{D}_P$. We have that J is* not accessible during interaction *and thus that its process counterpart represents the* common part of all normalization classes of P*; furthermore, P* is not reducible to 1.

Proof. 1. If $\mathcal{D}_P = |\mathcal{D}_P|$, then this holds for *all designs of $\mathscr{B}_P$*. This is due to how the restriction designs are built: being modifications of $\mathcal{D}_P$, and *strictly smaller* than it (they have shorter chronicles, either due to a ✠ or a pruning), then for each of their chronicles there is an interaction visiting it, since there is one visiting the corresponding chronicle on $\mathcal{D}_P$. Therefore, by theorem 4.2.14, P is *deadlock free* since any part of P is potentially synchronizable, being visitable in $[\![P]\!]$. We don't know if $P \to^* 1$, i.e. if there is a single execution path synchronizing all channels of P, but for each channel, *there is at least one execution sequence where it is synchronized with a dual*, i.e. there is not a part of P that cannot be accessed by execution. This means that there are no *cycles* in the order on synchronizations $\leqslant_{\mathcal{S}_P}$, and we can potentially communicate with any channel of P.

2. For the correspondence interaction-execution, if a part of $\mathcal{D}_P$ is *never visited*, then there are some channels of the process that *cannot be synchronized* (either there is no dual, or the execution is blocked for some other reason), and since each occurrence of a channel name is interpreted, we also know which these channels are (this can also tell us if a process is deadlocked for the meaning of [50]).

   The non visitable part is always relevant, since interaction in the *neutral part* of $\mathcal{D}_P$ (the $w[u, v]$ for *xor* conditions, which do not correspond to channels nor synchronizations) is always possible; therefore if a part

is never visited, it correspond either to locations, or synchronizations *and* locations. This also entails that P *cannot be reduced to the empty process* 1.

□

We have found a *sufficient condition* to check *on the interpretation* $[\![P]\!]$, that can tell us if P is *deadlock free*. Being *material* is a property of designs in a behaviour, thus an actual *ludical property* of the interpretation, which is what we wanted to find. However, this *is not* a *necessary condition*, since it could be the case that P is deadlock-free, but $\mathcal{D}_P$ *is not material* in $\mathscr{B}_P$. That is because there may be some other reason that *prevents execution* on some channels of P, without them being in a *deadlock*.

5.3.3 Superficial deadlocks

In this section we will focus on a specific case of deadlock, that is when it is possible to *directly act on channels part of the deadlocked cycle*, i.e. when some of these channels are *minimal prefixes* in the process, and thus their locations *principal*. In this case we talk about *superficial deadlocks*. The aim of this study is to determine if it is possible to *solve* this specific case of deadlock in a *MCCS* process P, by identifying them on $[\![P]\!]$ and adding a *minimal context* by *merging of interpretations* to "unlock" the process, and make further executions possible.

A deadlocked process looks like this:

$$P = a^1.\overline{b}^2 \mid b^3.\overline{c}^4 \mid c^5.\overline{a}^6 \mid R$$

where there are no possible synchronizations on R. We use a cycle of 3 synchronizations since it is a general representation of a cycle of length n, which arises by transitive closure; the case $n = 2$ is the minimal deadlock.

Our aim is to be able to *solve* a deadlock by giving a *minimal element* (a single channel name) able to break the cycle – by the introduction of a new synchronization such that an execution *on the channels in the deadlock* becomes possible. Of course the requisite is, at first, to be able to identify the deadlock, the channels in it, and if it can be acted on. The notion we are searching for is the one of *superficial* deadlock, in the sense that it is on the *surface* of the process, and thus we can act directly on the synchronizations in question by adding new channels. This means that each one of the synchronizations in the deadlock must be on a *minimal channel* (with respect to $<_P$), i.e. an external *prefix* of the process. We need to find a

good definition able to identify these kind of deadlock in a process in normal form.

At first, note that if a synchronization i who has a minimal channel is not part of any execution path, then we are in one of the following cases:

1. The synchronization i is blocked by another synchronization j, i.e. the non minimal channel of i is blocked by a channel of j, but i is not in a deadlock.

2. The synchronization is blocked by one or more lone channels $z_1 \ldots . z_n.a \mid \overline{a}.R$, on which there is no synchronization available ($\overline{z}_1 \vee \ldots \vee \overline{z}_n \notin P$, or are internal and not ready for synchronization).

3. The synchronization is in a deadlock.

For example in

$$P = a.R \mid b.\overline{a} \mid c.\overline{b}.S \mid d.\overline{e}.Q \mid e.f.\overline{d}$$

the synchronization $i = (a, \overline{a})$ is blocked by $j = (b, \overline{b})$, but they are not in a deadlock. The synchronization j is blocked by a lone channel c, which does not have an accessible dual. The synchronizations $u = (d, \overline{d})$ and $v = (e, \overline{e})$ are in a deadlock, and u is also blocked by another lone channel.

In case 1 we may repeat the reasoning on this synchronization j, until we fall in the second or third case, which as we saw may co-exists. We would like to be able to *identify* a deadlocked process by *unrestricted* reduction on the interpretation, i.e. reduction possibly on internal synchronizations, and give a general procedure to solve them via interpretation merging.

The property we want for a deadlock to be *superficial* is that, if the synchronizations $i_1, \ldots, i_n$ are in an deadlock, then it suffice to get rid of *any* one of them to have an execution path visiting *all the others*. Let us look again at the process

$$P = a^1.\overline{b}^2 \mid b^3.\overline{c}^4 \mid c^5.\overline{a}^6 \mid R$$

Here if we erase $u_1 = (a^1, \overline{a}^6)$ then we can perform the execution $\rightarrow_{u_2,u_3}$ with $u_2 = (\overline{b}^2, b^3)$ and $u_3 = (\overline{c^4}, c^5)$. If we erase u_2 we can do $\rightarrow_{u_3,u_1}$, and if we erase u_3 then $\rightarrow_{u_1,u_2}$ becomes possible. We can conclude that it is enough to erase a synchronization from the deadlock to break the cycle, and have an execution path including *all the other synchronizations*. On the other hand, if there is a set of not-executable synchronizations, each one with a minimal location, such that erasing any one of them generates an execution path in which all the others are synchronized – this must hold for each synchronization in the set – then we may conclude that these synchronizations are in a superficial deadlock, using this test as definition. In this way we can sensibly restrict

the deadlock cases that we are considering; for example, the test will yield the following result for the process

$$P \coloneqq a^1.\overline{b}^2 \mid b^3.\overline{c}^4.\overline{a}^5 \mid c^6.R$$

In P, erasing $u_1 = (a^1, \overline{a}^5)$ let us perform an execution on $u_2 = (\overline{b}^2, b^3)$ and then $u_3 = (\overline{c}^4, c^6)$, and erasing u_2 let us perform execution on u_3 and then u_1; but erasing u_3 do not free neither u_1 nor u_2. Thus P is in normal form, $\{u_1, u_2\}$ is a superficial deadlock, but u_3 is not part of the deadlock.

Instead, in

$$Q \coloneqq a^1.\overline{b}^2 \mid b^3.\overline{c}^4.\overline{a}^5 \mid R$$

with no available synchronizations, or deadlocks, on R, P is in normal form, and there is not any *superficial deadlock*, since erasing u_2 would not free u_1, for the interference of $\overline{c}^4$, which may be either a lone location or an internal synchronization (there is still a cycle in $\lessdot_{\mathcal{S}_P}$, though).

We are focusing on the *superficial* case because some of its channels are ready for synchronization, therefore it is possible to directly act on them, and the mutual dependence of multiple synchronizations is solvable by the elimination of any single synchronization in the cycle – i.e. always by adding a single element (any single channel dual to the external channel of any synchronization in the superficial deadlock). If a deadlocked process has an internal cycle, instead, *we would need to deal with other channels at first*, thus the deadlock itself is not what is blocking execution in the current state of the process.

We have reached a necessary and sufficient condition to check to identify superficial deadlocks; we can formalize this fact in the following definition:

Definition 5.3.11 (Superficial deadlock)**.** *Let P be a MCCS process in* normal form*, and $\vec{u} = u_1, \ldots, u_n \in \mathcal{S}_P$. Then $u_1, \ldots, u_n$ are in a* superficial deadlock *if and only if, for each i, in $P \setminus \{u_i\}$ (i.e. P where we have removed the channels of u_i) $\rightarrow_{\vec{u} \setminus \{u_i\}}$ is an admissible execution.*

The set of superficial deadlocks *of a process P is denoted $\mathscr{C}_P^{sup}$.*

At last, as a direct consequence of this fact, we can apply the reduction on $[\![P]\!]$ to check if a process P is (superficially) deadlocked *from the interpretation*, and which are the elements in the deadlock, information that the *visitable part* of $\mathcal{D}_P$ (its *incarnation*, definition 2.2.23) cannot give us. It suffices to consider, for each synchronization u, $([\![P]\!])_u$, and then check its *interaction paths* with respect to $[\![P]\!]$.

Lemma 5.3.12. *Let P be a MCCS process. If $[\![P]\!]$ yields no meaningful interactions (the only associated execution is the* empty one*) and there is a set $\{G[u_1], \ldots, G[u_n]\}$ with $2 \leq n$, $u_1, \ldots, u_n \in \mathcal{S}_P$, such that for each u_i in the sequence, in $([\![P]\!])_{u_i}$ exists an interaction path $\langle \mathcal{D} \leftarrow \mathcal{C} \rangle$ whose associated execution is* maximal*, and it visits* all the others *$G[u_j]$, with $1 \leq j \leq n$, then $u_1, \ldots, u_n$ are in a* superficial deadlock*.*

Proof. The lemma follows directly by definition of *superficial deadlock*, and theorem 4.2.14 on the correspondence execution-interaction.

□

It is actually enough to consider $u \in \mathcal{S}_P$ such that u has a *single* minimal location. However, this would require to directly look at the structure of the *restriction designs*. We can formalize this notion of *synchronization with a minimal location*, by giving the following definition

Definition 5.3.13. *Let P be a MCCS process and $u \in \mathcal{S}_P$. u is* semi-minimal *if it has only one minimal location, or, equivalently, if in $\mathcal{R}(u)$ there is a ✣ over only one $G[l]$, for some $l \in Loc_P$.*

Remark 5.3.14. *Beware that lemma 5.3.12 entails an* exponential explosion*. Indeed, two existentials quantifiers are hidden in the definition of* superficial deadlock*: P has a superficial deadlock if $\exists$ a sequence of synchronizations $u_1, \ldots, u_n$ such that $\exists$ a $\mathcal{C} \perp \mathscr{B}_P$* for each *$u_i$ such that interaction with it visits all the others u_j.*

It means taking one by one each semi-minimal synchronization*, erasing it from the interpretation by reduction on the interpretation, and checking interaction paths. Then, for each path of* maximal length*, take one by one each semi-minimal synchronization visited, and repeat the procedure, to see if there are any common synchronizations visited.*

Even with some restrictions on the length of the interaction paths (that is the reason behind the maximal lenght *requirement), and on which visited synchronizations to pick, the procedure still require, in the worst case, an* exponential number of steps with respect to the number of semi-minimal synchronizations *of a process P.*

Using lemma 5.3.12 as a *test*, we can find if there is a superficial deadlock in a *MCCS* process P, and which synchronizations are in it.

We are left with finding a procedure on $[\![P]\!]$ which will let us *unlock* the process P, i.e. after testing if its deadlocked and if there is a superficial deadlock, finding a minimal design (corresponding to a *single channel*) that will break the deadlock. Since the test automatically tells us which are the synchronizations involved, because we can check the presence of a deadlock

by checking the interactions generated by reduction on the interpretation, then it is trivial to find a minimal design which will break the cycle: we need to free any one of the *minimal locations* (with respect to $<_P$) in the deadlock, by adding a new $G[v]$, where v is a new synchronization in a *xor* relation with any of the synchronizations involved, along with a new $G[m]$, for a location m labeling the other channel of v. We can add to the interpretation an element corresponding to a *new channel labeled* by m, that is *dual to some minimal channel part of the superficial deadlock* (it doesn't matter which), which therefore *generates a new synchronization* v that can break the cycle. However, it could suffice to free other parts of the process, and still get rid of the deadlock, thus we need a procedure which will try to *blindly* free any external location, and check if there is interaction on the synchronizations in question: by definition of superficial deadlock, if there is interaction on *any one* of the channel in the deadlock, then there is interaction on *all of them*.

This procedure should be general enough to generate a *minimal context* able to *reduce* the number of superficial deadlocks in a process P. Indeed, P may have *multiple* deadlocks in it, and we would like to be able to solve them one by one by adding just what we need. By performing execution on the process directly, we could use a brute-force but effective procedure to solve the deadlock, the trivial one being adding in parallel composition the *dual process*, a dual channel for each one in the process, each in parallel composition with the others, reducing thus the process to 1. Of course this trivial solution is not interesting, and needs execution on the process, that in the worst case makes us try all the possible channels and paths.

Our aim is, instead, to act only *on the interpretation* by performing interaction. What we know is that, given a deadlocked process P, exists a context Q such that $P \mid Q \to^* 1$, which is the trivial context from above. What we are interested in is finding a context M minimal in *size* such that

- $M \mid P \to_{v,\vec{u}} P'$ for some synchronizations $v, \vec{u}$;
- $\mathscr{C}_{P'} \subset \mathscr{C}_P$ (there is at least one less cycle in $\preccurlyeq_{\mathcal{S}_P}$), and $\exists \sigma \in \mathscr{C}_P^{\text{sup}}$ such that $\vec{u} \in \sigma$.

By repeating the procedure, we can find an M^* such that

$$M^* \mid P \to^* 1$$

Using reduction on the interpretation and interaction we can avoid any reference to execution on the process, and act only and directly on the interpretation. As always we can work on $\mathcal{D}_P$, and then adjust the restriction designs to meet our needs.

Definition 5.3.15. *Let P be a* deadlocked *MCCS process, and let* $(+, \xi, I = \{1, \ldots, n-1\})$ *be the* first action *of $\mathcal{D}_P$. For any location $l \in Loc_P$ such that $G[l]$ has no $G[n]$, for another location n, as requisite in $[\![P]\!]$ (i.e. such that $G[n]$ ends with $\maltese$ in a restriction design of $\mathcal{R}(l)$)*[4]*, let $([\![l]\!], [\ ]_{[\![l]\!]})$ be the pair composed by the following design, noted $[\![l]\!]$,*

$$\dfrac{\dfrac{\dfrac{\xi.n.1.1 \vdash}{\vdash \xi.n.1}}{\xi.n \vdash} \quad \dfrac{\dfrac{\xi.n+1.1.1 \vdash}{\vdash \xi.n+1.1}}{\xi.n+1 \vdash} \quad \dfrac{\xi.n+2.1.1 \vdash \quad \xi.n+2.2.1 \vdash}{\vdash \xi.n+2.1 \quad \vdash \xi.n+2.2} \quad \cdots \quad \dfrac{\xi.n+i.1.1 \vdash \quad \xi.n+i.2.1 \vdash}{\vdash \xi.n+i.1 \quad \vdash \xi.n+i.2}}{\xi.n+2 \vdash \qquad\qquad\qquad\qquad \xi.n+i \vdash}}{\vdash \xi}$$

where each $\xi.n + j = xor^{u_j}$ & xor^{v}, with $2 \leq j \leq i$, and $i - 2$ is the number of u_j such that $G[u_j]$ is a requisite for $G[l]$ in $[\![P]\!]$ (the number of $u_j \in \mathcal{S}_P$ to which l belongs);

and the assignment function $[\ ]_{[\![l]\!]}$, *such that for a* new location m, *and a* new synchronization v, $[v]_{[\![l]\!]} = \xi.n$, $[m]_{[\![l]\!]} = \xi.n + 1$ *and* $[(u_j, v)]_{[\![l]\!]} = xor^{u_j}$ & $xor^{v} = \xi.n + i$ *for each u_j (we are treating the pairs (u_j, v) as* new *xor* conditions*).*

It follows:

Definition 5.3.16. *Let $[\![l]\!] = ([\![l]\!], [\ ]_{[\![l]\!]})$. The blind merging on l in $[\![P]\!]$ is*

$$[\![P]\!] \oplus [\![l]\!]$$

with the addition of the following restriction designs before the bi-orthogonal closure of the merging:

- *a new* restriction design *with $G[\xi.n]$ as requisite for $G[\xi.n + 1]$, meaning that the* new location m belongs to the synchronization v*;*

- *two restrictions designs for each clause $(u_j, \xi.n)$, as defined for any clause of $\mathcal{X}_P$, and the addition of a $\maltese$ in $\mathcal{R}(l)_{[\![P]\!] \oplus [\![l]\!]}$ as last rule of $G[\xi.n]$ – since l is minimal there is no immediate predecessor, and the only restriction is the design giving precedence to the synchronizations to which l belongs. This means that v is in* conflict *with any u_j to which l belongs, i.e. l* also belongs to *v, therefore* the channels of v are labeled by the locations l and m.

[4] This means that l is *minimal*.

The blind merging is taking, one at a time, the $G[l]$ corresponding to external locations, and adding to $\mathcal{D}_P$ an address able to free this locations, along with a synchronization in *conflict* (the *xor* relation) with each of the synchronization to which l belongs. This corresponds to adding by parallel composition a single channel on the process, able to synchronize with the channel named by this external location. Then we can check the interactions of the merged interpretations and see if there are any interaction paths describing execution on any synchronizations inside *one of the superficial deadlocks of* P, which we can identify by lemma 5.3.12. If interaction visits any one of them, then, there is an interaction of *maximal lenght* visiting all of them, except the one synchronization whose external location has been erased in order to break the deadlock; otherwise we may "reset" the interpretation *back to* $[\![P]\!]$, and try on another external location, until the deadlock is solved.

Remark 5.3.17. *Note that we said* by lemma 5.3.12*, but here we are satisfied by deleting a single location of a synchronization in the deadlock, and not both. That is because the only relevant locations blocking the execution are the external ones; however we defined the test on* synchronizations *in order to make it compatible with* reduction *on the interpretation* $[\![P]\!]$*, which must be on synchronizations to make sense (*in order to correspond to execution *on* P*).*

In the case of superficial deadlocks, we have that

Corollary 5.3.18. *Let* P *be a MCCS process such that* $[\![P]\!]$ *is in normal form. Then, the set of superficial deadlocks of* P*, noted* $\mathscr{C}_P^{sup}$*, is* non empty *if there is a sequence* $\sigma = \{u_1, \ldots, u_2\}$ *of semi-minimal synchronizations such that the* superficial deadlock test *(lemma 5.3.12) on* σ *ends successfully. In this case* P *is also* deadlocked*.*

Proof. The proof is straightforward by definition of *deadlocked process* (definition 5.3.5), of *superficial deadlock* (definition 5.3.11), and for the *superficial deadlock test* (lemma 5.3.12). □

Eventually, we can prove the following:

Lemma 5.3.19. *Let* P *be a MCCS deadlocked process. Then, if* $\mathscr{C}_P^{sup} \neq \varnothing$*, for any location* l *labeling a channel* u *belonging to a* $\sigma \in \mathscr{C}_P^{sup}$*,* $[\![l]\!]$ *is such that:*

1. $[\![P]\!] \mid [\![l]\!] = [\![Q]\!]$*, where* Q *is an MCCS process* not in normal form *such that* $Q = P \mid \overline{a}^m$*, with* $a^l \in P$*.*

2. *The reduction on the new synchronization v introduced by the blind merging is the interpretation of a process Q' such that $[\![Q]\!] \rightsquigarrow_v [\![Q']\!] = ([\![Q]\!])_v$ and $\mathscr{C}_{Q'} \subset \mathscr{C}_P$.*

Proof. For the first point, by the *blind merging on l* we are adding a new location m and a synchronization between the channels of l and m (which means that m is labeling the dual location of l) along with all its possible *xor* conditions with other synchronizations to which l belongs. Since the *merging of interpretations* corresponds to parallel composition, the result of the merging is the interpretation of $Q = P \mid \overline{a}^m$. Q is not in normal form since there is an interaction visiting the branch $G[v]$, assigned to a new synchronization v, and thus by theorem 4.2.14 on the correspondence execution-interaction, there is an execution $\rightarrow_{v=(\overline{a}^m, a^l)}$ on Q.

For the second point, reduction on v corresponds in the process to an execution step on v, which erases the channel a^l, and thus breaks the *superficial deadlock* in which a^l was. Since the addition of $\overline{a}^m$ by parallel composition is not creating any new cycles, then $\mathscr{C}_{Q'} \subset \mathscr{C}_P$.

□

For a more general result, in the case where there are no superficial deadlocks, in order to find a minimal context unlocking the process by reducing the number of cycles, we may need to do several combinations of blind merging until we find one that makes a deadlock superficial. This method would not be much different than a combinatorial method on the process itself; as is the superficial deadlock test, since it entails to check a large number of interactions. However, once we have found a superficial deadlock, building a minimal context to solve the deadlock become extremely easy. Let σ be a superficial deadlock, and $\{l_1, \ldots, l_n\}$ the set of external locations of synchronizations in σ; it suffices to merge the interpretation with $[\![l_i]\!]$ for any $l_i \in \{l_1, \ldots, l_n\}$. All the the successive mergings with $[\![m_i]\!]$, …, $[\![h_i]\!]$, each time for an external location of a synchronization in another superficial deadlock σ_i, will give us a minimal context which will break all the superficial deadlocks in the process – possibly counting those that arise in the procedure, that is those who become superficial after unlocking the previous ones. A minimal such context will thus look like this:

$$[\![l_i]\!] \mid [\![m_i]\!] \mid \ \ldots \ \mid [\![h_i]\!]$$

However this is only *one* minimal context, that can solve all the superficial deadlocks, while we want to determine *all* such minimal contexts.

Looking at processes we can notice the following facts: if $a^1 \mid b^2 \mid \ldots \mid c^n$ is a minimal context which removes all the superficial deadlocks of a process P, then also the following combinations are:

- $a^1.b^2.\cdots.d^{n-1}.c^n$
- $a^1.c^n \mid b^2.d^{n-1} \mid \cdots$
- $a^1.b^2 \mid \cdots \mid d^{n-1}.c^n$
- $\vdots$

Indeed, if each channel of the minimal context is found sequentially (by successive tests via reduction on the interpretation), then the structure of the context does not matter, as long as the sequential order is respected, which means that the n-th found channel must never be prefix of the channel named by the location $n-1$, i.e. found at the $n-1$-th step; however they can be independent with respect to $<_P$, i.e. in parallel composition.

This is because at each step we are adding a channel solving one particular deadlock, thus we just need to have the n-th channel available at the n-th step to be sure that there is at least *one* execution path that can solve all the superficial deadlocks of P, and that all such contexts are of minimal *size* (with respect to the number of channels).

Switching our focus to the interpretation, by using the merging we cannot quite catch all the combinations of the channels that can be considered minimal contexts. If we build them one by one, i.e. our $[\![l_i]\!]$, $[\![m_i]\!]$, etc., then we can put them together only in parallel composition (that is, the merging of interpretations), but not in an arbitrary prefix order. Yet, the merging can still serve this purpose with some slight modifications, that do not touch the core operation $\odot$ on the base designs. If we want to represent the *action prefix* operation $a^l.P$ on P, we can note that:

- l is smaller than all locations on P (the order get extended).
- No new synchronization is generated, since a is not added in parallel composition.

This bring us close to the question of *type assignment*. While our interpretation is not the result of a *typing system*, we can still try to define operations on the interpretation that correspond to the basic syntactical constructors of $MCCS$ processes (for the moment, without sums or recursion), being hence able to build the interpretation of a process by using these operations on elementary interpretations, that in our case are still *set of designs*. The next step is so to find an operation corresponding to the action prefix.

5.4 Interpreting the action prefix

The operation $a^l.P$ on a *MCCS* process P entails just the addition of a channel (and its location), and an extension of the partial order $<_P$. In $[\![P]\!]$, adding a channel as prefix – say a^l – entails the addition of a branch in $\mathcal{D}_P$ corresponding to that channel in the assignment $[\ \ ]_P$, and an extension of the partial order, for each minimal location $m_1, \ldots, m_k$ of P, by a new $\mathcal{R}(m_i)$ giving precedence to l with respect to $m_1, \ldots, m_k$, during interaction, with $1 \leq i \leq k$ (restriction designs that must be added before the bi-orthogonal closure). The final step is adding a^l in the domain of $[\ \ ]_P$, as in the merging of interpretations. Therefore:

Definition 5.4.1. *Let $\mathcal{D}_{a^l}$ be the single branch positive design*

$$\dfrac{\dfrac{\dfrac{}{\xi.n+1.1.1 \vdash}}{\vdash \xi.n+1.1}\ (+, \xi.n+1.1, \{1\})}{\xi.n+1 \vdash}\ (-, \xi.n+1, \{\{1\}\})}{\vdash \xi}\ (+, \xi, \{n+1\})$$

with n the cardinality of the directory of $\mathscr{B}_P$ (i.e. the cardinality of the ramification *of the first action of $\mathcal{D}_P$).*

We can now give the following definition;

Definition 5.4.2. *Let P be a MCCS process, $a^l \notin \mathcal{A}_P$ and*

$$\mathscr{B}_{a^l.P} = (\mathcal{D}_P \odot \mathcal{D}_{a^l} \cup \mathcal{R}(P)_{\mathcal{D}_P \odot \mathcal{D}_{a^l}} \cup \mathcal{R}(m_1) \cup \ldots \cup \mathcal{R}(m_k))^{\perp\perp}$$

where, for $1 \leq i \leq k$, each m_i is a minimal location *of P, and each $\mathcal{R}(m_i)$ is a restriction built on $\mathcal{D}_P \odot \mathcal{D}_{a^l}$, with $\maltese$ over $G[l]$, and $G[m_i]$ pruned. Then, the* **action prefix of a^l on $[\![P]\!]$** *is*

$$[\![a^l.P]\!] = (\mathscr{B}_{a^l.P}, [\ \ ]_P \cup [a^l = \xi.n+1])$$

Using this operation on $[\![l_i]\!]$, as $[\![l_1.l_2 \ldots l_n]\!] = [\![l_1]\!].[\![l_2]\!] \ldots [\![l_n]\!]$ for example, we can define the intended combinations of minimal contexts, given an ordered sequence of channels that can unlock the superficial deadlocks of the process. The procedure is nothing more than a merging of designs $[\![l_1]\!] \odot [\![l_2]\!] \odot \cdots \odot [\![l_n]\!]$, with the addition of the prefix order via a $\mathcal{R}(l_i)$ for each l_i in the sequence, defined on the merging itself.

Therefore, given a deadlocked process P and a sequence $([\![l_i]\!], [\![m_i]\!], \ldots, [\![h_i]\!])$ which can reduce $\mathscr{C}_P^{sup}$ to $\varnothing$ by successive mergings of the elements in the sequence with $[\![P]\!]$, then a *minimal context* achieving the same reduction with

a single merging is *any combination* of the elements of the sequence by *action prefix on the interpretation* together with *mergings* respecting the order of the sequence, i.e. if $[\![l_i]\!]$ is found before $[\![m_i]\!]$, then we cannot have $[\![m_i]\!].[\![l_i]\!]$.

If instead $\mathscr{C}_P^{sup} = \varnothing$ then we need to bring us back to a case where there is at least a superficial deadlock (since we cannot act directly on any deadlock), thus blind merge $[\![P]\!]$ with $[\![l]\!]$ for any minimal location l, then check if there are superficial deadlock (via the interpretation reduction test). If there aren't any, then repeat the procedure for another external location, and so on, possibly for every combination of external locations. This method is not simpler or faster, as said, than acting directly on the process trying all the possible combinations of channels, however we *do not need execution* to know if there is a superficial deadlock, but interpretation reduction and interaction. In conclusion, we have obtained a characterization in ludics of the action prefix operation, and found how to add a minimal resource to break a superficial deadlock.

5.5 Independent reductions

The normalization property is trivial for processes of *MCCS* (and *CCS* without replication in general), since *there are no infinite executions*, making normalization always *strong*. Thus, a standard normalization theorem, i.e. that every typable process is strongly normalizing (every execution path leads to a normal form) is a trivial result. The non trivial fact is that *every process* is interpretable in ludics, even deadlocked ones, which for interaction poses no problem at all.

As we saw, in our case *reduction on the interpretation* gives an expected property, i.e. a sort of strong inclusion of the interpretations, which is consistent with the meaning we gave to the dynamic of processes. We can actually have a version of normalization that takes into account *all the different execution paths*, and hence normal forms, while giving a *diamond property* for two synchronizations which are neither ordered (this is possible, being $\preccurlyeq_{\mathcal{S}_P}$ a partial order) nor in *conflict*: performing execution on these two synchronizations in any order yields the same result. In this case we talk about *independent synchronizations*, the equivalent of *concurrent events* in event structures, as defined in [27], which are events neither in a causal (or enabling) nor in a conflict relation. At the same time we can talk about *independent executions*: execution sequences separated due to a *xor* condition (or a sum, though, which we still have to interpret, but can be modeled by *xor* conditions), which meet at the end yielding the same result.

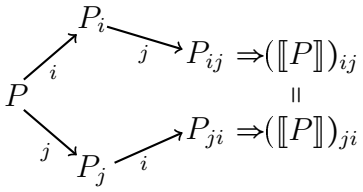
Definition 5.5.1. *We say that*

1. *two* synchronizations $u, v \in \mathcal{S}_P$ *are* independent *if neither* $u <_{\mathcal{S}_P} v$ *nor* $v <_{\mathcal{S}_P} u$, *and* $(u, v) \notin \mathcal{X}_P$;

2. *two* execution sequences $P \rightarrow_{\vec{u}} P'$ *and* $P \rightarrow_{\vec{v}} P''$, *such that* $\vec{u} \neq \vec{v}$, *are* independent *if* $P' = P''$. *We say that they* commute *if both* $P \rightarrow_{\vec{u},\vec{v}}$ *and* $P \rightarrow_{\vec{v},\vec{u}}$ *are* admissible *executions.*

Now we can give the following result

Lemma 5.5.2. *Let* P *be a MCCS process, and* $i, j \in \mathcal{S}_P$ *such that* i *and* j *are* independent *synchronizations. Then it holds the following property:*



i.e. the interpretation of P_{ij} *– the term resulting from the execution* $\rightarrow_{ij}$, *and the interpretation of* P_{ji} *– the term resulting from the execution* $\rightarrow_{ji}$, *are the same.*

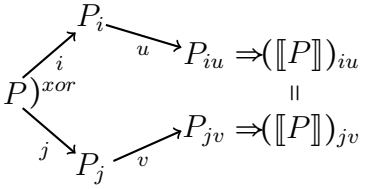
Proof. The property holds by reduction on the interpretation and definition of independent synchronizations.

□

The property is formulated and holds for *one step* executions: for an empty execution the diagram would obviously collapse. Note that even if the synchronizations are *not independent*, the result of an execution path could be the very same. Indeed, if

$$P = a^1.Q \mid \overline{a}^2.R \mid a^3.S \mid \overline{a}^4.T$$

let $i = (a^1, \overline{a}^2)$, $j = (\overline{a}^2, a^3)$, $u = (a^3, \overline{a}^4)$ and $v = (\overline{a}^4, a^1)$;
we have $(i, j); (j, u); (u, v); (v, i) \in \mathcal{X}_P$. However, it holds that:



and for P_{ui} and P_{vj} as well (always for reduction on the interpretation).

Still, the theorem *does not hold* for not independent synchronizations, as a *xor* condition leads, in general, to two different normal forms. For instance in

$$P = a^1.Q \mid \overline{a}^2.R \mid a^3.S$$

with $a \notin Q, R, S$, let $u = (a^1, \overline{a}^2)$ and $v = (a^3, \overline{a}^2$. Then $P_{u\vec{i}} \neq P_{v\vec{j}}$, for all execution sequences $\vec{i}, \vec{j}$, and, for reduction on the interpretation $(\llbracket P \rrbracket)_{u,\vec{i}} \neq (\llbracket P \rrbracket)_{v,\vec{j}}$; i.e. the theorem does not hold.

Remark 5.5.3. *Note that in a setting with the non-deterministic choice $+$, which we will introduce next, each sum $P + Q$, if $P \neq Q$, determines a* permanent fork *in execution sequences with the environment, since only one member is kept once synchronization on that member has occurred.*

For *MCCS* processes we have that each execution path leads to a normal form, and two normal forms are different when a *xor* condition forks permanently an execution sequence (which is not always the case). In these cases we talk about multiple normal forms, each associated to a different *normalization class* (definition 5.3.4) whose elements are all the execution sequences leading to that particular normal form.

6. On the extension of the interpretation to CCS

The objective of this chapter is to extend the interpretation to CCS, including the **non deterministic choice**, or sum, denoted +, the **private name operator**, called *new* and denoted ν, and eventually hint at some possible directions to represent **recursive definitions** (defined in section 3.2) inside standard ludics.

It is divided into two main parts: the first, composed of section 6.1 and section 6.2, is dedicated to *replication-free CCS* (without *recursion operator*), and shows how we can *extend the interpretation* to include both + and ν by using the *same tools* we already built and defined for the interpretation of $MCCS$. The interpretation is thus directly extended to + and ν *without modifying* the part about $MCCS$, and by using the very same machinery. All we need to do is explain how we are interpreting these new operations with the tools at our disposal, and how to adjust the proofs of the results we obtained in order to include these two new operators.

The second part, starting from subsection 6.3.3, instead is rather experimental, and discusses the problem of extending the interpretation to *recursion and replication*. It is the most delicate part of the interpretation, since, in the case of a un-restricted recursion, it requires to *extend ludics to non-linearity*. Being unable to reach a definitive conclusion, we stop with some suggestions on how recursion could be added to our interpretation, and a few attempted solutions that point at some research directions to explore, but that require further study.

The first part starts with a discussion on the non deterministic choice +, and its role in CCS processes. It is one of the main reasons, along with the *non uniqueness* of channel names, and thus the *conflict relation*, that make execution of CCS processes *non confluent*. By *choosing* to synchronize a channel of one member of +, the other is *discarded from the process*, thus

forming a *permanent fork* in an *execution sequence*, that eventually reaches different *normal forms*.

This property of the sum can be easily represented by the *conflict relation* $\mathcal{X}_P$. We can extend $\mathcal{X}_P$ with a new relation $+_P$ (definition 6.1.3), that includes pairs of synchronizations (u, v), where each element belongs to a different process member of a $+$. For instance, if $a^l.P + b^m.Q$, we will add in $+_P$ pairs of synchronizations of the form $((a^l, \overline{a}^x), (b^m, \overline{b}^y))$, as well as conflict-pairs on *minimal locations* of the two processes, as (l, m); then add the corresponding *restriction designs* for these pairs, as defined for standard pairs $(w, x) \in \mathcal{X}_P$.

Since no new construct is used, all the previous theorems will still hold for *replication-free CCS*, with some slight modifications, except theorem 5.2.12 about the diamond property for the *reduction on the interpretation*, where the (definition *trimming*) 5.2.2 needs to be extended with the specific case of *non deterministic choices*.

In section 6.2 we discuss the *private name operator* ν, whose role is to *isolate* some specific channels of a process P, in order to *prevent communication outside its scope*. In other words, ν is used to make some channel name occurrences *private*, i.e. channels that can communicate only between themselves, and are *inaccessible* by the *environment*: they are treated as *always fresh channels*, thus even if a dual *channel name* is present in a processes added by *parallel composition*, it cannot communicate with its duals inside the scope of the ν.

The simple solution to interpret the ν is to add a *second label* to channel names that are *inside the scope of a* ν. For instance, if $P = \nu a(a^1.b^2.\overline{a}^3) \mid Q$, then we rewrite the process as $a^1_\nu.b^2.\overline{a}^3_\nu \mid Q$ – which is consistent with the *syntactic rules* of ν. In this way, when interpreting the process P, we can *identify* these tagged channels, and *not consider synchronizations* pairs as $(a_\nu, \overline{a})$, i.e. pairs formed by a tagged and a non-tagged channel, even if duals: they *are not synchronizable pairs*, thus *do not appear as elements of* $\mathcal{S}_P$, and therefore will be simply *not interpreted* as branches of $\mathcal{D}_P$.

The second part starts with a discussion of replication in the πI-calculus of Varacca-Yoshida ([67]), in subsection 6.3.1. The controlled replication, and restrictions on the syntax to ensure *linearity*, make this kind of *soft* replication possibly easier to interpret, seemingly without the need to extend the syntax of ludics itself. However its not clear if its actually possible to interpret any possible process obtained with the syntax of the πI-calculus in the non-extended ludics syntax, especially considering the limit cases.

In subsection 6.3.2 a more solid solution is proposed, by *reformulating the interpretation* in Terui's *computational ludics* ([66]). In the syntax developed by K. Terui, closer to the one of *higher order* π*-calculus*, ludics is

naturally *non-linear* and admits *recursive design generators*, that seem to be perfectly suited to interpret standard recursion of CCS. However, Terui's syntax does have a few drawbacks in the *clarity of the correspondence*: absolute addresses disappear in favor of *pseudo-channel names*, therefore the clear $1-to-1$ correspondence between *addresses* and *elements of a process* becomes impossible to preserve.

Eventually in subsection 6.3.3, pushed by the need to remain as close as possible to the standard syntax and *interaction* of ludics, instead of reformulating our interpretation inside Terui's syntax, our attempted solution is to go the other way around, and reformulate Terui's *finite design generators* inside a *slightly-extended* syntax of ludics, using a special *exponential address* $\xi*$, whose role is close to the *linear logic exponential* $!A$.

We also discuss the non-linear version of *ludics with repetitions* of [6], as another possible solution to the problem of recursive definition, that however does not fit our needs.

6.1 Sum of processes

The non deterministic choice, or sum, of P and Q, denoted $P+Q$ (defined in 3.2.1) is an operation strongly related to $\oplus$, the linear logic connective representing an *external choice*. Indeed, if we have $A \oplus B$ we *do not know* on which member of the $\oplus$ we must take a further step in cut-elimination or proof search, as for focalization (subsection 2.2.1), which implies that the choice is not up to us, but depends upon the proof[1].

In the setting of CCS, the sum $+$ is a mutual exclusion between its two members, which is waiting for an *external* choice[2], i.e. a context in parallel composition, to select one process member of the sum to use, by synchronizing with one of its channels, dropping the other for that execution sequence. The $+$ may be seen as a *primitive* operation, precedent to parallel composition, as in [46], whose behaviour is defined through execution.

Execution is extended with the sum in the following way:

$$P = a.P' + Q' \mid \overline{a}.P'' + Q'' \rightarrow_{a,\overline{a}} P' \mid P''$$

as another instance we have

$$a.P' + b.P'' \mid \overline{a}.Q \mid \overline{b}.R \rightarrow_{a,\overline{a}} P' \mid Q \mid \overline{b}.R$$

[1] About this interpretation of connectives, and meaning of *internal* and *external* choice, see [40].

[2] As presented in [4].

or

$$a.P' + b.P'' \mid \overline{a}.Q \mid \overline{b}.R \rightarrow_{b,\overline{b}} P'' \mid R \mid \overline{a}.Q$$

and *structural congruence* is naturally extended to the sum:

Definition 6.1.1 (Structural congruence)**.** Structural congruence *is the smallest congruence relation, noted* $\equiv$*, such that parallel composition and* $+$ *are* commutative and associative*,* $+$ *also idempotent, and* 1 *is the* neutral element *of parallel composition and sum; i.e.* $P + P \mid R + Q \equiv Q + R \mid P$*;* $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$*;* $P \mid 1 \equiv P$ *and* $P + 1 = P$*.*

From the point of view of interaction, the non deterministic choice is equivalent to a *xor* condition on the *minimal channels* of the processes involved. For instance, if we take the process above

$$P = a.P' + b.P'' \mid \overline{a}.Q \mid \overline{b}.R,$$

we could interpret the sum with a xor clause on the synchronizations $u = (a, \overline{a})$ and $v = (b, \overline{b})$, thus add $(u, v) \in \mathcal{X}_P$. This is congruent with event structures (definition 3.2.9), where it is modeled by a *conflict* between the external channels of the processes. Indeed, once we have performed execution on one process in the sum, the other is excluded from the same execution sequence. On the other hand, both are possible until a choice is made, and choosing one of the two synchronizations u and v, by transitivity of $<_P$ and $\leqslant_{\mathcal{S}_P}$, necessarily excludes from any possible continuation all the internal synchronizations of P' (if v is chosen) or P'' (if u is chosen) – as in event structure, where the conflict relation *is hereditary w.r.t. causal implication.* This is effectively described by a **xor condition**, that can, therefore, mimic the non deterministic sum. So, if $P = a^l.P' + b^m.P'' \mid Q$, a possible solution would be to extend $\mathcal{X}_P$ with any pair (u, v) of synchronizations such that $u = (a^l, x^n)$ and $v = (b^m, y^h)$, with x varying on $\overline{a} \in \mathcal{S}_Q$, and y on $\overline{b} \in \mathcal{S}_Q$, though they have no channel in common.

However problems arise during *reduction on the interpretation.* Indeed, by simply putting a *xor* condition between all the synchronizations on the most external prefix of P and Q, then we are coding and keeping into the base design both members of the sum, *even after a reduction on the interpretation* on one of the the synchronizations in the new *xor* clause. Yet, by the execution rule, only one member is kept after synchronization, and the other is *erased* from the process. Therefore theorem 5.2.12 would fail, since if

$$R = a^l.P' + Q' \mid \overline{a^m}.P'' + Q'' \rightarrow_{u=(a^l,\overline{a}^m)} P' \mid P'' = R'$$

then $(\llbracket R \rrbracket)_u$, the reduction of $\llbracket R \rrbracket$ on the synchronization u, would **not be equal to** $\llbracket R' \rrbracket$, which **has no trace at all of** Q' and Q'', that instead are

still present in $([\![R]\!])_u$. This suggests that the sum has a *stronger commitment* than a *xor*, and needs something more than single standard *xor* clause, especially on the side of the reduction on the interpretation.

We saw in section 5.5, for *independent reduction paths*, that a fork in the execution sequence due to a *xor* is sometimes recoverable, and we could find two forked paths which in the end yield the same result. However this *is not the case* for the sum, since once a choice has been made the discarded process is lost for that execution sequence.

Still, a *xor* generalization is definitely suited to represent the sum, with the due precautions. From the point of view of the graph $\mathscr{G}_P$, this different xor would imply the erasing of not only a few **nodes** corresponding to the synchronizations in the added *xor* clause, but a **whole sub-graph** of **non-executed synchronizations**.

Example 6.1.2. *Regarding the* reduction on the interpretation *a more careful approach to + may require us to erase from the base design the branches associated to the discarded process (or processes). Let*

$$R = a^l.P' + b^n.Q' \mid \overline{a^m}.P'' + \overline{b}^o.Q''$$

After an execution step on $(a^l, \overline{a}^m)$ *we have*

$$R = a^l.P' + b^n.Q' \mid \overline{a^m}.P'' + \overline{b}^o.Q'' \rightarrow_{u=(a^l,\overline{a}^m)} P' \mid P'' = R'$$

It is clear that all branches associated to elements of Q' and Q'' *should be* erased *in* $(\mathcal{D}_R)_u$. *Therefore we need a way to* identify them on the interpretation itself, *by coding a* local conflict relation. *The way to achieve this goal is to put in a* new conflict relation *not only synchronizations, but also the* prefix channels *(all the minimal channels) of the two processes in the sum through their associated locations, to be able to* transitively recover all their respective channels, *through the partial order* $<_P$. *In our example, we would add the special xor clauses* (a^l, b^n) *and* $(\overline{a}^m, \overline{b}^o)$, *or simply* (l, n) *and* (m, o).

Adding clauses to $\mathcal{X}_P$ is still be an effective choice with respect to interaction, and since we only need the same relations and machinery already used for $MCCS$, the implementation of the sum in such a way would be rather natural. What the execution rule tells us is that there is a mutual exclusion between the whole processes P' and Q' (the same for P'' and Q''), starting from their prefix channel(s). The hard part is to be able to *identify* which channel occurrence belongs to which process in the sum *from the interpretation*; however we actually only need the *minimal ones* to transitively exclude all the others.

The solution is to add *xor*-like clauses on both synchronizations and locations, and then let reduction on the interpretation erase the one location which was excluded from execution on the sum, all its *greater locations* (thus channels), all synchronizations on these greater locations (simply by watching at restriction designs), and all xor conditions on these synchronizations, making an *hereditary erasing* with respect to $<_P$.

To summarize:

- We need to put in conflict synchronizations which are, respectively, on an external channel/minimal location of a different member of the sum. In example 6.1.2, between any pair (u, v) of synchronizations such that $u = (a^l, x^n)$ and $v = (b^m, y^h)$, with x varying on $\overline{a} \in \mathcal{S}_{P'}$, and y on $\overline{b} \in \mathcal{S}_{Q'}$.

  This is because *freedom requisites* (definition 4.1.11) require *interaction* to visit synchronizations *before* locations. Therefore an effective forking in interaction paths, and mutual exclusion of processes, must start with a *conflict relation between synchronizations*, that are visited just after their *xor* conditions (due to the restriction designs: subsubsection 4.1.3)

- We need a new kind of conflict relation for locations, i.e. pair of, in general, *not dual* channel name occurrences. This is because the *only way* to *transitively recover a sub-process* inside the interpretation is through the coded partial order on locations. In example 6.1.2, we need the conflict-pairs (a^l, b^n) and $(\overline{a}^m, \overline{b}^o)$; then, by checking the *restriction designs* where a^l, b^n, $\overline{a}^m$ and $\overline{b}^o$ are a requisite for an address assigned to *another location*, we can transitively recover the whole processes P', Q', P'' and Q''.

  In this way we can make theorem 5.2.12 and the reduction on the interpretation work as intended. We need to extend the *trimming* (definition 5.2.2) by taking into account if the synchronization on which we are performing the trimming is the prefix of a process in a sum or not.

Therefore, we can give the following definition

Definition 6.1.3 ($+_P$)**.** *Let P be a* replication-free *CCS process. The* **sum conflict relation** *of P is the set $+_P$, defined by cases:*

- *If P has* no sums*, then $+_P = \varnothing$.*
- *If P has at least one* sum*, and thus is, without loss of generality, of the form*

$$P = (a^i.P' \mid b^j.P'' \mid \cdots) + (c^k.Q' \mid d^h.Q'' \mid \cdots) \mid T$$

$$\textit{then let } (a^i.P' \mid b^j.P'' \mid \cdots) = R \textit{ and } (c^k.Q' \mid d^h.Q'' \mid \cdots) = R';$$
$$+_P = \{(x^n, y^m) \mid x \in \mathcal{A}_R, y \in \mathcal{A}'_R; n \in Loc_R, m \in Loc_{R'}, \textit{and}$$
$$n \textit{ is a minimal location of } Loc_R, m \textit{ is a minimal location of } Loc_{R'}\}$$
$$\cup$$
$$\{(u, v) \mid u = (x^n, w), v = (y^m, z) \wedge$$
$$w, z \in \mathcal{A}_T; u, v \in \mathcal{S}_P; x^n \in R, y^m \in R', \textit{and}$$
$$n \textit{ a minimal location of } Loc_R, m \textit{ a minimal location of } Loc_{R'}\}.$$

Note that we explicit the channel names labeled by their locations for clarity, but the pairs forming the set of $+_P$ can be considered as pairs of only *locations* (that stand for particular occurrences of channel names); therefore, when the channel names are either irrelevant or clear from the context, for brevity we note pairs $(a^l, b^m) \in +_P$ with just their locations (l, m).

Now, we need to explain how elements of $+_P$ are integrated in the **base design** of a process P, and which **restriction designs** are defined on them. The simple answer is the expected one, i.e. there are *xor* branches for each pair of $+_P$, and a *restriction design* defined exactly as the ones in $\mathcal{R}(\mathcal{X}_P)$ (definition 4.1.15).

Definition 6.1.4 (Base design with +)**.** *Let P be a* replication free *CCS process. Then, $\mathcal{D}_P$ is built as in definition 4.1.14 with the addition of the following branches:*

- *For each $(u, v) \in +_P$, with $u, v \in \mathcal{S}_P$, is added a branch $w[u, v]$*
- *For each $(l, m) \in +_P$, $l, m \in Loc_P$, is added a branch $w[l, m]$.*

The branches $w[u, v]$ and $w[l, m]$ are defined as in 4.1.14. Therefore, $\mathcal{D}_P$ for a replication free *CCS process P becomes:*

$$\mathcal{D}_P = (\bigotimes_{x \in \mathcal{S}_P \cup Loc_P,\ (u,v) \in \mathcal{X}_P \cup +_P,\ (l,m) \in +_P} \{G[x], w[u, v], w[l, m]\}).$$

And, for restriction designs, we have:

Definition 6.1.5 (Restriction designs for +)**.** *Let P be a* replication free *CCS process, let $(u, v) \in +_P$, with $u, v \in \mathcal{S}_P$ and $(l, m) \in +_P$ with $l, m \in Loc_P$; then both $\mathcal{R}(u, v)$ and $\mathcal{R}(l, m)$ are defined as $\mathcal{R}(i, j)$ for $(i, j) \in \mathcal{X}_P$ (definition 4.1.15).*

With $\mathcal{R}(+_P)$ we note the set *of all $\mathcal{R}(u, v)$ and $\mathcal{R}(l, m)$, for each $(u, v), (l, m) \in +_P$; therefore*

$$\mathcal{R}(P) = \mathcal{R}(\mathcal{S}_P) \cup \mathcal{R}(Loc_P) \cup \mathcal{R}(\mathcal{X}_P) \cup \mathcal{R}(+_P).$$

The *assignment function* must be extended accordingly, with addresses associated to the pairs $(u,v),(l,m) \in +_P$, such that $[(u,v)/(l,m)]_P = \xi.n$ for some n. The interpretation of a replication-free CCS process P, still denoted $[\![P]\!]$, becomes thus

$$((\{\mathcal{D}_P\} \cup \mathcal{R}(P))^{\perp\perp}, [\ \ ]_P)$$

with $\mathcal{D}_P$ as in definition 6.1.4, and $\mathcal{R}(P)$ as in definition 6.1.5.

We also need to make sure that all the previous results are preserved by this extension to +. In most cases they obviously still hold, since our interpretation have the very same form as for $MCCS$ processes, with the addition of a few *xor* clauses.

Regarding the *core theorems*, on the correspondence execution-interaction 4.2.14, the correspondence between *merging of interpretations* and *parallel composition* 5.1.18, and the diamond property of the *reduction on the interpretation* 5.2.12, there are minor revisions to be made.

Correspondence execution-interaction

The correspondence is not affected at all by the extension, since it does not require to modify either the base design or the restriction designs.

Lemma 6.1.6. *Let P be a* replication-free *CCS process. Then theorem 4.2.14 still holds for $[\![P]\!]$, i.e. $[\![P]\!]$* characterizes all executions on *P.*

Proof. We must prove that $+_P$ properly represent the non deterministic choice +. Since + *does not add execution sequences to P* but actually *prevents some*, then we only need to show that the corresponding *interaction paths* – by definition 4.2.4 – are not possible on $[\![P]\!]$.

This follows directly by definition of $+_P$ and $\mathcal{R}(+_P)$: the restriction designs in $\mathcal{R}(+_P)$ *prevent interaction to visit in the same path branches associated to elements of two members in a* +.

□

Merging of interpretations

The merging of interpretation requires some slight addition to be extended to +. The definition of *merging of base designs* (def. 5.1.9) is not affected by the introduction of $+_P$. This is because the only difference in $\mathcal{D}_P$ for a process P with sums is that there are more *xor* branches, which makes no difference at all for the merging.

Instead, the definition of $\mathcal{N}_{P \mid Q}$ (def. 5.1.8) must take into account also $new{+}_{P \mid Q}$, which is the set of the new sum conflict-pairs (u,v) for $u,v \in new\mathcal{S}_{P \mid Q}$, when u and v have each one channel in a different member

of a sum. The extension is however obvious, and requires little work. We have

$new{+}_{P \mid Q} = \{(u,v) \mid u,v \in new\mathcal{S}_{P \mid Q}$, and such that one channel of u and one channel of v are *minimal channels* belonging each to a different member of a sum + in either P or $Q\}$

This set is *recoverable from the assignment functions* $[\ ]_P$ *and* $[\ ]_Q$, and from $\mathcal{R}(Loc_P) \cup \mathcal{R}(Loc_Q)$ (to find which are the minimal locations). We don't need to add in $new{+}_{P \mid Q}$ pairs of *locations*, since there are *no new channels* generated in the parallel composition, but only *new synchronizations*.

We need to add to $\mathcal{N}_{P \mid Q}$ the branches corresponding to the new conflict-pairs of $new{+}_{P \mid Q}$, as we did for $new\mathcal{X}_{P \mid Q}$. Hence we have that $\mathcal{R}(new{+}_{P \mid Q})$ is defined exactly as $\mathcal{R}(new\mathcal{X}_{P \mid Q})$, but for the set $new{+}_{P \mid Q}$ defined above. Of course, $\mathcal{D}_P \odot \mathcal{D}_Q$ takes into account the extended $\mathcal{N}_{P \mid Q}$, as well as lemma 5.1.10, on the equality of base designs, that obviously still holds.

Lemma 5.1.15 thus becomes:

Lemma 6.1.7. *Let* P, Q *be* replication-free *CCS processes. It holds that* $\mathcal{R}(P \mid Q) = \mathcal{R}(P)_{\mathcal{D}_P \odot \mathcal{D}_Q} \cup \mathcal{R}(Q)_{\mathcal{D}_P \odot \mathcal{D}_Q} \cup \mathcal{R}(new\mathcal{S}_{P \mid Q}) \cup \mathcal{R}(new\mathcal{X}_{P \mid Q}) \cup \mathcal{R}(new{+}_{P \mid Q})$.

and still holds for the same reasons, along with the fact that the set $new{+}_{P \mid Q}$ obviously catches all the new +-conflict pairs arising in $P \mid Q$.

This leads to the extension of definition 5.1.16 with $\mathcal{R}(new{+}_{P \mid Q})$ as well, and to

Lemma 6.1.8. *Let* P *be a* replication-free *CCS process. Then, theorem 5.1.18 still holds, i.e.* $[\![P]\!] \oplus [\![Q]\!] = [\![P \mid Q]\!]$.

Proof. The proof is carried out exactly as for the *MCCS* case. □

Reduction on the interpretation

For our reduction on the interpretation a more complex extension is required. After a reduction step on a synchronization including a channel part of a sum, we need to recover the *whole sub-process* associated with the *other member of the sum*, and erase it. For instance, if $P = a.Q + Q' \mid \overline{a}.R$, and we want to perform a reduction on $u = (a, \overline{a})$, when *trimming* the base design $\mathcal{D}_P$ we need to associate to u also *all the branches corresponding to* Q'.

Definition 6.1.9 (Extended Trimming)**.** *Let* P *be a replication-free CCS process of the form*

$$P = a^l.Q + Q' \mid \overline{a}^m.R$$

To identify the sub-process to be erased along a synchronization $u = (a^l, \overline{a}^m)$*, we need to extend definition 5.2.2, the* **trimming** *of a synchronization on* $\mathcal{D}_P$*, by including the following to the* branches associated to the synchronization u *:*

- xor^u *&* xor^v *and* $G[v]$ *for all* $(u, v) \in +_P$*;*
- xor^v *&* xor^w *for all such* v*;*
- xor^l *&* xor^x *and* xor^m *&* xor^y *for all* $(l, x), (m, y) \in +_P$*;*
- $G[x]$ *and* $G[y]$ *for all such* x, y*;*
- *All branches found by* **repeating the previous two steps** *(both the xor and* $G[\ ]$ *branches) for all locations* n *and* o *such that* $x <_P n$ *and* $y <_P o$*, that can be transitively recovered by:*

  1. *finding the* immediately greater location *of each such* x *and* y *by checking in which restriction design* $G[x]$ *(or* $G[y]$*) ends with a* ✠ *(which means it is a requisite for another location).*
  2. *Repeating the above step for the found locations.*

  This is needed to identify all channels of Q'*.*

- *All* $G[z]$ *with* $z \in \mathcal{S}_P$*, such that* $G[z]$ *ends with* ✠ *in the restriction design of any of the locations found in the previous steps* $(x, y, n, o$ *and greater locations) – this is to find synchronizations that are on a channel labeled by one of the erased locations.*
- xor^z *&* xor^i *for all* $(z, i) \in \mathcal{X}_P \cup +_P$*, for all* z *found in the previous step.*

In this way, the **reduction on the interpretation** (definition 5.2.11), as well as *all the definition and lemmas based on the trimming*, are extended to replication-free *CCS* processes.

Lemma 6.1.10. *Theorem 5.2.12 still holds for* replication-free *CCS processes.*

Proof. By the extended definition of trimming (def. 6.1.9), all lemmas used in the proof for the *MCCS* case (lemma 5.2.5, corollary 5.2.6, and lemmas 5.2.9 and 5.2.10) still hold for the *replication-free* case, since we are erasing exactly the sub-process in a + excluded by execution, that is being performed on the other member of the sum.

□

Example 6.1.11. *Let:*

$$R = a^l.P' + Q' \mid \overline{a}^n.P'' + Q'' \rightarrow_{u=(a^l,\overline{a}^n)} P' \mid P'' = R'$$

The **reduction** *on* $u = (a^l, \overline{a}^n) \in \mathcal{S}_R$, *i.e.* $(\![R]\!)_u$, *implies erasing from the ramification of* $\mathcal{D}_R$, *in addition to the trimming for MCCS processes, also the branches* $xor^u \,\&\, xor^v, xor^l \,\&\, xor^n \in +_R$, *all branches* $G[x]$ *with* $x \in Loc_{Q'} \cup Loc_{Q''} \cup \mathcal{S}_{Q'} \cup \mathcal{S}_{Q''}$, *and all branches* $xor^x \,\&\, xor^y$ *for such* x *(considering both* $\mathcal{X}_R$ *and* $+_R$*).*

6.2 Name Hiding

The ν operator (called *new*), declares that a channel name is **hidden**, or **private**, inside its scope, i.e. that cannot communicate with channels outside it; we present it with the syntax and rules borrowed from the π-calculus. Its role is to forbid communication on some channels, making them private, in the sense that no external process in parallel composition can access to them. It is a key tool to ensure safe communication in concurrent programming, by respecting the communication protocols. The intended effect of the operator is that the channel names it refers to are always *fresh*, and therefore can be accessed (for instance to send/receive data) only by a dual channel also inside its scope. If $P = \nu a\ (a^l.R) \mid \overline{a}^m.Q$, a^l and $\overline{a}^m$ *cannot communicate*, therefore *are not a synchronizable*. For clarity we close the scope of the ν under parenthesis as we did in the example.

A channel name under ν is considered *bound*, otherwise *free*. We denote with $bn(P)$ the set of *bound* names, and with $fn(P)$ the set of *free* names. The rules which let us push ν inside a process, until it is applied to exactly its scope, are the following:

$$\nu a(P \mid Q) \equiv \ \nu a\ (P) \mid Q$$

if $a \notin fn(Q)$;

$$\nu a(P) \mid \nu b(Q) \equiv \nu a(\nu b(P \mid Q)) \equiv \nu b(\nu a(P \mid Q))$$

if $a \notin fn(Q)$ and $b \notin fn(P)$. $\equiv$ is the smallest congruence relation which extends structural congruence (def. 6.1.1). We note a sequence $\nu a_1(\nu a_2 \ldots (\nu a_n) \ldots)$ with $\nu \vec{a}$. The execution relation is also modified, either requiring that synchronizations *may happen only* between two dual channels inside the scope of the same ν, or that synchronizations simply *may not happen* between dual channels if *only one is inside the scope of* ν.

Definition 6.2.1 (Synchronizations under ν)**.** *Let P be a* replication-free *CCS process with* name hiding*. Then we have two possible choices of restrictions for $\mathcal{S}_P$:*

- *$\mathcal{S}_P$ contains* only *pairs of dual channels $(a, \overline{a})$ such that they are both* inside the scope of the same *νa. Therefore $(a, \overline{a}) \in \mathcal{S}_P$ if and only if P is of the form*

$$\nu a(a.P' \mid \overline{a}.P'') \mid Q.$$

- *$\mathcal{S}_P$ contains pairs of dual channels $(a, \overline{a})$ such that* either *they are both inside the scope of the same νa* or *they are* not bound *by any νa. Therefore $(a, \overline{a}) \in \mathcal{S}_P$ if and only if P is either of the form*

$$\nu a(a.P' \mid \overline{a}.P'') \mid Q,$$

or the form

$$a.P' \mid \overline{a}.P'' \mid Q$$

but not of the form

$$\nu a(a.P') \mid \overline{a}.P'' \mid Q.$$

this means that the ν is an operator that can also *enable* interaction, other than preventing it. This entails that the **execution rule** for replication-free CCS processes with ν is of the following forms:

- either **only**:

$$\nu a(a.P' \mid \overline{a}.P'') \mid Q \rightarrow_{(a,\overline{a})} P' \mid P'' \mid Q$$

- or **both**:

$$\nu a(a.P' \mid \overline{a}.P'') \mid Q \rightarrow_{(a,\overline{a})} P' \mid P'' \mid Q$$

and

$$a.P' \mid \overline{a}.P'' \mid Q \rightarrow_{(a,\overline{a})} P' \mid P'' \mid Q$$

(note that we are ignoring the + case only for brevity).

Internal choice

The ν has various purposes, and can be even used as an *alternative to the non deterministic choice*, even if not equivalent. Indeed it can be used to define a notion of *internal choice*, in the sense that it is independent by the environment/context, but which still consists of a mutual exclusion between two sub processes. The sum is an external choice since if $R[\ ] \coloneqq a.P + b.Q \mid [\ ]$, where $[\ \ ]$ is a *placeholder* that can be filled by another process[3], then execution can yield either P or Q as result, depending on which term will fill the empty space, that could synchronize on either a or b; thus different contexts may yield different results.

However, another kind of sum can be defined, that we denote $\boxplus$, but is usually denoted $\oplus$ (which, however, would be inconsistent with the differentiation between external and internal choice), in the following way:

Definition 6.2.2 (Internal choice)**.** *Let P, Q be CCS processes. Then,*

$$P \boxplus Q = \nu a(a^l.P \mid a^m.Q \mid \overline{a}^o) = \tau.P + \tau.Q$$

Here $a \notin P, Q$, and τ is an *internal action* (which may be executed or not independently of the environment), a sort of un-observable or *silent* action in the literature[4], in the sense that *we do not know if there has been a synchronization on it*, nor its content (which channel names are involved), and thus cannot be synchronized by our decision. The fact that the channel a is hidden means that it cannot interact with the environment, and thus *only an internal synchronization may let us communicate with* P or Q; however not with both in the same execution sequence.

Indeed, this is achieved by exactly a *xor* condition on two synchronizations, $u = (a^l, \overline{a}^o)$ and $v = (a^m, \overline{a}^o)$: since $\overline{a}^o$ is in common, and being a an external and hidden channel, then the xor condition $(u, v) \in \mathcal{X}_{P \boxplus Q}$ effectively forks execution in two parallel paths, which cannot be re-joint. The result of this *internal sum* has therefore the same effect as the external sum, though due to an internal choice: one of the members of the sum is discarded from the process, and cannot be synchronized on anymore in the same execution sequence. Thus, we do not need a new relation to represent the internal choice, but it suffices to use the ν and *xor* conditions properly, further justifying our choice of using a conflict-like relation to represent the non deterministic choice.

[3] Formally called an **evaluation context**: a process with a *placeholder* in it, an empty space which can be filled by any term of the syntax: [4].

[4] In [46] it is considered an *unobservable internal action*.

6.2.1 Representing ν inside $[\![P]\!]$

About the interpretation of ν, assuming we can isolate the scope of the operation, we can note that *it affects interaction only negatively*, i.e. it gives restrictions to the possible *executions* on the process, *without adding new ones*. Thus, we can just internalize in $[\![P]\!]$ the possible *restrictions* on the definition of synchronizations, i.e. the restrictions on $\mathcal{S}_P$ of definition 6.2.1, in order to consider only the intended pairs. Exploiting the *rules on* ν, and taking the lax restriction of $\mathcal{S}_P$ to be in a more general case, we can denote with a^l_ν a channel in the scope of a ν on its name; then we need to exclude from $\mathcal{S}_P$ any pair $(a^l, \overline{a}^m)$ such that only one channel is tagged with ν, i.e. either a^l_ν or $\overline{a}^m_\nu$. *If both are tagged with* ν *then the pair is still a synchronization* .

This solution implies that channel names tagged with ν must be *unique*, modulo renaming. Indeed, there is no loss in doing that, since one of the purposes of the operation is to note that a hidden channel a_ν *is not the same* that another a, even if they share the same name. An equivalent solution would be to rename the channel tagged by the ν with a new unused channel name: a constantly fresh channel, refreshed whenever the process is composed with a term containing that channel name. Obviously this is not a viable choice, since we would be required to rename the channel each time a new context is added by parallel composition, if the same name appear again; or impose a renaming on any new context, such that the name(s) under ν in the process is (are) renamed in the context, if present.

Instead of having multiple operations and notions (*fresh variable*, and *renamings*), we can just add the ν operator. Since this is the indented meaning, and is consistent with the given rules, it has no particular consequences the assumption that if we have $\nu a\ P$ and $\nu a\ Q$, once in a parallel composition, we have renamed one of the two a to a *fresh* channel b in order for it to be unique, with $\nu a(P) \mid \nu b(Q)$ as result. In this way there is no confusion between tagged channels: *if they are tagged and dual they can synchronize, otherwise they can't.*

The interpretation can thus take into consideration only synchronizations of the form $(a^l, \overline{a}^m)$ or $(a^l_\nu, \overline{a}^m_\nu)$, but not of the form $(a^l, \overline{a}^m_\nu)$. This means *we are not adding branches* $G[x]$ in $\mathcal{D}_P$ for these pairs, and there *is no associated address in the assignment* $[\ \ ]_P$, nor any kind of restriction or *xor* conditions: they are simply **not considered elements of** $\mathcal{S}_P$ **or the process** P. The consequence on the merging of interpretations is that, when unifying the assignments, the pairs of dual channels (represented by their locations) where one is tagged by ν and the other is not, are *not* counted as new synchronizations, and thus do not appear in the merged designs.

This can be an accurate representation of νa, since we are allowing execution only on dual channels inside the same scope of the new, scope that we can effectively determine. This implies that inside $\mathcal{D}_P$ and $\mathcal{R}(P)$ there is no trace at all of the νs that can occur in the process P, we can only check their presence from the assignment $[\ ]_P$, which only tells us singularly which channels occurrences are inside the scope of a ν – hence the assignment still does not need any additional information about the structure of the process. The result is that we forbid some interaction paths by not interpreting, instead of resorting to more restrictions designs.

Lemma 6.2.3. *Let P be a* replication-free *CCS process with* hidden names. *Then, its interpretation $[\![P]\!]$ still* characterizes all executions *on P.*

Proof. The proof is elementary, by definition 6.2.1: *there are no interaction paths* whose associated execution sequence contains a synchronization $(a^l, \overline{a}^m_\nu)$, since while $[a^l]$ and $[\overline{a}^m_\nu] \in [\ ]_P$ and have associated branches on $\mathcal{D}_P$, the pair $(a^l, \overline{a}^n_\nu)$ has none, and simply does not appear in $\mathcal{D}_P$. □

Clearly also all the other results are unaffected by the restriction on $\mathcal{S}_P$, and the introduction of tagged names.

Readback of the process

This discussion brings us to the problem of *reading back* a process P from the interpretation $[\![P]\!]$, something that we have defined only for *interaction paths*. Reading back the result of an execution associated to an interaction is simple enough, but *reconstructing the process from the interpretation* requires some more steps. We can get from the assignment $[\ ]_P$ all the channel names occurrences, and the partial order $<_P$ can be induced from $\mathcal{R}(l)$, for each $l \in Loc_P$, by transitive closure. Whenever two channels a^l, b^m are not ordered, with respect to their locations, thus neither $l <_P m$ nor $m <_P l$, then they are put in parallel composition. The problem lies with the ν, which leaves us with two choices: either simply put a ν at the end of the reconstruction of the process, thus $\nu\vec{a}(P)$, for each channel $a \in \vec{a}$ such that a_ν is in $[\ ]_P$, or identify the minimal scope by a step-by-step construction. The first way is however much simpler, therefore we will define the read back on the interpretation in the following way:

Definition 6.2.4. *The* **read back** *on the interpretation $[\![P]\!]$ of a replication-free CCS process P is the term build by the following steps:*

- **Base step***: $a_0^{l_0} + a_1^{l_1} \mid a_2^{l_2} + a_3^{l_3} \mid \ldots \mid a_{n-1}^{l_{n-1}} + a_n^{l_n}$ for each minimal location l_i, with $0 \leq i \leq n$, channel a_i which it labels, and xor condition*

on minimal locations $(l_i, l_j) \in +_P$. *Minimal locations* l_i *are such that* $G[l_i]$ *has no location requisite in* $\mathcal{R}(l_i)$.

- **Inductive step***:* $a_i^{l_i}.(b_j^{m_j} + c_h^{o_h})$ *for each location* m_j *such that* $G[l_i]$ *is a requisite for* $G[m_j]$ *(and* $G[o_h]$*), and* $(m_j, o_h) \in +_P$. *Or* $a_i^{l_i}.(b_j^{m_j} \mid c_h^{o_h})$ *if* b^{m_j} *and* c^{o_h} *are in no relation, but their are both one-step greater than* a^{l_i}. *This step is then repeated for the* 1*-step greater locations of the minimal ones (until there are locations).*

- **Final step***: After the inductive step(s), we need to add* $\nu\tilde{a}(P)$ *for each* $a_\nu \in [\ ]_P$, *with* P *the result of the previous two steps (note that we have omitted the possible* $_\nu$ *subscript in the first two steps only to lighten the notation).*

6.3 Non-linear extensions of ludics and recursive definitions

In this section we finally give some directions, and attempted solutions, to internalize *recursive definitions* in our interpretation. We have the option to deal with either the standard recursion operator of CCS, or the replication $!a.P$ of the π-calculus.

To handle replication in its π-calculus form (defined in section 3.3), we can possibly exploit its *controlled version* of [67], used to type event structures by considering a restricted version of the π-calculus: a linear typed version of Sangiorgi's πI-calculus [63]. This calculus has the same expressive power as the less-restricted version (with free *name passing*), and linearity only breaks for replicated outputs, but determinism is preserved by the uniqueness of inputs. This allow us to still use the partial orders $\preccurlyeq_{\mathcal{S}_P}$ and $<_P$ to manage synchronizations, and attempt a natural integration of replication in our interpretation, without extending or modifying the syntax and rules of ludics.

6.3.1 Replication in the πI-calculus

In [67] a more general version of the πI-calculus is considered, but with the same core restrictions to ensure *linearity*. Replication is of the form $!x(\vec{y}).P$, and follows the usual rule of the π-calculus:

$$!x(\vec{y}).P \mid \overline{x}(\vec{y}).Q \to !x(\vec{y}).P \mid (\nu\vec{y})(P \mid Q)$$

Firstly, only *bound names* are passed during interaction, thus an output can be only of the form $\nu\vec{y}(\overline{x}(\vec{y}).P)$. Secondly, the behaviour of processes is restricted with the following discipline:

- For each linear (not under !) name, there are an unique input and an unique output.
- For each replicated name there is an unique replicated input, with zero or more dual outputs.

We are particularly interested in the second condition, which forbids processes as

$$!b.\overline{a} \mid !\overline{b}.\overline{c} \mid \cdots \quad \text{or} \quad !b.a \mid !b.c \mid \cdots$$

while allows processes of the form

$$P = !b.\overline{a} \mid \overline{b}^1 \mid !c.\overline{b}^2$$

even if a b output appears twice, apparently violating the first rule, since there is only one replicated input per-name, the process is still valid. Indeed, each output can communicate with the same, and only, input. So, only inputs can be replicated, and each replicated channel must be *unique*. Only a replicated input allows multiple outputs, that cannot be a replicated as well; this entails that each replicated channel needs a linear and consumable dual for each execution step.

With respect to our interpretation, this means that we could treat a replicable channel name as an **infinite branch**, that can be visited each time after a synchronization with one of its duals. It would follow that all synchronizations having in common the replicable channel **would *not* be in a *xor* relation**.

Going back to the example above, it means that the synchronizations $i = (!b, b^1)$ and $j = (!b, b^2)$ would not be in $\mathcal{X}_P$. The base design would have as sub designs infinite sequences of the form $G[!b] =$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\vdash [!b].1.1.1}}{[!b].1.1 \vdash}}{\vdash [!b].1}}{[!b] \vdash}$$

Restrictions on channels prefixed by $!b$ would still be of the form

$$\dfrac{\dfrac{\dfrac{}{\vdash [!b].1}\ \text{✠}}{[!b] \vdash} \quad \cdots \quad \dfrac{}{[\overline{a}] \vdash}\ \text{p}}{\vdash \xi}$$

Infinite sequences are needed to preserve modularity (the number of synchronizations on $!b$ depends by the context), and to possibly have dynamic restrictions: the n-th rule of $G[!b]$ would be accessible after the n-th synchronization on $!b$, which can be determined if, at any execution step, *only one choice of synchronization* is possible on $!b$; i.e. if only one dual $\overline{b}$ is available at a time (even if multiple are present in the process): this would let us determine in advance the order in which each $\overline{b}$ synchronizing with $!b$ can be visited, and thus its restriction design.

Both synchronizations $i = (!b, b^1)$ and $j = (!b, b^2)$ can be part of the same execution sequence, thus we want that both $G[i]$ and $G[j]$ can be part of the same interaction path: this would be the case if there is no $xor^i \,\&\, xor^j$ branch, and restrictions forbidding interaction on both at the same time.

If the setting were *non-linear*, a lot of issues would arises regarding the suffix channels of $!b$ not under exponential: if $!b.\overline{a}$, we can synchronize the channel $\overline{a}$ only as many times as we already have synchronized the channel $!b$, and thus would need to *count* during interaction how many times we visited the branch of $G[!b]$ (i.e. performed synchronization on the channel $!b$), to let the context communicate multiple times with $\overline{a}$. However, for the linearity constraint, there can be only one input a, unless it is under exponential ($!a$). In the first case, there would be at most one synchronization on $\overline{a}$, and thus no need to count or express more complex restrictions. Instead, a possible issue is present in the second case, since there could be a potentially infinite execution sequence; for instance with a process as

$$P = !b.\overline{c} \mid \overline{b}^1 \mid !c.\overline{b}^2$$

However, this fixed-point-like cases (the process can only return to its starting form after 2 execution steps) could be treated via cycles in interaction (an interaction as in [6] defined via an *abstract machine*).

The uniqueness of linear inputs/outputs simplify the setting enough to attempt a solution without passing to non-linear ludics, or introducing *exponential* addresses. There is, however, another possibility that could let us represent a **full unrestricted replication**, also in the form of recursive definitions and without linear restrictions, that comes from an already existing version of ludics dedicated to achieving a stronger computational power.

6.3.2 A reformulation in Terui's computational ludics

K. Terui, in [66], formulated a complementary syntax for ludics, called **computational ludics** (c-ludics), closer to higher order π-calculus. His objective is to achieve practical advantages towards applications, with the general goal of developing an *interactive* theory of computability and *complexity* based on ludics. To this end, he drops the notation based on absolute addresses, the places in a proof, and replaces them by *names* binding variables. A variable bound by a name a (in a set A) let us link actions under the *justification* relation, and thus implicitly induce their relative addresses (one a sub-address of the other). This makes the manipulation of syntax easier, letting us naturally add *internal cuts* (while standard designs are *cut-free*), identities as variables in a ramification, and pass to a non-linear context.

The feature that seems most interesting to use is the latter, in particular the possibility to represent infinite designs by a *finite generator* (by definition,

all designs are representable in such a way[5]), allowing *recursive definitions*; using designs generators could let us easily extend the interpretation to CCS with an un-restricted replication.

C-ludics

Terui's syntax is based on a *signature* $\mathscr{A} = (A, ar)$, where A is a set of *names*, and $ar : A \to \mathbb{N}$ is a function giving an arity to each name; a denumerable set of variables V, denoted $x, y, z, \ldots$, is also needed. A *positive action* is either $✠$, Ω (denoting the divergence) or $\overline{a}$, with $a \in A$; a *negative* action is $x \in V$ or $a(x_1, \ldots, x_n)$, with $a \in A$ and $ar(a) = n$. $x_1, \ldots, x_n$ are distinct variables, and $\vec{x}_a$ notes a vector of the arity of a. Informally, a design $\mathcal{D}$ is co-inductively defined by

$$P ::= ✠ \mid \Omega \mid (N_0 \mid \overline{a}\langle N_1, \ldots, N_n\rangle),$$

$$N ::= x \mid \sum a(\vec{x}_a).P_a.$$

where P is the form of **positive** designs, N of **negative** ones, and $\vec{x}_a$ is a vector of variables of $ar(a)$. Here a name denotes both the polarity and cardinality of the ramification of a rule, and, in the negative rule, the variables stand for each sub-address of the ramification.

If in a positive design $P = N_0 \mid \overline{a}\langle N_1, \ldots, N_n\rangle$, $N_0 \neq x$, for a variable x, then we have a *cut* (however the name of N_0 must be dual to $\overline{a}$); otherwise $N_0 = x$, and, if it is bound by a negative action $a(\vec{x}_a)$, thus with $x \in \vec{x}_a$, then $a(\vec{x}_a)$ *justifies* P. If a variable is inside the scope of a positive action, i.e. $a\langle N_1, \ldots, x, \ldots, N_n\rangle$, then it is called an *identity*. Negative actions are instead justified by the positive action in which they are included. The $\sum$ notes an infinite sum of all names, with an Ω in place of P_a for all but finitely many of them – each member of the sum is a different premise.

Absolute addresses as *focus* of actions disappear from the syntax, becoming relative to the justification or cut relation, in favor of a more concise and dense presentation. While this might be useful for representing computational power, it hinders our interpretation, since it is based on a correspondence between absolute addresses and the sets $\mathcal{S}_P, Loc_P, \mathcal{X}_P$ (and $+_P$). If we use names in place of addresses, then we would not have a way to use the same name in a rule with a different number of premises, since each name has an unique arity, and the assignment is not naturally inherited by the justification relation on actions of the ramification; this leads to superimposition and further renamings when performing operations such as the merging or subject reduction. Shifting the correspondence from addresses to

[5] A universal generator can be defined: [66], section 2.2.

variables would not work properly either, since the variables stand for the *sub-addresses* of a rule, which could be replaced by a ✠ or a pruning, or their action erased in a trimming, leaving us with no way of determining which element correspond to the missing premise. Everything considered, a reformulation of our interpretation is still possible, since it holds the following:

Remark 6.3.1. Standard *(≠ Ω, linear, cut and identity-free) c-designs*[6] *correspond to the original designs.*

Example 6.3.2. *Assuming an assignment from* $Loc_P, \mathcal{S}_P, \mathcal{X}_P$ *to names, we have the following correspondence with definition 4.1.14:*

- $G[u] = [u](x_u).(x_u|\overline{[u.1]}\langle 0\rangle)$.
- $w[u,v] = [xor^u](x_u).x_u|\overline{[xor^u.1]}\langle 0\rangle + [xor^v](x_v).x_v|\overline{[xor^v.1]}\langle 0\rangle$.
- $\mathcal{D}_P = x_0|\overline{a}^{7}\langle G[x], \ldots, w[x,y], \ldots\rangle$, *with* x *varying on* $\mathcal{S}_P \cup Loc_P$, *and* (x,y) *on* $\mathcal{X}_P$.

where [ ] *notes the assignment function.*

Interaction is called **reduction**, and is defined in λ-calculus style on positive c-designs with a cut, by:

$$(\sum a(\vec{x}_a.P_a)|\overline{a}\langle\vec{N}\rangle \to P_a[\vec{N}/\vec{x}_a],$$

where $\vec{N}$ is of lenght $ar(a)$. The reduction relation selects the $a(\vec{x}_a)$ that matches with $\overline{a}\langle\vec{N}\rangle$, making sure that they have the same arity. Then, in the corresponding P_a, each variable is substituted by a negative design inside the scope of $\overline{a}$; reduction can then continue on $P_a[\vec{N}/\vec{x}_a]$, until a normal form is reached (a variable x or ✠), or it diverges (Ω).

Example 6.3.3. *Using the reduction relation, we can rewrite example 2.2.31 using prunings and ✠ easily:*

- $\mathcal{D} = x_0|\overline{a}_0\langle a_1(x_1).x_1|\overline{a_{11}}\langle 0\rangle,\ a_2(x_2).x_2|\overline{a_{21}}\langle 0\rangle\rangle$
- $\mathcal{C} = a_0(x_1,x_2).x_1|\overline{a}_1\langle a_{11}(x_{11}).x_2|\overline{a}_2\langle a_{21}(x_{21}).✠\rangle\rangle$.
- $\mathcal{E} = a_0(x_1,x_2).x_2|\overline{a}_2\langle a_{21}(x_{21}).x_1|\overline{a}_1\langle a_{11}(x_{11}).✠\rangle\rangle$
- $\mathcal{D}^* = x_0|\overline{a}_0\langle 0^p, a_2(x_2).✠\rangle$.

[6] We should add, *over the signature* $\mathscr{RAM} = (\mathcal{P}_f(\mathbb{N}), |\ |)$, the finite set of parts of $\mathbb{N}$, and the cardinality $|I|$, for $I \in \mathcal{P}_f(\mathbb{N})$. [66], remark 2.2.

[7] It does not matter which name we choose here, since the base is an arbitrary address.

where ^{p} *denotes a pruning, and* 0 *the empty negative action: it denotes that the* + *action has a premise, but there is no further action in the branch. Both reductions* $\mathcal{C}|\mathcal{D}$ *and* $\mathcal{E}|\mathcal{D}$ *reach* $\maltese$ *after five reduction steps. Instead, only* $\mathcal{E}$ *is orthogonal to* $\mathcal{D}^*$*. To perform reduction, we must substitute* x_0 *in* $\mathcal{D}^*$ *with* $\mathcal{E}$*, obtaining*

$$\mathcal{E}|\mathcal{D}^* = (a_0(x_1, x_2).x_2|\overline{a}_2\langle a_{21}(x_{21}).x_1|\overline{a}_1\langle a_{11}(x_{11}).\maltese\rangle\rangle)|\overline{a}_0\langle 0^p, a_2(x_2).\maltese\rangle$$

Reduction reach $\maltese$ *in only 2 steps:*

1. $(a_2(x_2).\maltese)|(\overline{a}_2\langle a_{21}(x_{21}).0^p)|\overline{a}_1\langle a_{11}(x_{11}).\maltese\rangle\rangle$.
2. $\maltese$.

Instead, the reduction

$$\mathcal{C}|\mathcal{D}^* = (a_0(x_1, x_2).x_1|\overline{a}_1\langle a_{11}(x_{11}).x_2|\overline{a}_2\langle a_{21}(x_{21}).\maltese\rangle\rangle)|\overline{a}_0\langle 0^p, a_2(x_2).\maltese\rangle$$

at the second step diverges:

1. $(0^p)|(\overline{a}_1\langle a_{11}(x_{11}).\maltese)|a_2(x_2).\maltese|\overline{a}_2\langle a_{21}(x_{21}).\rangle\rangle$.
2. Ω*; since* 0^p *has no* P_0 *and variables to perform the substitution on.*

The reformulation in *c*-ludics thus *does not affect restriction designs*, or the behaviour $\mathscr{B}_P$, and the correspondence between execution and interaction still holds.

About the *merging of interpretations*, the operation $\odot$ is simply an extension of the arity of a positive rule – or a substitution with a name of the arity we need – by putting together in the scope of this new action all the negative designs that we have. Therefore from

$$\mathcal{D} = x_0|\overline{a_1}\langle N_1, \ldots, N_k\rangle$$

and

$$\mathcal{C} = x_0|\overline{a_2}\langle N_{k+1}, \ldots, N_{k+n}\rangle$$

we obtain

$$\mathcal{D} \odot \mathcal{C} = x_0|\overline{a_3}\langle N_1, \ldots, N_k, N_{k+1}, \ldots, N_{k+n}\rangle.$$

The projection operation instead, used for the *reduction on the interpretation*, requires us to erase the sub-ramification associated to a certain synchronization from $\mathcal{D}_P$, in order to reduce the arity of the first action $\overline{a}$, and remove the corresponding negative designs (somewhat the dual operation of $\odot$). A change to a name of the right arity might be required, but the operation itself poses no issues. When reducing $\mathcal{D}_P$ to $\mathbb{O}ne$, the form we obtain is $\mathbb{O}ne = x_0|\overline{a}\langle\rangle = x_0|\overline{a}$, a positive action with a 0-ary name.

Finite design generators

What we can gain from the syntax of c-ludics is a natural extension to non-linearity and recursive definitions, via designs generators. Terui's definition used two sets, one of positive and one of negative *states*, and a labeling function assigning, respectively, positive and negative designs to them.

Definition 6.3.4. *A* generator G *is a triple* (S^+, S^-, l)*, with* S^+ *and* S^- *disjoint set of states, and* l *a labeling function, defined on* $S = S^+ \cup S^-$*, such that:*

- *For* $s^+ \in S^+$*,* $l(s^+)$ *is* $\maltese$*,* Ω *or of the form* $s_0|\overline{a}\langle s_1^-, \ldots, s_n^-\rangle$*, such that* $a \in A$*,* $ar(a) = n$*, and* $s_0^-, \ldots, s_n^- \in S^-$*.*
- *For* $s^- \in S^-$*,* $l(s^-)$ *is either a variable* x*, or of the form* $\sum a(\vec{x}_a).s_a^+$*, such that* $s_a^+ \in S^+$ *for every* $a \in A$*.*

A pointed generator *is a pair* (G, s_I)*, where* $s_I \in S$ *notes the root of the design generated by* G*.*

For instance, the ordinal ω can be represented using a **finite design generator** i.e. a generator with a *finite number of states* (the example can be found in Terui's paper [66], section 2.2). An *universal pointed generator* can be defined as well, therefore we can consider every design as the result of this generator.

Terui's syntax thus provides an unitary setting in which both linear and non-linear traits can be represented at the same time, with the original ludics recoverable by a *restriction of the syntax*. For instance, the *separation theorem* holds for standard c-designs[8]; but, at the same time, c–designs are particularly appropriate for representing functions and data-set, thanks to cuts and *identities* (when using variables as premises of the ramification, instead of designs), increasing ludics computational power, and reduction on design generators is possible through a Krivine's-like abstract machine.

Why the correspondence doesn't work

The main issue we have with Terui's syntax is, as we already noted, that the **assignment function** $[\ ]_P$ cannot be as straightforward as in standard designs. If we substitute addresses with names, then we could have different names that should correspond to the same element, since every time we change the arity of a rule, either by merging or reduction on the interpretation, we need to change also the name of the action. Another difficulty is that

[8] Theorem 3.9, [66].

we can track a *justified* sequence of actions only using *variables*: knowing which action corresponds to, for instance, the continuation of the branch of a $G[u]$, is not natural at all, since it requires us to make reference to two distinct set of elements – both names and variables – while the assignment $[\ ]_P$ should be to one set only.

In the standard syntax, once we have an assignment as $\xi.1 = [u]_P$, we know that $[u]_P.1$ is the address generated by the rule on $\xi.1$, i.e. $\xi.1.1$, and belongs to $G[u]$. Instead, in c-ludics we have that:

- if we start from a positive action, its premises have a name only if there actually *is* a rule on it; otherwise if it is the ending rule of a branch, we have a 0, no matter by which action it is justified. Therefore, we cannot link this name to $[u]_P$, since there is no name.

- If we start from a negative action $a(\vec{x}_a)$ we know which are the positive actions justified by it only by the variables in $\vec{x}_a$, thus $x \mid \overline{b}\langle\cdots\rangle$ is justified by $a(\vec{x}_a)$ if $x \in \vec{x}_a$. Only then we may associate to $\overline{b}$ the same element associated to a.

Using names instead of addresses forces us to modify the assignment function $[\ ]_P$, and to deduce the assignment from the c-design structure, reconstructing the justified sequences from the base design by tracking each variable. For instance, if $a(\vec{x}_a) = [u]_p$, for $u \in \mathcal{S}_P$, then we want to assign u also to any positive name $\overline{b}$ such that $x_i|\overline{b}\langle\cdots\rangle$, with $x_i \in \vec{x}_a$, is in $\mathcal{D}_P$; then to any negative name in the scope of $\overline{b}$, and so on.

This implies that to each element of $Loc_P, \mathcal{S}_P$ and $\mathcal{X}_P$ are assigned *multiple* names, depending by which ones are used in $\mathcal{D}_P$. We also need to substitute the *names* of the base design every time we perform a merging between $[\![P]\!]$ and $[\![Q]\!]$, and rename the assignments accordingly. This latter point is necessary to ensure that there are *no conflicts* between $[\ ]_P$ and $[\ ]_Q$, since it could be the case that the *same name* is used in both base designs, but is assigned to two distinct elements: one of P, the other of Q. There is no issue from the c-designs viewpoint, since there is no superimposition of actions if the variables are distinct, but it could happen between the assignments.

We must also note that most of the advantages of Terui's syntax go beyond the scope of our work: we do not need nor computational power, nor the ability to represent data sets. Our focus is on process calculi, and what we really lack is a way to represent *recursive definitions*. To this end, designs generators could meet our needs, however we *do not want* to have on them the kind of interaction used in Terui's work; i.e. the Krivine style normalization. Terui points out that the reduction-based normalization does not directly work on generators, because it involves substitution and renaming.

The problem is that we want to let finite generators require a possibly *infinite* number of steps to be normalized, to preserve the correspondence between execution and interaction; in other words, we want interaction to be *resource sensitive*. It could be the case that a process defined by recursion admits an infinite execution sequence; assuming that recursive definitions on processes can be represented by finite designs generators, then we want to be able to represent this infinite execution sequence by an infinite interaction path, taking into account how many steps, thus synchronizations and channels, are required to perform this execution.

This means that we want to be able to re-use a channel name only as many times as we have *already* replicated it, and not re-using a single sequence of actions for the potential number of times it can be replicated. This implies a definition of interaction that *cannot re-visit* actions already visited, as instead happens in ludics with repetitions [6], where, as in c-ludics, an abstract machine is used to define interaction on strategies. In [6] ludics is formulated in game-semantics terms, and the abstract machine used for interaction actually *duplicates* the whole designs which are interacting, once for every copy of the same address, thus effectively splitting the procedure into linear sub-interactions, forming a bigger cut-net. As we will see, this leads to other problems and solutions, that however do not fit our need to preserve the correspondence between execution and interaction.

The standard definition of interaction with cut-nets is actually already resource sensitive, since at each step a *sub* cut-net is considered, composed by the sub-designs of the ones on which we have performed the previous steps. This means that even if an action is repeated, it cannot continue interaction on a dual already visited, unless also this dual is replicated: this is exactly the kind of interaction we would like to use to represent recursion. The next step is, then, to *borrow design generators* from c-ludics, and *try to represent them in the original syntax* with the smallest possible extension, by using *standard interaction*, with the goal to understand if this extension is possible with no particular drawbacks.

6.3.3 Recursion and replication in standard designs

Let us look at a simple example of replicable process taken from [50], and see how we could translate it in standard ludics. The process is formulated in an extension of the π-calculus, therefore we leave the value-passing part of the calculus aside. Here ? notes an input channel, and ! an output channel, the parenthesis [ ] include the data (channel names themselves) to be sent, for the output channel, or to be substituted (once data has been received) for the input channel.

Example 6.3.5. *Let* $P = *f?[x,r].r![x]$*, the* identity function*, where* $*$ *is a recursive operator applied to* $f?[x,r]$ *(the function is replicated after communication). An equivalent recursive definition for the function is*

$$id(f,r) = f[x,r].(\overline{r}[x] \mid id(f,r))$$

since it is replicated after synchronization on f *– if it were replicated after* $\overline{r}$*, instead, we would have* $id(f,r) = f[x,r].\overline{r}[x].id(x,r)$*. Without considering the values to be sent or received during communication, a translation in the standard ludics syntax could use the following device:*

$$id(f,r) = \dfrac{\dfrac{id(f,r) \qquad \dfrac{\dfrac{[\overline{r}].1.1 \vdash *\xi}{\vdash [\overline{r}].1, *\xi}}{[\overline{r}] \vdash *\xi}}{\vdash [f].1, *\xi}}{[f] \vdash *\xi}\ (+, *\xi, \{1,2\})$$

where $[f] = \xi.1$*,* $[\overline{r}] = \xi.2$*, and* $*\xi$ *note a* special address*, that we call* exponential*, that is replicated after being the focus of a rule with a sort of implicit contraction rule, essentially having the same role as* $!A$ *in linear logic.*

This representation is motivated by our need to use such a structure as a premise for a base design*: we start by a negative rule, and we need to carry over the positive address of the base, since every premise is a sub-address of this latter; this lets us repeat the sequence of rules starting from the same address* ξ*, limited to a sub-ramification.*

We must also stress the fact that a generator, as $id(f,r)$*, refers to a specific* **recursive sequence of actions***, independently by the context. Indeed, in the second recursion step, where* $id(f,r)$ *appears as a premise, the base is not* $[f] \vdash *\xi$*, but* $[f] \vdash [f].1, *\xi$*, with the context larger than at step* 0*. This means that we still allow repetition of addresses, but in a controlled way: they may be generated only by multiple actions on a special exponential address. This also implies that we can* keep track *of the recursion steps, by the numbers of repeated* non-exponential *addresses in the context at any given time.*

Although we have just extended the syntax with a sort of exponential $*\xi$, note that this notation is only needed in the presentation of design as proof-trees: in the more game-semantical presentation as sequence of actions the context is *hidden*, and until there is an action on an address in the context, this exponential would be visible *only in the base of the design.* However,

using multiple copies of the same address, instead of an exponential as $*$, implies that a design generator requires an *infinite number of copies* of it in the base. In general, an exponential lets us duplicate an address only when needed, and as many times as we want, without having to know in advance how many copies we will use. This is essential to represent design generators, since a base cannot contain an infinite number of addresses. Using this kind of exponential, with a hidden contraction included in rules of which it is focus, could seemingly let us interpret recursion while making the identification of *justified* sequences simpler, as opposed to the syntax of c-ludics. Indeed, one problem with repetition of actions in ludics is that *justified sequences* of actions – views or chronicles, depending on the setting – are not easy to extract from a design.

As we noted in the example, repetition of standard addresses would be admitted in this setting, though *in a controlled way.* Rules on these repeated addresses are possible, but contrary to exponentials, they are not replicated in the premises: to use them again we need, first, to perform another action on the exponential generating them. This, along with a **resource-sensitive interaction** could provide a solution to represent recursive definition of CCS processes with a minimal extension of ludics' syntax.

We would have, so, two kind of addresses, the standard ones $(\xi, \zeta, \ldots)$ and the exponential ones $(*\xi, *\zeta, \ldots)$. At first, it is clear that an exponential (or replicable) address $*\xi$ (we prefer $*$ instead of !, since this latter is also used in the π-calculus to note output channels) cannot be in *negative position.* This is because negative addresses are necessarily unique, since they have priority in proof-search, and allowing multiple addresses at the left of $\vdash$ would bring us outside the world of focused linear logic proofs. Therefore, only a positive address can be replicated, by keeping a copy of it in the context each time a rule is performed on it: in this way, only one copy of each *exponential* address is present at a time in the context; while standard addresses, as noted, have no restrictions. Thus, if an address needs to be in negative position multiple times (as in the example), then it is the positive address generating it that must be replicated, to let us perform the same negative rule again. With respect to our example, since in $id(f, r)$ a rule on ξ is repeated, an orthogonal design would need a positive address $*\zeta$, such that $\xi = \zeta.1$. For instance, we would have $\mathcal{C} = (-, \zeta.1 = \xi, \mathcal{N}) \cdots (+, *\zeta, \{1\})(-, \zeta.1, \mathcal{N})$, with $\zeta.1 \vdash *\zeta$ as the base of $\mathcal{C}$. In general, positive rules would be of the following form:

$$\frac{\xi.1 \vdash *\xi \quad \cdots \quad \xi.n \vdash *\xi}{\vdash *\xi}\ (+, *\xi, I)$$

Using a replicable address instead of multiple copies of it does change how interaction works, but could be the best solution to treat design generators.

We will briefly describe how instead interaction is carried out *in ludics with repetitions*, show what are its problems and the proposed solutions, and why we can't adopt it as well if we want to internalize finite design generators while keeping valid the execution-interaction correspondence.

Ludics with repetitions of actions

In [6], by Faggian and Basaldella, designs are defined as Hyland-Ong *innocent linear strategies* on *an universal arena*, to then drop the restriction of linearity in the definition of their equivalent of *chronicles* (definition 2.2.6), to allow repeated actions with the same focus. The idea is that, during interaction, every time an action is repeated, a copy of the counter-design is created, then the address of this action is renamed to a fresh one in both the design and counter-design, to induce the same renaming in the whole justified sequence starting from the action in question, and thus continue interaction as in the standard case by breaking it into smaller linear pieces. In this way each time an action is repeated, the design is partially reverted to a linear one, and interaction can continue by *duplicating and renaming* the counter-design. To give an intuition, we borrow some examples from the original paper in figures 6.3.3 and 6.3.3.

Remark 6.3.6. *Dropping the linearity constraint on designs does not entail strong inconsistencies. While it is against the idea at the base of ludics, i.e. abstracting proofs to keep only their interactive part and forgetting the logical content, formally it only implies that the base of a design allows multiple times the same address, and thus is a* multi-set *of addresses. Standard interaction does becomes a little awkward, since there is no syntactic way to identify which of the many copies of an address ξ we are performing a rule on, and thus justified sequences (on which interaction is based) are superimposed, and become harder to isolate. Interaction as presented in* ludics with repetitions *overcomes this problem, as is shown by the examples.*

Problems of repetitions

The first issue encountered is that the *separation theorem* (explained in subsubsection 2.2.3) does not hold anymore (a counter-example is found in figure 6.3.3). Indeed, there are different strategies (the equivalent of designs) $\mathcal{D}_1$ and $\mathcal{D}_2$ which cannot be *separated* by a third design orthogonal to only one of the two: they have the very same orthogonal designs, and thus interact in the very same way. This somewhat nullifies one of the aims of ludics, to close the gap between syntax and semantics: the syntactic structure of a

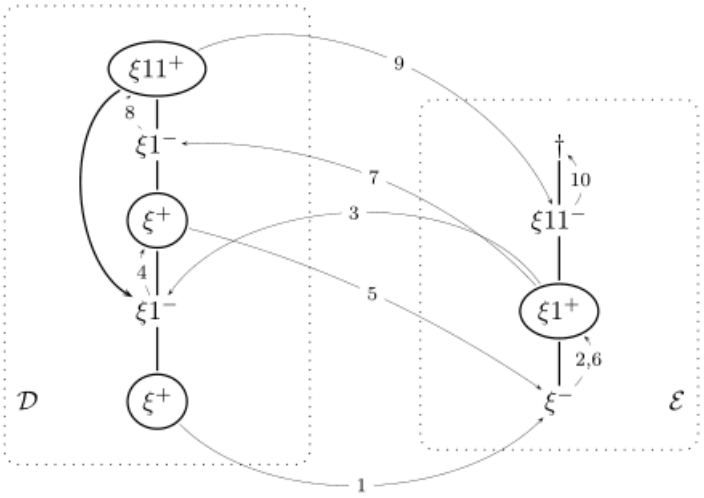


Figure 6.1: This is how interaction is intended to work. The numbers denote the interaction n-th step. Repeated addresses and the definition of interaction of ludics with repetitions allow to go back to already visited actions.

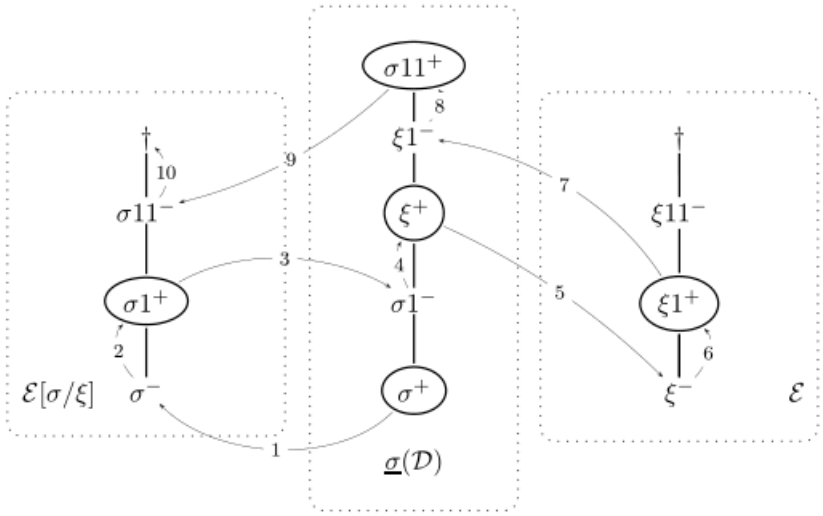


Figure 6.2: Formally the previous example works like this. $\underline{\sigma}(\mathcal{D})$ denotes the design $\mathcal{D}$ where the ξ focus of the first rule has been substituted by the address σ (in the base). While $\mathcal{E}[\sigma/\xi]$ denotes the counter design $\mathcal{E}$, where σ is in place of ξ. For each repetition of the positive action ξ^+, a copy of $\mathcal{E}$ is necessary, and interaction is carried out with $\mathcal{D}$, where a different address is in place of each ξ, and thus for each copy of $\mathcal{E}$.

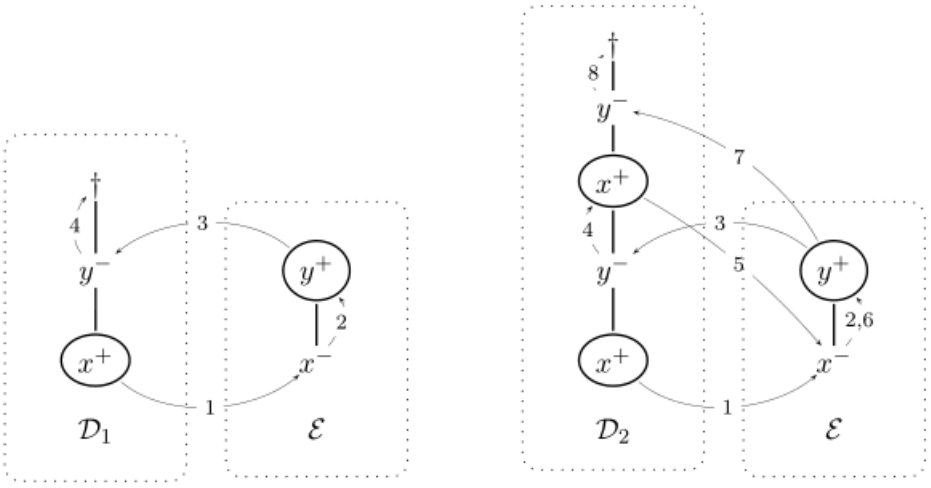


Figure 6.3: This is the counter-example to the separation theorem. However, with standard interaction, step 5 would be forbidden, since after step 2 the actions x^- and y^+ would not be in the cut-net anymore, effectively ending it before reaching a daimon (there noted † instead of ✠): this would separate the two designs.

design should explicitly express its semantics, and how it interacts with the other, therefore one should be able to identify and differentiate each design just by watching at its structure; however, this is not the case anymore once we allow repetitions.

The second problem is about having enough counter-designs, and is closely tied to the internal and full completeness theorems[9]. Without explaining the technical details, the problem lies in the interpretation of linear logic formulas into behaviours. Now that exponentials are admitted, the interpretation of $?A$ must be able to respect the *contraction rule*: if a design $\mathcal{D}$ of base ξ belongs to the behaviour interpreting $\vdash ?A$, then we should be able to transform it (using only substitutions and renamings), to a design $\mathcal{D}'$ of a behaviour interpreting $\vdash ?A, ?A$. To achieve this, *non-uniform* tests are introduced, a sort of *non-deterministic sum* of negative designs, which requires to choose an interaction path when encountered – one of the possible continuations of the design. Since a part of a design may be accessed several times, each time the same choice is presented; therefore two designs are orthogonal only if their interaction reach ✠ for every possible choice.

[9]The original statements and proofs can be found in [60] or [38].

6.3.4 Interaction with exponentials

On the side of our interpretation, we can easily see why this kind of interaction is *not* an optimal solution. In the first place, from a sheer theoretical point of view, we would like to keep the separation theorem. That is because it expresses the fact that the meaning of a design is caught by all the ways it can be used, which are the totality of the interactions with all the possible counter-designs. This is in accord with the meaning we gave to processes: their *dynamic*, i.e. all the possible (depending by the process itself) and potential (depending by the context) execution sequences. How each execution is carried out, and thus how a process can communicate with the environment (i.e, the other processes), at any given stage, defines the meaning of a process. Losing the separation theorem would nullify one of the reasons we used ludics to interpret process calculi in the first place.

In the second place, we already stressed the point of a resource sensitive interaction, which fails once we need to duplicate designs. Our correspondence execution-interaction is based on the assignment function and the sequence of visited actions; however, if we were to *rename* only a single occurrence of an (repeated) address at a time, the correspondence becomes problematic: we would need to extend the assignment every time there is an action on the address in question, and let a replicated channel in the process be associated to a potentially infinite number of different addresses. The substitution needs to be done on the whole behaviour, since it must affect also the restriction designs, with the effect that multiple, different, addresses will have the same restriction designs, that needs to be replicated for every repetition of a rule (and potential subsitution of an address during interaction). Moreover, a duplication of the counter-design would also shift the correspondence we have, since we would have two identical interaction paths – in the same interaction-session – that visit the same action sequence, even though on different addresses, thanks to the renaming. These two copies of the same interaction paths, however, correspond to two successive and different execution steps.

Interpreting interaction informally, instead, as repeated visits of a part of a design each time a rule is duplicated, would still have a problem of superimposition: on the side of the process, that sequence of actions corresponds to some synchronizations which already happened, therefore to elements already consumed by execution, which cannot be recovered, unless duplicated by a recursion. For each replication, one or more execution steps are needed, and each time a channel is used, it is consumed. This means that, by revisiting actions we lose the *memory* of the past execution, and at which recursion-step we are. The other strong issue is the same of *ludics with repetitions*:

there are not enough counter-designs to express all the possible execution sequences. That is because non-linear strategy, using this kind of interaction, are *uniform* by default: at a repeated action from the main strategy, they respond always in the same way. This means that, once we duplicate an address, or perform a recursion step, we have no way of using that address, corresponding to a replicated channel, in a different way than before.

Instead of using non-uniform tests, which extend the syntax of ludics in such a way to bring it closer to the syntax of process algebras itself, the solution we would like to achieve is the reformulation of design generators in the standard syntax, by using the construct we showed in the example 6.3.5. This, paired with *standard* interaction, could give a us a way to fully represent recursive definitions, while keeping the separation theorem valid, at the cost of extending the syntax by introducing *exponential addresses*, essentially the same solution used in linear logic to recover the lost expressive power. However, this naive extension may not be without consequences.

We can note right away that an exponential blurs the distinction between *open* and *closed* cut-nets, if using standard interaction. That is, if an address $*\xi$ is replicated, then the fact that it is cut in the base is not relevant anymore, since it is kept in the context. That matters, however, only if there actually is a repeated rule on the replicated address. The other problem is that we cannot counter an exponential $\vdash *\xi$ with an exponential in dual position – i.e. cut an exponential only with another exponential – since $*\xi \vdash$ cannot be admitted for negative rules, as we explained. It does neither work to require that $\xi = \zeta.1$ and that $*\zeta$ appears in the dual, since we do not know if $*\zeta$ will be focus of a rule or not. A partial solution could be to re-phrase interaction to include both cases (open and closed) at once, and let interaction continue only if, for each repetition of a rule on $\vdash *\xi$, there is a dual rule on $\xi \vdash$. Let us take again the identity function of the previous example:

Example 6.3.7. *Let* $P = \nu f(id(f,y) \mid \overline{f}^c.y^c.1)$*, with* $id(f,y) = f.(\overline{y} \mid id(f,y))$*. The synchronizations are* $(f, \overline{f}^c) = u$ *and* $(\overline{y}, y^c) = v$*. There are no xor conditions, and the partial order is* $f <_P \overline{y}$ *and* $\overline{f}^c <_P y^c$*. The* base design *is, thus:*

$$id(f,y) = \dfrac{\dfrac{\dfrac{id(f,y)\quad G[\overline{y}]}{\vdash [f].1, *\xi}\,(+, *\xi, \{1,2\})}{[f] \vdash *\xi}\,(-, \xi.1, \{\{1\}\}) \qquad G[\overline{y}]\ G[\overline{f}^c]\ G[y^c]\ G[u]\ G[v]}{\vdash *\xi}\,(+, *\xi, I)$$

with $I = \{1,2,3,4,5,6\}$*. Without expliciting the obvious restriction designs,*

interaction starts on $G[u]$*, then visits* $(-,[f],\{\{1\}\})$ *and* $(-,[\overline{f}^c],\{\{1\}\})$ *in any order, to then possibly continue on* $G[v]$*, and its channels* $\overline{y}$ *and* y^c*.*

An interaction could then continue on $(-,\xi.1 = [f],\{\{1\}\})$ *and necessarily visit the successive positive action* $(+,*\xi,\{1,2\})$*.*

At this point, a counter-design needs to have a negative action of the form $(-,\xi,\{\dots,\{1,2\},\dots\})$*, to then possibly continue on another branch with the next positive action. This implies, however, that* $\mathcal{C}$ *has as base* $\xi \vdash *\zeta$*, with* $\xi = \zeta.1$*, since there is at least a second negative action on* ξ*.*

While it is not exempt from problems, exponential addresses seem able to represent design generators in the standard syntax, without the complications on the assignment that c-ludics entails, and with a resource-sensitive interaction. A slight modification of the interaction procedure is still needed, since we cannot know anymore if we are in the open or closed case: an exponential address cut in the base could reappear as focus of a rule, but not have a match anymore.

Another matter left to examine, to understand the consequence of the introduction of an exponential, are the **full and internal completeness theorems** (subsection 2.2.4), which still hold in c-ludics (as well as ludics with repetitions). This possible extension of ludics, while based on an already existing formulation, needs however further study to achieve a more defined shape.

7. Relations with other works

In this chapter we briefly present other works with some points of contact with ours, both to see what we have gained with respect to the state of the art, and if our results can be applied or be useful in other contexts. The chapter is divided into three main parts, that will end with the *conclusions* of the thesis, in section 7.4.

The first part, section 7.1, is about the ideas of [9] by E. Beffara (continuation of a work with V. Mogbil [12]), were CCS is *typed* into **linear logic proof-nets**, to see more in detail which are its objectives and limits. It is an important step with respect to our interpretation, since one of our early goals was to try to get a similar result while avoiding its drawbacks: we will discuss how the interpretation in ludics does overcome these limits, even if taking a different route than a standard *typing*. The aim of the work of Beffara (continuation of a previous work with Mogbil, [12]) is to find a correspondence, rather than between CCS processes and proof-nets, between proofs and *execution*, where cut elimination corresponds to performing an execution sequence. We share the same focus on the *dynamic* of the two systems begin what gives meaning to the objects themselves, however we try to work on the logical system, ludics, instead that on CCS, to form a correspondence: while in the work of Beffara execution is *scheduled*, in order to make it compatible with a *confluent procedure* as cut-elimination, we try instead to find a way such that the dynamic of ludics is able to describe the *non-confluence* of process execution.

The paper in question starts from the simple setting of $MCCS$ as we do, in the standard *multiplicative* linear logic grammar with the addition of *modalities* to represent *channel names*. A translation is carried out between the two syntaxes, and to each process-construction rule are associated a fixed sequence of linear logic rules, in such a way that to each $MCCS$ process P is associated a *proof-net* $[P]_s$; then, the **execution rule** is interpreted as the **linear implication** $\multimap$.

The main theorem of the paper shows that, taken a $MCCS$ process P,

there is an execution $P \to^* P'$ if and only if $[P]_s \multimap [P']_s$ is *provable* in *multiplicative linear logic*; this leads to the fact that, to obtain the *proof net* corresponding to P' it suffices to consider the *cut* between $[P]_s$ and $[P]_s \multimap [P']_s$, and perform *cut elimination*.

Therefore in [9] a correspondence is built between process execution and cut-elimination, however this connection is strongly limited by the fact that we have to *know in advance* which execution step we are going to make, and thus is tied to a *scheduling* of execution: in this way, *each execution step corresponds to its own, different proof-net*, and we lack a *single object as interpretation* for each process. Our interpretation in ludics manages to form an equally stronger correspondence in *dynamics*, while giving a single, though complex, object that can characterize all the executions at the same time.

The second part is about **session types**, which we shortly introduce in section 7.2. We focus in particular on the work of Kobayashi, Saito and Sumii ([50]), about a *typing* for a version of the π-calculus that allows to check for *deadlocks*, with a very specific meaning of deadlock: a process is *deadlocked* if there is *no synchronization* on specifically *tagged* channels on which *communication is expected*. Being able to tell by the type of a process if there are deadlocks in this sense is the main concern of the paper, since session types are used to ensure a safe communication protocol between agents, and a typing deduction system is built to make sure that communication actually will happen on certain channels during execution of a given process. Our interpretation, when adapted to the π-calculus by ignoring *name-passing*, can achieve the same result in a trivial way; thanks to the correspondence execution-interaction, and the information of the assignment $[\ \ ]_P$, checking in $[\![P]\!]$ if certain channels actually synchronize is extremely simple: it just suffice to check the interactions on $[\![P]\!]$, or, in a constructive way, attempt to build a counter-design of $[\![P]\!]$ that visits the desired branches on $\mathcal{D}_P$. Inside [50] is also found the example of the *identity function* (example 6.3.7) that we used to show how exponential addresses could work in ludics.

The last part, subsubsection 3.2.2, shows a few obvious connections with event structures, through the *conflict* and *causal dependency* relations, which can be associated to, respectively our *xor* conditions and the *partial prefix order*. We also talk about the work of Faggian and Piccolo [27] that focus on forming a correspondence between *typed confusion-free event structures* and *linear strategies* (the game semantical definition of *designs*), and how our interpretation could form a bridge between *event structures* and *ludics* without the restriction to *confusion-freedom*, but further study is needed to clarify this connection.

7.1 $MCCS$ to proof-nets: scheduling in concurrency

In [9] the general goal is to be able to match linear logic *proofs* with process *execution*, in such a way that *cut elimination* corresponds to an *execution sequence*. We made ours that same *shift* in the standard approach to a correspondence: instead of searching a correspondence between terms of a process algebras, and linear logic (formulas), we focus on their respective dynamics.

The main difference in our approach is the choice of the logical system, in order to avoid necessary limitations or compromises on the process algebra to make a correspondence work. Since the *meaning of proofs* is their normal form, cut elimination is and should be confluent; processes of a concurrent system, instead, do not have a single irreducible normal forms, therefore their meaning becomes rather all their potential execution paths and thus interactions with the environment: the solution to form a correspondence of [9] is to make the concurrent system deterministic by **scheduling execution**, i.e. making it confluent and thus compatible with cut-elimination. This entails that, *for each execution* $P \to^* P'$, for a process P, *there is an associated proof-net*, but only for that *specific execution step*. Instead, we are explicitly trying to catch the *non-confluence of execution* in the chosen logical system, ludics: we searched for a way to make *interaction* able to represent non-confluence, rather than restricting execution to make it confluent. The main drawback of our approach is that standard ludics lacks the *exponential* part of linear logic, and as we saw extensions to non-linearity are not fit for our interpretation.

Outline of the work

The paper starts by considering $MCCS$ processes, and *multiplicative* linear logic extended with modalities (called MLL_a) $\langle a \rangle A$ and $\langle \overline{a} \rangle A$, attached to a formula A. Two cases are distinguished: the *synchronous* one (where execution is considered step-by-step), and the *asynchronous* one (where maximal execution sequences are considered). In both cases, a translation is defined, from terms of $MCCS$ to MLL_a formulas, using modalities to denote channel names, leading to a proof $[P]_s$ (where the s stands for *synchronous*) for every $MCCS$ process P.

The main result of the *synchronous translation* part is a theorem stating that *there is an execution* $P \to^* Q$ *if and only if* $[P]_s \multimap [Q]s$ *is provable in*

MLL without modality rules[1]; the main result of the *asynchronous translation* part instead deal with reduction sequences to the empty process 1, and say that *there is an execution* $P \to^* 1$ *if and only if* $[P]_a \multimap [1]_a$ *is provable in MLL (without modality rules)*[2]. These theorems form a correspondence between execution of $MCCS$ processes and cut-elimination of proofs, however there is a clear compromise to be made, that is *scheduling execution*. For each execution sequence of a process P there is a MLL proof corresponding to it, however *there is not a single proof for all executions*, and therefore there is not a translation in linear logic that can catch the *whole behaviour of a process*, in the sense of all its possible executions, by giving a single correspondent in the logical system to each process of the calculus. Furthermore, the scheduling of execution requires to know in advance which execution step to trasnlate, giving a strong limitation to the correspondence; for instance, *deadlocked* processes cannot be translated in a proof-net (the result of the translation would be an incorrect proof structure).

Comparison with our interpretation

Our interpretation gives a complex translation for each $MCCS$ process P, but is able to represent with a well-structured object its whole behaviour, with a superimposition of terms with ludics. It is a *multiplicity* of elements, being a *set* together with the assignment function, however it is the output of an operation of interpretation that can *characterize* all the executions of P at once, via *interaction* with its orthogonal. In this way we can overcome the strongest limitation, that was also the solution, of the work of Beffara and Mogbil, that is the *scheduling* of executions, by catching both the *non-determinism* and *non-confluence* of the dynamic of CCS through a *set of interactions*, instead of performing cut-elimination on a single element.

Not only we give to P an interpretation $[\![P]\!]$ that can catch the *meaning* of P, in the sense of all its possible execution sequences, but the same interpretation can *account for all reduced forms of* P, via *reduction on the interpretation* (definition 5.2.11). The interpretation $[\![P]\!]$ has thus a clear relation with any P' such that exists an execution sequence $P \to^* P'$, and there is a nice correspondence with reduction sequences $P \to^* 1$, by applying our *reduction* on $[\![P]\!]$ until we obtain the design $\mathbb{O}ne$, if that is possible. Moreover, we can also trivially interpret the *name hiding* operator ν, that instead gives some troubles with the proof-nets correspondence, and also deadlocked process, that have no correspondence at all in a proof-net translation.

[1] [9], p.8 for the full statement and proof of the theorem.
[2] [9], p.11 for the full statement and proof of the theorem.

However, while an extension to the non deterministic choice + is rather straightforward in both works – Beffara mentions that it suffices to use *additive proof-nets* for the sum – as we know treating replication is not a simple matter in ludics. For linear logic, there is a solid proof-net interpretation of the exponential $!A$ using *boxes*[3], instead for ludics there is no definitive extension to exponentials and non-linearity, and the existing ones do not let us directly transfer our interpretation in a non-linear setting for ludics.

7.2 About a deadlock free calculus

We attempt to apply our interpretation to **session types** by simplifying the π-calculus there used and taking a few simple examples from [50], to see what we can accomplish with our results.

Session types aims to ensure security and absence of errors during communication in parallel and distributed systems. These programs heavily rely on communications of various agents, as in web services and client-server interactions. Typing should ensure safe and correct communication, by making sure both parts are respecting a communication protocol – if their types match, then communications will happen as intended. Type systems have been devised in particular for extensions of the π-calculus, as in the work of E. Sumii and N. Kobayashi (later refined with S. Saito) about **session types for deadlock free processes**. Their motivations come from practical implementation problems, in particular being able to guarantee communication on specific channels – i.e. that sending an output and receiving an input on a chosen channel will actually happen; the absence of communication on such channels is considered a *deadlock*.

The first interpretation was restricted to *reliable channels*, either linear (used once), recursive or *mutex* channels; then it is generalized to extend the deadlock-free part of processes considered (other than reliable channels[4]), and finally in [50] it gets reformulated to an implicit interpretation with type reconstruction, for easier interpretation in programming. The addressed problem is thus deadlocks in a very broad sense: a process is deadlocked when there are particular channels waiting for synchronization, on which communication is *expected*, that cannot find a dual in the context. Therefore a process is considered deadlocked if it is in normal form, different from 1 (there noted 0), and contains special channels waiting for communication. In addition, to describe the behaviour of each channel (if its used as input or output, and which values sends/receive), a *time tag* relation defines a *partial*

[3] For instance, see [40].

[4] [64], introduction.

order on them, which must be respected during communication. A process is, then, *deadlock free* if reduction *on the type*, given a *time tag* ordering mirroring the *prefix partial order*, yields an empty set.

In CCS, as we know, there is no data passing; therefore we need to simplify the setting to be able to apply our interpretation to a few examples. Using $[\![P]\!]$, checking if there is synchronization on a certain channel is extremely simple, since all we need to do is look at its interactions with the orthogonal: if some interaction visits the ramification corresponding to the channel in question, for the correspondence execution-interaction, then the channel in question will synchronize with a dual in the associated execution sequence. Since the interpretation fully characterizes execution of the process, we do not need a tag ordering (which corresponds to the partial order on locations, coded in the restriction designs), and any interaction does correspond to a possible reduction (that we call *admissible execution*); therefore we can easily use our interpretation to describe and check deadlocks in this broad sense.

Applying our translation in a different context

We show some simple examples, taken from the cited work, to see what can we achieve using $[\![P]\!]$. In the version of the π-calculus considered, the syntax is as follows: $x!$ is the output channel (still noted $\overline{x}$ in other version of the calculus); $x?$ is the input; $[\ ]$ denotes that no data is sent on the channel; and ^{c} denotes that the labeled channel is *expected to synchronize*, or succeed, in the sense that the data should actually be received from or sent to another channel.

Example 7.2.1. *The processes*

$$P = (\nu x)(x?^c[\ ].0)$$

and

$$P' = (\nu x)(\nu y)(x?^c[\ ].y![\ ] \mid y?[\ ].x![\ ])$$

are both in a deadlock, because the input $x?$ is labeled with **c** *but cannot synchronize with a dual channel. Instead, neither $Q = (\nu x)(x?[\ ].0)$ nor $Q' = (\nu x)(x![\ ] \mid x?^c[\ ].0)$ are in a deadlock.*

If we treat them as CCS processes, we have that P has no synchronizations (a single private channel), so in $[\![P]\!]$ there would be no meaningful interactions, as in $[\![P']\!]$, which, however, has two synchronizations that form a cycle in $<_{\mathcal{S}_P}$, and thus is *deadlocked* even by our definition. This implies

that in $[\![P']\!]$ no interaction visits the branches associated to any of the two synchronizations.

Regarding Q, $[\![Q]\!]$ has the exact same interactions as $[\![P]\!]$, however the label c is missing (hence is not considered deadlocked). Q' instead has a possible synchronization between $x!$ and $x?^c$, reducing the process to 1. We have that in $[\![Q']\!]$ the execution $\rightarrow_{u=(x!,x?^c)}$ is associated to at least one interaction, and that the reduction $([\![Q']\!])_u$ on the interpretation yields the design $\mathbb{O}ne$.

Another example shows how to use typing rules and time tags.

Example 7.2.2. *Let $P = *f?[x,r].r![x]$ (that is the identity function), where $*$ is a recursive operator applied to $f?[x,r]$ (the function is replicated after communication), and x, r are channels sent as data. It receives the argument on x and a channel on r, to then send back the same argument on the received channel. It holds the following judgment:*

$$\varnothing; \{(t_f, t_y)\} \vdash (\nu f)(P \mid (\nu y) f!^c[true, y].y?^c[z].0)$$

The pair (t_f, t_y) means that communication on f must succeed before communication on y (it is, indeed, the prefix order), and $\varnothing$ that interpretation reduction ends successfully, i.e. all channels will communicate. $\varnothing; \{(t_f, t_y)\}$ is the type judgment for the process. What happens in the parallel composition is that $f!^c[true, y]$ synchronizes with $*f?[x,r].r![x]$, which then becomes $*f?[x,r].r![x] \mid y![true]$, and sends back to $y?^c[z]$ the value $[true]$ received.

This is the identity function we took as an example of recursive definition to define exponentials in ludics. As we saw we can attempt to represent the replication, though not the name passing, in such a way that interaction on $[\![P]\!]$ will respect the order $(f!^c, *f?) \preccurlyeq_{\mathcal{S}_P} (y!, y?^c)$ in their associated executions. Interaction can visit $G[x]$ for every channel x ($[f!], [f?], [y!]$ and $[y?]$), therefore the base design (the one of example 6.3.7) is *material* in $[\![P]\!]$, which means that P is *deadlock free* by lemma 5.3.10.

The partial order given by prefixes is implicit in the interpretation thanks to the restriction designs, therefore we do not need to require that the interpretation reduces to $\mathbb{O}ne$ to have a general property of deadlock-freedom: a process is thus deadlock free in the sense of [50] simply if interaction visits all channels noted with ^{c}.

7.3 A close connection with event structures

The similarities between event structures and our interpretation are rather direct, since event structures (that we introduced in 3.2.9) are a *model for*

CCS-like calculi[5].

Recently event structures have been showed to model the πI-calculus (the *internal* π-calculus [63]) by D. Varacca and N. Yoshida. In [27] event structures and linear strategies, which are designs in game-semantic terms as well as a class of Hyland-Ong innocent strategies, are related in order to see the latter as a particular case of the first, a sub-class of *confusion free* event structures. The work of C. Faggian and M. Piccolo in particular aims to see typed confusion free event structures as a generalization of linear strategies, and to extend thus linear strategies with *non-determinism* in order to model a linear finitary π-calculus (without replication). Event structures, being a model of CCS and the π-calculus (to some extent) have some direct superposition of concepts with out interpretation, and seem to be directly represented in ludics by our interpretation. Indeed, we are essentially representing the relations used to define event structures through interaction in a behaviour *directed* by the restriction designs. Both notions of causal dependency and *conflict relation* are essential to build $[\![P]\!]$, the first being essentially the partial order on locations $<_P$ – which induces the order $\preccurlyeq_{\mathcal{S}_P}$ on synchronizations – the second the relations $\mathcal{X}_P$ and $+_P$, while *concurrent events* correspond to *independent* synchronizations(definition 5.5.1). The main difference here is the level of abstraction: we are focusing on its executable part, i.e. the synchronizable pairs, instead of working only with single channel name occurrences, that are interpreted into *events*.

The relevant relations we need for the interpretation are the partial order on locations $<_P$, and the *xor* relations, which are defined on *pairs* of dual channels instead of single ones (and also pairs of locations for $+_P$). Yet, we can switch from $\leq$ to $<_P$ and from $\smile$ to $\mathcal{X}_P$ with ease: it suffices to see events as synchronizations, or to label them with *actions* that correspond to channels, as in the typing of [27]. In our case, it seems possible to reformulate $\mathscr{G}_P$, that we can translate in ludics-terms, as an *event structure* without the requisite to be *confusion free*. The notion expresses the fact that *choices are local*, which is the equivalent of *confluence* in a non-determinist setting. While linear strategies might be seen as a special case of confusion free event structure, we could associate to an event structure $\mathscr{E}$ an interpretation $[\![\mathscr{E}]\!]$, without particular constraints on it. Since event structures are a model for CCS, we could start from a process P, then pass to an event structure $\mathscr{E}$ modeling it, and then interpret it in ludics; we would so achieve a correspondence between $\mathscr{E}$ and $[\![P]\!]$.

In the interpretation of processes into event structures, events are labeled with actions and *polarities*: a polarity can be $+$, $-$ or *neutral* $\pm$, a label is of

[5] Recall the work by Winskel [68].

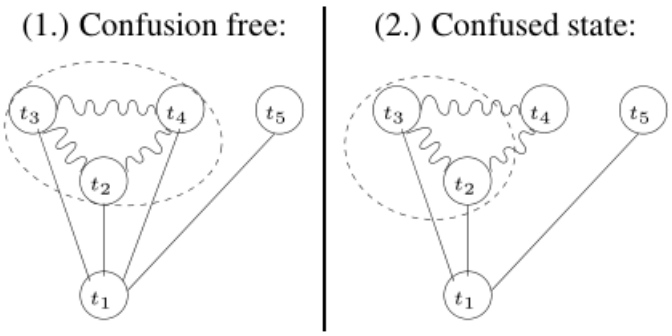


Figure 7.1: Regarding the similarities with $\mathscr{G}_P$: t_1 is a minimal synchronization, the edges are oriented arrows aiming downwards; the conflict relation (the irregular edges in the figure) stands for the sum +, and can be translated as a generic *xor* relation. Being or not confusion free does not matter for the ludics interpretation.

the form $(a, +)$ or $(a, -)$, or $(a, \pm)$, where the neutral polarity denotes a pair of dual actions (so a synchronization). By identifying channel occurrences and the locations labeling them, and giving neutral polarity to synchronizations, the relations $\leq$ and $\smile$ between the elements would then be rather straightforward to interpret.

Regarding the correspondence with linear strategies, we can relax the constrains on the conflict relation, which in confusion free event structures is confined to *cells* (recall that they are closed units of events in conflict with the same predecessor). Another interesting point with [27] is that the notion of cell is very close to the & rule of linear logic, but there are some relevant distinctions. In particular, only *negative cells*, that contains by negative actions, are considered as corresponding to the & rule, while positive or neutral cells, which represent *internal choice*, are without correspective in linear logic, but with a very similar behaviour. Instead, by using the *xor* relation we can represent both the non deterministic choice + and the *internal choice* $\boxplus$, that we defined in 6.2.2, as well as the mutual exclusion between synchronizations with a common channel. In $\mathcal{D}_P$ they have a common correspective in a negative binary rule, which is used as an intermediate step to separate the interaction paths visiting the members in conflict, via the restriction designs. The causal order or *enabling relation* as well is represented locally via the restriction designs on the partial order $<_P$, while the full relation is implicitly respected by interaction on $[\![P]\!]$. This differs from the usual game semantic interpretation, where the enabling relation is interpreted in the *justification* between actions, i.e. the *sub-address* relation of ludics. What we

have gained with our interpretation is thus an uniform representation for the elements and relations of event structures inside *directed behaviours*.

Discussion

In conclusion, we hope that our interpretation can give some new insights in the relation between process calculi, the game semantical formulation of ludics, and event structures, via the interpretation of $\leq$ and $\smile$ into directed behaviours, thus giving, to an extent, the inverse relation of *Event structures and linear strategies*. Despite being a direct correspondence, it will require further work to be explored in depth. What there seems to be gained from this correspondence is a way to model event structures in the sequential setting of ludics, without the need to confine the first to *confusion free* event structures, or add non-determinism to linear strategies.

7.4 Conclusions

This work started from studies on the extension of the *Curry-Howard* correspondence outside its standard intuitionistic setting, to models of concurrent and parallel computation. The main motivation behind the thesis was to find a way to form a satisfying Curry-Howard counterpart for a process algebra, in our case CCS, that would overcome the intrinsic problems and limitations found in the previous research on this topic. Linear logic has been the focus in most of these studies, as a logic particularly tied to interaction and thus apparently apt to represent process algebras, however the confluent nature of cut-elimination necessarily hinders a correspondence in *dynamic* between processes and proofs (or proof-nets), since execution on processes is, by default, non deterministic and non confluent. The proposed solutions mostly offer a partial correspondence between cut-elimination and execution; a common way to proceed is to restrict the syntax of the calculus to ensure *linearity* of channel names, i.e. to limit the occurrences of the same name, in such a way to avoid possible *conflicts* in the form of multiple choices of synchronization at each execution step, that as we saw generally split execution in two parallel sequences, that bring to different normal forms.

A different kind of solution is found in *proofs as executions*, where the correspondence is shifted in order to match a proof with an execution sequence; however the compromise here is to have a different proof for each execution sequence, and a different proof-net for each reduced form of the process, that must be known *in advance* to be able to form the correspondence in the first place.

On our end, we took a similar approach, but with a different goal in mind. The objective was to form a correspondence in dynamic between process algebras and a logical system without restrictions or compromises. We choose CCS as the process algebras, being the most basic one and paradigm for the others, and ludics as the target system, since it is a reformulation of logic from scratch ever more tied to interaction than linear logic, from which it generates: interaction is the central concept around which ludics is built, and designs are its syntactical support. Thus, the objective became to *interpret* CCS inside ludics, in such a way that *execution* would correspond to *interaction* as proofs do to programs in the original Curry-Howard correspondence.

This goal has been partially achieved by our work, with the exception of *replication* and *recursive definitions*, that elude a clear and straightforward representation in standard ludics. The main results we managed to obtain are:

- A translation of *replication-free CCS* processes inside ludics as *sets of designs*, that can be closed to bi-orthogonality without consequences for the technical results, obtaining thus a correspondence between *processes and behaviour*s, the types of ludics.

- A logical characterization of the dynamic of (replication free) processes by the correspondence between execution on the process and interaction on the interpreting set of designs, achieved without forcing the process into functional computation, resorting to multiple translations by partially determinizing execution via scheduling, or sacrificing the non-determinism or non-confluence itself (by imposing linearity and other constraints on the syntax). This characterization entails the possibility to check on the interpretation the possible executions of the interpreted process, by a simple read-back procedure: while there are multiple interaction paths describing the same execution sequence, every sequence is described, and no interaction path describes an incorrect or impossible execution on the process. We are able to identify through interaction each step of an execution, and not only its final result; this property also highlights the fact that the meaning of interaction lies in the same place of the one of execution: what is important is the *path* it takes, and not the final result, that in ludics case is only "convergence" ($\maltese$) or "divergence".

- A partial *modularity* of the interpretation, which let us combine the interpreting structures, behaviours in our case, as we do with processes via parallel composition. We are able to represent the composition via a ludical operation on behaviours that, in the trivial case where there is no communication between two processes, exactly interprets the linear logic tensor $\otimes$. Otherwise, some more artificial and non-ludical steps are required, by working on the assignment functions, but the core operation $\odot$ is kept intact on base designs.

- Insights on the dynamics of processes, as: expliciting what parallel composition entails when two process communicate, what causes forks in an execution paths, and how the different reduced forms of a process are related through their interpretations.

- An interpretation that is somewhat close to a *typing* since it enjoys a type-construction-like property, i.e. the interpretation of a process can be built via operations on behaviours that matches, step-by-step, its syntactical constructors: other than parallel composition, also the

action prefix ($a.P$), and non deterministic sum (P+Q) are representable via modifications of the same basic operation.

- A *reduction on the interpretation* that matches execution, and describes a particular inclusion between the interpretations of a process and one of its reduced forms, with respect to their structures and possible interactions. This is in contrast with the standard *subject-reduction* property of typing systems, that was inconsistent with the intended meaning of processes. This reduction let us further deepen the connection in dynamic, since we are able to recover from the interpretation alone all the *reduced forms* of the process, as well as its final *normal forms*. As a consequence, we found a clear connection between the *empty process* and the *linear logic multiplicative unit* 1, through the design $\mathbb{O}ne$.

- Along with execution, *deadlocks* are also characterized. Instead of being a property of the interpretation – as it is with typing systems, where if a process is typable, then it is deadlock free – the interpretation in ludics is oblivious of their presence, as is interaction on behaviours. Still, we have a way to know if a process is deadlock-free, or if it can't be reduced to 1, via the *visitable paths*, or incarnation, of the base design in the behaviour $\mathscr{B}_P$.

Future research directions

While we found most of the results we were hoping for, and successfully formed a Curry-Howard correspondence between CCS and ludics, in its current state it only holds for *replication-free* processes. The recursive or exponential operator, in either the form found in CCS or the π-calculus, is the hardest part of the interpretation, and we are only able to point at some possible paths to follow, that could bring to a solution after further research. In order to represent *recursive processes* an extension of ludics to non-linearity seems necessary, however a first attempt could be made with the controlled replication of the πI-calculus of [67], that could dispense us of such a step. In the work by Varacca and Yoshida communication cannot happen between two replicable channels, each synchronization is unique, and input channels are always unique: this let us ignore the complications of a replicable sub-process as $!a.P$, which would require us to *count* the number of synchronizations with its replicable prefix $!a$ to know how many times P is available for synchronization. A possible way to represent this controlled replication is via *infinite branches*, that are admissible on designs; however the limit cases of the syntax still seem problematic to carry out a clean correspondence without replication of addresses.

A REFORMULATION IN C-LUDICS. While we have some doubts on controlled replication, reformulating the whole interpretation inside Terui's *computational ludics* would be a much cleaner option. Finite design generators seem to exactly fit our needs, being built with the purpose in mind of representing recursive programs. Unfortunately the issue with *c-ludics* is found at the core of computational ludics itself, that is its π-calculus like syntax. While it is much more suited to represent functions and data types, it heavily hinders our correspondence: the lack of *absolute addresses* makes much harder to identify *justified sequences of actions*, and which *element of the process* is associated to them, heavily affecting the *assignment function*, that becomes impossible to define, being dependent by the changes/renaming that could happen to the names of a c-design during composition or reduction (its *names* need to change to make their arity consistent with a composed/reduced c-design).

IMPORTING DESIGN GENERATORS INTO STANDARD LUDICS. After examining interaction inside *ludics with repetitions*, we reach the conclusion that it is not the solution we seek. Interaction should be *resource sensitive* to be able to accurately describe execution, and keep all the properties of the correspondence in dynamic we defined; instead, in ludics with repetitions, designs must be duplicated, and sequences of rules visited multiple times: an expected step in a non-linear setting, that however is inconsistent with the properties we want the *characterization* of execution to have.

The solution we propose is, thus, to try to re-formulate finite design generators in standard ludics, with the only addition of special *exponential addresses*. Instead of allowing addresses to be replicated at will, with no restrictions, we introduce special *positive only* addresses marked with $*$, as $*\xi$, that are kept in the context after a rule on them, effectively mimicking a *contraction rule*. For this reason, their role is similar to ! for linear logic, meaning that they are not a linear resource. Using this kind of *exponential addresses* is not hard to translate into the standard syntax finite design generators: the resulting designs seem able to represent recursive functions, as the one of Example 6.3.5. However, such an extension to exponentials requires further study to prove its consistency with the core theorems of ludics, and the results of our work.

EVENT STRUCTURES. A final direction that has yet to be fully explored is the connection with event structures. Winskel showed that event structures are a model for CCS-like calculi, then, in what aspects do they differ with our interpretation? Can we prove the same results using event structures? Can we show that ludics is a model for CCS-like calculi in the same

way that event structures are? The evident close relation with our interpretation points at the possibility to see our results as an interpretation of event structures, rather than CCS, inside ludics. By transitivity, it would automatically hold that ludics is a model for CCS-like calculi: if a behaviour corresponds to an event structure, then it also corresponds to a CCS process, and vice-versa, if a CCS process is modeled by an event structure, then is modeled by a behaviour as well. This connection, once based on more solid grounds, would provide a bridge between ludics, hence also game semantics, and true models of concurrency, without the need to extend the correspondence specifically for each different process calculus that can be modeled by event structures.

Bibliography

[1] Samson Abramsky. Proofs as processes. *Theor. Comput. Sci.*, 135(1):5–9, 1994.

[2] Samson Abramsky and Paul-André Melliès. Concurrent games and full completeness. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 431–442, 1999.

[3] V. Michele Abrusci and Paul Ruet. Non-commutative logic I: the multiplicative fragment. *Ann. Pure Appl. Logic*, 101(1):29–64, 1999.

[4] Roberto M. Amadio. Operational methods in concurrency. *lecture notes*, 2015.

[5] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

[6] Michele Basaldella and Claudia Faggian. Ludics with repetitions (exponentials, interactive types and completeness). *Logical Methods in Computer Science*, 7(2), 2011.

[7] Emmanuel Beffara. *Logique, Réalisabilité et Concurrence. (Logic, Realisability and Concurrency)*. PhD thesis, Paris Diderot University, France, 2005.

[8] Emmanuel Beffara. A concurrent model for linear logic. *Electr. Notes Theor. Comput. Sci.*, 155:147–168, 2006.

[9] Emmanuel Beffara. A logical view on scheduling in concurrency. In *Proceedings Fifth International Workshop on Classical Logic and Computation, CL&C 2014, Vienna, Austria, July 13, 2014.*, pages 78–92, 2014.

[10] Emmanuel Beffara. Unifying type systems for mobile processes. *CoRR*, abs/1505.07794, 2015.

[11] Emmanuel Beffara and François Maurel. Concurrent nets: A study of prefixing in process calculi. *Theor. Comput. Sci.*, 356(3):356–373, 2006.

[12] Emmanuel Beffara and Virgile Mogbil. Proofs as executions. In *Theoretical Computer Science - 7th IFIP TC 1/WG 2.2 International Conference, TCS 2012, Amsterdam, The Netherlands, September 26-28, 2012. Proceedings*, pages 280–294, 2012.

[13] Gianluigi Bellin and Philip J. Scott. On the pi-calculus and linear logic. *Theor. Comput. Sci.*, 135(1):11–65, 1994.

[14] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theor. Comput. Sci.*, 195(2):205–226, 1998.

[15] Pierre-Louis Curien. Introduction to linear logic and ludics, part I. *CoRR*, abs/cs/0501035, 2005.

[16] Pierre-Louis Curien. Introduction to linear logic and ludics, part II. *CoRR*, abs/cs/0501039, 2005.

[17] Pierre-Louis Curien and Claudia Faggian. L-nets, strategies and proof-nets. In *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*, pages 167–183, 2005.

[18] Pierre-Louis Curien and Claudia Faggian. L-nets, strategies and proof-nets. In *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*, pages 167–183, 2005.

[19] Pierre-Louis Curien and Claudia Faggian. An approach to innocent strategies as graphs. *Inf. Comput.*, 214:119–155, 2012.

[20] Pierre-Louis Curien and Hugo Herbelin. Abstract machines for dialogue games. *CoRR*, abs/0706.2544, 2007.

[21] Vincent Danos and Laurent Regnier. The structure of multiplicatives. *Arch. Math. Log.*, 28(3):181–203, 1989.

[22] Thomas Ehrhard and Olivier Laurent. Interpreting a finitary pi-calculus in differential interaction nets. *Inf. Comput.*, 208(6):606–633, 2010.

[23] Thomas Ehrhard and Laurent Regnier. Differential interaction nets. *Theor. Comput. Sci.*, 364(2):166–195, 2006.

[24] Claudia Faggian and Martin Hyland. Designs, disputes and strategies. In *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings*, pages 442–457, 2002.

[25] Claudia Faggian and François Maurel. Ludics nets, a game model of concurrent interaction. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pages 376–385, 2005.

[26] Claudia Faggian and Mauro Piccolo. Ludics is a model for the finitary linear pi-calculus. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, pages 148–162, 2007.

[27] Claudia Faggian and Mauro Piccolo. Partial orders, event structures and linear strategies. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, pages 95–111, 2009.

[28] Christophe Fouqueré and Myriam Quatrini. Incarnation in ludics and maximal cliques of paths. *Logical Methods in Computer Science*, 9(4), 2013.

[29] Christophe Fouqueré and Myriam Quatrini. Ludics characterization of multiplicative-additive linear behaviours. *CoRR*, abs/1403.3772, 2014.

[30] Christophe Fouqueré and Myriam Quatrini. Study of behaviours via visitable paths. *Submitted to journal*, 2016.

[31] G. Gentzen. Untersuchungen über das logische schließen i. *Mathematische Zeitschrift*, 39:176–210, 1935.

[32] G. Gentzen. Untersuchungen über das logische schließen ii. *Mathematische Zeitschrift*, 39:405–431, 1935.

[33] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[34] Jean-Yves Girard. Geometry of interaction 2: deadlock-free algorithms. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, pages 76–93, 1988.

[35] Jean-Yves Girard. Geometry of interaction 1: Interpretation of system f. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic*

Colloquium '88, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221 – 260. Elsevier, 1989.

[36] Jean-Yves Girard. Geometry of interaction (abstract). In *CONCUR '94, Concurrency Theory, 5th International Conference, Uppsala, Sweden, August 22-25, 1994, Proceedings*, page 1, 1994.

[37] Jean-Yves Girard. On the meaning of logical rules i: syntax vs. semantics. *Computational Logic (U. Bergerand H. Schwichtenberg eds) Heidelberg Springer-Verlag*, pages 215–272, 1999.

[38] Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[39] Jean-Yves Girard. From foundations to ludics. *Bulletin of Symbolic Logic*, 9(2):131–168, 2003.

[40] Jean-Yves Girard. *Le Point Aveugle I*, volume 1. Hermann, 2006.

[41] Jean-Yves Girard. *Le Point Aveugle II*, volume 2. Hermann, 2007.

[42] Kohei Honda and Olivier Laurent. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theor. Comput. Sci.*, 411(22-24):2223–2238, 2010.

[43] Kohei Honda, Nobuko Yoshida, and Martin Berger. An observationally complete program logic for imperative higher-order functions. *Theor. Comput. Sci.*, 517:75–101, 2014.

[44] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: i, ii, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[45] Martin Hyland and Andrea Schalk. Games on graphs and sequentially realizable functionals. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 257–264, 2002.

[46] T.Basten J.C.M. Baeten and M.A. Reiniers. *Process Algebra: Equational Theories of Communicating Processes*, volume 50. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2010.

[47] Naoki Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998.

[48] Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, pages 233–247, 2006.

[49] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 358–371, 1996.

[50] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, pages 489–503, 2000.

[51] Yves Lafont. Interaction nets. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 95–108, 1990.

[52] Kim G. Larsen Luca Aceto and Anna Ingolfsdottir. An introduction to milner's ccs. *lecture notes*, 2015.

[53] Paul-André Melliès. Asynchronous games 2: The true concurrency of innocence. *Theor. Comput. Sci.*, 358(2-3):200–228, 2006.

[54] Dale Milner. The π-calculus as a theory in linear logic: Preliminary result. *Lecture Notes in Computer Science*, 660(2-3):200–228, 1992.

[55] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[56] Robin Milner. Calculi for synchrony and asynchrony. *Theor. Comput. Sci.*, 25:267–310, 1983.

[57] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[58] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

[59] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theor. Comput. Sci.*, 13:85–108, 1981.

[60] Myriam Quatrini. *La Ludique : une théorie de l'interaction, de la logique mathématique au langage naturel*. Habilitation à diriger des recherches, Université Aix-Marseille, May 2014.

[61] Grzegorz Rozenberg and P. S. Thiagarajan. Petri nets: Basic notions, structure, behaviour. In *Current Trends in Concurrency, Overviews and Tutorials*, pages 585–668. 1986.

[62] Davide Sangiorgi. Internal mobility and agent-passing calculi. In *Automata, Languages and Programming, 22nd International Colloquium, ICALP95, Szeged, Hungary, July 10-14, 1995, Proceedings*, pages 672–683, 1995.

[63] Davide Sangiorgi. pi-calculus, internal mobility, and agent-passing calculi. *Theor. Comput. Sci.*, 167(1&2):235–274, 1996.

[64] Eijiro Sumii and Naoki Kobayashi. A generalized deadlock-free process calculus. *Electr. Notes Theor. Comput. Sci.*, 16(3):225–247, 1998.

[65] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, pages 398–413, 1994.

[66] Kazushige Terui. Computational ludics. *Theor. Comput. Sci.*, 412(20):2048–2071, 2011.

[67] Daniele Varacca and Nobuko Yoshida. Typed event structures and the linear pi-calculus. *Theor. Comput. Sci.*, 411(19):1949–1973, 2010.

[68] Glynn Winskel. Event structure semantics for CCS and related languages. In *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*, pages 561–576, 1982.

[69] Glynn Winskel. Event structures. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, pages 325–392, 1986.

[70] Glynn Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, pages 364–397, 1988.

[71] Glynn Winskel and Mogens Nielsen. Models for concurrency. *DAIMI Report Series*, 21(429), 1992.

[72] Glynn Winskel and Mogens Nielsen. *Models for Concurrency*, volume 4. Oxford Clarendon Press, 1995.